ricecooker Documentation

Release 0.6.29

Learning Equality

Apr 16, 2019

Contents

CHAPTER 1

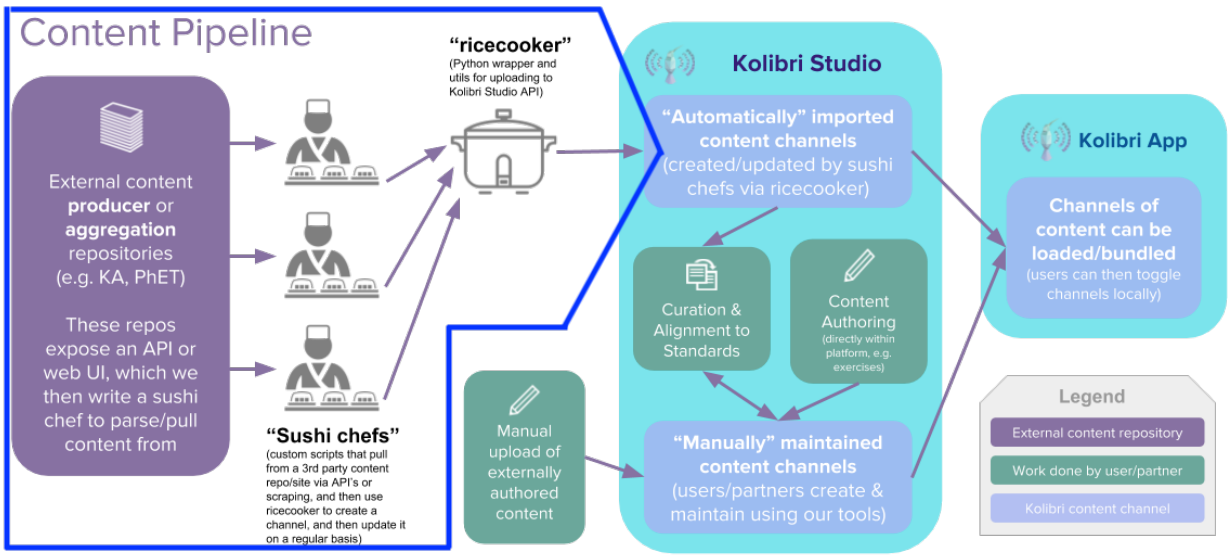
Quickstart

Developers who are new to the `ricecooker` library can get started here.

1.1 ricecooker

The `ricecooker` library is a framework for creating Kolibri content channels and uploading them to Kolibri Studio, which is the central content server that Kolibri applications talk to when they import content.

The Kolibri content pipeline is pictured below:



This `ricecooker` framework is the "main actor" in the first part of the content pipeline, and touches all aspects of the pipeline within the region highlighted in blue in the above diagram.

Before we continue, let's have some definitions:

- A **Kolibri channel** is a tree-like data structure that consist of the following content nodes:
  - Topic nodes (folders)
  - Content types:
    - Document (ePub and PDF files)
    - Audio (mp3 files)
    - Video (mp4 files)
    - HTML5App zip files (generic container for web content: HTML+JS+CSS)
    - Exercises
- A **sushi chef** is a Python script that uses the `ricecooker` library to import content from various sources, organize content into Kolibri channels and upload the channel to Kolibri Studio.

1.1.1 Overview

Use the following shortcuts to jump to the most relevant parts of the `ricecooker` documentation depending on your role:

- **Content specialists and Administrators** can read the non-technical part of the documentation to learn about how content works in the Kolibri platform.
  - The best place to start is the *Kolibri Platform overview*
  - Read more about the supported *content types here*
  - Content curators can consult this document for information about how to prepare "spec sheets" that guide developers how to import content into the Kolibri ecosystem.
  - The Non-technical of particular interest is the *CSV workflow* – channel metadata as spreadsheets
- **Chef authors** can read the remainder of this README, and get started using the `ricecooker` library by following these first steps:
  - *Quickstart*, which will introduce you to the steps needed to create a sushi chef script.
  - After the quickstart, you should be ready to take things into your own hands, and complete all steps in the ricecooker tutorial.
  - The next step after that is to read the *ricecooker usage docs*, which is also available in Jupyter notebooks under *docs/tutorial/*. More detailed technical documentation is available on the following topics:
  - *Installation*
  - *Content Nodes*
  - *File types*
  - *Exercises*
  - *HTML5 apps*
  - *Parsing HTML*
  - *Running chef scripts* to learn about the command line args, for controlling chef operation, managing caches, and other options.
  - Sushi chef style guide

- **Ricecooker developers** should read all the documentation for chef authors, and also consult the docs in the *developer* folder for additional information info about the "behind the scenes" work needed to support the Kolibri content pipeline:
  - *Running chef scripts*, also known as **chefops**.
  - *Running chef scripts in daemon mode*
  - *Managing the content pipeline*, also known as **sushops**.
  - *Command line interface*,
  - *Notes for ricecooker library developers*.

1.1.2 Installation

We'll assume you have a Python 3 installation on your computer and are familiar with best practices for working with Python codes (e.g. `virtualenv` or `pipenv`). If this is not the case, you can consult the Kolibri developer docs as a guide for setting up a Python virtualenv.

The `ricecooker` library is a standard Python library distributed through PyPI:

- Run `pip install ricecooker` to install You can then use `import ricecooker` in your chef script.
- Some of functions in `ricecooker.utils` require additional software:
  - Make sure you install the command line tool ffmpeg
  - Running javascript code while scraping webpages requires the phantomJS browser. You can run `npm install phantomjs-prebuilt` in your chef's working directory.

For more details and install options, see *the installation guide*.

1.1.3 Simple chef example

This is a sushi chef script that uses the `ricecooker` library to create a Kolibri channel with a single topic node (Folder), and puts a single PDF content node inside that folder.

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


class SimpleChef(SushiChef):
    channel_info = {
        'CHANNEL_TITLE': 'Potatoes info channel',
        'CHANNEL_SOURCE_DOMAIN': '<domain.org>',         # where you got the content␣
↪(change me!!)
        'CHANNEL_SOURCE_ID': '<unique id for channel>',  # channel's unique id␣
↪(change me!!)
        'CHANNEL_LANGUAGE': 'en',                        # le_utils language code
        'CHANNEL_THUMBNAIL': 'https://upload.wikimedia.org/wikipedia/commons/b/b7/A_
↪Grande_Batata.jpg', # (optional)
        'CHANNEL_DESCRIPTION': 'What is this channel about?',      # (optional)
    }

    def construct_channel(self, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        channel = self.get_channel(**kwargs)
        potato_topic = TopicNode(title="Potatoes!", source_id="<potatos_id>")
        channel.add_child(potato_topic)
        doc_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(doc_node)
        return channel


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

Let's assume the above code snippet is saved as the file `simple_chef.py`.

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

```
python simple_chef.py -v --reset --token=YOURTOKENHERE9139139f3a23232
```

The most important argument when running a chef script is `--token` which is used to pass in the Studio Access Token which you can obtain from your profile's settings page.

