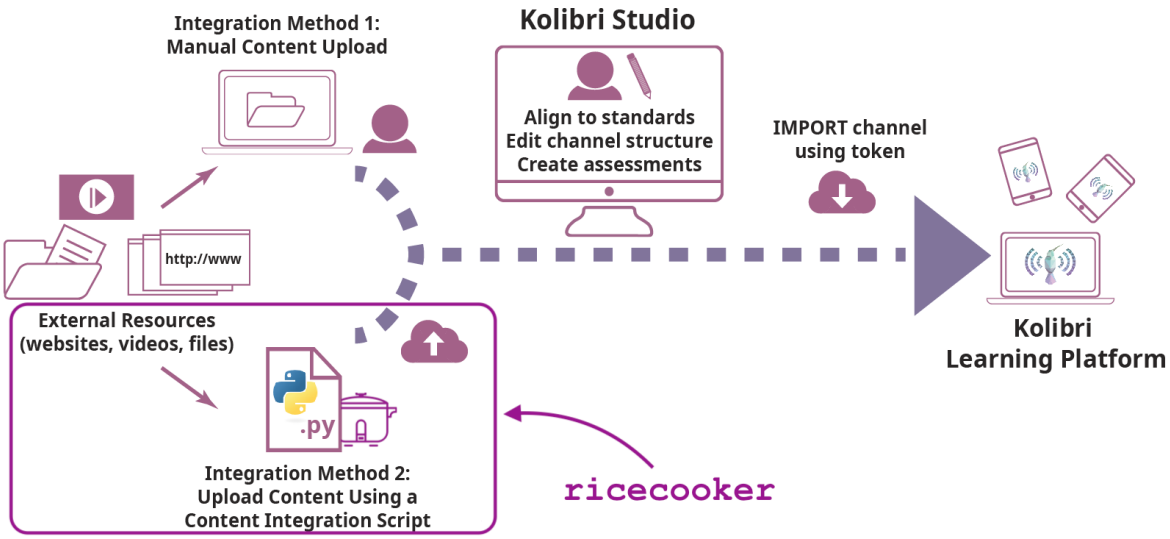
Ricecooker

Kolibri Content Integration Tools



Published by Learning Equality

Also available online at ricecooker.readthedocs.io

Contents

The Kolibri channels used by Kolibri are created using Kolibri Studio, which is a central content repository, an online channel editor, and the home of the Kolibri Content Library. Using `ricecooker` you can convert existing educational content (documents, audios, videos, HTML apps, etc.) into Kolibri channels ready to be uploaded to Kolibri Studio and used offline in Kolibri.

The basic process of getting new content into Kolibri is as follows:

- **UPLOAD** your content to Kolibri Studio using one of two methods: (1) manually uploading through the Kolibri Studio web interface, or (2) using a content integration script based on the `ricecooker` framework.
- PUBLISH the channel on Kolibri Studio to make it accessible for use in Kolibri.
- IMPORT the channel into Kolibri using the channel token displayed in Kolibri Studio after the PUBLISH step is done.

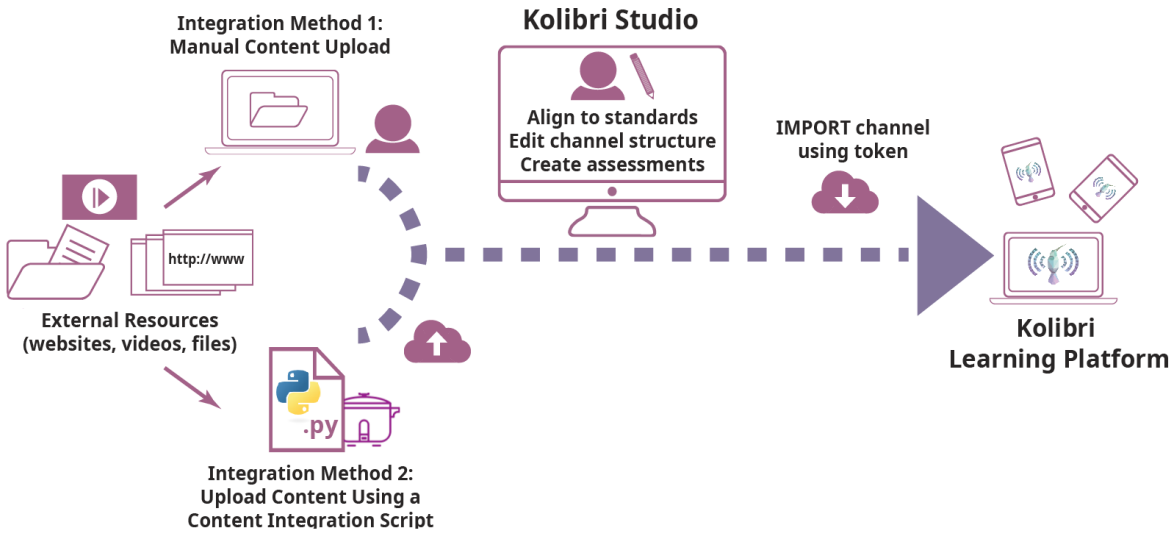


Fig. 1: **Content flow within the Kolibri ecosystem.** External content sources (left) are first uploaded to Kolibri Studio (middle) so they can be imported into the Kolibri Learning Platform (right).

Chapter 1

Installation

You can install `ricecooker` by running the command `pip install ricecooker`, which will install the Python package and all its Python dependencies. You'll need version 3.5 or higher of Python to use the `ricecooker` framework, as well as some software for media file conversions (`ffmpeg` and `poppler`).

In the next fifteen minutes or so, we'll setup your computer with all these things so you can get started writing your first content integration scripts.

1.1 System prerequisites

The first step will will be to make sure you have `python3` installed on your computer and two additional file conversion tools: `ffmpeg` for video compression, and the `poppler` library for manipulating PDFs.

Jump to the specific instructions for your operating system, and be sure to try the *Checklist* commands to know the installation was successful.

1.1.1 Linux

On a Debian or Ubuntu GNU/Linux, you can install the necessary packages using:

```
sudo apt-get install  git python3 ffmpeg poppler-utils
```

You may need to adjust the package names for other Linux distributions (CentOS/Fedora/OpenSuSE).

Checklist: verify your python version is 3.5 or higher by running `python3 --version`. If no `python3` command exists, then try `python --version`. Run the commands `ffmpeg -h` and `pdftoppm -h` to make sure they are available.

1.1.2 Mac

Mac OS X users can install the necessary software using Homebrew:

```
brew install  git python3 ffmpeg poppler
```

Checklist: verify you python version is 3.5 or higher by running `python3 --version`. Also run the commands `ffmpeg -h` and `pdftoppm -h` to make sure they are available.

1.1.3 Windows

On windows the process is a little longer since we'll have to download and install several programs and make sure their `bin`-directories are added to the `Path` variable:

1. Download Python from https://www.python.org/downloads/windows/. Look under the **Python 3.7.x** heading and choose the "Windows x86-64 executable installer" option to download the latest installer and follow usual installation steps. During the installation, make sure to check the box **"Add Python 3.7 to path"**.
   - *Checklist*: after installation, open a new command prompt (`cmd.exe`) and type in `python --version` and `pip --version` to make sure the commands are available.
2. Download `ffmpeg` from https://github.com/BtbN/FFmpeg-Builds/releases/download/latest/ffmpeg-master-latest-win64-gpl.zip. Extract the zip file to a permanent location where you store your code, like `C:\Users\User\Tools` for example. Next, you must add the `bin` folder that contains `ffmpeg` (e.g. `C:\Users\User\Tools\ffmpeg-4.1.4-win64-static\bin`) to your user Path variable following these instructions.
   - *Checklist*: Open a new command prompt and type in `ffmpeg -h` and `ffprobe -h` to verify the commands `ffmpeg` and `ffprobe` are available on your Path.
3. Download the file linked under "Latest binary" from poppler-windows. You will need to download and install 7-zip to "unzip" the `.7z` archive. Extract the file to a some permanent location in your files. Add the `bin` folder `poppler-0.xx.y\bin` to your Path variable.
   - *Checklist*: after installation, open a command terminal and type in `pdftoppm -h` to make sure the command `pdftoppm` is available.

We recommend you also download and install Git from https://git-scm.com/downloads. Using git is not a requirement for the getting started, but it's a great tool to have for borrowing code from others and sharing back your own code on the web.

If you find the text descriptions to be confusing, you can watch this video walkthrough that shows the installation steps and also explains the adding-to-Path process.

1.2 Installing Ricecooker

To install the `ricecooker` package, simply run this command in a command prompt:

```
pip install ricecooker
```

You will see lots of lines scroll on the screen as `pip`, the package installer for Python, installs all the Python packages required to create content integration scripts.

Reporting issues: If you run into problems or encounter an error in any of the above steps, please let us know by opening an issue on github.

Okay so now we have all the system software and Python libraries installed. Let's get started!

Chapter 2

Getting started

The purpose of these pages is to help you learn how to use the `ricecooker` framework.

2.1 Getting started tutorial

This short guide will walk you through the steps required to create a new Kolibri channel by using a content integration script (a.k.a. sushi chef script) based on the `ricecooker` framework. We'll write a Python script that creates a simple channel containing one PDF file, and run this script to upload the PDF to Kolibri Studio so it can then be imported in Kolibri.

Note: The software tools developed by the Learning Equality content team have food-related code names. Content integration scripts are called "sushi chefs" since they perform the detail-oriented task of taking external educational resources (fish), packaging them as individual Kolibri content nodes (sushi), and organizing them neatly into Kolibri channels (lunchbox).

2.1.1 Installation

If you haven't done so already, go through the steps on the installation page to install the `ricecooker` Python package and other system prerequisites.

2.1.2 Obtaining a Studio access token

You'll need a Kolibri Studio Access Token to create channels using ricecooker scripts. To obtain this token:

1. Create an account on Kolibri Studio
2. Navigate to the Settings page on Kolibri Studio.
3. Copy the given access token to a safe place on your computer.

You must pass the token on the command line as `--token=<your-access-token>` when calling your chef script. Alternatively, you can create a file to store your token and pass in the command line argument `--token=path/to/studiotoken.txt`.

2.1.3 Video overview

Watch this video tutorial to learn how to create a new content integration script and set the required channel metadata fields like `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID`.

2.1.4 Creating a sushichef script

In a new folder on your computer, create a file called `sushichef.py` with the code contents shown below. Alternatively you can use the "Save as" option on this link to achieve the same result. We'll use this simple example of a content integration script for this tutorial.

```
#!/usr/bin/env python
from ricecooker.chefs import SushiChef
from ricecooker.classes.files import DocumentFile
from ricecooker.classes.licenses import get_license
from ricecooker.classes.nodes import DocumentNode
from ricecooker.classes.nodes import TopicNode


class SimpleChef(SushiChef):
    channel_info = {
        "CHANNEL_TITLE": "Potatoes info channel",
        "CHANNEL_SOURCE_DOMAIN": "<yourdomain.org>",  # where content comes from
        "CHANNEL_SOURCE_ID": "<unique id for the channel>",  # CHANGE ME!!!
        "CHANNEL_LANGUAGE": "en",  # le_utils language code
        "CHANNEL_THUMBNAIL": "https://upload.wikimedia.org/wikipedia/commons/b/b7/
A_Grande_Batata.jpg",  # (optional)
```

(continues on next page)

(continued from previous page)

```
        "CHANNEL_DESCRIPTION": "What is this channel about?",  # (optional)
    }

    def construct_channel(self, **kwargs):
        channel = self.get_channel(**kwargs)
        potato_topic = TopicNode(title="Potatoes!", source_id="<potatoes_id>")
        channel.add_child(potato_topic)
        document_node = DocumentNode(
            title="Growing potatoes",
            description="An article about growing potatoes on your rooftop.",
            source_id="pubs/mafri-potatoe",
            license=get_license("CC BY", copyright_holder="University of Alberta"),
            language="en",
            files=[
                DocumentFile(
                    path="https://www.gov.mb.ca/inr/pdf/pubs/mafri-potatoe.pdf",
                    language="en",
                )
            ],
        )
        potato_topic.add_child(document_node)
        return channel


if __name__ == "__main__":
    """
    Run this script on the command line using:
        python sushichef.py  --token=YOURTOKENHERE9139139f3a23232
    """
    simple_chef = SimpleChef()
    simple_chef.main()
```

The code above is the equivalent of a “Hello, World!” content integration script based on the `ricecooker` framework that will create a Kolibri channel with a single topic node (Folder), and put a single PDF content node inside that folder.

As you can tell from the above code sample, most of the code in a content integration script is concerned with setting the right metadata for files, content nodes, topics nodes (folders), and the overall channel. This will be the running theme when you work on content integration scripts.

Attention: You need to modify the value of `CHANNEL_SOURCE_ID` before you continue, otherwise you'll get an error when you run the script in the next step. The combination of `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID` serve to create the channel's unique ID. If you use the same values as an already existing channel, you will either get a permissions error, or if you have editing permissions, you could overwrite the channel contents. Therefore, you want to be careful to use different values from the default ones used in the sample code.

2.1.5 Running the sushichef script

You can run a chef script by calling it on the command line:

```
python sushichef.py  --token=<your-access-token>
```

The most important argument when running a chef script is `--token`, which is used to pass in the Studio Access Token and authenticates you in Kolibri Studio. To see all the `ricecooker` command line options, run `python sushichef.py -h`. For more details about running chef scripts see the chefops page.

Note: If you get an error when running this command, make sure you have replaced `<your-access-token>` with the token you obtained from Kolibri Studio. Also make sure you've changed the value of `channel_info['CHANNEL_SOURCE_ID']` instead of using the value in the sample code.

If the command succeeds, you should see something like this printed in your terminal:

```
In SushiChef.run method. args={'command': 'uploadchannel', 'token': '
<your-access-token>', 'update': False, 'resume': False, 'stage': True, 'publish':
False} options={}
Logged in with username you@yourdomain.org
Ricecooker v0.6.42 is up-to-date.

* Starting channel build process *

Calling construct_channel...
   Setting up initial channel structure...
   Validating channel structure...
      Potatoes info channel (ChannelNode): 2 descendants
         Potatoes! (TopicNode): 1 descendant
```

```
            Growing potatoes (DocumentNode): 1 file
   Tree is valid

Downloading files...
Processing content...
    --- Downloaded 3641693a88b37e8d0484c340a83f9364.pdf
    --- Downloaded 290c80ed7ce4cf117772f29dda76413c.jpg
   All files were successfully downloaded


Checking if files exist on Kolibri Studio...
    Got file diff for 2 out of 2 files
Uploading files...
Uploading 0 new file(s) to Kolibri Studio...

Creating channel...
Creating tree on Kolibri Studio...
   Creating channel Potatoes info channel
    (0 of 2 uploaded)    Processing Potatoes info channel (ChannelNode)
    (1 of 2 uploaded)       Processing Potatoes! (TopicNode)
       All nodes were created successfully.
Upload time: 0.896938s


DONE: Channel created at https://api.studio.learningequality.org/channels/
47147660ecb850bfb71590bf7d1ca971/staging
```

Congratulations, you put the potatoes on the internet! You're probably already a legend in Ireland!

2.1.5.1 What just happened?

As you can tell from the above messages, running a `sushichef.py` involves all kinds of steps that are orchestrated by the ricecooker framework:

- The channel structure is created based on the output of the method `construct_channel` of the `SimpleChef` class
- The tree structure and metadata are validated
- All referenced files are downloaded locally (saved in the `storage` directory)
- New files are uploaded to Kolibri Studio (in the above case no new files are uploaded because the files already exist on Studio from a previous run)
- The channel structure and metadata is uploaded to Kolibri Studio

- A link is printed for you to view the channel draft you just uploaded

If you're interested, you can read this page to learn about the tech details behind these steps, but the details are not important for now. Let's continue to follow your channel's journey by clicking the Kolibri Studio link.

2.1.6 View your channel in Kolibri Studio

At the end of the chef run the complete channel (files and metadata) will be uploaded to a "draft version" of the channel called a "staging tree". Use the **DEPLOY** button in the Studio web interface to take your channel out of "draft mode." This step is normally important for reviewing changes between the new draft version and the current version of the channel.

The next step is to PUBLISH your channel using the button on Studio. The **PUBLISH** action exports all the channel metadata and files in the format that is used by Kolibri and so it is needed in order to import you channel in Kolibri.

At the end of the **PUBLISH** step, you will be able to see the **channel token** associated with your channel, which is a short two-word string that you'll use in the next step. You'll also receive an email notification telling you when the channel has finished publishing.

Tip: Running the chef script with the command arguments `--deploy --publish` will perform both the **DEPLOY** and **PUBLISH** actions after the chef run completes. This combination of arguments can be used for testing and development, but never for "production" channels, which must be reviewed before deploying.

2.1.7 Import your channel in Kolibri

The final step is to IMPORT your channel into Kolibri using the channel token you obtained after the Kolibri Studio PUBLISH step finished.

Congratulations! Thanks to your Python skills and perseverence through this multi-step process involving three software systems, you finally have access to your content in the offline-capable Kolibri Learning Platform.

This topic node "Potatoes!" is nice to look at no doubt, but it feels kind of empty. Not to worry—in the next step of this tutorial we'll learn how to add more nodes to your channel. Before that let's do a quick recap of what we've learned thus far.

2.1.8 Recap and next steps

We can summarize the entire process we the Kolibri channel followed through the three parts of the Kolibri ecosystem using the following diagram:

```
sushichef(ricecooker)       Kolibri Studio             Kolibri
UPLOADCHANNEL------->-------DEPLOY+PUBLISH------>------IMPORT (using channel token)
```

I know it seems like a complicated process, but you'll get used to it after going through it a couple of times. All the steps represent *necessary* complexity. The automated extraction and packaging of source materials ricecooker into Kolibri channels provides the "raw materials" on which educators can build by reusing and remixing in Kolibri Studio. **Ultimately the technical effort you invest in creating content integration scripts will benefit learners and teachers all around the world, this week and for years to come.** So get the metadata right!

As your next step for learning about Kolibri channels, we propose an optional, non-technical activity to get to know Kolibri Studio better. After that we'll resume the ricecooker training with the hands-on tutorial. If you're in a hurry, and want to skip ahead to API reference docs pages, check out content nodes and files.

2.1.8.1 Try the manual upload (optional)

Redo the steps from this tutorial but this time using the Kolibri Studio web interface. The manual upload process (Integration Method 1) is can be described as follows:

```
Kolibri Studio             Kolibri
UPLOAD+PUBLISH------>------IMPORT (using channel token)
```

Login to Kolibri Studio and try these steps:

1. Create a new channel
2. Add a topic node to your channel
3. Add content by uploading a PDF document (note which metadata fields are required and which are optional)
4. Use the ADD > Import from Channels feature to import the **Growing potatoes** document node from the **Potatoes info channel**.
5. PUBLISH your channel (a new channel token will be generated).
6. IMPORT your channel in Kolibri using the channel token.

Most of the channel creation operations steps you can do using `ricecooker`, you can also do through the Kolibri Studio web interface—in both cases you're creating Kolibri channels, that can be *PUBLISH*-ed and used offline in Kolibri.

Note: Channels created using a content integration script (ricecooker channels), cannot be modified manually through the Kolibri Studio web interface. This is because manual changes would get overwritten and lost on next chef runs. If you want to make manual edits/tweaks to the channel, you can create a "derivative channel" and import the content from the ricecooker channel using the **ADD** > **Import from Channels** feature as in step 4 above.

2.1.8.2 Hands-on tutorial

Now that we have a working example of a simple chef, we're ready to extend it by adding other kinds of nodes (nutritional groups) and `files` (ingredients). The next section will take you through a hands-on tutorial where you'll learn how to use the different content kinds and file types supported by the ricecooker framework to create Kolibri channels.

2.2 The jiro command line tool

New in 0.7

Named after the master sushi chef, the `jiro` command line tool provides a command line interface around sushi chef scripts and simplifies common tasks when operating on sushi chef scripts.

It has the following commands:

2.2.1 jiro new

```
jiro new [script name]
```

Create a new sushi-chef script. Will create a `sushi-chef-[name]` directory unless the command is run within a directory of that name.

2.2.2 jiro remote

Used to manage Studio server upload authentication.

```
jiro remote add [remote-name] [url] <token>
```

Registers a new Studio instance to upload to.

- `remote-name` is a short string used to refer to this server in `jiro` commands.
- `URL` should be the fully qualified URL to the root of a Studio server

- `token` - if specified, the token to use to authenticate to the server. If not specified, you will be prompted to provide the token before your first upload.

`jiro remote list`

List the Studio servers you have registered with `jiro`.

2.2.3 jiro prepare

`jiro prepare`

This command should be run in the root script directory containing `sushichef.py`.

Downloads content and creates the channel tree, but skips the upload step. Often used while iterating on scripts.

2.2.4 jiro serve

`jiro serve <remote-name>`

This command should be run in the root script directory containing `sushichef.py`.

Runs the same steps as `jiro prepare`, but also uploads the results to Studio. If `remote-name` is specified, it will upload to the remote server registered with that name. Otherwise, it will upload to production Studio.

If you have never registered an API token for the Studio server you're uploading to, it may prompt you to enter it when running this command.

2.3 Hands-on tutorial

In this tutorial, you'll start with a basic content integration script (sushi chef) and extend the code to construct a bigger channel based on your own content. In the process you'll learn about all the features of the `ricecooker` framework.

2.3.1 Prerequisite steps

The steps in this tutorial assume you have:

1. Completed the Installation steps
2. Created an account on Kolibri Studio and obtained your access token, which you'll need to to use instead of the text `<your-access-token>` in the examples below
3. Successfully managed to run the basic chef example in the Getting started tutorial

2.3.2 Step 1: Setup your environment

Create a directory called `tutorial` where you will run this code. In general it is recommended to have separate directories for each content integration script you will be working on. In order to prepare for the upcoming **Step 6**, find a `.pdf` document, a small `.mp4` video file, and an `.mp3` audio file. Save these files somewhere inside the `tutorial` directory.

2.3.3 Step 2: Copy the sample code

To begin, download the sample code from here and save it as the file *sushichef.py* in the tutorial directory.

Note all the `TODO` items in the code. These are the places left for you to edit.

2.3.4 Step 3: Edit the channel metadata

1. Open your terminal and `cd` into the folder where `sushichef.py` is located.
2. Open `sushichef.py` in a text editor.
3. Change `<yourdomain.org>` to any domain. The source domain specifies who is supplying the content.
4. Change `<yourid>` to any id. The source_id will distinguish your channel from other channels.
5. Change `The Tutorial Channel` to any channel name.

Try running the sushi chef by entering the following command in your terminal:

```
python sushichef.py  --token=<your-access-token>
```

Click the link to Kolibri Studio that shows up in the final step and make sure your channel looks OK.