The flags `-v` (verbose) and `--reset` are generally useful in development. These make sure the chef script will start the process from scratch and displays useful debugging information on the command line.

To see all the `ricecooker` command line options, run `python simple_chef.py -h`. For more details about running chef scripts see *the chefops page*.

If you get an error when running the chef, make sure you've replaced `YOURTOKENHERE9139139f3a23232` by the token you obtained from Studio. Also make sure you've changed the value of `channel_info['CHANNEL_SOURCE_DOMAIN']` and `channel_info['CHANNEL_SOURCE_ID']` instead of using the default values.

1.1.4 Next steps

- See the *usage docs* <*usage*> for more explanations about the above code.
- See *nodes* <*nodes*> to learn how to create different content node types.
- See *file* <*files*> to learn about the file types supported, and how to create them.

1.1.5 Further reading

- Read the Kolibri Studio docs to learn more about the Kolibri Studio features

- Read the Kolibri user guide to learn how to install Kolibri on your machine (useful for testing channels)
- Read the Kolibri developer docs to learn about the inner workings of Kolibri.

1.2 Tutorial

1.2.1 Tutorials

This folder contains interactive tutorials that demonstrate how to build sushi chefs.

- quickstart : basic steps for creating a channel from the README
- *languages*: examples of how to work with the internal language codes
- *exercises*: examples of exercise questions types supported by ricecooker
- nodes [TODO]
- files [TODO]

Install

```
pip install jupyter
```

then run

```
jupyter notebook
```

and go to the directory `docs/tutorial/` to see the notebooks

1.2.2 The `ricecooker` quick start

This mini-tutorial will walk you through the steps of running a simple chef script `SimpleChef` that uses the `ricecooker` framework to upload a content channel to the Kolibri Studio server.

We'll go over the same steps as described in the usage, but this time showing the expected output of each step.

Running the notebooks

To follow along and run the code in this notebook, you'll need to clone the `ricecooker` repository, crate a virtual environement, install `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.

Step 1: Obtain a Studio Authorization Token

You will need a‘ Studio Authorization Token to create a channel on Kolibri Studio. In order to obtain such a token: 1. Create an account on Kolibri Studio. 2. Navigate to the Tokens tab under your Settings page. 3. Copy the given authorization token to a safe place.

You must pass the token on the command line as `--token=<your-auth-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/file.txt"`.

Step 2: Creating a Sushi Chef class

We'll use following simple chef script as an the running example in this section. You can find the full source code of it here.

Mmmm, potato... potato give you power!

```
from ricecooker.chefs import SushiChef
from ricecooker.classes.nodes import ChannelNode, TopicNode, DocumentNode
from ricecooker.classes.files import DocumentFile
from ricecooker.classes.licenses import get_license


class SimpleChef(SushiChef):
    channel_info = {
        'CHANNEL_TITLE': 'Potatoes info channel',
        'CHANNEL_SOURCE_DOMAIN': '<yourdomain.org>',     # where you got the content
        'CHANNEL_SOURCE_ID': '<unique id for channel>',  # channel's unique id
        'CHANNEL_LANGUAGE': 'en',                        # le_utils language code
        'CHANNEL_THUMBNAIL': 'https://upload.wikimedia.org/wikipedia/commons/b/b7/A_
↪Grande_Batata.jpg', # (optional)
        '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="<potatos_id>")
        channel.add_child(potato_topic)
        doc_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(doc_node)
        return channel
```

Note: make sure you change the values of `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID` before you try running this script. The combination of these two values is used to compute the `channel_id` for the Kolibri channel you're creating. If you keep the lines above unchanged, you'll get an error because the channel with source domain 'gov.mb.ca' and source id 'website_docs' already exists on Kolibri Studio.

Run of you chef by creating an instance of the chef class and calling it's `run` method:

```
mychef = SimpleChef()
args = {'token': 'YOURTOKENHERE9139139f3a23232',
        'reset': True,
        'verbose': True,
        'publish': True,
        'nomonitor': True}
options = {}
mychef.run(args, options)
```

```
Logged in with username you@yourdomain.org
Ricecooker v0.6.15 is up-to-date.
```

```
Running get_channel...


* 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...
    Downloading https://www.gov.mb.ca/inr/pdf/pubs/mafri-potatoe.pdf
    --- Downloaded 3641693a88b37e8d0484c340a83f9364.pdf
    Downloading https://upload.wikimedia.org/wikipedia/commons/b/b7/A_Grande_
Batata.jpg
    --- Downloaded 290c80ed7ce4cf117772f29dda76413c.jpg
   All files were successfully downloaded
Getting file diff...

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
    Preparing fields...
(0 of 2 uploaded)    Processing Potatoes info channel (ChannelNode)
(1 of 2 uploaded)       Processing Potatoes! (TopicNode)
   All nodes were created successfully.
Upload time: 39.441051s
Publishing channel...

Publishing tree to Kolibri...


DONE: Channel created at https://contentworkshop.learningequality.org/
channels/47147660ecb850bfb71590bf7d1ca971/edit
```

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

Creating more nodes

Now that you have a working example of a simple chef you can extend it by adding more content types. - Complete the ricecooker hands-on tutorial: https://gist.github.com/jayoshih/6678546d2a2fa3e7f04fc9090d81aff6 - usage docs for more explanations about the above code. - See to learn how to create different content node types. - See files to learn about the file types supported, and how to create them.

1.2.3 Languages

This tutorial will explain how to set the `language` property for various nodes and file objects when using the `ricecooker` framework.

Explore language objects and language codes

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`.

```
from le_utils.constants import languages as languages


# can lookup language using language code
language_obj = languages.getlang('en')
language_obj
```

```
Language(native_name='English', primary_code='en', subcode=None, name='English', ka_
↪name=None)
```

```
# can lookup language using language name (the new le_utils version has not shipped␣
↪yet)
language_obj = languages.getlang_by_name('English')
language_obj
```

```
Language(native_name='English', primary_code='en', subcode=None, name='English', ka_
↪name=None)
```

```
# all `language` attributed (channel, nodes, and files) need to use language code
language_obj.code
```

```
'en'
```

```
from le_utils.constants.languages import getlang_by_native_name

lang_obj = getlang_by_native_name('français')
print(lang_obj)
print(lang_obj.code)
```

```
Language(native_name='Français, langue française', primary_code='fr', subcode=None,␣
↪name='French', ka_name='francais')
fr
```

The above language code is an internal representaiton that uses two-letter codes, and sometimes has a 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.

Create chef class

We now create subclass of `ricecooker.chefs.SushiChef` and defined its `get_channel` and `construct_channel` methods.