2.3.5 Step 4: Create a Topic

1. Locate the first **TODO** in the `sushichef.py` file. Here, you will create your first topic.
2. Copy/paste the example code and change `exampletopic` to `mytopic`.
3. Set the `source_id` to be something other than `topic-1` (the `source_id` will distinguish your node from other nodes in the tree)
4. Set the title.
5. Go to the next **TODO** and add `mytopic` to channel (use example code as guide)

```
Check Run sushi chef from your terminal. Your channel should look like this:
Channel
| Example Topic
| Your Topic
```

2.3.6 Step 5: Create a Subtopic

1. Go to the next **TODO** in the `sushichef.py` file. Here, you will create a subtopic
2. Copy/paste the example code and change `examplesubtopic` to `mysubtopic`
3. Set the `source_id` and `title`
4. Go to the next **TODO** and add `mysubtopic` to `mytopic` (use example code as guide)

```
Check Run the sushi chef from your terminal. Your channel should look like this:
Channel
| Example Topic
|      | Example Subtopic
| Your Topic
|      | Your Subtopic
```

2.3.7 Step 6: Create Files

1. Go to the next **TODO** in the sushichef.py file. Here, you will create a pdf file
2. Copy/paste the example code and change `examplepdf` to `mypdf`. `DocumentFile(...)` will automatically download a pdf file from the given path.
3. Set the `source_id`, the `title`, and the `path` (any url to a pdf file)
4. Repeat steps 1-3 for video files and audio files.
5. Finally, add your files to your channel (see last ** statements)

```
Check: Run the sushi chef from your terminal. Your channel should look like this:
Channel
| Example Topic
|      | Example Subtopic
|      |      | Example Audio
|      |  Example Video
| Your Topic
|      | Your Subtopic
|      |      | Your Audio
|      | Your Video
| Example PDF
| Your PDF
```

2.3.8 Next steps

You're now ready to start writing your own content integration scripts. The following links will guide you to the next steps:

- Ricecooker API reference
- Code examples
- Learn about the ricecooker utilities and helpers

2.4 Explanations

This page provides more details about the code structure and the metadata needs, which we glossed over in the getting started page.

Feel free to skip these explanations if you're in a hurry, but remember to come back here and learn the code and metadata technical details, as this knowledge will be very helpful when you try to create larger, more complicated channels.

2.4.1 How does a chef script work?

Let's look at the chef code we used in the getting started tutorial and walk through and comment on the most important parts of the code.

```
#!/usr/bin/env python
from ricecooker.chefs import SushiChef
from ricecooker.classes.nodes import TopicNode, DocumentNode
from ricecooker.classes.files import DocumentFile
from ricecooker.classes.licenses import get_license

class SimpleChef(SushiChef):                                                    # (1)
    channel_info = {                                                            # (2)
        'CHANNEL_TITLE': 'Potatoes info channel',
        'CHANNEL_SOURCE_DOMAIN': 'gov.mb.ca',                                   # change me!

        'CHANNEL_SOURCE_ID': 'website_docs',                                    # change me!

        'CHANNEL_LANGUAGE': 'en',
        'CHANNEL_THUMBNAIL': 'https://upload.wikimedia.org/wikipedia/commons/b/b7/
A_Grande_Batata.jpg',
        'CHANNEL_DESCRIPTION': 'A channel about potatoes.',
    }

    def construct_channel(self, **kwargs):
        channel = self.get_channel(**kwargs)                                    # (3)
        potato_topic = TopicNode(title="Potatoes!", source_id="patates")        # (4)
        channel.add_child(potato_topic)                                         # (5)
        doc_node = DocumentNode(                                                # (6)
            title='Growing potatoes',
            description='An article about growing potatoes on your rooftop.',
            source_id='inr/pdf/pubs/mafri-potatoe.pdf',
            author=None,
            language='en',                                                      # (7)
            license=get_license('CC BY', copyright_holder='U. of Alberta'),     # (8)
            files=[
              DocumentFile(                                                     # (9)
                path='https://www.gov.mb.ca/inr/pdf/pubs/mafri-potatoe.pdf', #
(10)
```

(continues on next page)

(continued from previous page)

```
                language='en',                                                    #
(11)
            )
        ],
    )
    potato_topic.add_child(doc_node)
    return channel

if __name__ == '__main__':                                                        #
(12)
    """
    Run this script on the command line using:
        python simple_chef.py  --token=YOURTOKENHERE9139139f3a23232
    """
    simple_chef = SimpleChef()
    simple_chef.main()                                                            #
(13)
```

2.4.1.1 Ricecooker Chef API

To use the `ricecooker` library, you create a **sushi chef** scripts that define a subclass of the base class `ricecooker.chefs.SushiChef`, as shown at (1) in the code. By extending `SushiChef`, your chef class will inherit all the standard functionality provided by the `ricecooker` framework.

2.4.1.2 Channel metadata

A chef class should have the attribute `channel_info` (dict), which contains the metadata for the channel, as shows on line (2). Define the `channel_info` as follows:

```
    channel_info = {
        'CHANNEL_TITLE': 'Channel name shown in UI',
        'CHANNEL_SOURCE_DOMAIN': '<sourcedomain.org>',
        'CHANNEL_SOURCE_ID': '<some unique identifier>',     #
        'CHANNEL_LANGUAGE': 'en',                            # use language codes
from le_utils
        'CHANNEL_THUMBNAIL': 'http://yourdomain.org/img/logo.jpg', # (optional)
local path or url to a thumbnail image
```

(continues on next page)

(continued from previous page)

```
        'CHANNEL_DESCRIPTION': 'What is this channel about?',      # (optional)
longer description of the channel
    }
```

The `CHANNEL_SOURCE_DOMAIN` identifies the domain name of the organization that produced or is hosting the content (e.g. `khanacademy.org` or `youtube.com`). The `CHANNEL_SOURCE_ID` must be set to some unique identifier for this channel within the domain (e.g. `KA-en` for the Khan Academy English channel). The combination of `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID` is used to compute the `channel_id` for the Kolibri channel you're creating.

2.4.1.3 Construct channel

The code responsible for building the structure of the channel your channel by adding `TopicNodes`, `ContentNodess`, files, and exercises questions lives here. This is where most of the work of writing a chef script happens.

You chef class should have a method with the signature:

```
def construct_channel(self, **kwargs) -> ChannelNode:
    ...
```

To write the `construct_channel` method of your chef class, start by getting the `ChannelNode` for this channel by calling `self.get_channel(**kwargs)`. An instance of the `ChannelNode` will be constructed for you, from the metadata provided in `self.channel_info`. Once you have the `ChannelNode` instance, the rest of your chef's `construct_channel` method is responsible for constructing the channel by adding various `Nodes` objects to the channel using `add_child`.

2.4.1.4 Topic nodes

Topic nodes are folder-like containers that are used to organize the channel's content. Line (4) shows how to create a `TopicNode` (folder) instance titled "Potatoes!". Line (5) shows how to add the newly created topic node to the channel. You can use topic nodes to build arbitrary hierarchies based on subject, language, grade levels, or any other organizational structure that is best suited for the specific content source. Reach out to the Learning Equality content team if you're not sure how to structure your channel. We've got experience with both technical and curriculum aspects of creating channels and will be able to guide you to a structure that best first the needs of learners and teachers.

2.4.1.5 Content nodes

The `ricecooker` library provides classes like `DocumentNode`, `VideoNode`, `AudioNode`, etc., to store the metadata associate with content items. Each content node also has one or more files associated with it, `EPubFile`, `DocumentFile`, `VideoFile`, `AudioFile`, `ThumbnailFile`, etc.

Line (6) shows how to create a `DocumentNode` to store the metadata for a pdf file. The `title` and `description` attributes are set. We also set the `source_id` attribute to a unique identifier for this document. The document does not specify authors, so we set the `author` attribute to `None`.

On (7), we set `language` attribute to the internal language code `en`, to indicate the content node is in English. We use the same language code later on line (11) to indicate the file contents are in English. The Python package `le-utils` defines the internal language codes used throughout the Kolibri platform (e.g. `en`, `es-MX`, and `zul`). To find the internal language code for a given language, you can locate it in the lookup table, or use one of the language lookup helper functions defined in `le_utils.constants.languages`.

Line (8) shows how we set the `license` attribute to the appropriate instance of `ricecooker.classes.licenses.License`. All non-topic nodes must be assigned a license upon initialization. You can obtain the appropriate license object using the helper function `get_license` defined in `ricecooker.classes.licenses`. Use the predefined license ids given in `le_utils.constants.licenses` as the first argument to the `get_license` helper function.

2.4.1.6 Files

On lines (9, 10, and 11), we create a `DocumentFile` instance and set the appropriate `path` and `language` attributes. Note that `path` can be either a local filesystem path, or a web URL (as in the above example). Paths that point to web URLs will be downloaded automatically when the chef runs and cached locally. Note the default ricecooker behaviour is to cache downloaded files forever. Use the `--update` argument to bypass the cached and re-download all files. The `--update` must be used whenever files are modified but the path stays the same.

2.4.1.7 Command line interface

You can run your chef script by passing the appropriate command line arguments:

```
./sushichef.py --token=YOURTOKENHERE9139139f3a23232
```

The most important argument when running a chef script is `--token` which is used to pass in the Studio Access Token obtained in Step 1.

To see the full list of `ricecooker` command line options, run `./sushichef.py -h`. For more details about running chef scripts see the *chefops page*.

2.4.2 Deploying the channel

At the end of the chef run the complete channel (files and metadata) will be uploaded to "draft version" of the channel called a "`staging` tree". The purpose of the staging tree is to allow channel editors can to review the changes in the "draft version" as compared to the current version of the channel. Use the **DEPLOY** button in the Studio web interface to activate the "draft copy" and make it visible to all Studio users.

2.4.3 Publishing the channel

The **PUBLISH** button on Studio is used to save and export a new version of the channel. The **PUBLISH** action exports all the channel metadata to a sqlite3 DB file served by Studio at the URL `/content/{{channel_id}}.sqlite3` and ensure the associated files exist in `/content/storage/` which is served by a CDN. This step is a prerequisite for getting the channel out of Studio and into Kolibri.

2.4.4 Next steps

After these tutorial and explanations, you are ready to take things into your own hands and learn about:

- *Content Nodes*
- *File types*
- *Exercises*
- *Parsing HTML* and creating *HTML5 apps*
- *Command line arguments* for controlling chef operation, managing caches, and other options
- See also the Cheffing techniques doc which provides links to tips and code examples for handling various special cases and content sources.

Chapter 3

Concepts

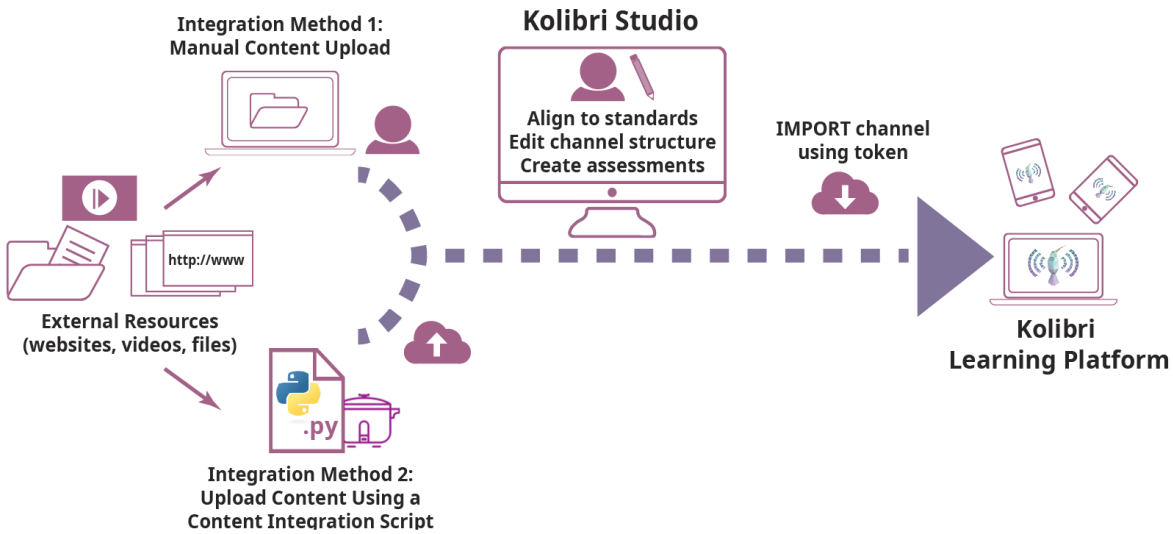
The purpose of this page is to help you understand how content integration work fits more broadly within the Kolibri ecosystem.

The links below establish the shared vocabulary to facilitate communication between content partners, the Learning Equality team, Kolibri users, and other stakeholders.

3.1 Kolibri content ecosystem overview

Educational content in the Kolibri platform is organized into **content channels**. The `ricecooker` framework is used for creating content channels and uploading them to Kolibri Studio, which is the central content server that Kolibri applications talk to when importing their content.

Content flow within the Kolibri ecosystem is pictured below.



This `ricecooker` framework is the main tool used to facilitate **Integration Method 2**.

3.1.1 Kolibri channels

A Kolibri channel is the combination of a topic tree (a nested folder structure) and number of self-contained "content items" packaged for offline use and distribution. Each content item within the channel is represented as a content node with one or more files associated with it. In summary, a channel is a nested structure of `TopicNodes` (folders) that contain `ContentNode` objects similar to how files are organized into folders on computers.

The Kolibri channel is the fundamental structure common to all parts of the Kolibri ecosystem: the Kolibri Learning Platform is where Kolibri channels are used by learners and teachers, Kolibri Studio is the editor for Kolibri Channels (think five Rs), and Ricecooker scripts are used for content integrations that pull in OER from external sources, package them for offline use, and upload them to Kolibri Studio.

3.1.2 Supported content kinds

Kolibri channels are tree-like structures that consist of the following types of nodes:

- **Topic nodes** (folders): the nested folders structure is the is main way of representing structured content in Kolibri. Depending on the particular channel, a topic node could be a language, a subject, a course, a unit, a module, a section, a lesson, or any other structural element. Rather than impose a particular fixed structure, we let educators decide the folder structure that is best suited for the learners needs.
- **Content nodes**:
  - Document (either an `epub` or a `pdf` file)
  - Audio (`mp3` files of audio lessons, audiobooks, podcasts, radio shows, etc.)
  - Video (`mp4` files with `h264` video codec and `aac` audio codec)
  - HTML5App (`zip` files containing web content like HTML, JavaScript, css and images)
  - H5PApp (self-contained `h5p` files)
  - Slideshow (a sequence of `jpg` and `png` slide images)
  - Exercises containing questions like multiple choice, multiple selection, and numeric inputs

3.1.3 Further reading

- Kolibri channel as explained in the Kolibri documentation.
- Kolibri Studio User Guide

3.2 Terminology

This page lists key concepts and technical terminology used as part of the content integration work within Learning Equality.

3.2.1 Content Pipeline

The combination of software tools and procedures used to convert content from an external content source to becoming a Kolibri Channel available for use in the Kolibri Learning Platform. The Kolibri Content Pipeline is a collaborative effort between educational experts and software developers.

3.2.2 Channel Spec

A content specification document, or Channel Spec, is a blueprint document that specifies the structure of the Kolibri channel that is to be created.

Channel Specs are an important aspect of the content integration process for two reasons:

1. It specifies what needs to be done. The channel spec establishes an agreement between the curriculum specialist and the developer who will be writing the content integration script.
2. It serves to define when the work is done. Used as part of the review process to know when the channel is "Spec Compliant," i.e. the channel structure in Kolibri matches the blueprint.

A Channel Spec document includes the following information:

- Channel Title: usually of the form `{Source Name} ({lang})` where `{Source Name}` is chosen to be short and descriptive, and `{lang}` is included in the title to make it easy to search for content in this language.
- Channel Description: a description (up to 400 characters) of the channel and its contents.
- Languages: notes about content language, and special handling for multilingual content, subtitles, or missing translations
- Files Types: info about what content kinds and file types to look for
- Channel Structure: a specification of the desired topic structure for the channel. This is the key element in the Channel Spec and often requires domain expertise to take into account the needs of the teachers and learners who will be accessing this content.

- Links and sample content
- Credentials: info about how to access the content (e.g. info about API access)
- Technical notes: The Channel Spec can include guidance about technical aspects like content transformations (for example, the need to compress the videos so that they take up less space).

For more info about each of these aspects, see the section "Creating a Content Channel Spec" in the Kolibri Content Integration Guide.

3.2.3 Content Integration Script (aka SushiChef)

The content integration scripts that use the `ricecooker` library to generate Kolibri Channels are commonly referred to as **SushiChef** scripts. The responsibility of a `SushiChef` script is to download the source content, perform any necessary format or structure conversions to create a content tree viewable in Kolibri, then to upload the output of this process to Kolibri Studio for review and publishing.

Conceptually, `SushiChef` scripts are very similar to web scrapers, but with specialized functions for optimizing the content for Kolibri's data structures and capabilities.

3.3 Content integration methods

There are two methods that you can use to create Kolibri channels:

- **Manual content upload**: This method is suitable for content that is saved on your local computer such as files or folders. You can directly upload your content through the Kolibri Studio web interface. This method is appropriate for small and medium content sets. See the Kolibri Studio User Guide for more information.
- **Uploading content using a content integration script**: You can use a content integration script (a.k.a. sushichef script) to integrate content from websites, content repositories, APIs, or other external sources. A content integration script is a Python program.

More information about each of these methods provided below.

3.3.1 Manual content upload

You can use the Kolibri Studio web interface to upload various content types and organize them into channels. Kolibri Studio allows you to explore pre-organized libraries of open educational resources, and reuse them in your channels. You can also add tags, re-order, re-mix content, and create exercises to support student's learning process.

To learn more about Studio, we recommend reading the following pages in the Kolibri Studio User Guide:

- Accessing Studio
- Working with channels
- Adding content to channels

When creating large channels (100+ content items) or channels that need to be updated regularly, you should consider using a content integration script, as described below.

3.3.2 Content integration scripts

The `ricecooker` framework is a tool that programmers can use to upload content to Kolibri Studio in an automated fashion. We refer to these import scripts as **sushi chefs**, because their job is to chop-up the source material (e.g. an educational website) and package the content items into tasty morsels (content items) with all the associated metadata.

Using the bulk import option requires the a content developer (sushi chef author) to prepare the content, content metadata, and run the chef script to perform the upload to Kolibri Studio.

Educators and content specialists can assist the developers by preparing a **spec sheet** for the content source that provides detailed guidance for how content should be structured and organized within the channel. The content specialist also plays a role during the channel review process.

The following alternative options are available for specifying the metadata for content nodes that can be used in special circumstances.

3.3.2.1 CSV metadata workflow

In addition to the web interface and the Python interface (`ricecooker`), there exists an option for creating Kolibri channels by:

- Organizing content items (documents, videos, mp3 files) into a folder hierarchy on the local file system
- Specifying metadata in the form of CSV files created using Excel

The CSV-based workflow is a good fit for non-technical users since it doesn't require writing any code, but instead can use Excel to provide all the metadata.

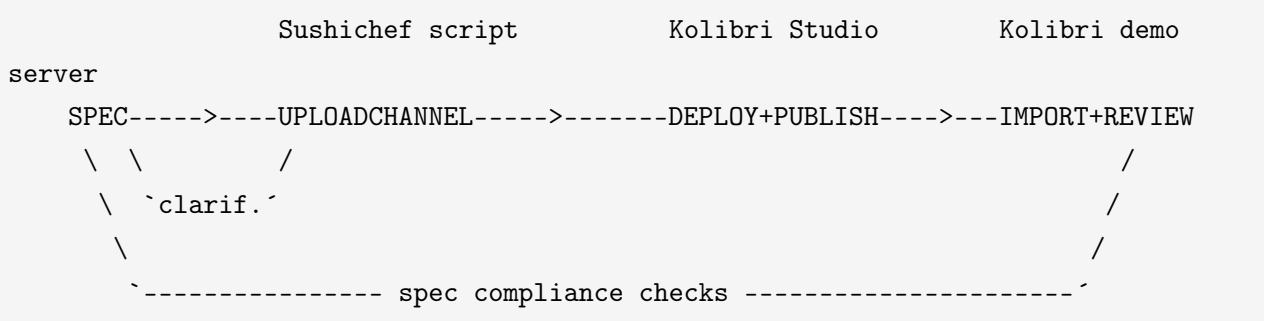
- CSV-based workflow README
- Example content folder
- Example Channel.csv metadata file
- Example Content.csv metadata file
- CSV-based exercises info

Organizing the content into folders and creating the CSV metadata files is most of the work, and can be done by non-programmers. The generic sushi chef script (`LineCook`) is then used to upload the channel.

3.4 Developer workflows

3.4.1 The Kolibri UPLOADCHANNEL-PUBLISH-IMPORT content workflow

Running a sushichef script is only one of the steps in a channel's journey within the Kolibri platform. Here is the full picture:



It is the responsibility of the chef author to take a content channel all the way through this workflow and make sure that the final channel works in Kolibri.

Notes on specific steps:

- `SPEC`: the **channel spec** describes the target channel structure, licensing, and technical notes about content transformations that might be necessary. All this information will be available on the notion card for this source.
- `UPLOADCHANNEL`: the main task of the chef author is to implement all the extraction and content transformation described in the spec. If anything in the spec is missing or requires further clarification, post comments on the Notion card. If you run into any kind of difficulties during the cheffing process, post a question in the LE slack channel `#sushi-chefs` and someone from the content team will be able to assist you. For example, "Hello @here I'm having trouble with the {{cookiecutter.channel_name}} chef because X and Y cannot be organized according to the spec because Z."
- `DEPLOY` and `PUBLISH`: once the channel is uploaded to Kolibri Studio you can preview the structure and update the information on the notion card. Use the `DEPLOY` button to take the channel out of "draft mode" (staging). The next step is to export the channel in the format necessary for use in Kolibri using the `PUBLISH` button on Studio. The PUBLISH action exports all the channel metadata to a sqlite3 DB file `https://studio.learningequality.org/content/databases/{channel_id}.sqlite3` the first time a channel is PUBLISH-ed a secret token is generated

that can be used to import the channel in Kolibri.

- IMPORT: the next step is to import your channel into a Kolibri instance. You can use Kolibri installed on your local machine or an online demo server. Admin (devowner user) credentials for the demo server will be provided for you so that you can import and update the channel every time you push a new version. Follow these steps to import your channel Device > IMPORT > KOLIBRI STUDIO (online) > Try adding a token, add the channel token, select all nodes > IMPORT.
- REVIEW: You can now go to the Kolibri Learn tab and preview your channel to see it the way learners will see it. Take the time to click around and browse the content to make sure everything works as expected. Update the notion card and leave a comment. For example "First draft of channel uploaded to demo server." This would be a good time to ask a member of the LE content team to review the channel. You can do this using the @Person Name in your notion comment. Consult the content source notion card to know who the relevant people to tag. For example, you can @-comment the Library person on the card to ask them to review the channel—be sure to specify the channel's "level of readiness" in your comment, e.g., if it's a draft version for initial feedback, or the near-final, spec-compliant version ready for detailed review and QA. For async technical questions tag the SushOps person on the card or post your question in the #sushi-chefs channel. For example, "I downloaded this html and js content, but it doesn't render right in Kolibri because of the iframe sandboxing." or "Does anyone have sample code for extracting content X from a shared drive link Y of type Z?".

3.4.2 The Kolibri UPLOADCHANNEL-PUBLISH-UPDATE content workflow

The process is similar to the initial upload and import, but some steps are different because a version of the channel is already available:

- UPLOADCHANNEL: to upload a new version of a content channel, simply re-run the chef script. The newly uploaded channel tree will replace the old tree on Studio (assuming the source_domain and source_id of the channel haven't changed).
  - The new version of the channel will be uploaded to a "staging tree" instead of replacing the current tree, which will allow you to preview the changes to the channel on Studio before replacing the existing tree. Use the DEPLOY button on Studio to complete the upload and replace the current tree.
  - By default, ricecooker will cache all files that have been previously uploaded to Studio in order to avoid the need to download and compress files each time the chef runs. You can pass the --update command line argument to the chef script to bypass this caching mechanism. This is useful when the source files have changed, but their URLs and filenames haven't changed.
- DEPLOY and PUBLISH: every time you upload a new content to your channel, you must repeat the deploy and publish step on Studio to regenerate the sqlite3 DB file that is used for import-

ing into Kolibri. At the end of the PUBLISH process, the channel version number will increase by one. The channel token stays the same.