For the purpose of this example, we'll create three topic nodes in different languages that contain one document in each.

```
from ricecooker.chefs import SushiChef
from ricecooker.classes.nodes import ChannelNode, TopicNode, DocumentNode
from ricecooker.classes.files import DocumentFile
from le_utils.constants import languages
from le_utils.constants import licenses


class MySushiChef(SushiChef):
    """
    A sushi chef that creates a channel with content in EN, FR, and SP.
    """
    def get_channel(self, **kwargs):
        channel = ChannelNode(
            source_domain='testing.org',
            source_id='lang_test_chanl',
            title='Languages test channel',
            thumbnail='http://themes.mysitemyway.com/_shared/images/flags.png',
            language = languages.getlang('en').code    # set global language for␣
↪channel (will apply as default option to all content items in this channel)
        )
        return channel

    def construct_channel(self, **kwargs):
        # create channel
        channel = self.get_channel(**kwargs)

        # create the English topic, add a DocumentNode to it
        topic = TopicNode(
            source_id="<en_topic_id>",
            title="New Topic in English",
            language=languages.getlang('en').code,
        )
        doc_node = DocumentNode(
            source_id="<en_doc_id>",
            title='Some doc in English',
            description='This is a sample document node in English',
            files=[DocumentFile(path='samplefiles/documents/doc_EN.pdf')],
            license=licenses.PUBLIC_DOMAIN,
            language=languages.getlang('en').code,
        )
        topic.add_child(doc_node)
        channel.add_child(topic)

        # create the Spanish topic, add a DocumentNode to it
        topic = TopicNode(
            source_id="<es_topic_id>",
            title="Topic in Spanish",
            language=languages.getlang('es-MX').code,
        )
        doc_node = DocumentNode(
            source_id="<es_doc_id>",
            title='Some doc in Spanish',
            description='This is a sample document node in Spanish',
            files=[DocumentFile(path='samplefiles/documents/doc_ES.pdf')],
            license=licenses.PUBLIC_DOMAIN,
            language=languages.getlang('es-MX').code,
        )
```

(continues on next page)

(continued from previous page)

```
        topic.add_child(doc_node)
        channel.add_child(topic)

        # create the French topic, add a DocumentNode to it
        topic = TopicNode(
            source_id="<fr_topic_id>",
            title="Topic in French",
            language=languages.getlang('fr').code,
        )
        doc_node = DocumentNode(
            source_id="<fr_doc_id>",
            title='Some doc in French',
            description='This is a sample document node in French',
            files=[DocumentFile(path='samplefiles/documents/doc_FR.pdf')],
            license=licenses.PUBLIC_DOMAIN,
            language=languages.getlang('fr').code,
        )
        topic.add_child(doc_node)
        channel.add_child(topic)

        return channel
```

Run of you chef by creating an instance of the chef class and calling it's `run` method:

```
mychef = MySushiChef()
args = {'token': 'YOURTOKENHERE9139139f3a23232',
        'reset': True,
        'verbose': True,
        'publish': True}
options = {}
mychef.run(args, options)
```

```
Logged in with username you@yourdomain.org
Ricecooker v0.6.19 is up-to-date.
Running get_channel...
run_id: 27a7726c4b2b418fb0f7b1842f6abe84


* Starting channel build process *


Calling construct_channel...
   Setting up initial channel structure...
   Validating channel structure...
      Languages test channel (ChannelNode): 6 descendants
         New Topic in English (TopicNode): 1 descendant
            Some doc in English (DocumentNode): 1 file
         Topic in Spanish (TopicNode): 1 descendant
            Some doc in Spanish (DocumentNode): 1 file
         Topic in French (TopicNode): 1 descendant
            Some doc in French (DocumentNode): 1 file
   Tree is valid

Downloading files...
Processing content...
```

```
    --- Downloaded e8b1fe37ce3da500241b4af4e018a2d7.pdf
    --- Downloaded cef22cce0e1d3ba08861fc97476b8ccf.pdf
    --- Downloaded 6c8730e3e2554e6eac0ad79304bbcc68.pdf
    --- Downloaded de498249b8d4395a4ef9db17ec02dc91.png
   All files were successfully downloaded
Getting file diff...

Checking if files exist on Kolibri Studio...
    Got file diff for 4 out of 4 files
Uploading files...

Uploading 0 new file(s) to Kolibri Studio...
Creating channel...

Creating tree on Kolibri Studio...
   Creating channel Languages test channel
    Preparing fields...
(0 of 6 uploaded)    Processing Languages test channel (ChannelNode)
(3 of 6 uploaded)       Processing New Topic in English (TopicNode)
(4 of 6 uploaded)       Processing Topic in Spanish (TopicNode)
(5 of 6 uploaded)       Processing Topic in French (TopicNode)
   All nodes were created successfully.
Upload time: 6.641212s
Publishing channel...

Publishing tree to Kolibri...


DONE: Channel created at https://contentworkshop.learningequality.org/
channels/cba91822d3ab5a748cd19532661d690f/edit
```

Congratulations, you put three languages on the internet!

Example 2: YouTube video with subtitles in multiple languages

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.

```
import youtube_dl

ydl = youtube_dl.YoutubeDL({
    'quiet': True,
    'no_warnings': True,
    'writesubtitles': True,
    'allsubtitles': True,
})


youtube_id =  'FN12ty5ztAs'

info = ydl.extract_info(youtube_id, download=False)
subtitle_languages = info["subtitles"].keys()

print(subtitle_languages)
```

```
dict_keys(['en', 'fr', 'zu'])
```

Full sushi chef example

The `YoutubeVideoWithSubtitlesSushiChef` class below shows how to create a channel with youtube video and upload subtitles files with all available languages.

```
from ricecooker.chefs import SushiChef
from ricecooker.classes import licenses
from ricecooker.classes.nodes import ChannelNode, TopicNode, VideoNode
from ricecooker.classes.files import YouTubeVideoFile, YouTubeSubtitleFile
from ricecooker.classes.files import is_youtube_subtitle_file_supported_language


import youtube_dl
ydl = youtube_dl.YoutubeDL({
    'quiet': True,
    'no_warnings': True,
    'writesubtitles': True,
    'allsubtitles': True,
})


# Define the license object with necessary info
TE_LICENSE = licenses.SpecialPermissionsLicense(
    description='Permission granted by Touchable Earth to distribute through Kolibri.
↪',
    copyright_holder='Touchable Earth Foundation (New Zealand)'
)


class YoutubeVideoWithSubtitlesSushiChef(SushiChef):
    """
    A sushi chef that creates a channel with content in EN, FR, and SP.
    """
    channel_info = {
        'CHANNEL_SOURCE_DOMAIN': 'learningequality.org',        # change me!
        'CHANNEL_SOURCE_ID': 'sample_youtube_video_with_subs',  # change me!
        'CHANNEL_TITLE': 'Youtube subtitles downloading chef',
        'CHANNEL_LANGUAGE': 'en',
        'CHANNEL_THUMBNAIL': 'http://themes.mysitemyway.com/_shared/images/flags.png',
        'CHANNEL_DESCRIPTION': 'This is a test channel to make sure youtube subtitle␣
↪languages lookup works'
    }

    def construct_channel(self, **kwargs):
        # create channel
        channel = self.get_channel(**kwargs)

        # get all subtitles available for a sample video
        youtube_id =  'FN12ty5ztAs'
        info = ydl.extract_info(youtube_id, download=False)
        subtitle_languages = info["subtitles"].keys()
        print('Found subtitle_languages = ', subtitle_languages)
```

(continues on next page)