- `UPDATE`: the process of updating a channel in Kolibri is similar to the `IMPORT` step described above, but the actions are different because a version of the channel is already available. Go to the `Device` > `Channels` page in Kolibri and use the `OPTIONS` button next to your channel, select `Import more` > `KOLIBRI STUDIO (online)` > `UPDATE`, select all nodes then click `IMPORT`. Updating a channel takes much less time than the initial import because Kolibri only needs to import the new files that have been added to the channel. Once the update process is complete, you can notify the relevant members of the LE content team to let them know a new version is available for review.

3.4.3 Rubric

Use the following rubric as a checklist to know when a sushi chef script is done:

3.4.3.1 Main checks

1. Does the channel correspond to the spec provided?
2. Does the content render as expected when viewed in Kolibri?

3.4.3.2 Logistic checks

1. Is the channel uploaded to Studio and `PUBLISH`-ed?
2. Is the channel imported to a demo server where it can be previewed?
3. Is the information about the channel token, the studio URL, and demo server URL on the notion card up to date? See the Studio Channels table. If a card for your channel doesn't exist yet, you can create one using the `[+ New]` button at the bottom of the table.

3.4.3.3 Metadata checks

1. Do all nodes have appropriate titles?
2. Do all nodes have appropriate descriptions (when available in the source)?
3. Is the correct language code set on all nodes and files?
4. Is the `license` field set to the correct value for all nodes?
5. Is the `source_id` field set consistently for all content nodes? Try to use unique identifiers based on the source website or permanent url paths.

3.5 Reviewing Kolibri content channels

Every content channel on the Kolibri platform benefits from a the review process that ensures the content structure, metadata, and functionality is up to standard. This is broadly referred to as "channel review," "providing feedback," or "QA." Everyone on the LE team is a potential channel reviewer, and external partners can also be asked to review channels when they have capacity.

3.5.1 Issue tracker

Channel reviewers can use the "Issue tracker" table to report problems so that developers responsible for creating the channel can address them.

3.5.1.1 Issue tracker columns

- `Issue ID`: internal numeric identifier (or `github:nn` for two-way-synced issues with the chef's github repo)
- `Type` (multi select): what type of issue is this (see full list of options below)
- `Severity` (Blocker || Nice to have): how bad is the issue
- `URL`: A link to studio, a demo server, or the source website where the issue is visible
- `Screenshots` (files): screenshot that shows the issue in action
- `Issue description` (text): provide detailed description of what the issue is, how to reproduce, and any additional info (e.g. copy-paste of errors from the JavaScript console)
- `Possible fixes` (text): provide suggestions (technical or not) for how issue could be fixed and ideas for workarounds
- `Assigned to` (notion user): track the person that is supposed to fix this issue
- `Status` (Not started||In progress||Fixed): track progress on issue fix
- `Created`: record the date when the issue was added
- `Created by`: record who filed the issue

Issue types

- `Missing content`: some content from the source was not imported
- `Structure`: problem with the channel structure
- `Title`: problem with titles, e.g. titles that are too long or not informative
- `Description`: use to flag description problems (non-informative or repeating junk text)
- `Metadata`: problem with metadata associated with nodes (language, licensing info, author, role visibility, tags)
- `Thumbnails`: flag broken or missing thumbnails on the channel, topics, or content nodes

- `Display issue`: the content doesn't look right (HTML/CSS issues) or doesn't work as expected (JavaScript issues)
- `Learning UX`: any problem that might interfere with learning user experience
- `Video compression`: if videos are not compressed enough (files too large) or alternatively too compressed (cannot read text)
- `Bulk corrections`: flag issues that might require bulk metadata edits on numerous content nodes
- `Translation`: content files or metadata are partially or completely in the wrong language
- `Enhancement`: use to keep track of possible enhancements or additions that could be made to improve coach or learner experience

Issue severity

- `Blocker`: this issue must be fixed before the channel can go into QA
- `Nice to have`: non-blocking issues like corrections, enhancements, and minor learning UX problems

3.5.2 Who can be a channel reviewer?

You can. Whenever you need a distraction, take 20 minutes and place yourself in the learner's shoes and go explore the channel on the demo server link provided on the notion card. If you notice any issues while browsing, add them to the Issue tracker table. That's it. Learn something today.

Chapter 4

Examples

Below are some examples that demonstrate certain aspects of the content integration process that require careful consideration and are best explained in code:

```
{
    "cells": [
        {
            "cell_type": "markdown", "metadata": {}, "source": [
                "# Languagesn", "n", "This tutorial will explain how to set the *language* property for various nodes and file objects when using the *ricecooker* framework."
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "## Explore language objects and language codesn", "n", "First we must import the *le-utils* pacakge. The languages supported by Kolibri and the Content Curation Server are provided in *le_utils.constants.languages*.n"
            ]
        }, {
            "cell_type": "code", "execution_count": 1, "metadata": {}, "outputs": [
                {
                  "data": {
                    "text/plain": [
                        "Language(native_name='English', primary_code='en', subcode=None, name='English', ka_name=None)"
                    ]
                  }, "execution_count": 1, "metadata": {}, "output_type": "execute_result"
                }
```

], “source”: [
“from le_utils.constants import languagesn”, “n”, “n”, “# can lookup language using language coden”, “language_obj = languages.getlang(‘en’)n”, “language_obj”
]
}, {
“cell_type”: “code”, “execution_count”: 2, “metadata”: {}, “outputs”: [
{
“data”: {
“text/plain”: [
“Language(native_name=’English’, primary_code=’en’, subcode=None, name=’English’, ka_name=None)”
]
}, “execution_count”: 2, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# can lookup language using language name (the new le_utils version has not shipped yet)n”, “language_obj = languages.getlang_by_name(‘English’)n”, “language_obj”
]
}, {
“cell_type”: “code”, “execution_count”: 3, “metadata”: {}, “outputs”: [
{
“data”: {
“text/plain”: [
“‘en’”
]
}, “execution_count”: 3, “metadata”: {}, “output_type”: “execute_result”
}
], “source”: [
“# all *language* attributed (channel, nodes, and files) need to use language coden”, “language_obj.code”
]
}, {
“cell_type”: “code”, “execution_count”: 4, “metadata”: {}, “outputs”: [
{
“name”: “stdout”, “output_type”: “stream”, “text”: [
“Language(native_name=’Français, langue française’,

```
                primary_code='fr', subcode=None, name='French',
                ka_name='francais')n", "frn"
            ]
        }
    ], "source": [
        "from le_utils.constants.languages import getlang_by_native_namen",
        "n", "lang_obj = getlang_by_native_name('français')n",
        "print(lang_obj)n", "print(lang_obj.code)n"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "The above language code is an internal representaiton that uses two-
        letter codes, and sometimes has locale information, e.g., pt-BR for
        Brazilian Portuiguese. Sometimes the internal code representaiton for
        a language is the three-letter vesion, e.g., zul for Zulu."
    ]
}, {
    "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [],
    "source": []
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
        "## Create chef classn", "n", "We now create subclass of rice-
        cooker.chefs.SushiChef and defined its get_channel and con-
        struct_channel methods.n", "n", "For the purpose of this example,
        we'll create three topic nodes in different languages that contain one
        document in each."
    ]
}, {
    "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [],
    "source": [
        "from ricecooker.chefs import SushiChefn", "from rice-
        cooker.classes.nodes import ChannelNode, TopicNode, Docu-
        mentNoden", "from ricecooker.classes.files import Document-
        Filen", "from le_utils.constants import licensesn", "n", "from
        le_utils.constants.languages import getlangn", "n", "n", "n", "class
        MultipleLanguagesChef(SushiChef):n", " """n", " A sushi chef that cre-
        ates a channel with content in EN, FR, and SP.n", " """n", " channel_info
        = {n", " 'CHANNEL_TITLE': 'Languages test channel',n", " 'CHAN-
        NEL_SOURCE_DOMAIN': '<yourdomain.org>', # where you got the
        contentn", " 'CHANNEL_SOURCE_ID': '<unique id for channel>', #
```

```
            channel's unique id CHANGE ME!!n", " 'CHANNEL_LANGUAGE': getlang('mul').code, # set global language for channeln", " 'CHANNEL_DESCRIPTION': 'This channel contains nodes in multiple languages',n", " 'CHANNEL_THUMBNAIL': None, # (optional)n", " }n", "n", " def construct_channel(self, **kwargs):n", " # create channeln", " channel = self.get_channel(**kwargs)n", "n", " # create the English topic, add a DocumentNode to itn", " topic = TopicNode(n", " source_id="<en_topic_id>",n", " title="New Topic in English",n", " language=getlang('en').code,n", " )n", " doc_node = DocumentNode(n", " source_id="<en_doc_id>",n", " title='Some doc in English',n", " description='This is a sample document node in English',n", " files=[DocumentFile(path='samplefiles/documents/doc_EN.pdf')],n", " license=licenses.PUBLIC_DOMAIN,n", " language=getlang('en').code,n", " )n", " topic.add_child(doc_node)n", " channel.add_child(topic)n", "n", " # create the Spanish topic, add a DocumentNode to itn", " topic = TopicNode(n", " source_id="<es_topic_id>",n", " title="Topic in Spanish",n", " language=getlang('es-MX').code,n", " )n", " doc_node = DocumentNode(n", " source_id="<es_doc_id>",n", " title='Some doc in Spanish',n", " description='This is a sample document node in Spanish',n", " files=[DocumentFile(path='samplefiles/documents/doc_ES.pdf')],n", " license=licenses.PUBLIC_DOMAIN,n", " language=getlang('es-MX').code,n", " )n", " topic.add_child(doc_node)n", " channel.add_child(topic)n", "n", " # create the French topic, add a DocumentNode to itn", " topic = TopicNode(n", " source_id="<fr_topic_id>",n", " title="Topic in French",n", " language=languages.getlang('fr').code,n", " )n", " doc_node = DocumentNode(n", " source_id="<fr_doc_id>",n", " title='Some doc in French',n", " description='This is a sample document node in French',n", " files=[DocumentFile(path='samplefiles/documents/doc_FR.pdf')],n", " license=licenses.PUBLIC_DOMAIN,n", " language=getlang('fr').code,n", " )n", " topic.add_child(doc_node)n", " channel.add_child(topic)n", "n", " return channeln"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Run of you chef by creating an instance of the chef class and calling it's *run* method:"
        ]
    }, {
```

“cell_type”: “code”, “execution_count”: 6, “metadata”: {}, “outputs”: [
{
“name”: “stderr”, “output_type”: “stream”, “text”: [
“u001b[32mINFO u001b[0m u001b[34mIn SushiChef.run method. args={‘command’: ‘dryrun’, ‘reset’: True, ‘verbose’: True, ‘token’: ‘YOURTO…’} options={}u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mn”, “n”, “* Starting channel build process *n”, “n”, “u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mCalling construct_channel… u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Setting up initial channel structure… u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Validating channel structure…u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Languages test channel (ChannelNode): 6 descendantsu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m New Topic in English (TopicNode): 1 descendantu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Some doc in English (DocumentNode): 1 fileu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Topic in Spanish (TopicNode): 1 descendantu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Some doc in Spanish (DocumentNode): 1 fileu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Topic in French (TopicNode): 1 descendantu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Some doc in French (DocumentNode): 1 fileu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m Tree is validn”, “u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mDownloading files…u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mProcessing content…u001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mt— Downloaded e8b1fe37ce3da500241b4af4e018a2d7.pdfu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mt— Downloaded cef22cce0e1d3ba08861fc97476b8ccf.pdfu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mt— Downloaded 6c8730e3e2554e6eac0ad79304bbcc68.pdfu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34m All files were successfully downloadedu001b[0mn”, “u001b[32mINFO u001b[0m u001b[34mCommand is dryrun so we are not uploading chanel.u001b[0mn”

```
            ]
        }
    ], “source”: [
        “mychef = MultipleLanguagesChef()n”, “args = {n”, “ ‘command’: ‘dryrun’, # use ‘uploadchannel’ for real runn”, “ ‘verbose’: True,n”, “ ‘token’: ‘YOURTOKENHERE9139139f3a23232’n”, “}n”, “options = {}n”, “mychef.run(args, options)”
    ]
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “Congratulations, you put three languages on the internet!”
    ]
}, {
    “cell_type”: “code”, “execution_count”: null, “metadata”: {}, “outputs”: [], “source”: []
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “## Example 2: YouTube video with subtitles in multiple languagesn”, “n”, “You can use the library *youtube_dl* to get lots of useful metadata about videos and playlists, including the which language subtitle are vailable for a video.n”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 7, “metadata”: {}, “outputs”: [
        {
            “name”: “stdout”, “output_type”: “stream”, “text”: [
                “[youtube] FN12ty5ztAs: Downloading webpagen”, “[youtube] FN12ty5ztAs: Downloading MPD manifestn”, “dict_keys([‘en’, ‘fr’, ‘zu’])n”
            ]
        }
    ], “source”: [
        “import youtube_dln”, “n”, “ydl = youtube_dl.YoutubeDL({n”, “ #’quiet’: True,n”, “ ‘no_warnings’: True,n”, “ ‘writesubtitles’: True,n”, “ ‘allsubtitles’: True,n”, “})n”, “n”, “n”, “youtube_id = ‘FN12ty5ztAs’n”, “n”, “info = ydl.extract_info(youtube_id, download=False)n”, “subtitle_languages = info["subtitles"].keys()n”, “n”, “print(subtitle_languages)”
    ]
}, {
    “cell_type”: “code”, “execution_count”: null, “metadata”: {}, “outputs”: [],
```

“source”: [
“n”
]
}, {
“cell_type”: “markdown”, “metadata”: {}, “source”: [
“### Full sushi chef examplen”, “n”, “The *YoutubeVideoWithSubtitlesSushiChef* class below shows how to create a channel with youtube video and upload subtitles files with all available languages.”
]
}, {
“cell_type”: “code”, “execution_count”: 10, “metadata”: {}, “outputs”: [], “source”: [
“from ricecooker.chefs import SushiChefn”, “from ricecooker.classes import licensesn”, “from ricecooker.classes.nodes import ChannelNode, TopicNode, VideoNoden”, “from ricecooker.classes.files import YouTubeVideoFile, YouTubeSubtitleFilen”, “from ricecooker.classes.files import is_youtube_subtitle_file_supported_languagen”, “n”, “n”, “import youtube_dln”, “ydl = youtube_dl.YoutubeDL({n”, “ ‘quiet’: True,n”, “ ‘no_warnings’: True,n”, “ ‘writesubtitles’: True,n”, “ ‘allsubtitles’: True,n”, “})n”, “n”, “n”, “# Define the license object with necessary infon”, “TE_LICENSE = licenses.SpecialPermissionsLicense(n”, “ description=’Permission granted by Touchable Earth to distribute through Kolibri.’,n”, “ copyright_holder=’Touchable Earth Foundation (New Zealand)’n”, “)n”, “n”, “n”, “class YoutubeVideoWithSubtitlesSushiChef(SushiChef):n”, “ """n”, “ A sushi chef that creates a channel with content in EN, FR, and SP.n”, “ """n”, “ channel_info = {n”, “ ‘CHANNEL_SOURCE_DOMAIN’: ‘<yourdomain.org>’, # where you got the contentn”, “ ‘CHANNEL_SOURCE_ID’: ‘<unique id for channel>’, # channel’s unique id CHANGE ME!!n”, “ ‘CHANNEL_TITLE’: ‘Youtube subtitles downloading chef’,n”, “ ‘CHANNEL_LANGUAGE’: ‘en’,n”, “ ‘CHANNEL_THUMBNAIL’: ‘https://edoc.coe.int/4115/postcard-47-flags.jpg’,n”, “ ‘CHANNEL_DESCRIPTION’: ‘This is a test channel to make sure youtube subtitle languages lookup works’n”, “ }n”, “n”, “ def construct_channel(self, **kwargs):n”, “ # create channeln”, “ channel = self.get_channel(**kwargs)n”, “n”, “ # get all subtitles available for a sample videon”, “ youtube_id =’FN12ty5ztAs’n”, “ info = ydl.extract_info(youtube_id, download=False)n”, “ subtitle_languages = info["subtitles"].keys()n”, “ print(‘Found subtitle_languages = ‘,

```
            subtitle_languages)n", " n", " # create video noden", " video_node
            = VideoNode(n", " source_id=youtube_id,n", " title='Youtube
            video',n", " license=TE_LICENSE,n", " derive_thumbnail=True,n",
            "  files=[YouTubeVideoFile(youtube_id=youtube_id)],n", "
            )n", "n", " # add subtitles in whichever languages are
            available.n", " for lang_code in subtitle_languages:n", " if
            is_youtube_subtitle_file_supported_language(lang_code):n",
            "  video_node.add_file(n", " YouTubeSubtitleFile(n", "
            youtube_id=youtube_id,n", " language=lang_coden", " )n", " )n", "
            else:n", " print('Unsupported subtitle language code:', lang_code)n",
            "n", " channel.add_child(video_node)n", "n", " return channeln", "n", " "
        ]
    }, {
        "cell_type": "code", "execution_count": 11, "metadata": {}, "outputs": [
            {
                "name": "stderr", "output_type": "stream", "text": [
                    "u001b[32mINFO u001b[0m u001b[34mIn SushiChef.run
                    method. args={'command': 'dryrun', 'reset': True, 'ver-
                    bose': True, 'token': 'YOURTO…'} options={}u001b[0mn",
                    "u001b[32mINFO u001b[0m u001b[34mn", "n", "*
                    Starting channel build process *n", "n", "u001b[0mn",
                    "u001b[32mINFO u001b[0m u001b[34mCalling con-
                    struct_channel…  u001b[0mn", "u001b[32mINFO
                    u001b[0m u001b[34m Setting up initial chan-
                    nel structure…  u001b[0mn", "u001b[32mINFO
                    u001b[0m u001b[34m Validating channel struc-
                    ture…u001b[0mn", "u001b[32mINFO u001b[0m
                    u001b[34m Youtube subtitles downloading chef (Chan-
                    nelNode): 1 descendantu001b[0mn", "u001b[32mINFO
                    u001b[0m u001b[34m Youtube video (VideoNode):
                    4 filesu001b[0mn", "u001b[32mINFO u001b[0m
                    u001b[34m Tree is validn", "u001b[0mn", "u001b[32mINFO
                    u001b[0m u001b[34mDownloading files…u001b[0mn",
                    "u001b[32mINFO u001b[0m u001b[34mProcessing
                    content…u001b[0mn", "u001b[32mINFO
                    u001b[0m u001b[34mt— Downloaded (YouTube)
                    a144a6af6977247684d2a3977dc6f841.mp4u001b[0mn"
                ]
            }, {
                "name": "stdout", "output_type": "stream", "text": [
```

```
            "Found subtitle_languages = dict_keys(['en', 'fr', 'zu'])n"
          ]
        }, {
          "name": "stderr", "output_type": "stream", "text": [
            "u001b[32mINFO u001b[0m u001b[34mt— Downloaded 5f22a71e53271eb2d2abe013457a625d.jpgu001b[0mn",
            "u001b[31mERROR u001b[0m u001b[34m 3 file(s) have failed to downloadu001b[0mn", "u001b[33mWARNING u001b[0m u001b[34mtVideo FN12ty5ztAs: http://www.youtube.com/watch?v=FN12ty5ztAs n", "t Subtitle with langauge en is not available for http://www.youtube.com/watch?v=FN12ty5ztAsu001b[0mn",
            "u001b[33mWARNING u001b[0m u001b[34mtVideo FN12ty5ztAs: http://www.youtube.com/watch?v=FN12ty5ztAs n", "t Subtitle with langauge fr is not available for http://www.youtube.com/watch?v=FN12ty5ztAsu001b[0mn",
            "u001b[33mWARNING u001b[0m u001b[34mtVideo FN12ty5ztAs: http://www.youtube.com/watch?v=FN12ty5ztAs n", "t Subtitle with langauge zul is not available for http://www.youtube.com/watch?v=FN12ty5ztAsu001b[0mn",
            "u001b[32mINFO u001b[0m u001b[34mCommand is dryrun so we are not uploading chanel.u001b[0mn"
          ]
        }
      ], "source": [
        "chef = YoutubeVideoWithSubtitlesSushiChef()n", "args = {n", " 'command': 'dryrun', # use 'uploadchannel' for real runn", " 'verbose': True,n", " 'token': 'YOURTOKENHERE9139139f3a23232'n", "}n", "options = {}n", "chef.run(args, options)"
      ]
    }, {
      "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": []
    }, {
      "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": []
    }, {
```

```
            "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [],
            "source": []
        }, {
            "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [],
            "source": []
        }
    ], "metadata": {
        "kernelspec": {
            "display_name": "Python 3", "language": "python", "name": "python3"
        }, "language_info": {
            "codemirror_mode": {
                "name": "ipython", "version": 3
            }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python",
            "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "ver-
            sion": "3.6.9"
        }
    }, "nbformat": 4, "nbformat_minor": 2
}
{
    "cells": [
        {
            "cell_type": "markdown", "metadata": {}, "source": [
                "# ricecooker exercisesn", "n", "n", "This mini-tutorial will walk you through
                the steps of running a simple chef script ExercisesChef that creates two ex-
                ercises nodes, and four exercises questions.n", "n", "n", "### Running the
                notebooksn", "To follow along and run the code in this notebook, you'll
                need to clone the ricecooker repository, crate a virtual environement, in-
                stall ricecooker using pip install ricecooker, install Jypyter notebook using
                pip install jupyter, then start the jupyter notebook server by running jupyter
                notebook. You will then be able to run all the code sections in this notebook
                and poke around."
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "### Creating a Sushi Chef classn", "n"
            ]
        }, {
            "cell_type": "code", "execution_count": 1, "metadata": {}, "outputs": [], "source":
            [
```