(continued from previous page)

```
        # create video node
        video_node = VideoNode(
            source_id=youtube_id,
            title='Youtube video',
            license=TE_LICENSE,
            derive_thumbnail=True,
            files=[YouTubeVideoFile(youtube_id=youtube_id)],
        )

        # add subtitles in whichever languages are available.
        for lang_code in subtitle_languages:
            if is_youtube_subtitle_file_supported_language(lang_code):
                video_node.add_file(
                    YouTubeSubtitleFile(
                        youtube_id=youtube_id,
                        language=lang_code
                    )
                )
            else:
                print('Unsupported subtitle language code:', lang_code)

        channel.add_child(video_node)

        return channel
```

```
chef = YoutubeVideoWithSubtitlesSushiChef()
args = {'token': 'YOURTOKENHERE9139139f3a23232',
        'reset': True,
        'verbose': True,
        'publish': True}
options = {}
chef.run(args, options)
```

```
Logged in with username you@yourdomain.org
Ricecooker v0.6.19 is up-to-date.
Running get_channel...
run_id: 682e56ae42c246eb8c307bae35122e9e


* Starting channel build process *


Calling construct_channel...
```

```
Found subtitle_languages =  dict_keys(['en', 'fr', 'zu'])
```

```
   Setting up initial channel structure...
   Validating channel structure...
      Youtube subtitles downloading chef (ChannelNode): 1 descendant
         Youtube video (VideoNode): 4 files
   Tree is valid

Downloading files...
Processing content...
    --- Downloaded (YouTube) 987257c13adb6d2f2c86849be6031a4c.mp4
```

(continues on next page)

(continued from previous page)

```
    --- Downloaded subtitle f589321457f81efd035bb72cb57a1b3b.vtt
    --- Downloaded subtitle 99d24a5240d64e505a6343f50f851d2e.vtt
    --- Downloaded subtitle a1477da82f45e776b7f889b67358e761.vtt
    --- Extracted thumbnail 2646f5028c7925c0d304c709d39cf5b0.png
    --- Downloaded de498249b8d4395a4ef9db17ec02dc91.png
   All files were successfully downloaded
Getting file diff...

Checking if files exist on Kolibri Studio...
    Got file diff for 6 out of 6 files
Uploading files...

Uploading 0 new file(s) to Kolibri Studio...
```

1.2.4 `ricecooker` exercises

This mini-tutorial will walk you through the steps of running a simple chef script `ExercisesChef` that creates two exercises nodes, and four exercises questions.

We'll go over the same steps as described in the Exercises section of the page nodes, but this time showing the expected output of each step.

Running the notebooks

To follow along and run the code in this notebook, you'll need to clone the `ricecooker` repository, crate a virtual environement, install `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.

Creating a Sushi Chef class

```
from ricecooker.chefs import SushiChef
from ricecooker.classes.nodes import TopicNode, ExerciseNode
from ricecooker.classes.questions import SingleSelectQuestion, MultipleSelectQuestion,
↪ InputQuestion, PerseusQuestion
from ricecooker.classes.licenses import get_license
from le_utils.constants import licenses
from le_utils.constants import exercises
from le_utils.constants.languages import getlang


class SimpleChef(SushiChef):
    channel_info = {
        'CHANNEL_TITLE': 'Sample Exercises',
        'CHANNEL_SOURCE_DOMAIN': '<yourdomain.org>',    # where you got the content
        'CHANNEL_SOURCE_ID': '<unique id for channel>',  # channel's unique id
        'CHANNEL_LANGUAGE': 'en',                        # le_utils language code
        'CHANNEL_DESCRIPTION': 'A test channel with different types of exercise␣
↪questions',      # (optional)
        'CHANNEL_THUMBNAIL': None, # (optional)
    }
```

(continues on next page)

(continued from previous page)

```
    def construct_channel(self, **kwargs):
        channel = self.get_channel(**kwargs)
        topic = TopicNode(title="Math Exercises", source_id="folder-id")
        channel.add_child(topic)

        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',
                    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.'],
                )
            ]
        )
        topic.add_child(exercise_node)

        # LOAD JSON DATA (as string) FOR PERSEUS QUESTIONS
        RAW_PERSEUS_JSON_STR = open('../../examples/data/perseus_graph_question.json',
↪ 'r').read()
        # or
        # import requests
        # RAW_PERSEUS_JSON_STR = requests.get('https://raw.githubusercontent.com/
↪learningequality/sample-channels/master/contentnodes/exercise/perseus_graph_
↪question.json').text
        exercise_node2 = ExerciseNode(
```

(continues on next page)

(continued from previous page)

```
                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={
                    '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'
                    ),
                ]
        )
        topic.add_child(exercise_node2)

        return channel
```

Note: make sure you change the values of `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID` before you try running this script. The combination of these two values is used to compute the `channel_id` for the Kolibri channel you're creating. If you keep the lines above unchanged, you'll get an error because the channel with source domain 'gov.mb.ca' and source id 'website_docs' already exists on Kolibri Studio.

Run of you chef by creating an instance of the chef class and calling it's `run` method:

```
mychef = SimpleChef()
args = {'token': '70aec3d11849e6691a8806d17f05b18bc5ca5ed4',
        'reset': True,
        'verbose': True,
        'publish': True,
        'nomonitor': True}
options = {}
mychef.run(args, options)
```

```
Logged in with username ivan.savov@gmail.com
Ricecooker v0.6.15 is up-to-date.
Running get_channel...


* Starting channel build process *


Calling construct_channel...
   Setting up initial channel structure...
   Validating channel structure...
      Sample Exercises (ChannelNode): 3 descendants
         Math Exercises (TopicNode): 2 descendants
            Basic questions (ExerciseNode): 3 questions
```

```
            An exercise containing a perseus question (ExerciseNode): 1␣
↪question
   Tree is valid

Downloading files...
Processing content...
    * Processing images for exercise: Basic questions
    * Images for Basic questions have been processed
    * Processing images for exercise: An exercise containing a perseus␣
↪question
    * Images for An exercise containing a perseus question have been processed
   All files were successfully downloaded
Getting file diff...

Checking if files exist on Kolibri Studio...
Uploading files...

Uploading 0 new file(s) to Kolibri Studio...
Creating channel...

Creating tree on Kolibri Studio...
   Creating channel Sample Exercises
    Preparing fields...
(0 of 3 uploaded)    Processing Sample Exercises (ChannelNode)
(1 of 3 uploaded)       Processing Math Exercises (TopicNode)
   All nodes were created successfully.
Upload time: 36.425115s
Publishing channel...

Publishing tree to Kolibri...


DONE: Channel created at https://contentworkshop.learningequality.org/
channels/47147660ecb850bfb71590bf7d1ca971/edit
```

Congratulations, you put some math exercises on the internet!

CHAPTER 2

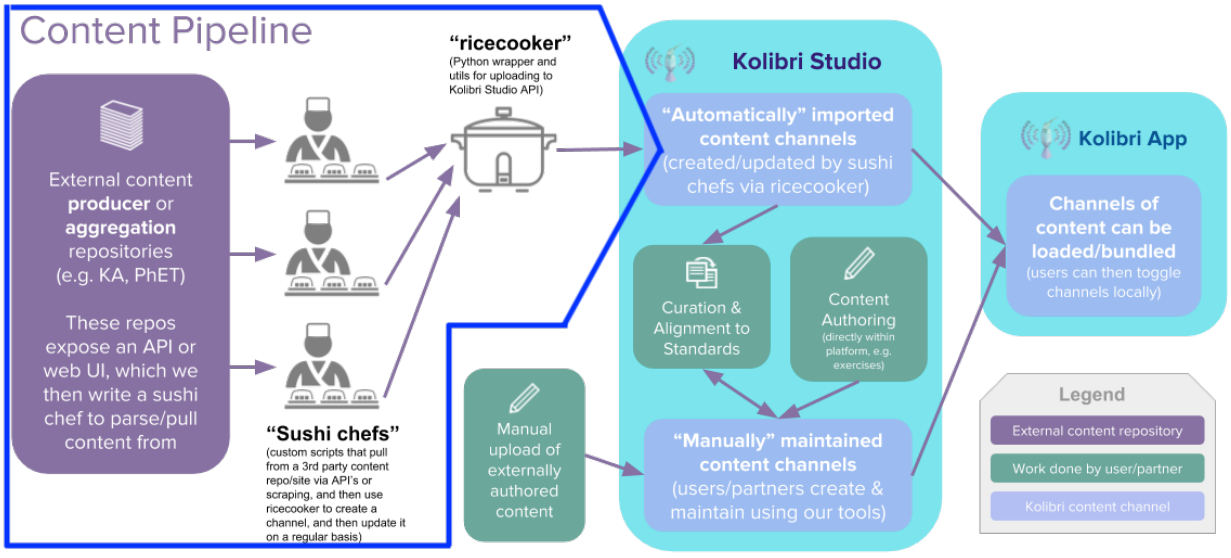
Kolibri content platform

The Kolibri content platform is described in the following docs, which should be accessible to both technical and non-technical audiences.

2.1 Kolibri content platform

Educational content in the Kolibri platform is organized into **content channels**. The `ricecooker` library 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.

The Kolibri content pipeline is pictured below:



The Kolibri Content Pipeline

This `ricecooker` framework is the "main actor" in the first part of the content pipeline, and touches all aspects of the pipeline within the region highlighted in blue in the above diagram.

2.1.1 Supported Content types

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

- Topic nodes (folders)
- Content types:
  - Document (ePub and PDF files)
  - Audio (mp3 files)
  - Video (mp4 files)
  - HTML5App zip files (generic container for web content: HTML+JS+CSS)
  - Exercises, which contain different types of questions:
    - SingleSelectQuestion (multiple choice)
    - MultipleSelectQuestion (multiple choice with multiple correct answers)
    - InputQuestion (good for numeric inputs)
    - PerseusQuestion (a rich exercise question format developed at Khan Academy)

You can learn more about the content types supported by the Kolibri ecosystem here.

2.1.2 Content import workflows

The following options are available for importing content into Kolibri Studio.

Kolibri Studio web interface

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 (50+ content items) or channels that need will be updated regularly, you should consider using one of the bulk-import options below.

Bulk-importing content programatically

The `ricecooker` library 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 (usually a shared google doc), which provides detailed instructions for how content should be structured and organized within the channel.

Consult this document for more info about writing spec sheets.

CSV metadata workflow

In addition to the web interface and the Python interface (`ricecooker`), there exists a third 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

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.

2.1.3 Further reading

- Kolibri Studio User Guide
- Sample channels

2.2 Supported content types

2.2.1 Audio

The `AudioNode` and `AudioFile` are used to store mp3 files.

2.2.2 Videos

The `VideoNode` and `VideoFile` are used to store videos.

2.2.3 Documents

The `DocumentNode` class supports two type of files:

- Use the `DocumentFile` for `.pdf` documents
- Use the `EPubFile` for `.epub` files

2.2.4 HTML5Apps

The most versatile and extensible option for importing content into Kolibri is to package the content as HTML5App nodes. The HTML5 content type on Kolibri, consists of a zip file with web content inside it. The Kolibri application serves the file `index.html` from the root of the zip folder inside an iframe. It is possible to package any web content in this manner: text, images, CSS, fonts, and JavaScript code. The `iframe` rendering the content in Kolibri is sandbox so no plugins are allowed (no swf/flash). In addition, it is expected that oh web resources are stored within the zip file, and referenced using relative paths. This is what enables Kolibri to used in offline settings.

Here are some samples:

- Sample Vue.js App: Proof of concept of minimal webapp based on the vue.js framework. Note the shell script tweaks the output to make references relative paths.
- Sample React App: Proof of concept of minimal webapp based on the React framework. Note the shell script tweaks required to make paths relative.

2.2.5 Exercises

Kolibri exercises are based on the `perseus` exercise framework developed by Khan Academy. Perseus provides a free-form interface for questions based on various "widgets" buttons, draggables, expressions, etc. This is the native format for exercises on Kolibri. An exercise question item is represented as a giant json file, with the main question field stored as Markdown. Widgets are included in the "main" through a unique-Unicode character and then widget metadata is stored separately as part of the json data.

Exercises can be created programmatically or interactively using the perseus editor through the web: http://khan.github.io/perseus/ (try adding different widgets in the Question area and then click the JSON Mode checkbox to "view source" for the exercise.

You can then copy-paste the results as a .json file and import into Kolibri using ricecooker library (Python).

Sample: https://github.com/learningequality/sample-channels/blob/master/contentnodes/exercise/sample_perseus04.json

Kolibri Studio provides helper classes for creating single/multiple-select questions, and numeric input questions: https://github.com/learningequality/ricecooker/blob/master/docs/exercises.md

A simple multiple choice (single select) question can be created as follows:

```
SingleSelectQuestion(
    question = "What was the main idea in the passage you just read?",
    correct_answer = "The right answer",
    all_answers = ["The right answer", "Another option", "Nope, not this"]
    ...
```

Exercise activities allow student answers to be logged and enable progress reports for teachers and coaches. Exercises can also be used as part of individual assignments (playlist-like thing with a mix of content and exercises), group assignments, and exams.

2.2.6 Extending Kolibri

New content types and presentation modalities will become available and supported natively by future versions of Kolibri. The Kolibri software architecture is based around the plug-in system that is easy to extend. All currently supported content type renderers are based on this plug-in architecture. It might be possible to create a Kolibri plugin for rendering specific content in custom ways.

CHAPTER 3

Ricecooker API reference

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

3.1 Using the `ricecooker` library

The `ricecooker` library is used to transform various educational content types into Kolibri-compatible formats and upload content to Kolibri Studio. The following steps will guide you through the creation of a sushi chef script that uses all the features of the `ricecooker` library.

3.1.1 Step 1: Obtain a Studio Authorization Token

You will need a Studio Authorization Token to create a channel on Kolibri Studio. In order to obtain such a token:

1. Create an account on Kolibri Studio.
2. Navigate to the Tokens tab under your Settings page.
3. Copy the given authorization token to a safe place.