“from ricecooker.chefs import SushiChefn”, “from ricecooker.classes.nodes import TopicNode, ExerciseNoden”, “from ricecooker.classes.questions import SingleSelectQuestion, MultipleSelectQuestion, InputQuestion, PerseusQuestionn”, “from ricecooker.classes.licenses import get_licensen”, “from le_utils.constants import licensesn”, “from le_utils.constants import exercisesn”, “from le_utils.constants.languages import getlangn”, “n”, “n”, “class ExercisesChef(SushiChef):n”, “ channel_info = {n”, “ ‘CHANNEL_TITLE’: ‘Sample Exercises’,n”, “ ‘CHANNEL_SOURCE_DOMAIN’: ‘<yourdomain.org>’, # where you got the contentn”, “ ‘CHANNEL_SOURCE_ID’: ‘<unique id for channel>’, # channel’s unique id CHANGE MEn”, “ ‘CHANNEL_LANGUAGE’: ‘en’, # le_utils language coden”, “ ‘CHANNEL_DESCRIPTION’: ‘A test channel with different types of exercise questions’, # (optional)n”, “ ‘CHANNEL_THUMBNAIL’: None, # (optional)n”, “ }n”, “n”, “ def construct_channel(self, **kwargs):n”, “ channel = self.get_channel(**kwargs)n”, “ topic = TopicNode(title="Math Exercises", source_id="folder-id")n”, “ channel.add_child(topic)n”, “n”, “ exercise_node = ExerciseNode(n”, “ source_id=’<some unique id>’,n”, “ title=’Basic questions’,n”, “ author=’LE content team’,n”, “ description=’Showcase of the simple question type supported by Ricecooker and Studio’,n”, “ language=getlang(‘en’).code,n”, “ license=get_license(licenses.PUBLIC_DOMAIN),n”, “ thumbnail=None,n”, “ exercise_data={n”, “ ‘mastery_model’: exercises.M_OF_N, # \n”, “ ‘m’: 2, # learners must get 2/3 questions correct to complete exercisen”, “ ‘n’: 3, # /n”, “ ‘randomize’: True, # show questions in random ordern”, “ },n”, “ questions=[n”, “ MultipleSelectQuestion(n”, “ id=’sampleEX_Q1’,n”, “ question = "Which numbers the following numbers are even?",n”, “ correct_answers = ["2", "4",],n”, “ all_answers = ["1", "2", "3", "4", "5"],n”, “ hints=[‘Even numbers are divisible by 2.’],n”, “),n”, “ SingleSelectQuestion(n”, “ id=’sampleEX_Q2’,n”, “ question = "What is 2 times 3?",n”, “ correct_answer = "6",n”, “ all_answers = ["2", "3", "5", "6"],n”, “ hints=[‘Multiplication of a by b is like computing the area of a rectangle with length a and width b.’],n”, “),n”, “ InputQuestion(n”, “ id=’sampleEX_Q3’,n”, “ question = "Name one of the *factors* of 10.",n”, “ answers = ["1", "2", "5", "10"],n”, “ hints=[‘The factors of a number are the divisors of the number that leave a whole remainder.’],n”, “)n”, “]n”, “)n”, “ topic.add_child(exercise_node)n”, “n”, “ # LOAD JSON DATA (as string)

```
            FOR PERSEUS QUESTIONS n", " RAW_PERSEUS_JSON_STR =
            open('../../examples/exercises/chefdata/perseus_graph_question.json',
            'r').read()n", " # orn", " # import re-
            questsn", " # RAW_PERSEUS_JSON_STR = re-
            quests.get('https://raw.githubusercontent.com/learningequality/sample-
            channels/master/contentnodes/exercise/perseus_graph_question.json').textn",
            " exercise_node2 = ExerciseNode(n", " source_id='<another unique
            id>',n", " title='An exercise containing a perseus question',n", "
            author='LE content team',n", " description='An example exercise
            with a Persus question',n", " language=getlang('en').code,n", " li-
            cense=get_license(licenses.CC_BY, copyright_holder='Copyright
            holder name'),n", " thumbnail=None,n", " exercise_data={n",
            " 'mastery_model': exercises.M_OF_N,n", " 'm': 1,n", "
            'n': 1,n", " },n", " questions=[n", " PerseusQuestion(n", "
            id='ex2bQ4',n", " raw_data=RAW_PERSEUS_JSON_STR,n",
            " source_url='https://github.com/learningequality/sample-
            channels/blob/master/contentnodes/exercise/perseus_graph_question.json'n",
            " ),n", " ]n", " )n", " topic.add_child(exercise_node2)n", "n", " return
            channeln"
        ]
    }, {
        "cell_type": "markdown", "metadata": {
            "collapsed": true
        }, "source": [
            "### Running the chefn"
        ]
    }, {
        "cell_type": "markdown", "metadata": {}, "source": [
            "Run of you chef by creating an instance of the chef class and calling
            it's *run* method:"
        ]
    }, {
        "cell_type": "code", "execution_count": 2, "metadata": {}, "outputs": [
            {
                "name": "stderr", "output_type": "stream", "text": [
                    "u001b[32mINFO u001b[0m u001b[34mIn SushiChef.run
                    method. args={'command': 'dryrun', 'reset': True, 'ver-
                    bose': True, 'token': 'YOURTO…'} options={}u001b[0mn",
                    "u001b[32mINFO u001b[0m u001b[34mn", "n", "*
                    Starting channel build process *n", "n", "u001b[0mn",
```

```
                "u001b[32mINFO u001b[0m u001b[34mCalling construct_channel… u001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Setting up initial channel structure… u001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Validating channel structure…u001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Sample Exercises (ChannelNode): 3 descendantsu001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Math Exercises (TopicNode): 2 descendantsu001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Basic questions (ExerciseNode): 3 questionsu001b[0mn", "u001b[32mINFO u001b[0m u001b[34m An exercise containing a perseus question (ExerciseNode): 1 questionu001b[0mn", "u001b[32mINFO u001b[0m u001b[34m Tree is validn", "u001b[0mn", "u001b[32mINFO u001b[0m u001b[34mDownloading files…u001b[0mn", "u001b[32mINFO u001b[0m u001b[34mProcessing content…u001b[0mn", "u001b[32mINFO u001b[0m u001b[34mt*** Processing images for exercise: Basic questionsu001b[0mn", "u001b[32mINFO u001b[0m u001b[34mt*** Images for Basic questions have been processedu001b[0mn", "u001b[32mINFO u001b[0m u001b[34mt*** Processing images for exercise: An exercise containing a perseus questionu001b[0mn", "u001b[32mINFO u001b[0m u001b[34mt*** Images for An exercise containing a perseus question have been processedu001b[0mn", "u001b[32mINFO u001b[0m u001b[34m All files were successfully downloadedu001b[0mn", "u001b[32mINFO u001b[0m u001b[34mCommand is dryrun so we are not uploading chanel.u001b[0mn"
            ]
        }
    ], "source": [
        "chef = ExercisesChef()n", "args = {n", " 'command': 'dryrun', # use 'uploadchannel' for real runn", " 'verbose': True,n", " 'token': 'YOURTOKENHERE9139139f3a23232'n", "}n", "options = {}n", "n", "chef.run(args, options)"
    ]
}, {
    "cell_type": "markdown", "metadata": {}, "source": [
```

```
                "Congratulations, you put some math exercises on the internet!"
            ]
        }, {
            "cell_type": "markdown", "metadata": {}, "source": [
                "**Note**: you will need to change the value of *CHANNEL_SOURCE_ID* if youn", "before you try running this script with *{'command': 'upload-channel', ...}*.n", "The combination of source domain and source id are used to compute the *channel_id*n", "for the Kolibri channel you're creating. If you keep the lines above unchanged,n", "you'll get an error because you don't have edit rights on that channel."
            ]
        }, {
            "cell_type": "code", "execution_count": null, "metadata": {}, "outputs": [], "source": []
        }
    ], "metadata": {
        "kernelspec": {
            "display_name": "Python 3", "language": "python", "name": "python3"
        }, "language_info": {
            "codemirror_mode": {
                "name": "ipython", "version": 3
            }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.7.4"
        }
    }, "nbformat": 4, "nbformat_minor": 2
}

{
    "cells": [
        {
            "cell_type": "markdown", "metadata": {}, "source": [
                "# Document conversionsn", "n", "In this tutorial we'll look at some examples of converting various document formats to PDF.n", "This step may be needed in cases when you want to addn", "word processor documents (*.doc*, *.docx*, *.odt*),n", "spreadheeets (*.xls*, *.xlsx*, *.ods*),n", "or simple presentation (*.ppt*, *.pptx*, *.odp*).n", "Learning Equality operates a web service called *microwave* that you can usen", "to convert all kinds of documents into PDF format, which can be viewed in Kolibri.n", "n", "Before we begin, click [this link](http://35.185.105.222:8989/healthz) to checkn", "that the *microwave*
```

conversion service is running. You should see a message about the uptime.n", "If you see an error get in touch with the Learning Equality content teamn", "so we can fix it."
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"## Setup"
]
}, {
"cell_type": "code", "execution_count": 1, "metadata": {}, "outputs": [], "source": [
"import requestsn", "n", "# This is the main URL for the microwave servicen", "microwave_url = "http://35.185.105.222:8989/unoconv/pdf"n""
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"## Convert a .docx file to .pdf"
]
}, {
"cell_type": "code", "execution_count": 2, "metadata": {}, "outputs": [], "source": [
"# helpern", "def save_response_content(response, filename):n", " with open(filename, 'wb') as localfile:n", " localfile.write(response.content)n", "n", "# Download a sample .docx filen", "docx_url = 'https://calibre-ebook.com/downloads/demos/demo.docx'n", "response1 = requests.get(docx_url)n", "save_response_content(response1, 'document.docx')n", "n", "# Convert itn", "microwave_url = 'http://35.185.105.222:8989/unoconv/pdf'n", "files = {'file': open('sample.docx', 'rb')}n", "response = requests.post(microwave_url, files=files)n", "save_response_content(response, 'document.pdf')"
]
}, {
"cell_type": "markdown", "metadata": {}, "source": [
"You should now be able to see the file *document.pdf* in the current directory, and add this PDF file to a Kolibri channel using the *DocumentFile* class as part of a *DocumentNode*."
]
}, {

```
    “cell_type”: “code”, “execution_count”: null, “metadata”: {}, “outputs”: [],
    “source”: []
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “## Convert a .png file to .pdf”
    ]
}, {
    “cell_type”: “code”, “execution_count”: 3, “metadata”: {}, “outputs”: [], “source”:
    [
        “# helpern”, “def save_response_content(response, file-
        name):n”, “ with open(filename, ‘wb’) as localfile:n”, “ lo-
        calfile.write(response.content)n”, “n”, “# Let’s GET the
        postern”, “png_url = ‘https://www.who.int/images/default-
        source/health-topics/coronavirus/risk-communications/general-
        public/protect-yourself/infographics/masks-infographic—final.tmb-
        1920v.png’n”, “response1 = requests.get(png_url)n”,
        “save_response_content(response1, ‘infographic.png’)n”, “n”,
        “# Convert itn”, “files = {‘file’: open(‘infographic.png’, ‘rb’)}n”,
        “response = requests.post(microwave_url, files=files)n”,
        “save_response_content(response, ‘infographic.pdf’)”
    ]
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “You should now be able to see the file infographic.pdf in the current
        directory, and add this PDF file to a Kolibri channel using the Docu-
        mentFile class as part of a DocumentNode.”
    ]
}, {
    “cell_type”: “code”, “execution_count”: null, “metadata”: {}, “outputs”: [],
    “source”: []
}, {
    “cell_type”: “markdown”, “metadata”: {}, “source”: [
        “The procedure works the same for spreadsheets and presentations.
        Just send the file to the microwave service using a POST request and
        you’ll get a PDF version back.”
    ]
}, {
    “cell_type”: “code”, “execution_count”: null, “metadata”: {}, “outputs”: [],
    “source”: []
}
```

], “metadata”: {
“kernelspec”: {
“display_name”: “Python 3”, “language”: “python”, “name”: “python3”
}, “language_info”: {
“codemirror_mode”: {
“name”: “ipython”, “version”: 3
}, “file_extension”: “.py”, “mimetype”: “text/x-python”, “name”: “python”, “nbconvert_exporter”: “python”, “pygments_lexer”: “ipython3”, “version”: “3.7.4”
}
}, “nbformat”: 4, “nbformat_minor”: 2
}

4.1 Jupyter notebooks

Jypyter notebooks are a very powerful tool for interactive programming. You type in commands into an online shell, and you immediately see the results.

To install jupyter notebook on your machine, you run:

```
pip install jupyter
```

then to start the jupyter notebook server, run

```
jupyter notebook
```

If you then navigate to the directory *docs/examples/* in the ricecooker source code repo, you'll find the same examples described above in the form of runnable notebooks that will allow you to experiment and learn hands-on.

You'll need to press CTRL+C in the terminal to stop the jupyter notebook server, or use the Shutdown button in the web interface.

Watch the beginning of this Video tutorial to learn how to use the Jypyter notebook environment for interactively coding parts of the chef logic.

4.2 Advanced examples

The links below will take you to the GitHub repositories of content integration scripts we use to create some of the most popular Kolibri channels in the library:

- Khan Academy chef
- Open Stax chef
- SHLS Toolkit chef

You can get a list of ALL the content integration scripts by searching for sushi-chef on GitHub.

Chapter 5

Ricecooker API reference

The detailed information for content developers (chef authors) is presented here:

5.1 Nodes

Kolibri channels are tree-like structures that consist of different types of topic nodes (folders) and various content nodes (document, audio, video, html, exercise). The module `ricecooker.classes.nodes` defines helper classes to represent each of these supported content types and provide validation logic to check channel content is valid before uploading it to Kolibri Studio.

The purpose of the Node classes is to represent the channel tree structure and store metadata necessary for each type of content item, while the actual content data is stored in file objects (defined in `ricecooker.classes.files`) and exercise questions object (defined in `ricecooker.classes.questions`) which are created separately.

5.1.1 Overview

The following diagram lists all the node classes defined in `ricecooker.classes.nodes` and shows the associated file and question classes that content nodes can contain.

```
      ricecooker.classes.nodes
      |
      |                                          ricecooker.classes.files
class Node(object)                               |
    class ChannelNode(Node)                      |
    class TreeNode(Node)                         |
```

(continues on next page)

(continued from previous page)

```
        class TopicNode(TreeNode)                       |
        class ContentNode(TreeNode)                     |
            class AudioNode(ContentNode)     files = [AudioFile]
            class DocumentNode(ContentNode)  files = [DocumentFile, EPubFile]
            class HTML5AppNode(ContentNode)  files = [HTMLZipFile]
            class H5PAppNode(ContentNode)    files = [H5PFile]
            class SlideshowNode(ContentNode) files = [SlideImageFile]
            class VideoNode(ContentNode)     files = [VideoFile, WebVideoFile,
YouTubeVideoFile,
                                                      SubtitleFile,
YouTubeSubtitleFile]
            class ExerciseNode(ContentNode)  questions = [SingleSelectQuestion,
                                                          MultipleSelectQuestion,
                                                          InputQuestion,
                                                          PerseusQuestion]
                                                          |
                                                          |
                                                          ricecooker.classes.
questions
```

In the remainder of this document we'll describe in full detail the metadata that is needed to specify different content nodes.

For more info about file objects see page *files* and to learn about the different exercise questions see the page *exercises*.

5.1.2 Content node metadata

Each node has the following attributes:

- **source_id** (str): content's original id
- **title** (str): content's title
- **license** (str or License): content's license id or object
- **language** (str or lang_obj): language for the content node
- **description** (str): description of content (optional)
- **author** (str): who created the content (optional)
- **aggregator** (str): website or org hosting the content collection but not necessarily the creator or copyright holder (optional)
- **provider** (str): organization that commissioned or is distributing the content (optional)

- **role** (str): set to `roles.COACH` for teacher-facing materials (default `roles.LEARNER`)
- **thumbnail** (str or ThumbnailFile): path to thumbnail or file object (optional)
- **derive_thumbnail** (bool): set to True to generate thumbnail from contents (optional)
- **files** ([FileObject]): list of file objects for node (optional)
- **extra_fields** (dict): any additional data needed for node (optional)
- **domain_ns** (uuid): who is providing the content (e.g. learningequality.org) (optional)

IMPORTANT: nodes representing distinct pieces of content MUST have distinct `source_ids`. Each node has a `content_id` (computed as a function of the `source_domain` and the node's `source_id`) that uniquely identifies a piece of content within Kolibri for progress tracking purposes. For example, if the same video occurs in multiple places in the tree, you would use the same `source_id` for those nodes – but content nodes that aren't for that video need to have different `source_ids`.

5.1.2.1 Usability guidelines

- Thumbnails: 16:9 aspect ratio ideally (e.g. 400x225 pixels)
- Titles: Aim for titles that make content items reusable independently of their containing folder, since curators could copy content items to other topics or channels. e.g. title for pdf doc "{lesson_name} - instructions.pdf" is better than just "Instructions.pdf" since that PDF could show up somewhere else.
- Descriptions: aim for about 400 characters (about 3-4 sentences)
- Licenses: Any non-public domain license must have a copyright holder, and any special permissions licenses must have a license description.

5.1.2.2 Licenses

All content nodes within Kolibri and Kolibri Studio must have a license. The file le_utils/constants/licenses.py contains the constants used to identify the license types. These constants are meant to be used in conjunction with the helper method `ricecooker.classes.licenses.get_license` to create `Licence` objects.

To initialize a license object, you must specify the license type and the `copyright_holder` (str) which identifies a person or an organization. For example:

```
from ricecooker.classes.licenses import get_license
from le_utils.constants import licenses

license_obj = get_license(licenses.CC_BY, copyright_holder="Khan Academy")
```

Note: The `copyright_holder` field is required for all License types except for the public domain license for which `copyright_holder` can be None. Everyone owns the stuff in the public domain.

5.1.2.3 Languages

The Python package `le-utils` defines the internal language codes used throughout the Kolibri platform (e.g. `en`, `es-MX`, and `zul`). To find the internal language code for a given language, you can locate it in the lookup table, or use one of the language lookup helper functions defined in `le_utils.constants.languages`:

- `getlang(<code>) --> lang_obj`: basic lookup used to ensure `<code>` is a valid internal language code (otherwise returns `None`).
- `getlang_by_name(<Language name in English>) --> lang_obj`: lookup by name, e.g. `French`
- `getlang_by_native_name(<Language autonym>) --> lang_obj`: lookup by native name, e.g., `français`
- `getlang_by_alpha2(<two-letter ISO 639-1 code>) --> lang_obj`: lookup by standard two-letter code, e.g `fr`

You can either pass `lang_obj` as the `language` attribute when creating nodes, or pass the internal language code (str) obtained from the property `lang_obj.code`:

```
from le_utils.constants.languages import getlang_by_native_name

lang_obj = getlang_by_native_name('français')
print(lang_obj        # Language(native_name='Français', primary_code='fr',
subcode=None, name='French')
print(lang_obj.code)  # fr
```

See [languages][./languages.md] to read more about language codes.

5.1.2.4 Thumbnails

Thumbnails can be passed in as a local filesystem path to an image file (str), a URL (str), or a `ThumbnailFile` object. The recommended size for thumbnail images is 400px by 225px (aspect ratio 16:9). Use the command line argument `--thumbnails` to automatically generate thumbnails for all content node that don't have a thumbnail specified.

5.1.3 Topic nodes

Topic nodes are folder-like containers that are used to organize the channel's content.

```
from ricecooker.classes import TopicNode
from le_utils.constants.languages import getlang

topic_node = TopicNode(
    title='The folder name',
    description='A longer description of what the folder contains',
    source_id='<some unique identifier for this folder>',
    language='en',
    thumbnail=None,
    author='',
)
```

It is highly recommended to find suitable thumbnail images for topic nodes. The presence of thumbnails will make the content more appealing and easier to browse. Set `derive_thumbnail=True` on a topic node or use the `--thumbnails` command line argument and Ricecooker will generate thumbnails for topic nodes based on the thumbnails of the content nodes they contain.

5.1.4 Content nodes

The table summarizes summarizes the content node classes, their associated files, and the file formats supported by each file class:

```
  ricecooker.classes.nodes  ricecooker.classes.files
  |                         |
  AudioNode      --files-->  AudioFile                                  # .mp3
  DocumentNode   --files-->  DocumentFile                               # .pdf
                             EPubFile                                   # .epub
  SlideshowNode --files-->  SlideImageFile                              # .png/.
jpg
  HTML5AppNode   --files-->  HTMLZipFile                                # .zip
  VideoNode      --files-->  VideoFile, WebVideoFile, YouTubeVideoFile,  # .mp4
                             SubtitleFile, YouTubeSubtitleFile          # .vtt
```

For your copy-paste convenience, here is the sample code for creating a content node (`DocumentNode`) and an associated (`DocumentFile`)

```
content_node = DocumentNode(
      source_id='<some unique identifier within source domain>',
      title='Some Document',
      author='First Last (author\'s name)',
      description='Put node description here',
      language=getlang('en').code,
      license=get_license(licenses.CC_BY, copyright_holder='Copyright holder name
'),
      thumbnail='some/local/path/name_thumb.jpg',
      files=[DocumentFile(
                path='some/local/path/name.pdf',
                language=getlang('en').code
             )]
)
```

Files can be passed in upon initialization as in the above sample, or can be added after initialization using the content_node's `add_files` method.

Note you also use URLs for `path` and `thumbnail` instead of local filesystem paths, and the files will be downloaded for you automatically.

You can replace `DocumentNode` and `DocumentFile` with any of the other combinations of content node and file types.

Specify `derive_thumbnail=True` and leave thumbnail blank (`thumbnail=None`) to let Ricecooker automatically generate a thumbnail for the node based on its content. Thumbnail generation is supported for audio, video, PDF, and ePub, and HTML5 files.

5.1.4.1 Role-based visibility

It is possible to include content nodes in any channel that are only visible to Kolibri coaches. Setting the visibility to "coach-only" is useful for pedagogical guides, answer keys, lesson plan suggestions, and other supporting material intended only for teachers to see but not students. To control content visibility set the `role` attributes to one of the constants defined in `le_utils.constants.roles` to define the "minimum role" needed to see the content.

- if `role=roles.LEARNER`: visible to learners, coaches, and administrators
- if `role=roles.COACH`: visible only to Kolibri coaches and administrators

5.1.5 Exercise nodes

The `ExerciseNode` class (also subclasses of `ContentNode`), act as containers for various assessment questions types defined in `ricecooker.classes.questions`. The question types currently supported are:

- **SingleSelectQuestion**: questions that only have one right answer (e.g. radio button questions)
- **MultipleSelectQuestion**: questions that have multiple correct answers (e.g. check all that apply)
- **InputQuestion**: questions that have as answers simple text or numeric expressions (e.g. fill in the blank)
- **PerseusQuestion**: perseus json question (used in Khan Academy chef)

The following code snippet creates an exercise node that contains the three simple question types:

```
exercise_node = ExerciseNode(
        source_id='<some unique id>',
        title='Basic questions',
        author='LE content team',
        description='Showcase of the simple question type supported by Ricecooker
and Studio',
        language=getlang('en').code,
        license=get_license(licenses.PUBLIC_DOMAIN),
        thumbnail=None,
        exercise_data={
            'mastery_model': exercises.M_OF_N,  # \
            'm': 2,                             #   learners must get 2/3 questions
correct to complete exercise
            'n': 3,                             # /
            'randomize': True,                  # show questions in random order
        },
        questions=[
            MultipleSelectQuestion(
                id='sampleEX_Q1',
                question = "Which numbers the following numbers are even?",
                correct_answers = ["2", "4",],
                all_answers = ["1", "2", "3", "4", "5"],
                hints=['Even numbers are divisible by 2.'],
            ),
            SingleSelectQuestion(
                id='sampleEX_Q2',
```

(continues on next page)

(continued from previous page)

```
                question = "What is 2 times 3?",
                correct_answer = "6",
                all_answers = ["2", "3", "5", "6"],
                hints=['Multiplication of $a$ by $b$ is like computing the area of
a rectangle with length $a$ and width $b$.'],
            ),
            InputQuestion(
                id='sampleEX_Q3',
                question = "Name one of the *factors* of 10.",
                answers = ["1", "2", "5", "10"],
                hints=['The factors of a number are the divisors of the number that
leave a whole remainder.'],
            )
        ]
)
```

Creating a `PerseusQuestion` requires first obtaining the perseus-format `.json` file for the question. You can questions using the web interface. Click here to see a samples of questions in the perseus json format.

To following code creates an exercise node with a single perseus question in it:

```
# LOAD JSON DATA (as string) FOR PERSEUS QUESTIONS
RAW_PERSEUS_JSON_STR = open('ricecooker/examples/data/perseus_graph_question.json',
'r').read()
# or
# import requests
# RAW_PERSEUS_JSON_STR = requests.get('https://github.com/learningequality/sample-
channels/blob/master/contentnodes/exercise/perseus_graph_question.json').text
exercise_node2 = ExerciseNode(
        source_id='<another unique id>',
        title='An exercise containing a perseus question',
        author='LE content team',
        description='An example exercise with a Persus question',
        language=getlang('en').code,
        license=get_license(licenses.CC_BY, copyright_holder='Copyright holder name
'),
        thumbnail=None,
        exercise_data={
```

(continues on next page)

(continued from previous page)

```
            'mastery_model': exercises.M_OF_N,
            'm': 1,
            'n': 1,
        },
        questions=[
            PerseusQuestion(
                id='ex2bQ4',
                raw_data=RAW_PERSEUS_JSON_STR,
                source_url='https://github.com/learningequality/sample-channels/
blob/master/contentnodes/exercise/perseus_graph_question.json'
            ),
        ]
)
```

The example above uses the JSON from this question, for which you can also a rendered preview here.

5.1.6 SlideshowNode nodes

The `SlideshowNode` class and the associated `SlideImageFile` class are used to create powerpoint-like presentations. The following code sample shows how to create a `SlideshowNode` that contains two slide images:

```
slideshow_node = SlideshowNode(
      source_id='<some unique identifier within source domain>',
      title='My presentations',
      author='First Last (author\'s name)',
      description='Put slideshow description here',
      language=getlang('en').code,
      license=get_license(licenses.CC_BY, copyright_holder='Copyright holder name
'),
      thumbnail='some/local/path/slideshow_thumbnail.jpg',
      files=[
          SlideImageFile(
              path='some/local/path/firstslide.png',
              caption="The caption text to be displayed below the slide image.",
              descriptive_text="Description of the slide for users that cannot see
the image",
```

(continues on next page)

(continued from previous page)

```
            language=getlang('en').code,
        ),
        SlideImageFile(
            path='some/local/path/secondslide.jpg',
            caption="The caption for the second slide image.",
            descriptive_text="Alternative text for the second slide image",
            language=getlang('en').code,
        )
    ]
)
```

Note this is a new feature in Kolibri 0.13 and prior version of Kolibri will not be able to import and view this content kind.

5.2 Files

Each `ricecooker` content node is associated with one or more files stored in a content-addressable file storage system. For example, to store the file `sample.pdf` we first compute `md5` hash of its contents (say `abcdef00000000000000000000000000`) then store the file at the path `storage/a/b/abcdef00000000000000000000000000.pdf`. The same storage mechanism is used on Kolibri Studio and Kolibri applications.

5.2.1 File objects

The following file classes are defined in the module `ricecooker.classes.files`:

```
AudioFile             # .mp3
DocumentFile          # .pdf
EPubFile              # .epub
HTMLZipFile           # .zip containing HTML,JS,CSS
H5PFile               # .h5p
VideoFile             # .mp4 (`path` is local file system or url)
  WebVideoFile        # .mp4 (downloaded from `web_url`)
  YouTubeVideoFile    # .mp4 (downloaded from youtube based on `youtube_id`)
  SubtitleFile        # .vtt (`path` is local file system or url)
  YouTubeSubtitleFile # .vtt (downloaded from youtube based on `youtube_id` and
`language`)
```

(continues on next page)

(continued from previous page)

```
SlideImageFile        # .png/.jpg an image that is part of a SlideshowNode
ThumbnailFile         # .png/.jpg/.jpeg  (`path` is local file system or url)
```

5.2.2 Base classes

The file classes extent the base classes `File(object)` and `DownloadFile(File)`. When creating a file object, you must specify the following attributes:

- `path` (str): this can be either local path like `dir/subdir/file.ext`, or a URL like 'http://site.org/dir/file.ext'.
- `language` (str or `le_utils` language object): what is the language is the file contents.

5.2.2.1 Path

The `path` attribute can be either a path on the local filesystem relative to the current working directory of the chef script, or the URL of a web resource.

5.2.2.2 Language

The Python package `le-utils` defines the internal language codes used throughout the Kolibri platform (e.g. `en`, `es-MX`, and `zul`). To find the internal language code for a given language, you can locate it in the lookup table, or use one of the language lookup helper functions defined in `le_utils.constants.languages`:

- `getlang(<code>) --> lang_obj`: basic lookup used to ensure `<code>` is a valid internal language code (otherwise returns `None`).
- `getlang_by_name(<Language name in English>) --> lang_obj`: lookup by name, e.g. `French`
- `getlang_by_native_name(<Language autonym>) --> lang_obj`: lookup by native name, e.g., `français`
- `getlang_by_alpha2(<two-letter ISO 639-1 code>) --> lang_obj`: lookup by standard two-letter code, e.g `fr`

You can either pass `lang_obj` as the `language` attribute when creating nodes and files, or pass the internal language code (str) obtained from the property `lang_obj.code`. See [languages][./languages.md] to read more about language codes.

5.2.3 Audio files

Use the `AudioFile(DownloadFile)` class to store `mp3` files.

```
audio_file = AudioFile(
    path='dir/subdir/lecture_recording.mp3',
    language=getlang('en').code
)
```

5.2.4 Document files

Use the `DocumentFile` class to add PDF documents:

```
document_file = DocumentFile(
    path='dir/subdir/lecture_slides.pdf',
    language=getlang('en').code
)
```

Use the `EPubFile` class to add ePub documents:

```
document_file = EPubFile(
    path='dir/subdir/lecture_slides.epub',
    language=getlang('en').code
)
```

5.2.5 HTML files

The `HTML5ZipFile` class is a generic zip container for web content like HTML, CSS, and JavaScript. To be a valid `HTML5ZipFile` file, the file must have a `index.html` in its root. The file `index.html` will be loaded within a sandboxed iframe when this content item is accessed on Kolibri.

Chef authors are responsible for scraping the HTML and all the related JS, CSS, and images required to render the web content, and creating the zip file. Creating a `HTML5ZipFile` is then done using

```
document_file = HTML5ZipFile(
    path='/tmp/interactive_js_simulation.zip',
    language=getlang('en').code
)
```

Use the `H5PFile` class to add H5P files:

```
h5p_file = H5PFile(
    path='dir/subdir/presentation.h5p',
    language=getlang('en').code
)
```

5.2.6 Videos files

The following file classes can be added to the `VideoNodes`:

```
class VideoFile(DownloadFile)
class WebVideoFile(File)
class YouTubeVideoFile(WebVideoFile)
class SubtitleFile(DownloadFile)
class YouTubeSubtitleFile(File)
```

To create `VideoFile`, you need the code

```
video_file = VideoFile(
    path='dir/subdir/lecture_video_recording.mp4',
    language=getlang('en').code
)
```

VideoFiles can also be initialized with **ffmpeg_settings** (dict), which will be used to determine compression settings for the video file.

```
video_file = VideoFile(
    path = "file:///path/to/file.mp4",
    ffmpeg_settings = {"max_height": 480, "crf": 28},
    language=getlang('en').code
)
```

WebVideoFiles must be given a **web_url** (str) to a video on YouTube or Vimeo, and YouTubeVideoFiles must be given a **youtube_id** (str).

```
video_file2 = WebVideoFile(
    web_url = "https://vimeo.com/video-id",
    language=getlang('en').code,
)
```

(continues on next page)

(continued from previous page)

```
video_file3 = YouTubeVideoFile(
    youtube_id = "abcdef",
    language=getlang('en').code,
)
```

WebVideoFiles and YouTubeVideoFiles can also take in **download_settings** (dict) to determine how the video will be downloaded and **high_resolution** (boolean) to determine what resolution to download.

Subtitle files can be created using

```
subs_file = SubtitleFile(
    path = "file:///path/to/file.vtt",
    language = languages.getlang('en').code,
)
```

Kolibri uses the `.vtt` subtitle format internally, but the following formats can be automatically converted: `.srt`, `.ttml`, `.scc`, `.dfxp`, and `.sami`. The subtitle format is inferred from the file extension of the `path` argument. Use the `subtitlesformat` keyword argument in cases where the path does not end on a format extension:

```
subs_file = SubtitleFile(
    path = "http:/srtsubs.org/subs/29323923",
    subtitlesformat = 'srt',                    # specify format because not in URL
    language = languages.getlang('en').code,
)
```

You can also get subtitles using `YouTubeSubtitleFile` which takes a `youtube_id` and youtube `language` code (may be different from internal language codes). Use the helper method `is_youtube_subtitle_file_supported_language` to test if a given youtube language code is supported by `YouTubeSubtitleFile` and skip the ones that are not currently supported. Please let the LE content team know when you run into language codes that are not supported so we can add them.

5.2.7 Thumbnail files

The class `ThumbnailFile` defined thumbnails that can be added to channel, topic nodes, and content nodes. The extensions `.png`, `.jpg`, and `.jpeg` and supported.

The recommended size for thumbnail images is 400px by 225px (aspect ratio 16:9).

5.2.8 SlideImageFile files

The `SlideImageFile` class is used in conjunction with the `SlideshowNode` class to create powerpoint-like slideshow presentations.

```
slide_image_file = SlideImageFile(
    path='some/local/path/firstslide.png',
    caption="The caption text to be displayed below the slide image",
    descriptive_text="Description of the slide for users that cannot see the image
",
    language=getlang('en').code,
)
```

Use the `caption` field to provide the text that will be displayed under the slide image as part of the presentation. Use the `descriptive_text` field to provide the "alt text" the image contents for visually impaired users.

5.2.9 File size limits

Kolibri Studio does not impose any max-size limits for files uploaded, but chef authors need to keep in mind that content channels will often be downloaded over slow internet connections and viewed on devices with limited storage.

Below are some general guidelines for handling video files:

- Short videos (5-10 mins long) should be roughly less than 15MB
- Longer video lectures (1 hour long) should not be larger than 200MB
- High-resolution videos should be converted to lower resolution formats: Here are some recommended choices for video vertical resolution:
  - Use max height of `480` for videos that work well in low resolution (most videos)
  - Use max height of `720` for high resolution videos (lectures with writing on board)
- Ricecooker can handle the video compression for you if you specify the `--compress` command line argument, or by setting the `ffmpeg_settings` property when creating `VideoFiles`. The default values for `ffmpeg_settings` are as follows:

```
ffmpeg_settings = {'crf':32, 'max_height':480 }
```

- The `ffmpeg` setting `crf` stands for Constant Rate Factor and is very useful for controlling overall video quality. Setting `crf=24` produces high quality video (and possibly large file size), `crf=28` is a mid-range quality, and values of `crf` above 30 produce highly-compressed videos with small size.

PDF files are usually not large, but PDFs with many pages (more than 50 pages) can be difficult to views and browse on devices with small screens, so we recommend that long PDF documents be split into separate parts.

Note: Kolibri Studio imposes a file storage quota on a per-user basis. By default the storage limit for new accounts is 500MB. Please get in touch with the content team by email (`content@le...`) if you need a quota increase.

5.3 HTML5 Apps

Kolibri supports web content through the use of `HTML5AppNode`, which renders the contents of a `HTMLZipFile` in a sandboxed iframe. The Kolibri HTML5 viewer will load the `index.html` file which is assumed to be in the root of the zip file. All `hrefs` and other `src` attributes must be relative links to resources within the zip file. The `iframe` rendering of the content in Kolibri is sandbox so there are some limitations about use of plugins and parts of the web API.

5.3.1 Technical specifications

- An `HTMLZipFile` *must* have an `index.html` file at the root of the zip file.
- A web application packaged as a `HTMLZipFile` *must* not depend on network calls for it to work (cannot load resources references via http/https links)
- A web application packaged as a `HTMLZipFile` *should* not make unnecessary network calls (analytics scripts, social sharing functionality, tracking pixels). In an offline setting none of these functions would work so it is considered best practices to “clean up” the web apps as part of packaging for offline use.
- The web application *must* not use plugins like swf/flash.

5.3.2 HTML5AppNode examples

- A raw HTML example that consists of basic unstyled HTML content taken from the "Additional Online Resources" section of this source page. Note links are disabled (removed blue link, and replaced by display of target URL. If the links were to useful resources (documents, worksheets, sound clips), they could be included in the zip file (deep scraping) with link changed to a relative path. By modern cheffing standards, this HTML node would be flagged as "deficient" since it lacks basic styled and readability. See the recommended approach to basic HTML styling in the next example.
- A basic styled HTML example. The code uses a basic template which was copy-pasted from html-app-starter. This presentation applies basic fonts, margins, and layout to make HTML content more readable. See the section "Usability guidelines" below for more details.
- An example of an interactive app. Complete javascript interactive application packaged as a zip file. Source: sushi-chef-phet.
- A flipbook reader application that is built by this code.
- A section from a math textbook that includes text, images, and scripts for rendering math equations.
- A interactive training activity. The code for packaging this HTML app ensures all js, css, and media assets are included in the .zip file.
- Proof of concept of a Vue.js App. This is a minimal webapp example based on the vue.js framework. Note the shell script used to tweak the links inside index.html and build.js to make references relative paths.
- Proof of concept React App: A minimal webapp example based on the React framework. Note the shell script tweaks required to make paths relative.
- A complete task-oriented coding environment which is obtained by taking the source page content and packaging it for offline use.
- A powerpoint sideshow presentation packaged as a standalone zip with PREV/NEXT buttons.

5.3.3 Extracting Web Content

Most content integration scripts for web content require some combination of *crawling* (visiting web pages on the source website to extract the structure), and *scraping* (extracting the metadata and files from detail pages).

The two standard tools for these tasks in the Python community are the `requests` library for making HTTP requests, and the BeautifulSoup library.

The page *Parsing HTML* contains some basic info and code examples that will allow you to get started with crawling and scraping. You can also watch this cheffing video tutorial that will show the basic steps of using `requests` and `BeautifulSoup` for crawling a website. See the sushi-chef-shls code repo for the final version of the web crawling code that was used for this content source.

5.3.3.1 Static assets download utility

We have a handy function for fetching all of a webpage's static assets (JS, CSS, images, etc.), so that, in theory, you could scrape a webpage and display it in Kolibri exactly as you see it in the website itself in your browser.

See the source in `utils/downloader.py`, example usage in a simple app: MEET chef, which comprises articles with text and images, and another example in a complex app: Blockly Games chef, an interactive JS game with images and sounds.

5.3.4 Usability guidelines

- Text should be legible (high contrast, reasonable font size)
- Responsive: text should reflow to fit screens of different sizes. You can preview on a mobile device (or use Chrome's mobile emulation mode) and ensure that the text fits in the viewport and doesn't require horizontal scrolling (a maximum width is OK but minimum widths can cause trouble).
- Ensure navigation within HTML5App is easy to use:
  - consistent use of navigation links (e.g. side menu with sections)
  - consistent use of previous/next links
- Ensure links to external websites are disabled (remove `<a></a>` tag), and instead show the `href` in brackets next to the link text (so that users could potentially access the URL by some other means). For example "some other text **link text**(http://link.url) and more text continues"

5.3.4.1 Links and navigation

It's currently not possible to have navigation links between different HTML5App nodes, but relative links within the same zip file work (since they are rendered in same iframe).

5.3.4.2 Packaging considerations

It's important to "cut" the source websites content into appropriately sized chunks:

- As small as possible so that resources are individually trackable, assignable, remixable, and reusable accross channels and in lessons.
- But not too small, e.g., if a lesson contains three parts intended to be followed one after the other, then all three parts should be included in a same HTML5App with internal links.
- Use nested folder structure to represent complex sources. Whenever an HTML page that acts as a "container" with links to other pages and PDFs, turn it into a TopicNode (Folder) and put content items inside it.

5.3.4.3 Starter template

We also have a starter template for apps, particularly helpful for displaying content that's mostly text and images, such as articles. It applies some default styling on text to ensure readability, consistency, and mobile responsiveness.

It also includes a sidebar for those apps where you may want internal navigation. However, consider if it would be more appropriate to turn each page into its own content item and grouping them together into a single folder (topic).

How to decide between the static assets downloader (above) and this starter template? Prefer the static assets downloader if it makes sense to keep the source styling or JS, such as in the case of an interactive app (e.g. Blockly Games) or an app-like reader (e.g. African Storybook). If the source is mostly a text blob or an article – and particularly if the source styling is not readable or appealing—using the template could make sense, especially given that the template is designed for readability.

The bottom line is ensure the content meets the usability guidelines above: legible, responsive, easy to navigate, and "look good" (you define "good" :P). Fulfilling that, use your judgment on whatever approach makes sense and that you can use effectively!

5.3.5 Using Local Kolibri Preview

The *kolibripreview.py* script can be used to test the contents of `webroot/` in a local installation of Kolibri without needing to go through the whole content pipeline.

5.3.6 Creating a HTMLZipFile

No special technique is required to create HTMLZipFile files—as long as the .zip file contain the index.html in it's root (not in a subfolder), it can be used as a `HTMLZipFile` and added as a file to an `HTML5AppNode`.

Since creating the zip files is such a common task of the cheffing process, we provide two helpers to save you time: the `create_predictable_zip` method and the `HTMLWriter` class.

5.3.6.1 Zipping a folder

The function `create_predictable_zip` can be used to create a zip file from a given directory. This is the recommended approach for creating zip files since it strips out file timestamps to ensure that the content hash will not change every time the chef script runs.

Here is some sample code that show how to use this function:

```
# 1. Create a temporary directory
webroot = tempfile.mkdtemp()

# 2. Create the index.html file inside the temporary directory
indexhtmlpath = os.path.join(webroot, 'index.html')
with open(indexhtmlpath, 'w') as indexfile:
    indexfile.write("<html><head></head><body>Hello, World!</body></html>")
# add images the webroot dir
# add css files the webroot dir
# add js files to the webroot dir
# ...

# 3. Zip it!                        (see https://youtu.be/BODSCrj9FHQ for a laugh)
zippath = create_predictable_zip(webroot)
```

You can then use this zippath as follows `zipfile = HTMLZipFile(path=zippath, ...)` and add the `zipfile` to a `HTML5AppNode` object using its `add_file` method. See here for a full code sample.

5.3.6.2 The `HTMLWriter` utility class

The class `HTMLWriter` in `ricecooker.utils.html_writer` provides a basic helper methods for creating zip files directly in compressed form, without the need for creating a temporary directory first.

To use the `HTMLWriter` class, you must enter the `HTMLWriter` context:

```
from ricecooker.utils.html_writer import HTMLWriter
with HTMLWriter('./myzipfile.zip') as zipper:
    # Add your code here
```

To write the main file (`index.html` in the root of the zip file), use the `write_index_contents` method:

```
contents = "<html><head></head><body>Hello, World!</body></html>"
zipper.write_index_contents(contents)
```

You can also add other files (images, stylesheets, etc.) using `write_file`, `write_contents`, and `write_url` methods:

```
# Returns path to file "styles/style.css"
css_path = zipper.write_contents("style.css", "body{padding:30px}", directory=
"styles")
extra_head = "<link href='{}' rel='stylesheet'></link>".format(css_path)          #
Can be inserted into <head>

img_path = zipper.write_file("path/to/img.png")                                   #
Note: file must be local
img_tag = "<img src='{}'>...".format(img_path)                                    #
Can be inserted as image

script_path = zipper.write_url("src.js", "http://example.com/src.js", directory=
"src")
script = "<script src='{}' type='text/javascript'></script>".format(script_path) #
Can be inserted into html
```

To check if a file exists in the zipfile, use the `contains` method:

```
# Zipfile has "index.html" file
zipper.contains('index.html')     # Returns True
zipper.contains('css/style.css')  # Returns False
```

You can then call `zipfile = HTMLZipFile(path='./myzipfile.zip', ...)` and add the `zipfile` to a `HTML5AppNode` object using its `add_file` method.

See the source code for more details: ricecooker/utils/html_writer.py.

5.3.7 Further reading

- Conceptually, we could say that `.epub` files are a subkind of the `.zip` file format, but Kolibri handles them differently, using `EPubFile`.
- The new H5P content format (experimental support) is also conceptually similar but contains much more structure and metadata about the javascript libraries that are used. See `H5PAppNode` and the `H5PFile` for more info.

5.4 Exercises

Exercises (assessment activities) are an important part of every learning experience. Kolibri exercises are graded automatically and provide immediate feedback learners. Student answers to be logged and enable progress reports for teachers and coaches. Exercises can also be used as part of lessons and quizzes.

An `ExerciseNodes` are special kind of content node contains one or more questions. In order to set the criteria for completing exercises, you must set **exercise_data** to a dict containing a `mastery_model` field based on the mastery models provided in `le_utils.constants.exercises`. If no data is provided, `ricecooker` will default to mastery at 3 of 5 correct. For example:

```
node = ExerciseNode(
    exercise_data={
        'mastery_model': exercises.M_OF_N,
        'randomize': True,
        'm': 3,
        'n': 5,
    },
    ...
)
```

To add a question to an exercise node, you must first create a question model from `ricecooker.classes.questions`. Your sushi chef is responsible for determining which question type to create. Here are the available question types:

- **SingleSelectQuestion**: questions that only have one right answer (e.g. radio button questions)
- **MultipleSelectQuestion**: questions that have multiple correct answers (e.g. check all that apply)
- **InputQuestion**: questions that have text-based answers (e.g. fill in the blank)
- **PerseusQuestion**: special question type for pre-formatted perseus questions

Each question class has the following attributes that can be set at initialization:

- **id** (str): question's unique id
- **question** (str): question body, in plaintext or Markdown format; math expressions must be in Latex format, surrounded by `$`, e.g. `$f(x) = 2^3$`.
- **correct_answer** (str) or **answers** ([str]): the answer(s) to question as plaintext or Markdown
- **all_answers** ([str]): list of choices for single select and multiple select questions as plaintext or Markdown
- **hints** (str or [str]): optional hints on how to answer question, also in plaintext or Markdown

To set the correct answer(s) for MultipleSelectQuestions, you must provide a list of all of the possi-

ble choices as well as an array of the correct answers (`all_answers [str]`) and `correct_answers [str]` respectively).

```
question = MultipleSelectQuestion(
    question = "Select all prime numbers.",
    correct_answers = ["2", "3", "5"],
    all_answers = ["1", "2", "3", "4", "5"],
    ...
)
```

To set the correct answer(s) for SingleSelectQuestions, you must provide a list of all possible choices as well as the correct answer (`all_answers [str]` and `correct_answer str` respectively).

```
question = SingleSelectQuestion(
    question = "What is 2 x 3?",
    correct_answer = "6",
    all_answers = ["2", "3", "5", "6"],
    ...
)
```

To set the correct answer(s) for InputQuestions, you must provide an array of all of the accepted answers (`answers [str]`).

```
question = InputQuestion(
    question = "Name a factor of 10.",
    answers = ["1", "2", "5", "10"],
)
```

To add images to a question's question, answers, or hints, format the image path with `'![](path/to/some/file.png)'` and `ricecooker` will parse them automatically.

Once you have created the appropriate question object, add it to an exercise object with `exercise_node.add_question(question)`.

5.4.1 Further reading

- See also the section `Exercise Nodes <nodes.html#exercise-nodes>__` on the nodes page.

5.5 Kolibri Language Codes

The file le_utils/constants/languages.py and the lookup table in le_utils/resources/languagelookup.json define the internal representation for languages codes used by Ricecooker, Kolibri, and Kolibri Studio to identify content items in different languages.

The internal representation uses a mixture of two-letter codes (e.g. `en`), two-letter-and-country code (e.g. `pt-BR` for Brazilian Portuguese), and three-letter codes (e.g., `zul` for Zulu).

In order to make sure you have the correct language code when interfacing with the Kolibri ecosystem (e.g. when uploading new content to Kolibri Studio), you must lookup the language object using the helper method `getlang`:

```
>>> from le_utils.constants.languages import getlang
>>> language_obj = getlang('en')        # lookup language using language code
>>> language_obj
Language(native_name='English', primary_code='en', subcode=None, name='English',
ka_name=None)
```

The function `getlang` will return `None` if the lookup fails. In such cases, you can try lookup by name or lookup by alpha2 code (ISO_639-1) methods defined below.

Once you've successfully looked up the language object, you can obtain the internal representation language code from the language object's `code` attribute:

```
>>> language_obj.code
'en'
```

The `ricecooker` API expects these internal representation language codes will be supplied for all `language` attributes (channel language, node language, and files language).

5.5.1 More lookup helper methods

The helper method `getlang_by_name` allows you to lookup a language by name:

```
>>> from le_utils.constants.languages import getlang_by_name
>>> language_obj = getlang_by_name('English')  # lookup language by name
>>> language_obj
Language(native_name='English', primary_code='en', subcode=None, name='English',
ka_name=None)
```

The module `le_utils.constants.languages` defines two other language lookup methods:

- Use `getlang_by_native_name` for lookup up names by native language name, e.g., you look for 'Français' to find French.
- Use `getlang_by_alpha2` to perform lookups using the standard two-letter codes defined in ISO_639-1 that are supported by the `pycountries` library.

5.6 Running chef scripts

The base class `SushiChef` provides a lot of command line arguments that control the chef script's operation. Chef scripts often have a README.md file that explains what are the recommended command line arguments and options for the chef script.