You must pass the token on the command line as `--token=<your-auth-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/file.txt"`.

3.1.2 Step 2: Create a Sushi Chef script

We'll use following simple chef script as an the running example in this section. You can copy-paste this code into a file `mychef.py` and use it as a starting point for the chef script you're working on.

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

(continues on next page)

(continued from previous page)

```
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="les_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)
                    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 -v --reset --token=YOURTOKENHERE9139139f3a23232
    """
    simple_chef = SimpleChef()
    simple_chef.main()                                                          #␣
↪(13)
```

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 the following methods:

- `run`, which performs all the work of uploading your channel to the Kolibri Studio. A sushi chef run consists of multiple steps, the most important one being when the we call the chef class' `construct_channel` method.
- `main`, which your is the function that runs when the sushi chef script is called on the command line.

Chef class attributes

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>',       # who is providing the␣
↪content (e.g. learningequality.org)
    'CHANNEL_SOURCE_ID': '<some unique identifier>',     # an unique identifier for␣
↪this channel within the domain
    '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
    'CHANNEL_DESCRIPTION': 'What is this channel about?',      # (optional) longer␣
↪description of the channel
}
```

Note: make sure you change the values of `CHANNEL_SOURCE_DOMAIN` and `CHANNEL_SOURCE_ID` before you try running this script. The combination of these two values is used to compute the `channel_id` for the Kolibri channel you're creating. If you keep the lines above unchanged, you'll get an error because the channel with source domain 'gov.mb.ca' and source id 'website_docs' already exists on Kolibri Studio.

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`.

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.

Content nodes

The `ricecooker` library provides classes like `DocumentNode`, `VideoNode`, `AudioNode`, etc., to store the metadata associate with media 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 on the source domain `gov.mb.ca`. 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.

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 a web URL as in the above example, or a local filesystem path.

Command line interface

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

```
python mychef.py -v --reset --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.

The flags `-v` (verbose) and `--reset` are generally useful in development. These make sure the chef script will start the process from scratch and displays useful debugging information on the command line.

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

If you get an error when running the chef, make sure you've replaced `YOURTOKENHERE9139139f3a23232` by the token you obtained from Studio. Also make sure you've changed the value of `channel_info['CHANNEL_SOURCE_DOMAIN']` and `channel_info['CHANNEL_SOURCE_ID']` instead of using the default values.

If the channel run was successful, you should be able to see your single-topic channel on Kolibri Studio server. The topic node "Potatoes!" is nice to look at, but it feels kind of empty. Let's add more nodes to it!

3.1.3 Step 3: Add more content nodes and files

Once your channel is created, you can start adding nodes. To do this, you need to convert your data to `ricecooker` objects. Here are the classes that are available to you (import from `ricecooker.classes.nodes`):

- **TopicNode**: folders to organize to the channel's content
- **AudioNode**: content containing mp3 file
- **DocumentNode**: content containing pdf and epub files
- **HTML5AppNode**: content containing zip of html files (html, js, css, etc.)
- **VideoNode**: content containing mp4 file
- **ExerciseNode**: assessment-based content with questions

Once you have created the node, add it to a parent node with `parent_node.add_child(child_node)`

To read more about the different nodes, read the *nodes page*.

To add a file to your node, you must start by creating a file object from `ricecooker.classes.files`. Your sushi chef is responsible for determining which file object to create. Here are the available file models:

- **AudioFile**: mp3 file
- **DocumentFile**: pdf file
- **EPubFile**: epub file
- **HTMLZipFile**: zip of html files (must have `index.html` file at topmost level)
- **VideoFile**: mp4 file (can be high resolution or low resolution)
- **WebVideoFile**: video downloaded from site such as YouTube or Vimeo
- **YouTubeVideoFile**: video downloaded from YouTube using a youtube video id
- **SubtitleFile**: .vtt subtitle files to be used with VideoFiles
- **YouTubeSubtitleFile**: subtitles downloaded based on youtube video id and language code
- **ThumbnailFile**: png or jpg thumbnail files to add to any kind of node

Each file class can be passed a **preset** and **language** at initialization (SubtitleFiles must have a language set at initialization). A preset determines what kind of file the object is (e.g. high resolution video vs. low resolution video). A list of available presets can be found at `le_utils.constants.format_presets`.

ThumbnailFiles, AudioFiles, DocumentFiles, HTMLZipFiles, VideoFiles, and SubtitleFiles must be initialized with a **path** (str). This path can be a url or a local path to a file.

To read more about the different nodes, read the *nodes files*.

3.1.4 Step 4: Adding exercises

See the *exercises page*.

3.2 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.

3.2.1 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)                               |
        class TopicNode(TreeNode)                      |
        class ContentNode(TreeNode)                    |
            class AudioNode(ContentNode)      files = [AudioFile]
            class DocumentNode(ContentNode)   files = [DocumentFile, EPubFile]
            class HTML5AppNode(ContentNode)   files = [HTMLZipFile]
            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*.

3.2.2 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)
- **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`.

Usability guidelines

- Thumbnails: 16:9 aspect ratio ideally (e.g. 420x236 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.

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.

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.

Thumbnails

Thumbnails can be passed in as a local filesystem path to an image file (str) or a `ThumbnailFile` object. The recommended size for thumbnail images is 420px by 236px (aspect ratio 16:9).

3.2.3 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. The `--thumbnails` command line argument can be used to generate thumbnails for topic nodes based on the thumbnails of the content nodes they contain.

3.2.4 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
```

(continues on next page)

(continued from previous page)

```
  DocumentNode   --files-->  DocumentFile                                  # .pdf
                             EPubFile                                      # .epub
  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 file 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. VideoNodes also have a **derive_thumbnail** (boolean) argument, which will automatically extract a thumbnail from the video if no thumbnail is provided.

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

3.2.5 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',
                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>',
```

(continues on next page)

(continued from previous page)

```
        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={
            '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.

3.3 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.

3.3.1 File objects

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

```
AudioFile             # .mp3
DocumentFile          # .pdf
HTMLZipFile           # .zip containing HTML,JS,CSS
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`)
ThumbnailFile         # .png/.jpg/.jpeg  (`path` is local file system or url)
```

3.3.2 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.

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.

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.

3.3.3 Audio files

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

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

3.3.4 Document files

Use the `DocumentFile` class to add PDF documents:

```
document_file = DocumentFile(
    path='dir/subdir/lecture_slides.mp4',
    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
)
```

3.3.5 HTMLZip 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
)
```

3.3.6 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_width": 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,
)

video_file3 = YouTubeVideoFile(
```

(continues on next page)

(continued from previous page)

```
    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,
)
```

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.

3.3.7 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 420px by 236px (aspect ratio 16:9).

3.3.8 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 then 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 compressions 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_width':"'trunc(oh*a/2)*2:min(ih,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.

3.4 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).

3.4.1 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.

3.5 HTML5App nodes and HTML5Zip files

Kolibri supports rendering of generic HTML content through the use of `HTML5Apps` nodes, which correspond to `HTML5Zip` files. The Kolibri application serves the contents of `index.html` in the root of the zip file inside an iframe. All `href`s and img `src` attributes must be relative links to resources within the zip file.