5.6.1 Ricecooker CLI

This listing shows the `ricecooker` command line interface (CLI) arguments:

```
usage: sushichef.py  [-h] [--token TOKEN] [-u] [-v] [--quiet] [--warn]
                     [--debug] [--compress] [--thumbnails]
                     [--resume]  [--step {CONSTRUCT_CHANNEL, CREATE_TREE,
                                          DOWNLOAD_FILES, GET_FILE_DIFF,
                                          START_UPLOAD, UPLOAD_CHANNEL}]
                     [--deploy] [--publish]

required arguments:
  --token TOKEN         Studio API Access Token (specify wither the token
                        value or the path of a file that contains the token).

optional arguments:
  -h, --help            show this help message and exit
  --debug               Print extra debugging infomation.
  -v, --verbose         Verbose mode (default).
  --compress            Compress videos using ffmpeg -crf=32 -b:a 32k mono.
  --thumbnails          Automatically generate thumbnails for content nodes.
  --resume              Resume chef session from a specified step.
  --step  {INIT, ...    Step to resume progress from (must be used with --resume
flag)
  --update              Force re-download of files (skip .ricecookerfilecache/
check)
  --sample SIZE         Upload a sample of SIZE nodes from the channel.
```

(continues on next page)

(continued from previous page)

```
--deploy              Immediately deploy changes to channel's main tree.
                      This operation will overwrite the previous channel
                      content. Use only during development.
--publish             Publish newly uploaded version of the channel.
```

As you can tell, there are lot of arguments to choose from, and this is not even the complete list: you'll have to run `./sushichef.py -h` to see the latest version. Below is a short guide to some of the most important and useful ones arguments.

5.6.1.1 Compression and thumbnail globals

You can specify video compression settings (see this page) and thumbnails for specific nodes and files in the channel, or use `--compress` and `--thumbnails` to apply compression to ALL videos, and automatically generate thumbnails for all the supported content kinds. **We recommend you always use the** `--thumbnails` in order to create more colorful, lively channels that learners will want to browse.

5.6.1.2 Caching

Use `--update` argument to skip checks for the `.ricecookerfilecache` directory. This is required if you suspect the files on the source website have been updated.

Note that some chef scripts implement their own caching mechanism, so you need to disable those caches as well if you want to make sure you're getting new content. Use the commands `rm -rf .webcache` to clear the webcache if it is present, and `rm -rf .ricecookerfilecache/* storage/* restore/*` to clean all ricecooker directories and start from scratch.

5.6.1.3 Extra options

In addition to the command line arguments described above, the `ricecooker` CLI supports passing additional keyword options using the format `key=value key2=value2`.

It is common for a chef script to accept a "language option" like `lang=fr` which runs the French version of the chef script. This way a single chef codebase can create multiple Kolibri Studio channels, one for each language.

These extra options will be parsed along with the `riceooker` arguments and passed as along to all the chef's methods: `pre_run`, `run`, `get_channel`, `construct_channel`, etc.

For example, a script started using `./sushichef.py ... lang=fr` could. Subclass the method `get_channel(self, **kwargs)` to choose the channel's name and description based on the value `fr` you'll receive in `kwargs['lang']`. The language code `fr` will can passed in to the `construct_channel` method, and the `pre_run` and `run` methods as well.

5.6.2 Using Python virtual env

The recommended best practice is to keep the Python packages required to run each sushichef script in a self-contained Python environment, separate from the system Python packages. This per-project software libraries isolation is easy to accomplish using the Python `virtualenv` tool and simple `requirements.txt` files.

By convention all the sushichef code examples and scripts we use in production contain a `requirements.txt` file that lists what packages must be installed for the chef to run, including `ricecooker`.

To create a virtual environment called `venv` (the standard naming convention for virtual environments) and install all the required packages, run the following:

```
cd Projects/sushi-chef-{source_name}      # cd into the chef repo
virtualenv -p python3 venv                # create a Python3 virtual environment
source venv/bin/activate                  # go into the virtualenv `venv`
pip install -r requirements.txt           # install a list of python packages
```

Windows users will need to replace the third line with `venv\Scripts\activate` as the commands are slightly different on Windows.

When the virtual environment is "activated," you'll see `(venv)` at the beginning of your command prompt, which tells you that you're in project-specific environment where you can experiment, install, uninstall, upgrade, test Python things out, without your changes interfering with the Python installation of your operating system. You can learn more about virtualenvs from the Python docs

5.6.3 Executable scripts

On UNIX systems, you can make your sushi chef script (e.g. `sushichef.py`) run as a standalone command line application. To make a script into a program, you need to do three things:

- Add the line `#!/usr/bin/env python` as the first line of `sushichef.py`
- Add this code block at the bottom of `sushichef.py` if it is not already there:

```
if __name__ == '__main__':
    chef = MySushiChef()  # replace with you chef class name
    chef.main()
```

- Make the file `sushichef.py` executable by running `chmod +x sushichef.py`

You can now call your sushi chef script using `./sushichef.py ...` or `sushichef.py ...` on Windows.

5.6.4 Long running tasks

Certain chefs that require lots of downloads and video transcoding take a long time to complete so it is best to run them on a dedicated server for this purpose.

- Connect to the remove server via `ssh`
- Clone the sushi chef git repository in the `/data` folder on the server
- Run the chef script as follows `nohup <chef cmd> &`, where `<chef cmd>` is contains the entire script name and command line options, e.g. `./sushichef.py --token=... --thumbnails lang=fr`.
- By default `nohup` logs stderr and stdout output to a file called `nohup.out` in the current working directory. Use `tail -f nohup.out` to follow this log file.

5.7 Code examples

- See the examples directory on GitHub full code examples.
- See the examples page for literate code examples that explain how to do specific tasks (find language codes, download subtitles, and exercises questions, etc). These examples are available as runnable Jupyter notebooks so you can try things out interactively and learn.
- See the Cheffing techniques doc which provides links to tips and code examples for handling various special cases and content sources.

Chapter 6

Working with content

Ricecooker includes a number of utility functions to help chef authors with common content extraction and transformation tasks.

6.1 Downloading web content

6.1.1 The ArchiveDownloader class

New in 0.7

6.1.1.1 Overview

The `ArchiveDownloader` class encapsulates the functionality of downloading URLs, their related resources, and rewriting page links to point to their downloaded location.

All the downloaded content becomes part of an archive, which can be saved and reused on future runs of the script. After the download completes, the script could even be run offline.

This enables many things, including the ability to update and re-run scripts even if the original content was removed, easily and automatically create a dependency zip of content shared by various pages in the archive, and create HTML5 zips for pages that correctly include all necessary resources without extra code.

6.1.1.2 Using ArchiveDownloader

`ArchiveDownloader` has the following general workflow:

- Create an `ArchiveDownloader` instance before downloading any web content.
- Call `get_page` on that instance passing in full URLs to pages you wish to download.
- Once content has been downloaded, you need to create an HTML5 zip of the content.
  - If the content does not need to be modified, call `export_page_as_zip` and use the zip created as a file for an HTML5 app node.
  - If you need to make modifications, call `create_zip_dir_for_page`, then modify the files in the directory it returns as needed. (Not modifying the original sources allows you to keep a clean copy at all times.) Finally, create a ZIP by calling `ricecooker.utils.create_predictable_zip` and use the zip created as a file for an HTML5 app node.

Usage example:

```
from ricecooker.utils.downloader import ArchiveDownloader

sushi_url = 'https://en.wikipedia.org/wiki/Sushi'

archive = ArchiveDownloader("downloads/archive_1")
# Download and store page in the archive
archive.get_page(sushi_url)
# Convert page into a Kolibri-friendly HTML5 zip file
zip_file = archive.export_page_as_zip(sushi_url)
# ... code to add zip_file to an HTML5AppNode ...
```

Example scripts:

The COVID 19 Simulations sushi chef provides a relatively small and simple example of how to use `ArchiveDownloader`.

6.1.1.3 Using `get_page`

By default, `get_page` will download the page and all its related resources, including CSS, image, and JS files. You can also pass a few optional arguments to modify its behavior:

`refresh` [True | False]: If `True`, this will re-download the page, even if it is already in the archive.

`run_js` [True | False]: If `True`, the page will be loaded using `pyppeteer` and wait until page load handlers have run before downloading the content. If `False` (default), it will use `requests` to download the content.

`link_policy` [Dict or None]: Defines how to handle scraping of page links. The default, `None`, indicates no scraping. If a dictionary is passed, it may contain the following keys:

- `levels` [Integer]: (Required) Number of levels deep to scrape links.
- `scope` [String]: (Optional) Defaults to "internal", which only scrapes links on the same domain. Change to "all" to scrape links from external domains as well.
- `whitelist` [List]: (Optional) A list of strings containing URLs to be whitelisted. They will be compared against complete URLs, so the strings can be as complete as desired. e.g. `www.mydomain.com/subdir/subdir2/` can be used to match against only URLs in that particular subdir.
- `blacklist` [List]: (Optional) A list of strings containing URLs to be blacklisted. URLs can be specified using the same rules as the `whitelist` argument.

6.1.2 downloader.py Functions

The Ricecooker module `utils/downloader.py` provides a `read` function that can be used to read the file contents from both urls and local file paths.

Usage examples:

```
from ricecooker.utils.downloader import read

local_file_content = read('/path/to/local/file.pdf')              # Load local
file
web_content = read('https://example.com/page')                    # Load web page
contents
web_content2 = read('https://example.com/loadpage', loadjs=True)  # Load js before
getting contents
```

The `loadjs` option will run the JavaScript code on the webpage before reading the contents of the page, which can be useful for scraping certain websites that depend on JavaScript to build the page DOM tree.

If you need to use a custom session, you can also use the `session` option. This can be useful for sites that require login. See the sushi-chef-firki code for an example of this.

6.1.3 Caching

Requests made with the `read` method are cached by default, and the cache doesn't have an expiration date. The cached files are stored the folder `.webcache` in the chef repository. You must manually delete this folder when the source website changes.

```
rm -rf .webcache
```

This sample code shows how to setup requests session caching that expires after one day.

6.1.4 Further reading

- Tutorial on the Python requests module.

6.2 Parsing HTML using BeautifulSoup

BeautifulSoup is an HTML parsing library that allows you to "select" various DOM elements, and extract their attributes and text contents.

6.2.1 Video tutorial

To get started, you can watch this cheffing video tutorial that will show the basic steps of using `requests` and `BeautifulSoup` for crawling a website. See the sushi-chef-shls code repo for the final version of the web crawling code that was used for this content source.

6.2.2 Scraping 101

The basic code to GET the HTML source of a webpage and parse it:

```
import requests
from bs4 import BeautifulSoup

url = 'https://somesite.edu'
html = requests.get(url).content
doc = BeautifulSoup(html, "html5lib")
```

You can now call `doc.find` and `doc.find_all` methods to select various DOM elements:

```
special_ul = doc.find('ul', class_='some-special-class')
section_lis = special_ul.find_all('li', recursive=False)  # search only immediate
children
for section_li in section_lis:
    print('processing a section <li> right now...')
    print(section_li.prettify())  # useful seeing HTML in when developing...
```

The most commonly used parts of the BeautifulSoup API are:

- `.find(tag_name, <spec>)`: find the next occurrence of the tag `tag_name` that has attributes specified in `<spec>` (given as a dictionary), or can use the shortcut options `id` and `class_` (note extra underscore).
- `.find_all(tag_name, <spec>)`: same as above but returns a list of all matching elements. Use the optional keyword argument `recursive=False` to select only immediate child nodes (instead of including children of children, etc.).
- `.next_sibling`: find the next element (for badly formatted pages with no useful selectors)
- `.get_text()` extracts the text contents of the node. See also helper method called `get_text` that performs additional cleanup of newlines and spaces.
- `.extract()`: to extract an element from the DOM tree
- `.decompose()`: useful to remove any unwanted DOM elements (same as `.extract()` but throws away the extracted element)

6.2.2.1 Example 1

Here is some sample code for getting the text of the LE mission statement:

```
from bs4 import BeautifulSoup
from ricecooker.utils.downloader import read

url = 'https://learningequality.org/'
html = read(url)
doc = BeautifulSoup(html, 'html5lib')

main_div = doc.find('div', {'id': 'body-content'})
mission_el = main_div.find('h3', class_='mission-state')
mission = mission_el.get_text().strip()
print(mission)
```

6.2.2.2 Example 2

To print a list of all the links on the page, use the following code:

```
links = doc.find_all('a')
for link in links:
    print(link.get_text().strip(), '-->', link['href'])
```

6.2.3 Further reading

For more info about BeautifulSoup, see the docs.

There are also some excellent tutorials online you can read:

- http://akul.me/blog/2016/beautifulsoup-cheatsheet/
- http://youkilljohnny.blogspot.com/2014/03/beautifulsoup-cheat-sheet-parse-html-by.html
- http://www.compjour.org/warmups/govt-text-releases/intro-to-bs4-lxml-parsing-wh-press-briefings/

6.3 Debugging HTML5 app rendering in Kolibri

6.3.1 The problem

The edit-preview loop for HTML5App nodes is very time consuming since it requires running the sushichef script, waiting for the channel to publish in Studio, then going through the channel UPDATE steps in Kolibri before you can see the edits.

6.3.2 Local HTMLZip replacement hack

It is possible to have a quick edit-refresh-debug loop for HTML5Apps using a local Kolibri instance by zipping and putting the work-in-progress `webroot/` content into an existing zip file in the local `.kolibrihome/content/storage/` directory.

Under normal operations files in `content/storage/` are stored based on md5 hash of their contents, but if you replace a file with a different contents, Kolibri will still load it.

We provide the script kolibripreview.py to help with this file-replacement process used for HTML5App debugging and dev.

6.3.3 Prerequisites

1. Install Kolibri on your machine.
2. Find the location of `KOLIBRI_HOME` directory for your Kolibri instance. By default Kolibri will use the directory `.kolibri` in your User's home folder.
3. Import the **HTML5App Dev Channel** using the token `bilol-vivol` into Kolibri. Note you can use any channel that contains .zip files for this purpose, but the code examples below are given based on this channel, which contains the placeholder file `9cf3a3ab65e771abfebfc67c95a8ce2a.zip` which we'll be replacing. After this step, you can check the file `$KOLIBRI_HOME/content/storage/9/c/9cf3a3ab65e771abfebfc67c95a8ce2a.zip` exists on your computer and view it at http://localhost:8080/en/learn/#/topics/c/60fe072490394595a9d77d054f7e3b52
4. Download the helper script `kolibripreview.py` and make it executable:

```
wget https://raw.githubusercontent.com/learningequality/ricecooker/master/ricecooker/utils/kolibripreview.py
chmod +x kolibripreview.py
```

6.3.4 Usage

Assuming you have prepared work-in-progress draft directory `webroot`, you can load int into Kolibri by running:

```
./kolibripreview.py --srcdir webroot --destzip ~/.kolibri/content/storage/9/c/9cf3a3ab65e771abfebfc67c95a8ce2a.zip
```

The script will check that the file `webroot/index.html` exists then create a zip file from the `webroot` directory and replace the placeholder .zip file. Opening and refreshing the page http://localhost:8080/en/learn/#/topics/c/60fe072490394595a9d77d054f7e3b52 will show you to the result of your work-in-progress `HTML5App`.

You'll need to re-run the script whenever you make changes to the `webroot` then refresh the Kolibri page. It's not quite webpack live dev server, but much faster than going through the ricecooker uploadchannel > Studio PUBLISH > Kolibri UPDATE, Kolibri IMPORT steps.

6.3.5 Testing in different releases

If you need to test your `HTML5App` works in a specific version of Kolibri, you can quickly download the `.pex` file and run it as a "one off" test in temporary `KOLIBRI_HOME` location (to avoid clobbering your main Kolibri install). A `.pex` file is a self-contained Python EXecutable file that contains all libraries and is easy to run without requiring setting up a virtual environment or installing dependencies. You can download Kolibri `.pex` files from the Kolibri releases page on github.

The instructions below use the pex file `kolibri-0.13.2.pex` which is the latest at the time of writing this, but you can easily adjust the commands to any version.

```
# Download the .pex file
wget https://github.com/learningequality/kolibri/releases/download/v0.13.2/kolibri-
0.13.2.pex

# Create a temporary directory
mkdir -p ~/.kolibrihomes/kolibripreview
export KOLIBRI_HOME=~/.kolibrihomes/kolibripreview

# Setup Kolibri so you don't have to go through the setup wizard
python kolibri-0.13.2.pex manage provisiondevice \
  --facility "$USER's Kolibri Facility" \
  --preset informal \
  --superusername devowner \
  --superuserpassword admin123 \
  --language_id en \
  --verbosity 0 \
  --noinput

# Import the HTML5App Dev Channel
python kolibri-0.13.2.pex manage importchannel network
0413dd5173014d33b5a98a8c00943724
python kolibri-0.13.2.pex manage importcontent network
0413dd5173014d33b5a98a8c00943724

# Start Kolibri (and leave it running)
python kolibri-0.13.2.pex start --foreground
```

After that you can use the script as usual:

1. Replace placeholder .zip with contents of `webroot`:

```
./kolibripreview.py  --srcdir webroot --destzip=~/.kolibrihomes/
kolibripreview/content/storage/9/c/9cf3a3ab65e771abfebfc67c95a8ce2a.zip
```

2. Open and refresh http://localhost:8080/learn/#/topics/c/60fe072490394595a9d77d054f7e3b52

6.3.6 Further reading

See the docs page on *HTML Apps* for info about technical details and best practices for packaging web content for use in the Kolibri Learning Platform.

6.4 PDF Utils

The module `ricecooker.utils.pdf` contains helper functions for manipulating PDFs.

6.4.1 PDF splitter

When importing source PDFs like books that are very long documents (100+ pages), it is better for the Kolibri user experience to split them into multiple shorter PDF content nodes.

The `PDFParser` class in `ricecooker.utils.pdf` is a wrapper around the `PyPDF2` library that allows us to split long PDF documents into individual chapters, based on either the information available in the PDF's table of contents, or user-defined page ranges.

6.4.1.1 Split into chapters

Here is how to split a PDF document located at `pdf_path`, which can be either a local path or a URL:

```
from ricecooker.utils.pdf import PDFParser

pdf_path = '/some/local/doc.pdf' or 'https://somesite.org/some/remote/doc.pdf'
with PDFParser(pdf_path) as pdfparser:
    chapters = pdfparser.split_chapters()
```

The output `chapters` is list of dictionaries with `title` and `path` attributes:

```
[
  {'title':'First chapter',  'path':'downloads/doc/First-chapter.pdf'},
  {'title':'Second chapter', 'path':'downloads/doc/Second-chapter.pdf'},
```

(continues on next page)

(continued from previous page)

```
  ...
]
```

Use this information to create an individual `DocumentNode` for each PDF and store them in a `TopicNode` that corresponds to the book:

```
from ricecooker.classes import nodes, files

book_node = nodes.TopicNode(title='Book title', description='Book description')
for chapter in chapters:
    chapter_node = nodes.DocumentNode(
        title=chapter['title'],
        files=[files.DocumentFile(chapter['path'])],
        ...
    )
    book_node.add_child(chapter_node)
```

By default, the split PDFs are saved in the directory `./downloads`. You can customize where the files are saved by passing the optional argument `directory` when initializing the `PDFParser` class, e.g., `PDFParser(pdf_path, directory='somedircustomdir')`.

The `split_chapters` method uses the internal `get_toc` method to obtain a list of page ranges for each chapter. Use `pdfparser.get_toc()` to inspect the PDF's table of contents. The table of contents data returned by the `get_toc` method has the following format:

```
[
  {'title': 'First chapter',  'page_start': 0,  'page_end': 10},
  {'title': 'Second chapter', 'page_start': 10, 'page_end': 20},
  ...
]
```

If the page ranges automatically detected form the PDF's table of contents are not suitable for the document you're processing, or if the PDF document does not contain table of contents information, you can manually create the title and page range data and pass it as the `jsondata` argument to the `split_chapters()`.

```
page_ranges = pdfparser.get_toc()
# possibly modify/customize page_ranges, or load from a manually created file
chapters = pdfparser.split_chapters(jsondata=page_ranges)
```

6.4.1.2 Split into chapters and subchapters

By default the `get_toc` will detect only the top-level document structure, which might not be sufficient to split the document into useful chunks. You can pass the `subchapters=True` optional argument to the `get_toc()` method to obtain a two-level hierarchy of chapters and subchapter from the PDF's TOC.

For example, if the table of contents of textbook PDF has the following structure:

```
Intro
Part I
   Subchapter 1
   Subchapter 2
Part II
   Subchapter 21
   Subchapter 22
Conclusion
```

then calling `pdfparser.get_toc(subchapters=True)` will return the following chapter-subchapter tree structure:

```
[
  { 'title': 'Part I', 'page_start': 0,  'page_end': 10,
    'children': [
        {'title': 'Subchapter 1',  'page_start': 0,  'page_end': 5},
        {'title': 'Subchapter 2',  'page_start': 5,  'page_end': 10}
     ]},
  { 'title': 'Part II', 'page_start': 10,  'page_end': 20,
    'children': [
        {'title': 'Subchapter 21',  'page_start': 10, 'page_end': 15},
        {'title': 'Subchapter 22',  'page_start': 15,  'page_end': 20}
     ]},
  { 'title': 'Conclusion', 'page_start': 20,  'page_end': 25 }
]
```

Use the `split_subchapters` method to process this tree structure and obtain the tree of title and paths:

```
[
  { 'title': 'Part I',
    'children': [
```

(continues on next page)

(continued from previous page)

```
        {'title': 'Subchapter 1', 'path': '/tmp/0-0-Subchapter-1.pdf'},
        {'title': 'Subchapter 2', 'path': '/tmp/0-1-Subchapter-2.pdf'},
    ]},
  { 'title': 'Part II',
    'children': [
        {'title': 'Subchapter 21', 'path': '/tmp/1-0-Subchapter-21.pdf'},
        {'title': 'Subchapter 22', 'path': '/tmp/1-1-Subchapter-22.pdf'},
    ]},
  { 'title': 'Conclusion', 'path': '/tmp/2-Conclusion.pdf'}
]
```

You'll need to create a `TopicNode` for each chapter that has `children` and create a `DocumentNode` for each of the children of that chapter.

6.4.2 Accessibility notes

Do not use `PDFParser` for tagged PDFs because splitting and processing loses the accessibility features of the original PDF document.

6.5 Video compression tools

Importing video files into Kolibri requires special considerations about the file size of the video resources that will be imported.

Below are some general guidelines for importing video files:

- Use the `.mp4` file format
- Use the `H.264` (a.k.a. `x264`) video codec to ensure video will play in web browsers
- Use the `aac` audio codec
- Use compression
  - Short videos (5-10 mins long) should be roughly less than 15MB
  - Longer video lectures (1 hour long) should not be larger than 200MB
  - High-resolution videos should be converted to lower resolution formats: Here are some recommended choices for video vertical resolution:
    * Use max height of `480` for videos that work well in low resolution (most videos)
    * Use max height of `720` for high resolution videos (lectures with writing on board)

Using video compression and low resolutions is important for the context of use. **Think of the learners and the device they will be using to view the videos.** Consider also the overall size of the

channel—**how much storage space** will be required for the entire collection of videos?

Let's now look at compression tools that you can use to ensure a good video experience for all Kolibri users, regardless of their device.

6.5.1 Automated conversion

The `ricecooker` library can handle the video compression for you if you specify the `--compress` command line argument to the chef script, e.g. `python chef.py ... --compress`. Under the hood, the `ffmpeg` video conversion program will be called to compress video files before uploading them to Kolibri Studio. Specifying `--compress` on the command line will use the following default settings:

```
ffmpeg -i inputfile.mp4 \
  -b:a 32k -ac 1 \
  -vf scale="'w=-2:h=trunc(min(ih,480)/2)*2'" \
  -crf 32 \
  -profile:v baseline -level 3.0 -preset slow -v error -strict -2 -stats \
  -y outputfile.mp4
```

This command takes the `inputfile.mp4` and outputs the file `outputfile.mp4` that has the following transformations applied to it:

- Limits the audio codec to 32k/sec
- Scale the video to max-height of 480 pixels
- Compress the video with CRF of 32 (constant rate factor)

To overwrite these defaults, chef authors can pass the argument `ffmpeg_settings` (dict), when creating `VideoFile` object, and specify these options: `crf`, `max_height`, and `max_width`.

6.5.2 Manual conversion

For optimal control of the compression options, users should perform the conversion and compression steps before uploading their videos to Kolibri Studio. We highly recommend the command line tool `ffmpeg`. You'll need to use it through the command prompt (Terminal in linux, CMD in Windows). Any video conversion and compression operation can be performed by setting the appropriate parameters.

6.5.2.1 Installing ffmpeg

Before proceeding, please go and download the `ffmpeg` program for you OS:

Links:

- Homepage: https://www.ffmpeg.org/
- Downloads for windows users: https://web.archive.org/web/20200918193047/https://ffmpeg.zeranoe.com Choose 64bit "static" version to download, unzip the archive, then go to the folder called `bin/` inside the zip file. Copy the files `ffmpeg.exe` and `ffprobe.exe` to the folder on your computer where your videos are stored.

To check the installation was successful you can open a command line prompt (cmd.exe on Windows, or terminal on mac/linux), and try typing in the command:

```
ffmpeg -h
```

which will print command help information. You can see the full list command line options for `ffmpeg` here: https://www.ffmpeg.org/ffmpeg.html. Don't worry you won't need to use all of them.

If you see the error message "ffmpeg is not recognized as an internal or external command, operable program or batch file," you will have to change directory to the folder where you saved the program files `ffmpeg.exe` and `ffprobe.exe` (e.g. use `cd Desktop` if saved on the desktop or `cd %HOMEPATH%\Documents` to go to your Documents folder).

6.5.2.2 Looking around with ffprobe

Equally useful is the command `ffprobe` which prints detailed information for any video files. To illustrate the usefulness, let's see what info `ffprobe` can tells us about some video files downloaded from the internet. You can download the same files from here if you want to follow along (download the three different video formats available in the sidebar: `ogv`, `mpg`, and `mp4`)

To check what's in the file `CM_National_Rice_Cooker_1982.ogv` use the command:

```
ffprobe  CM_National_Rice_Cooker_1982.ogv

Input #0, ogg, from 'CM_National_Rice_Cooker_1982.ogv':
  Duration: 00:00:15.03, start: 0.000000, bitrate: 615 kb/s
    Stream #0:0: Video: theora, yuv420p,
                 400x300 [SAR 1:1 DAR 4:3], 29.97 fps, 29.97 tbr, 29.97 tbn, 29.97
tbc
    Stream #0:1: Audio: vorbis, 44100 Hz, stereo, fltp, 128 kb/s
```

The video codec is `theora` and the audio codec is `vorbis`, so this video will need to be converted before uploading to Studio.

Similarly we can check the codecs for `CM_National_Rice_Cooker_1982.mpg` using

```
ffprobe  CM_National_Rice_Cooker_1982.mpg

Input #0, mpeg, from 'CM_National_Rice_Cooker_1982.mpg':
Duration: 00:00:15.02, start: 0.233367, bitrate: 6308 kb/s
  Stream #0:0[0x1e0]: Video: mpeg2video (Main), yuv420p(tv, smpte170m, top first),
                      720x480 [SAR 8:9 DAR 4:3], 29.97 fps, 29.97 tbr, 90k tbn, 59.
94 tbc
  Stream #0:1[0x1c0]: Audio: mp2, 48000 Hz, stereo, s16p, 224 kb/s
```

The video codec is `mpeg2video` and the audio codec is `mp2`, so this video too will need to be converted.

Finally, to check the codecs for `CM_National_Rice_Cooker_1982.mp4`, we use

```
ffprobe  CM_National_Rice_Cooker_1982.mp4

Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'CM_National_Rice_Cooker_1982.mp4':
  Duration: 00:00:15.05, start: -0.012585, bitrate: 835 kb/s
    Stream #0:0(und): Video: h264 (Constrained Baseline) (avc1 / 0x31637661),
yuv420p,
                      640x480 [SAR 1:1 DAR 4:3], 700 kb/s, 29.97 fps, 29.97 tbr,
30k tbn, 59.94 tbc (default)
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 44100 Hz, stereo, fltp,
129 kb/s (default)
```

Here we see the `h264` video codec and `aac`/`mp4a` audio codec so this file can be uploaded to Studio as is. These codecs are relatively well supported by most browsers. This video can be uploaded to Kolibri.

6.5.2.3 Converting files using ffmpeg

Recall the file `CM_National_Rice_Cooker_1982.mpg` that we downloaded above, which uses the Kolibri-incompatibe codecs `mpeg2video` and `mp2`. Let's see how to use the `ffmpeg` command to convert it to the supported codecs:

```
ffmpeg -i CM_National_Rice_Cooker_1982.mpg \
  -b:a 32k -ac 1 \
  -vf scale="'w=-2:h=trunc(min(ih,480)/2)*2'" \
  -crf 32 \
  -profile:v baseline -level 3.0 -preset slow -v error -strict -2 -stats \
  -y compressed.mp4
```

Note the \ character denotes line-continuation and works only on UNIX. Windows users should put the entire command on a single line:

```
ffmpeg -i CM_National_Rice_Cooker_1982.mpg   -b:a 32k -ac 1   -vf scale="'w=-2:
h=trunc(min(ih,480)/2)*2'"   -crf 32   -profile:v baseline -level 3.0 -preset slow
-v error -strict -2 -stats   -y compressed.mp4
```

This command will run for some time (video transcoding takes a lot of CPU power). In the end, if you check using `ffprobe compressed.mp4`, you'll see that the converted output file has video codec `h264` and audio codec `aac`. The resolution `720x480` and bitrate `534 kb/s` are also very good parameters. Note the file size of `compressed.mp4` is 1MB which is twice smaller than the file `mp4` file which we obtained directly form the web `CM_National_Rice_Cooker_1982.mp4`. Clearly the compression option `-crf 32` had an effect.

The video `compressed.mp4` is now ready for upload to Studio!

6.5.2.4 Using the ffmpeg helper scripts

We provide a helper script to help run the ffmpeg command. The instructions are different depending if your operating systems is Windows or Mac/Linux:

- For Windows users, download the file convertvideo.bat and save it to your computer. Make sure the extension is `.bat` (Windows batch file). Put the `convertvideo.bat` file in the same folder where you copied `ffmpeg.exe`. To convert `inputfile.mp4` to `outputfile.mp4` using the conversion script, open a command line prompt, navigate to the folder where `convertvideo.bat` and `ffmpeg.exe` are stored, and type the following command:

  ```
  convertvideo.bat  inputfile.mp4  outputfile.mp4
  ```

- Linux and Mac users should download convertvideo.sh, save it to the folder where all the videos are. Next open a command prompt and change directory to that folder. Make the script executable using `chmod u+x convertvideo.sh`, then you can start converting videos using:

```
./convertvideo.sh  inputfile.mp4  outputfile.mp4
```

See https://youtu.be/oKbCbuDlRmY for a video walkthrough of the steps and example usage of the batch script.

The conversion scripts provided are just wrappers for the `ffmpeg` command, to make it easier for you so you won't have to remember all the command line options. If you need to adjust the conversion parameters, you edit the scripts—they are ordinary text files, so you can edit them with notepad.

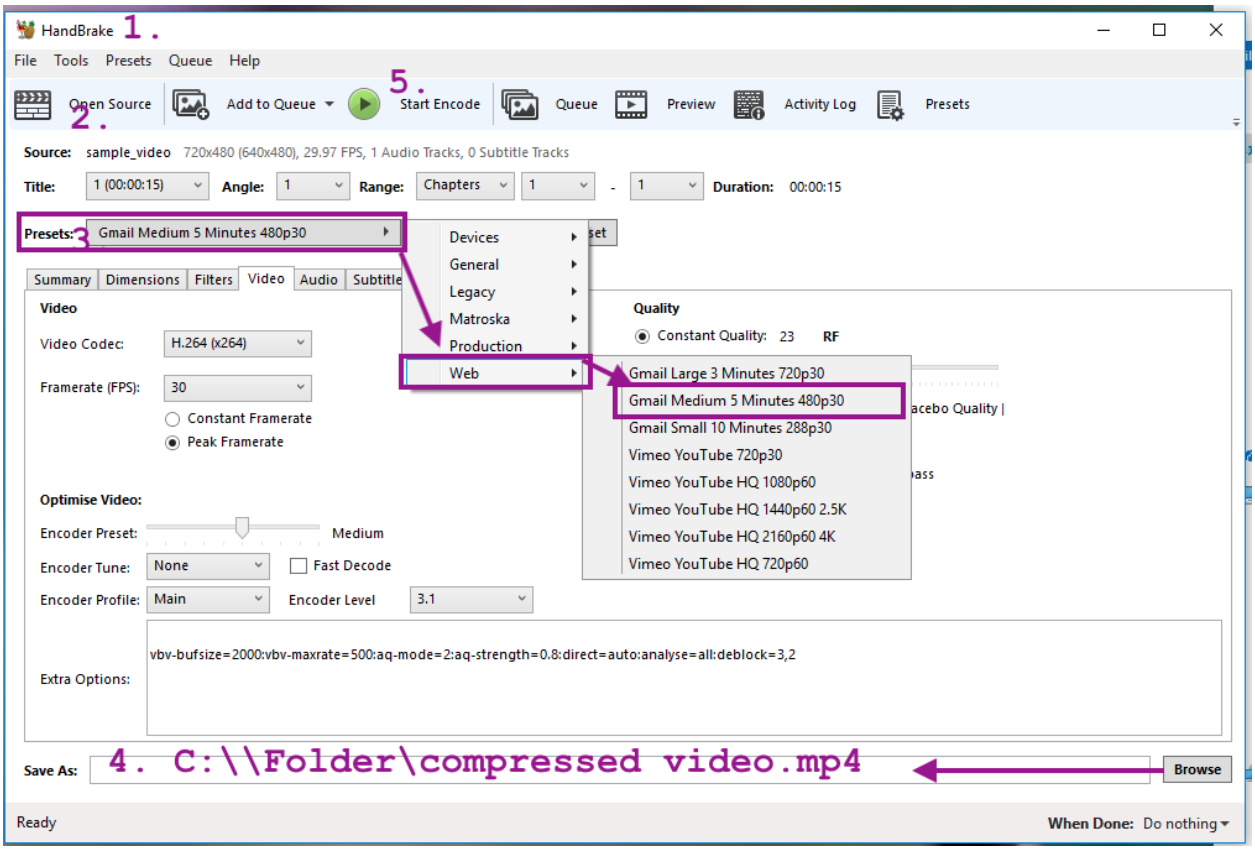
Note video conversion takes a long time, so be prepared to get a coffee or two.

6.5.2.5 HandBrake

If you don't have many videos to convert, you can use HandBrake, which is a video conversion tool with a graphical user interface. Handbrake uses `ffmpeg` under the hood, so the same compression results can be achieved as with the more technical options presented above.

Here are steps for converting videos using HandBrake:

1. **Download** and install handbrake from here https://handbrake.fr/
2. **Open** the video file you want to compress.
3. From the presets menu, choose **Web > Gmail Medium 5 Minutes 480p30**
4. Set the output filename (e.g. you could use the same as input filename, but append `_compressed.mp4`). Make sure to use the `.mp4` extension.
5. Click the **Start Encode** button.



Here is a video guide to using HandBrake for compressing videos.

The **Web > Gmail Medium 5 Minutes 480p30** preset will use the `x264` video codec, `aac` audio codec, and `480` vertical resolution, and compression rate `crf=23`. The 480 vertical resolution is a good choice for most videos, but if you find the compressed output to be too low quality, you can try the preset **Web > Gmail Large 3 Minutes 720p30**, which will result in larger videos files with 720 vertical resolution.

If your channel contains many videos, or very long videos, you should consider increasing the "Constant Rate Factor" compression parameter in the Video settings. Using the value RF=32 will result in highly compressed videos, with very small file sizes.

6.5.2.6 Experimenting

Since every content source is unique, we recommend that you experiment with different compression options. The command line tool `ffmpeg` offers a very useful option called `crf` which stands for Constant Rate Factor. **Setting this single parameter allows for controlling overall video quality.** For example, setting `crf=24` produces high quality video (and possibly large file size), `crf=28` is a mid-range quality, and values of `crf` above 30 produce highly-compressed videos with small size.

Here are the steps to preview different compression factors in Kolibri:

- Choose a sample video from your collection, let's call it `video.mp4`
- Try different compression options for it:
  - Create a CRF=24 version using `ffmpeg -i video.mp4 ... -crf 24 video_crf24.mp4`
  - Create a CRF=28 version using `ffmpeg -i video.mp4 ... -crf 28 video_crf28.mp4`
  - Create a CRF=30 version using `ffmpeg -i video.mp4 ... -crf 30 video_crf30.mp4`
- Upload the original and the compressed version to a Studio channel
- PUBLISH the channel and record the channel token
- Import the channel into a Kolibri instance using the channel token
- Test video playback on different devices (desktop and mobile browsers on all OSs)

6.6 Spreadsheet Metadata Workflow

It is possible to create Kolibri channels by specifying all the channel metadata in the form of spreadsheet or CSV files instead of through Python code.

6.6.1 CSV Metadata Workflow

It is possible to create Kolibri channels by:

- Organizing content items (documents, videos, mp3 files) into a folder hierarchy on the local file system
- Specifying metadata in the form of CSV files

The CSV-based workflow is a good fit for non-technical users since it doesn't require writing any code, but instead can use the Excel to provide all the metadata.

- CSV-based workflow README
- Example content folder
- Example Channel.csv metadata file
- Example Content.csv metadata file

Organizing the content into folders and creating the CSV metadata files is most of the work, and can be done by non-programmers. The generic sushi chef script (`LineCook`) is then used to upload the channel.

6.6.1.1 CSV Exercises

You can also use the CSV metadata workflow to upload simple exercises to Kolibri Studio. See this doc for the technical details about creating exercises.

6.6.2 CSV Exercises Workflow

In addition to content nodes (files) and topics (folders), we can also specify exercises using CSV metadata files (and associated images).

Exercises nodes store the usual metadata that all content nodes have (title, description, author, license, etc.) and contain multiple types of questions. The currently supported question types for the CSV workflow are:

- `input_question`: Numeric input question, e.g. What is 2+2?
- `single_selection`: Multiple choice questions where a single correct answer.
- `multiple_selection`: Multiple choice questions with multiple correct answers/

To prepare a CSV content channel with exercises, you need the usual things (A channel directory `channeldir`, `Channel.csv`, and `Content.csv`) and two additional metadata files `Exercises.csv` and `ExerciseQuestions.csv`, the format of which is defined below.

You can download template HERE https://github.com/learningequality/sample-channels/tree/master/channels/csv_exercises

6.6.2.1 Exercises.csv

A CSV file that contains the following fields:

- `Path *`:
- `Title *`:
- `Source ID *`: A unique identifier for this exercise, e.g., `exrc1`
- `Description`:
- `Author`:
- `Language`:
- `License ID *`:
- `License Description`:
- `Copyright Holder`:
- `Number Correct`: (integer, optional) This field controls how many questions students must get correct in order to complete the exercise.

- `Out of Total`: (integer, optional) This field controls how many questions students are presented in a row, if not specified the value will be determined automatically based on the number of questions available (up to maximum of 5).
- `Randomize`: (bool) True or False
- `Thumbnail`:

6.6.2.2 ExerciseQuestions.csv

Individual questions

- `Source ID *`: This field is the link (foreign key) to the an exercise node, e.g. `exrc1`
- `Question ID *`: A unique identifier for this question within the exercise, e.g. q1
- `Question type *`: (str) Question types are defined in le-utils. The currently supported question types for the CSV workflow are:
  - `input_question`: Numeric input question, e.g. What is 2+2?
  - `single_selection`: Multiple choice questions where a single correct answer.
  - `multiple_selection`: Multiple choice questions with multiple correct answers/
- `Question *`: (markdown) contains the question setup and the prompt, e.g. "What is 2+2?"
- `Option A`: (markdown) The first answer option
- `Option B`: (markdown)
- `Option C`: (markdown)
- `Option D`: (markdown)
- `Option E`: (markdown) The fifth answer option
- `Options F...`: Use this field for questions with more than five possible answers. This field can contain a list of multiple "sushi"-separated string values, e.g., "Answer FsushiAnswer Gsushi-Answer H"
- `Correct Answer *`: The correct answer
- `Correct Answer 2`: Another correct
- `Correct Answer 3`: A third correct answer
- `Hint 1`: (markdown)
- `Hint 2`:
- `Hint 3`:
- `Hint 4`:
- `Hint 5`:
- `Hint 6+`: Use this field for questions with more than five hints. This field stores a list of "sushi"-separated string values, e.g., "Hint 6 textsushiHint 7 textsushiHing 8 text"

The question, options, answers, and hints support Markdown and LaTeX formatting:

- Use two newlines to start a new paragraph
- Use the syntax `![](relative/path/to/figure.png)` to include images in text field

- Use dollar signs as math delimiters `$\alpha\beta$`

Markdown image paths

Note that image paths used in Markdown will be interpreted as relative to the location where the chef is running. For example, if the sushi chef project directory looks like this:

```
csvchef.py
figures/
  exercise3/
    somefig.png
content/
  Channel.csv
  Content.csv
  Exercises.csv
  ExerciseQuestions.csv
  channeldir/
     somefile.mp4
     anotherfile.pdf
```

Then the code for including `somefig.png` a Markdown field of an exercise question is `![](figures/exercise3/somefig.png)`.

6.6.2.3 Ordering

The order that content nodes appear in the channel is determined based on their filenames in alphabetical order, so the choice of filenames can be used to enforce a particular order of items within each folder.

The filename part of the `Path *` attribute of exercises specified in Exercises.csv gives each exercise a "virtual filename" so that exercises will appear in the same alphabetical order, intermixed with the CSV content items defined in `Content.csv`.

6.6.2.4 Implementation details

- To add exercises to a certain channel topic, the folder corresponding to this topic must exist inside the `channeldir` folder (even if it contains no files). A corresponding entry must be added to `Content.csv` to describe the metadata for the topic node containing the exercises.

Chapter 7

Developer docs

To learn about the inner workings of the `ricecooker` library, consult the following pages:

7.1 SushOps

SushOps engineers (also called ETL engineers) are responsible for making sure the overall content pipeline runs smoothly. Assuming the *chefops* is done right, running the chef script should be as simple as running a single command. SushOps engineers need to make sure not only that chef is running correctly, but also monitor content in Kolibri Studio, in downstream remixed channels, and in Kolibri installations.

SushOps is an internal role to Learning Equality but we'll document the responsibilities here for convenience, since this role is closely related to the `ricecooker` library.

7.1.1 Project management and support

SushOps manage and support developers working on new chefs scripts, by reviewing spec sheets, writing technical specs, crating necessary git repos, reviewing pull requests, chefops, and participating in QA.

7.1.2 Cheffing servers

Chef scripts run on various cheffing servers, equipped with appropriate storage space and processing power (if needed for video transcoding). Currently we have:

- CPU-intensive chefs running on `vader`
- various other chefs running on partner orgs infrastructure

7.1.2.1 Cheffing servers conventions

- Put all the chef repos in `/data` (usually a multi-terabyte volume), e.g., use the directory `/data/sushi-chef-{{nickname}}/` for the `nickcname` chef.
- Use the name `sushichef.py` for the chef script
- Document all the instructions and options needed to run the chef script in the chef's `README.md`
- Use the directory `/data/sushi-chef-{{nickname}}/chefdata/tmp/` to store tmp files to avoid cluttering the global `/tmp` directory.
- For long running chefs, use the command `nohup <chef cmd> &` to run the chef so you can close the ssh session (hangup) without the process being terminated.

7.1.3 SushOps tooling and automation

Some of the more repetitive system administration tasks have been automated using `fab` commands:

```
fab -R vader   setup_chef:nickname     # clones the nickname repo and installs
requirements
fab -R vader   update:nickname         # git fetch and git reset --hard to get
latest chef code
fab -R vader   run_chef:nickname       # runs the chef
```

See the content-automation-scripts project for more details.

7.2 Computed identifiers

7.2.1 Channel ID

The `channel_id` (uuid hex str) property is an important identifier that:

- Is used in the "wire format" between `ricecooker` and Kolibri Studio
- Appears as part of URLs on both Kolibri Studio and Kolibri
- Determines the filename for the channel sqlite3 database file that Kolibri imports from Kolibri Studio, from local storage, or from other Kolibri devices via peer-to-peer content import.

To compute the `channel_id`, you need to know the channel's `source_domain` (a.k.a. `channel_info['CHANNEL_SOURCE_DOMAIN']`) and the channel's `source_id` (a.k.a `channel_info['CHANNEL_SOURCE_ID']`):

```
import uuid
domain_namespace = uuid.uuid5(uuid.NAMESPACE_DNS, source_domain)
channel_id = uuid.uuid5(domain_namespace, source_id).hex
```

This above code snippet is useful if you know the `source_domain` and `source_id` and you want to determine the `channel_id` without crating a `ChannelNode` object.

The `ChannelNode` class implements the following methods:

```
class ChannelNode(Node):
    def get_domain_namespace(self):
        return uuid.uuid5(uuid.NAMESPACE_DNS, self.source_domain)
    def get_node_id(self):
        return uuid.uuid5(self.get_domain_namespace(), self.source_id)
```

Given a channel object `ch`, you can find its id using `channel_id = ch.get_node_id().hex`.

7.2.2 Node IDs

Content nodes within the Kolibri ecosystem have the following identifiers:

- `source_id` (str): arbitrary string used to identify content item within the source website, e.g., the a database id or URL.
- `node_id` (uuid): an identifier for the content node within the channel tree
- `content_id` (uuid): an identifier derived from the channel source_domain and the content node's `source_id` used for tracking a user interactions with the content node (e.g. video watched, or exercise completed).

When a particular piece of content appears in multiple channels, or in different places within a tree, the `node_id` of each occurrence will be different, but the `content_id` of each item will be the same for all copies. In other words, the `content_id` keeps track of the "is identical to" information about content nodes. See next section for more info about Content IDs.

Content nodes inherit from the `TreeNode` class, which implements the following methods:

```
class TreeNode(Node):
    def get_domain_namespace(self):
        return self.domain_ns if self.domain_ns else self.parent.get_domain_
namespace()
    def get_content_id(self):
        return uuid.uuid5(self.get_domain_namespace(), self.source_id)
    def get_node_id(self):
        return uuid.uuid5(self.parent.get_node_id(), self.get_content_id().hex)
```

The `content_id` identifier is computed based on the channel source domain, and the `source_id` attribute of the content node. To find the `content_id` hex value for a content node `node`, use `content_id = node.get_content_id().hex`.

The `node_id` of a content nodes within a tree is computed based on the parent node's `node_id` and current node's `content_id`.

7.2.3 Content IDs

Every content node within the Kolibri platform is associated with a `content_id` field that is attached to it throughout its journey: from the content source, to content integration, transformations, transport, remixing, distribution, use, and analytics. All these "downstream" content actions preserve the original `content_id` which was computed based on the source domain (usually the domain name of the content producer), and a unique source identifier that identifies this content item within the source domain.

7.2.3.1 Implementation

```
domain_namespace = uuid.uuid5(uuid.NAMESPACE_DNS, source_domain)
content_id = uuid.uuid5(domain_namespace, source_id)
```

The values of `source_domain` and `source_id` can be arbitrary strings. By convention `source_domain` is set to the domain name of the content source, e.g. `source_domain="khanacademy.org"`. The `source_id` is set to "primary key" of the source database or a canonical URI.

7.2.3.2 Applications

- `content_id`s allows us to correctly assign "activity completed" credit for content items that appears in multiple places within a channel, or in remixed versions.
- Serves as a common identifier for different formats of the same content item, like "high res" and "low res" versions of a Khan Academy video, or ePub and HTML versions of an African Storybook story.
- Enables content analytics and tracking usage of a piece of content regardless of which channel it was obtained from.
- Allows us to match identical content items between different catalogues and content management systems since `content_id`s is based on canonical source ids.

Note it is a non-trivial task to set `source_domain` and `source_id` to the correct values and often requires some "reverse engineering" of the source website. Chef authors must actively coordinate the dev work across different projects to ensure the values of `content_id` for the same content items in different channels are computed consistently. For example, if the current chef you are working on has content overlap with items in another channel, you must look into how it computes its source_domain and source_id and use the same approach to get matching `content_id`s. This cheffing-time deduplication effort is worth investing in, because it makes possible all the applications described above.

7.3 Ricecooker content upload process

This page describes the "behind the scenes" operation of the `ricecooker` framework. The goal is to give an overview of the processing steps that take place every time you run a sushichef script. The goal of this page to help developers know which parts of the code to look at when debugging ricecooker issues, adding support for new content kinds and file types, or when implement performance optimizations.

Each section below describes one of the steps in this process.

7.3.1 Build tree

The ricecooker tree consists of `Node` and `File` objects organized into a tree data structure. The chef script must implement the `construct_channel` method, which gets called by the ricecooker framework:

```
    channel = chef.construct_channel(**kwargs)
```

7.3.2 Validation logic

Every ricecooker `Node` has a `validate` method that performs basic checks to make sure the node's metadata is set correctly and necessary files are provided. Each `File` subclass comes turn has it's own validation logic to ensure the file provided has the appropriate extension.

The tree validation logic is initiated here when the channel's `validate_tree` method is called.

Note: the files have not been processed at this point, so the node and file `validate` methods cannot do "deep checks" on the file contents yet.

7.3.3 File processing

The next step of the ricecooker run occurs when we call the `process_files` method on each node object. The file processing is initiated here and proceeds recursively through the tree.

7.3.3.1 Node.process_files

Each `Node` subclass implements the `process_files` method which includes the following steps:

- call `process_file` on all files associated with the node (described below)
- if the node has children, `process_files` is called on all child nodes
- call the node's `generate_thumbnail` method if it doesn't have a thumbnail already, and the node has `derive_thumbnail` set to True, or if the global command line argument `--thumbnail` (config.THUMBNAILS) is set to True. See notes section "Node.generate_thumbnail".

The result of the `node.process_file()` is a list of processed filenames, that reference files in the content-addressable storage directory `/content/storage/`.

The list of files names can contain `None` values, which indicate that some the file processing for a certain files has failed. These None values are filtered out here before the list is passed onto the file diff and file upload steps.

7.3.3.2 File.process_file

Each `File` subclass implements the `process_file` method that takes care of:

- downloading the `path` (a web URL or a local filepath) and possibly, possibly performing format conversions (e.g. for videos and subtitles)
- saves the file to the content-hash based filesystem in `/storage` and keeping track of the file saved in `.ricecookerfilecache`

- optionally runs video compression on video file and records the output compressed version in `/storage` and `.ricecookerfilecache`

7.3.3.3 Node.generate_thumbnail

Content Node subclasses can implement a the `generate_thumbnail` method that can be used to automatically generate a thumbnail based on the node content. The `generate_thumbnail` will return a `Thumbnail` object if the thumbnail generation worked and the thumbnail will be added to the Node during inside the `Node.process_files` method.

The actual thumbnail generation happens using one of the `pressurcooker` helper functions that currently support PDF, ePub, HTML5, mp3 files, and videos.

7.3.4 File diff

Ricecooker then sends the list of filenames (using the content-hash based names) to Studio to check which files are already present.

```
    get_file_diff(tree, files_to_diff)
        tree.get_file_diff(files_to_diff)
            config.SESSION.post(config.file_diff_url())
```

See managers/tree.py for the code details. Any files that have been previously uploaded to Studio do not need to be (re)uploaded, since Studio already has those files in storage. Studio will reply with the "file difference" list of files that Studio does not have and need to be uploaded, as described in the next section.

7.3.5 File upload

Guess what happens in this step?

```
    upload_files(tree, file_diff)
        tree.upload_files(file_diff)
        tree.reattempt_upload_fails()
```

At the end of this process all the files from the local `storage/` directory will also exist in the Studio's storage directory. You can verify this by trying to access one of the files at `https://studio.learningequality.org/content/storage/c/0/c0ntetha5h0fdaf1le0a0e.ext` with `c0ntetha5h0fdaf1le0a0e.ext` replaced by one of the filenames you find in your local `storage/` directory. Note path prefix `c/0/` is used for filenames starting with `c0`.

See managers/tree.py for details.

7.3.6 Structure upload

The final step happens in the function `tree.upload_tree()`, which repeatedly calls the `add_nodes` method to upload the json metadata to Kolibri Studio, and finally calls the `commit_channel` to finalize the process.

At the end of this chef step the complete channel (files, tree structure, and metadata) is now on Studio. By default, the content is uploaded to a `staging` tree of the channel, which is something like a "draft version" of the channel that is hidden from Studio channel viewers but visible to channel editors. The purpose of the staging tree is to allow channel editors can to review the proposed changes in the "draft version" in the Studio web interface for changes like nodes modified/added/removed and the total storage space requirements.

7.3.7 Deploying the channel (optional)

Studio channel editors can use the `DEPLOY` button in the Studio web interface to activate the "draft copy" and make it visible to all Studio users. This is implemented by replacing the channel's `main` tree with the `staging` tree. During this step, a "backup copy" of channel is saved, called the `previous_tree`.

7.3.8 Publish channel (optional)

The `PUBLISH` channel button on Studio is used to save and export a new version of the channel. The PUBLISH action exports all the channel metadata to a sqlite3 DB file served by Studio at the URL `/content/{{channel_id}}.sqlite3` and ensure the associated files exist in `/content/storage/` which is served by a CDN. This step is a prerequisite for getting the channel out of Studio and into Kolibri. The combination of `{{channel_id}}.sqlite3` file and the files in `/content/storage` define the Kolibri Channels content format. This is what gets exported to the folder `KOLIBRI_DATA` on sdcard or external drives when you use the `EXPORT` action in Kolibri.

7.4 Command line interface

This document describes control flow used by the `ricecooker` framework, and the particular logic used to parse command line arguments. Under normal use cases you shouldn't need modify the default command line parsing, but you need to understand how things work in case want to customize the command line args for your chef script.

7.4.1 Summary

A sushi chef script like this:

```
#!/usr/bin/env python
...
...
class MySushiChef(SushiChef):
    def get_channel(**kwargs)` -> ChannelNode (bare channel, used just for
metadata)
        ...
    def construct_channel(**kwargs) -> ChannelNode (with populated topic tree)
        ...
...
...
if __name__ == '__main__':
    chef = MySushiChef()
    chef.main()
```

7.4.2 Flow diagram

The call to `chef.main()` results in the following sequence of six calls:

```
MySushiChef -----extends----> SushiChef                  commands.uploadchannel
--------------------------    ----------------------     ----------------------
                              1. main()
                              2. parse_args_and_options()
                              3. run(args, options)
                                                         4. uploadchannel(chef,
*args, **options)
                                                         ...
```

(continues on next page)

(continued from previous page)

```
5. get_channel(**kwargs)
                                                        ...
6. construct_channel(**kwargs)
                                                        ...
                                                        ...
                                                        DONE
```

- The chef script must define a subclass of `ricecooker.chefs.SushiChef` that implement the methods `get_channel` and `construct_channel`:

```
class MySushiChef(ricecooker.chefs.SushiChef):
    def get_channel(**kwargs)` -> ChannelNode (bare channel, used just for
info)
        ...
    def construct_channel(**kwargs): --> ChannelNode (with populated Tree)
        ...
```

- Each chef script is a standalone python executable. The `main` method of the chef instance is the entry point used by a chef script:

```
#!/usr/bin/env python
...
...
...
if __name__ == '__main__':
    chef = MySushiChef()
    chef.main()
```

- The `__init__` method of the sushi chef class configures an `argparse` parser. The base class `SushiChef` creates `self.arg_parser` and a chef subclass can adds to this shared parser its own command line arguments. This is used only in vary special cases; for most cases, using the CLI `options` is sufficient.
- The `main` method of the `SushiChef` class parses the command line arguments and calls the `run` method:

```
class SushiChef():
    ...
    def main(self):
        args, options = self.parse_args_and_options()
        self.run(args, options)
```

- The chef's `run` method calls `uploadchannel`

```
class SushiChef():
    ...
    def run(self, args, options):
        ...
        uploadchannel(self, **args.__dict__, **options)
```

  note the chef instance is passed as the first argument, and not path.
- The `uploadchannel` function expects the sushi chef class to implement the following two methods:
  - `get_channel(**kwargs)`: returns a ChannelNode (previously called `create_channel`)
    - as an alternative, if `MySushiChef` has a `channel_info` attribute (a dict) then the default SushiChef.get_channel will create the channel from this info
  - `construct_channel(**kwargs)`: create the channel and build node tree
- Additionally, the `MySushiChef` class can implement the following optional methods that will be called as part of the run
  - `__init__`: if you want to add custom chef-specific command line arguments using argparse
  - `pre_run`: if you need to do something before chef run starts (called by `run`)
  - `run`: in case you want to call `uploadchannel` yourself

7.4.3 Args, options, and kwargs

There are three types of arguments involved in a chef run:

- `args` (dict): command line args as parsed by the sushi chef class and its parents
  - SushiChef: the `SushiChef.__init__` method configures argparse for the following:
    - `compress`, `download_attempts`, `prompt`, `publish`, `resume`, `stage`, `step`, `thumbnails`, `token`, `update`, `verbose`, `warn`
  - MySushiChef: the chef's `__init__` method can define additional cli args
- `options` (dict): additional [OPTIONS...] passed at the end of the command line
  - often used to pass the language option (`./sushichef.py ... lang=fr`)
- `kwargs` (dict): chef-specific keyword arguments not handled by ricecooker's `uploadchannel` method
  - the chef's `run` method makes the call `uploadchannel(self, **args.__dict__, **options)` while the definition of `uploadchannel` looks like `uploadchannel(chef, verbose=False, update=False, ... stage=False, **kwargs)` so `kwargs` contains a mix of both `args` and `options` that are not explicitly expected by the `uploadchannel` function

- The function `uploadchannel` will pass `**kwargs` on to the `chef`'s `get_channel` and `construct_channel` methods as part of the chef run.

7.5 Studio bulk corrections

The command line script `corrections` allows to perform bulk corrections of titles, descriptions, and other attributes for the content nodes of a channel.

Use cases:

- Bulk modify titles and descriptions (e.g. to fix typos)
- Translate titles and/or descriptions (for sources with missing structure translations)
- Enhance content by adding description (case by case detail work done during QA)
- Add missing metadata like author, copyright holder, and tags to content nodes
- Perform basic structural edits to channel (remove unwanted topics and content nodes)

Not use cases:

- Modify a few node attributes (better do manually through the Studio web interface)
- Structural changes (the corrections workflow does not support node moves)
- Global changes (if the same modification must be performed on all nodes in the channel, it would be better to implement these changes during cheffing)

7.5.1 Credentials

In order to use the corrections workflow as part of a chef script, you need to create the file `credentials/studio.json` in the chef repo that contains the following information:

```
{
  "token": "YOURTOKENHERE9139139f3a23232fefefefefefe",
  "username": "your.name@yourdomain.org",
  "password": "yourstudiopassword",
  "studio_url": "https://studio.learningequality.org"
}
```

These credentials will be used to make the necessary Studio API calls. Make sure you have edit rights for this channel.

7.5.2 Corrections workflow

The starting point is an existing channel available on Studio, which we will identify through its Channel ID, denoted `<channel_id>` in code examples below.

7.5.2.1 Step 1: Export the channel metadata to CSV

Export the complete metadata of the source channel as a local `.csv` file using:

```
corrections export <channel_id>
```

This will create the file `corrections-export.csv` which can be opened with a spreadsheet program (e.g. LibreOffice). In order to allow for collaboration, the content of the spreadsheet must be copied to a shared google sheet with permissions set to allow external edits.

7.5.2.2 Step 2: Edit metadata

In this step the content expert (internal or external) edits the metadata for each content node in the shared google sheet. The possible actions (first column) to apply to each row are as follows:

- `modify`: to apply metadata modifications to the topic or content node
- `delete`: to remove the topic or content node from the channel
- Leaving the Action column blank will leave the content node unchanged

All rows with the `modify` keyword in the Action column will undergo metadata modifications according to the text specified in the `New *` columns of the sheet.

For example, to correct typos in the title and description of a content node you must:

- Mark the row with Action=`modify` (first column)
- Add the desired title text in the column `New Title`
- Add the desired description text in the column `New Description`

Note that not all metadata columns need to be specified. The choice of fields that will be edited during the `modify` operation will be selected in the next step.

7.5.2.3 Step 3: Apply the corrections from a google sheet

Once the google sheet has been edited to contain all desired changes in the `New *` columns, the next step is apply the corrections:

```
corrections apply <channel_id> --gsheet_id='<gsheet_id>' --gid=<gsheet_gid>
```

where `<gsheet_id>` is the google sheets document identifier (take from the URL) and `<gsheet_gid>` is identifier of the particular sheet within the spreadsheet document that contains the corrections (usually `<gsheet_gid>=0`).

The attributes that will be edited during the `modify` operation is specified using the `--modifyattrs` command line argument. For example to apply modifications only to the `title` and `description` attributes use the following command:

```
corrections apply <channel_id> --gsheet_id='<gsheet_id>' --gid=<gsheet_gid> --
modifyattrs='title,description'
```

Using the above command will apply only the modifications only from the `New Title` and `New Description` columns and ignore modifications to copyright holder, author, and tags attributes. The default settings is `--modifyattrs=title,description,author,copyright_holder`.

7.5.3 Status

Note the corrections workflows is considered “experimental” and to be used only when no other options are viable (too many edits to do manually through the Studio web interface).

Chapter 8

Community

Learn how to contribute to the project and how you can become part of the community of content developers working to integrate content into the Kolibri platform, and more broadly into all offline learning tools.

8.1 History

8.1.1 0.6.46 (2020-09-21)

- Add `ricecooker.utils.youtube`, containing utilities for handling and caching YouTube videos and playlists.
- Improve validation of `m` and `n` mastery model values in `ExerciseNode`.

8.1.2 0.6.45 (2020-07-25)

- Remove SushiBar remote monitoring and remote command functionality

8.1.3 0.6.44 (2020-07-16)

- Documentation overhaul and refresh, see ricecooker.readthedocs.io
- Add support for specifying a Channel tagline
- Ensure we send `extra_fields` data for all node types
- Make `--reset` behavior the default
- Remove legacy code around `compatibiliry_mode` and BaseChef class
- Improved caching logic (no caching for thumbnails and local paths, always cache youtube downloads)

- Ensure chefs clean up after run (automatic temp files removal)
- Added `save_channel_tree_as_json` full channel metadata as part of every run.
- Added support for web content archiving
- Further improvements to logging
- Bugfix: deep deterministic cache keys for nested dicts values in `ffmpeg_settings`

8.1.4 0.6.42 (2020-04-10)

- Added `--sample N` command line option. Run script with `--sample 10` to produce a test version of the channel with 10 randomly selected nodes from the full channel. Use this to check transformations are working as expected.
- Added `dryrun` command. Use the command `./sushichef.py dryrun` to run the chef as normal but skip the step where the files get uploaded to Studio.
- Added HTTP proxy functionality for YouTubeVideoFile and YouTubeSubtitleFile Set the `PROXY_LIST` env variable to a `;`-separated list of `{ip}:{port}`. Ricecooker will detect the presence of the `PROXY_LIST` and use it when accessing resources via YoutubeDL. Alternatively, set `USEPROXY` env var to use a list of free proxy servers, which are very slow and not reliable.
- Improved colored logging functionality and customizability of logging output.

8.1.5 0.6.40 (2020-02-07)

- Changed default behaviour to upload the staging tree instead of the main tree
- Added `--deploy` flag to reproduce old bahavior (upload to main tree)
- Added thumbnail generating methods for audio, HTML5, PDF, and ePub nodes. Set the `derive_thumbnail=True` when creating the Node instance, or pass the command line argument `--thumbnails` to generate thumbnails for all nodes. Note: automatic thumbnail generation will only work if `thumbnail` is None.

8.1.6 0.6.38 (2019-12-27)

- Added support the `h5p` content kind and `h5p` file type
- Removed monkey-patching of `localStorage` and `document.cookie` in the helper method `download_static_assets`
- Added validation logic for tags
- Improved error reporting

8.1.7 0.6.36 (2019-09-25)

- Added support for tags using the `JsonChef` workflow
- Added validation step to ensure subtitles file are unique for each language code
- Document new `SlidesShow` content kind coming in Kolibri 0.13
- Added docs with detailed instruction for content upload and update workflows
- Bugfixes to file extension logic and improved error handling around subtitles

8.1.8 0.6.32 (2019-08-01)

- Updated documentation to use top-level headings
- Removed support for Python 3.4
- Removed support for the "sous chef" workflow

8.1.9 0.6.31 (2019-07-01)

- Handle more subtitle convertible formats: `SRT`, `TTML`, `SCC`, `DFXP`, and `SAMI`

8.1.10 0.6.30 (2019-05-01)

- Updated docs build scripts to make ricecooker docs available on read the docs
- Added `corrections` command line script for making bulk edits to content metadata
- Added `StudioApi` client to support CRUD (created, read, update, delete) Studio actions
- Added pdf-splitting helper methods (see `ricecooker/utils/pdf.py`)

8.1.11 0.6.23 (2018-11-08)

- Updated `le-utils` and `pressurcooker` dependencies to latest version
- Added support for ePub files (`EPubFile` s can be added of `DocumentNode` s)
- Added tag support
- Changed default value for `STUDIO_URL` to `api.studio.learningequality.org`
- Added `aggregator` and `provider` fields for content nodes
- Various bugfixes to image processing in exercises
- Changed validation logic to use `self.filename` to check file format is in `self.allowed_formats`
- Added `is_youtube_subtitle_file_supported_language` helper function to support importing youtube subs
- Added `srt2vtt` subtitles conversion

- Added static assets downloader helper method in `utils.downloader.download_static_assets`
- Added LineCook chef functions to `--generate` CSV from directory structure
- Fixed the always `randomize=True` bug
- Docs: general content node metadata guidelines
- Docs: video compression instructions and helper scripts `convertvideo.bat` and `convertvideo.sh`

8.1.12 0.6.17 (2018-04-20)

- Added support for `role` attribute on ConentNodes (currently `coach` || `learner`)
- Update pressurecooker dependency (to catch compression errors)
- Docs improvements, see https://github.com/learningequality/ricecooker/tree/master/docs

8.1.13 0.6.15 (2018-03-06)

- Added support for non-mp4 video files, with auto-conversion using ffmpeg. See `git diff b1d15fa 87f2528`
- Added CSV exercises workflow support to `LineCook` chef class
- Added –nomonitor CLI argument to disable sushibar functionality
- Defined new ENV variables: * PHANTOMJS_PATH: set this to a phantomjs binary (instead of assuming one in node_modules) * STUDIO_URL (alias CONTENTWORKSHOP_URL): set to URL of Kolibri Studio server where to upload files
- Various fixes to support sushi chefs
- Removed `minimize_html_css_js` utility function from `ricecooker/utils/html.py` to remove dependency on `css_html_js_minify` and support Py3.4 fully.

8.1.14 0.6.9 (2017-11-14)

- Changed default logging level to –verbose
- Added support for cronjobs scripts via *–cmdsock* (see docs/daemonization.md)
- Added tools for creating HTML5Zip files in utils/html_writer.py
- Added utility for downloading HTML with optional js support in utils/downloader.py
- Added utils/path_builder.py and utils/data_writer.py for creating souschef archives (zip archive that contains files in a folder hierarchy + Channel.csv + Content.csv)

8.1.15 0.6.7 (2017-10-04)

- Sibling content nodes are now required to have unique source_id
- The field *copyright_holder* is required for all licenses other than public domain

8.1.16 0.6.7 (2017-10-04)

- Sibling content nodes are now required to have unique source_id
- The field *copyright_holder* is required for all licenses other than public domain

8.1.17 0.6.6 (2017-09-29)

- Added *JsonTreeChef* class for creating channels from ricecooker json trees
- Added *LineCook* chef class to support souschef-based channel workflows

8.1.18 0.6.4 (2017-08-31)

- Added *language* attribute for *ContentNode* (string key in internal repr. defined in le-utils)
- Made *language* a required attribute for ChannelNode
- Enabled sushibar.learningequality.org progress monitoring by default Set SUSHIBAR_URL env. var to control where progress is reported (e.g. http://localhost:8001)
- Updated le-utils and pressurecooker dependencies to latest

8.1.19 0.6.2 (2017-07-07)

- Clarify ricecooker is Python3 only (for now)
- Use https:// and wss:// for SuhiBar reporting

8.1.20 0.6.0 (2017-06-28)

- Remote progress reporting and logging to SushiBar (MVP version)
- New API based on the SuchiChef classes
- Support existing old-API chefs in compatibility mode

8.1.21 0.5.13 (2017-06-15)

- Last stable release before SushiBar functionality was added
- Renamed –do-not-activate argument to –stage

8.1.22 0.1.0 (2016-09-30)

- First release on PyPI.

Chapter 9

Non-technical documentation redirects

The `ricecooker` docs are mainly a *technical* documentation project intended for readers with prior experience with programming and running command line scripts. If you're not a tech person, no worries. You can do everything that `ricecooker` scripts can do through the Kolibri Studio web interface. We redirect you to the channel create and add your content info in the Kolibri Studio docs.

We recommend reading the Kolibri Content Integration Guide which is a comprehensive guide to the decisions, processes, and tools for integrating external content sources for use in the Kolibri Learning Platform.

Chapter 10

Technical documentation

If you're still reading this, we'll **assume you've got access to the internet**, and some **prior experience with Python**, in which case the technical documentation is for you.

10.1 Install

Guide to installing the `ricecooker` package and system dependencies on Linux, Mac, and Windows.

10.2 Getting started

A step-by-step guide to writing your first content integration script based on the Ricecooker framework including sample code.

Chapter 11

Intermediate topics

Once you've completed the *Getting started* steps, you can read the following pages to take your `ricecooker` game to the next level.

11.1 Ricecooker API reference

Discover the kinds of content nodes like HTML5App, video, audio, document, and exercises nodes you can use to build Kolibri channels.

11.2 HTML5 Apps

Learn how to scrape websites, parse HTML, and debug the HTML5 App nodes.

11.3 Concepts and workflow

Understand the main concepts, terminology, and workflows for creating Kolibri channels, as well as the review process.

Chapter 12

Advanced topics

Developing your Python automation superpowers, requires mastering lots of different techniques specific for different kinds of content sources. Use the links below to jump to the specific topics that you want to learn about.

12.1 Command line interface

Use command line options like `--thumbnails` (auto-generating thumbnails), and `--compress` (video compression) to create better channels.

12.2 Utility functions

Learn what tools are available for: downloading, creating and debugging HTMLZip files, video compression, splitting PDFs into chapters, and the CSV metadata workflow.

12.3 Developer docs

Explanations about how the `ricecooker` works in full details, including the computed ids, the upload process, and bulk corrections scripts.