3.5.1 Example of HTML5App nodes

- simple example
  - Note links are disabled (removed blue link) because A) external links are disable in iframe and B) because wouldn't have access offline
  - If link is to a PDF, IMG, or other useful resource than can be included in zip file then keep link but change to relative path
- medium complexity example
  - Download all parts of a multi-part lesson into a single HTML5Zip file
  - Original source didn't have a "table of contents" so added manually (really bad CSS I need to fix in final version)
- complex example
  - Full javascript application packaged as a zip file
  - Source: sushi-chef-phet

3.5.2 Usability guidelines

- There *must* be an index.html file at the topmost level of the zip file, otherwise no app will be shown
- 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 prev/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 potentially users could access URL by other means)
  - e.g., "some other text link text (http://link.url) and more text continues"

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

- As small as possible so that resources are individually trackable, assignable, and reusable in multiple places

- 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 single 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 we try to turn it into a Folder and put content items inside it. Nested folders is main way of representing structured content.

3.5.4 HTML Writer utility class

The class `HTMLWriter` in `ricecooker.utils.html_writer` provides a basic helper methods for creating files within a zip file.

See the source code: ricecooker/utils/html_writer.py

3.5.5 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: `ricecooker.utils.downloader.download_static_assets()`
- example usage in a simple app: MEET chef, which comprises articles with text and images
- example usage in a complex app: Blockly Games chef, an interactive JS game with images and sounds

3.5.6 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 guidelines layed out above – legible, responsive, easy to navigate, and "look good" (you define "good" :P). Fulfilling that, use your judgement on whatever approach makes sense and that you can use effectively!

3.6 Exercise and exercise questions

`ExerciseNodes` are special objects that have questions used for assessment.

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 your exercise, 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$`.
- **answers** ([{'answer':str, 'correct':bool}]): answers to question, also in 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 possible 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)`

3.7 Installation

The `ricecooker` library is published as a Python3-only package on PyPI.

3.7.1 Software prerequisites

The `ricecooker` library requires Python 3.5+ and some additional tools like `ffmpeg` for video compression, and `phantomjs` for scraping webpages that require JavaScript to run before the DOM is rendered.

On a Debian-like linux box, you can install all the necessary packages using:

```
apt-get install build-essential gettext pkg-config \
    python3 python3-pip python3-dev python3-virtualenv virtualenv python3-tk \
    linux-tools libfreetype6-dev libxft-dev libwebp-dev libjpeg-dev libmagickwand-dev␣
↪\
    ffmpeg phantomjs
```

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

```
brew install freetype imagemagick@6 ffmpeg phantomjs
brew link --force imagemagick@6
```

3.7.2 Stable release

To install `ricecooker`, run this command in your terminal:

```
pip install ricecooker
```

This is the preferred method to install `ricecooker`, as it will always install the most recent stable release.

If you don't have `pip` installed, then this Python installation guide will guide you through the process of setting up.

Note: We recommend you install `ricecooker` in a Python `virtualenv` specific for cheffing work, rather that globally for your system python. For information about creating and activating a virtualenv, you can follow the instructions provided here.

3.7.3 Install from github

You can install `ricecooker` directly from the github repo using the following command:

```
pip install git+https://github.com/learningequality/ricecooker
```

Occasionally, you'll want to install a `ricecooker` version from a specific branch, instead of the default branch version. This is the way to do this:

```
pip install -U git+https://github.com/learningequality/ricecooker@somebranchname
```

The `-U` flag forces the update instead of reusing any previously installed/cached versions.

3.7.4 Install from source

Another option for installing `ricecooker` is to clone the repo and install using:

```
git clone git://github.com/learningequality/ricecooker
cd ricecooker
pip install -e .
```

The flag `-e` installs `ricecooker` in "editable mode," which means you can now make changes to the source code and you'll see the changes reflected immediately. This installation method very useful if you're working around a bug in `ricecooker` or extending the crawling/scraping/http/html utilities in `ricecooker/utils/`.

Speaking of bugs, if you ever run into problems while using `ricecooker`, you should let us know by opening an issue.

3.8 Running chef scripts

The base class `SushiChef` provides a lot of command line arguments that control the chef script's operation. It is expected that **every chef script will come with a README** that explains the desired command line arguments for the chef script.

3.8.1 Executable scripts

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

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

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

- Make the file `chef.py` executable by running `chmod +x chef.py` on the
  command line.
```

You can now call your sushi chef script using `./chef.py ...` instead of the longer `python chef.py ...`.

3.8.2 Ricecooker CLI

You can run `./chef.py -h` to see an always-up-to-date info about the `ricecooker` CLI interface:

```
usage: tutorial_chef.py [-h] [--token TOKEN] [-u] [-v] [--quiet] [--warn]
                        [--debug] [--compress] [--thumbnails]
                        [--download-attempts DOWNLOAD_ATTEMPTS]
                        [--reset | --resume]
                        [--step {INIT, CONSTRUCT_CHANNEL, CREATE_TREE, DOWNLOAD_FILES,
↪ GET_FILE_DIFF,
                           START_UPLOAD, UPLOADING_FILES, UPLOAD_CHANNEL, PUBLISH_
↪CHANNEL,DONE, LAST}]
                        [--prompt] [--stage] [--publish] [--daemon]
                        [--nomonitor] [--cmdsock CMDSOCK]
```

(continues on next page)

(continued from previous page)

```
required arguments:
  --token TOKEN         Authorization token (can be token or path to file with token)

optional arguments:
  -h, --help            show this help message and exit
  -u, --update          Force re-download of files (skip .ricecookerfilecache/ check)
  -v, --verbose         Verbose mode
  --quiet               Print only errors to stderr
  --warn                Print warnings to stderr
  --debug               Print debugging log info to stderr
  --compress            Compress high resolution videos to low resolution
                        videos
  --thumbnails          Automatically generate thumbnails for topics
  --download-attempts DOWNLOAD_ATTEMPTS
                        Maximum number of times to retry downloading files
  --reset               Restart session, overwriting previous session (cannot
                        be used with --resume flag)
  --resume              Resume from ricecooker step (cannot be used with
                        --reset flag)
  --step  {INIT, ...    Step to resume progress from (must be used with --resume flag)
  --prompt              Prompt user to open the channel after creating it
  --stage               Upload to staging tree to allow for manual
                        verification before replacing main tree
  --publish             Publish newly uploaded version of the channel
  --daemon              Run chef in daemon mode
  --nomonitor           Disable SushiBar progress monitoring
  --cmdsock CMDSOCK     Local command socket (for cronjobs)

extra options:
  You can pass arbitrary key=value options on the command line
```

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 `./chef.py ... lang=fr` could:

- Subclass the method `get_channel` to set the channel name to `"Channel Name ({})".format(getlang('fr').native_name)`
- Use the language code `fr` in `pre_run`, `run`, and `construct_channel` to crawl and scrape the French version of the source website

Resuming interrupted chef runs

If your `ricecooker` session gets interrupted, you can resume from any step that has already completed using `--resume --step=<step>` option. If step is not specified, `ricecooker` will resume from the last step you ran. The "state" necessary to support these checkpoints is stored in the directory `restore` in the folder where the chef runs.

Use the `--reset` flag to skip the auto-resume prompt.

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.

3.8.3 Run scripts

For complicated chef scripts that run in multiple languages or with multiple options, the chef author can implement a "run script" that can be run as:

```
./run.sh
```

The script should contain the appropriate command args and options (basically the same thing as the instructions in the chef's README but runnable).

3.8.4 Daemon mode

Starting a chef script with the `--daemon` argument makes it listen for remote control commands from the sushibar host. For more information, see *Daemonization*.

CHAPTER 4

Ricecooker Utils

The `ricecooker` library includes a number of utilities and helper functions:

4.1 Parsing HTML using `BeautifulSoup`

Basic code to GET the HTML source of a webapge and parse it:

```
import requests
from bs4 import BeautifulSoup

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

Basic API uses `find` and `find_all`:

```
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...
```

4.1.1 Further reading

You can learn more about BeautifulSoup from these excellent tutorials:

- http://akul.me/blog/2016/beautifulsoup-cheatsheet/
- http://youkilljohnny.blogspot.ca/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/

4.2 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.

4.2.1 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.

4.3 CSV Exercises Workflow

In addition to content nodes (files) and topics (folders), we can also add 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

4.3.1 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`:

4.3.2 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 ""-separated string values, e.g., "Answer FAnswer GAnswer 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 “”-separated string values, e.g., “Hint 6 textHint 7 textHing 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)`.

4.3.3 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`.

4.3.4 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 desribe the metadata for the topic node containing the exercises.

4.4 Writing a SousChef

Kolibri is an open source educational platform to distribute content to areas with little or no internet connectivity. Educational content is created and edited on Kolibri Studio, which is a platform for organizing content to import from the Kolibri applications.

A *souchef* is a program that scrapes content from a source website source and puts the content into a format that can be imported into Kolibri Studio. This project will read a given source's content and parse and organize that content into a folder + csv structure, which will then be imported into Kolibri Studio.

4.4.1 Definitions

A `sous chef` script is responsible for scraping content from a source and putting it into a folder and CSV structure.

4.4.2 Installation

- Install Python 3 if you don't have it already.
- Install pip if you don't have it already.
- Create a Python virtual environment for this project (optional, but recommended):
  - Install the virtualenv package: `pip install virtualenv`
  - The next steps depends if you're using UNIX (Mac/Linux) or Windows:
    - For UNIX systems:
      - Create a virtual env called `venv` in the current directory using the following command: `virtualenv -p python3 venv`
      - Activate the virtualenv called `venv` by running: `source venv/bin/activate`. Your command prompt will change to indicate you're working inside `venv`.
    - For Windows systems:
      - Create a virtual env called `venv` in the current directory using the following command: `virtualenv -p C:/Python36/python.exe venv`. You may need to adjust the `-p` argument depending on where your version of Python is located.
      - Activate the virtualenv called `venv` by running: `.\venv\Scripts\activate`
- Run `pip install -r requirements.txt` to install the required python libraries.

4.4.3 Getting started

Here are some notes and sample code to help you get started writing a sous chef.

Downloader

The Ricecooker module `utils/downloader.py` provides a `read` function that can read from both urls and file paths. To use:

```
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
js_content = 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 information.

HTML parsing using BeautifulSoup

BeautifulSoup is an HTML parsing library that allows to select various DOM elements, and extract their attributes and text contents. 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)
page = BeautifulSoup(html, 'html.parser')

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

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 remove a element from the DOM tree (useful to remove labels, and extra stuff)

For more info about BeautifulSoup, see the docs.

4.4.4 Using the DataWriter

The DataWriter (`ricecooker.utils.data_writer.DataWriter`) is a tool for creating channel `.zip` files in a standardized format. This includes creating folders, files, and `CSV` metadata files that will be used to create the channel on Kolibri Studio.

Step 1: Open a DataWriter

The `DataWriter` class is meant to be used as a context manager. To use it, add the following to your code:

```
from ricecooker.utils.data_writer import DataWriter
with DataWriter() as writer:
    # Add your code here
```

You can also pass the argument `write_to_path` to control where the `DataWriter` will generate a zip file.

Step 2: Create a Channel

Next, you will need to create a channel. Channels need the following arguments:

- `title` (str): Name of channel
- `source_id` (str): Channel's unique id
- `domain` (str): Who is providing the content
- `language` (str): Language of channel
- `description` (str): Description of the channel (optional)
- `thumbnail` (str): Path in zipfile to find thumbnail (optional)

To create a channel, call the `add_channel` method from DataWriter

```
from ricecooker.utils.data_writer import DataWriter

CHANNEL_NAME = "Channel name shown in UI"
CHANNEL_SOURCE_ID = "<some unique identifier>"
CHANNEL_DOMAIN = "<yourdomain.org>"
CHANNEL_LANGUAGE = "en"
CHANNEL_DESCRIPTION = "What is this channel about?"

with DataWriter() as writer:
    writer.add_channel(CHANNEL_NAME, CHANNEL_SOURCE_ID, CHANNEL_DOMAIN, CHANNEL_
↪LANGUAGE, description=CHANNEL_DESCRIPTION)
```

To add a channel thumbnail, you must write the file to the zip folder

```
thumbnail = writer.add_file(CHANNEL_NAME, "Channel Thumbnail", CHANNEL_THUMBNAIL,␣
↪write_data=False)
writer.add_channel(CHANNEL_NAME, CHANNEL_SOURCE_ID, CHANNEL_DOMAIN, CHANNEL_LANGUAGE,␣
↪description=CHANNEL_DESCRIPTION, thumbnail=thumbnail)
```

The DataWriter's `add_file` method returns a filepath to the downloaded thumbnail. This method will be covered more in-depth in Step 4.

Every channel must have language code specified (a string, e.g., `'en'`, `'fr'`). To check if a language code exists, you can use the helper function `getlang`, or lookup the language by name using `getlang_by_name` or `getlang_by_native_name`:

```
from le_utils.constants.languages import getlang, getlang_by_name, getlang_by_native_
↪name
getlang('fr').code                          # = 'fr'
getlang_by_name('French').code              # = 'fr'
getlang_by_native_name('Français').code     # = 'fr'
```

The same language codes can optionally be applied to folders and files if they differ from the channel language (otherwise assumed to be the same as channel).

Step 3: Add a Folder

In order to add subdirectories, you will need to use the `add_folder` method from the DataWriter class. The method `add_folder` accepts the following arguments:

- `path` (str): Path in zip file to find folder

- `title` (str): Content's title
- `source_id` (str): Content's original ID (optional)
- `language` (str): Language of content (optional)
- `description` (str): Description of the content (optional)
- `thumbnail` (str): Path in zipfile to find thumbnail (optional)

Here is an example of how to add a folder:

```
# Assume writer is a DataWriter object
TOPIC_NAME = "topic"
writer.add_folder(CHANNEL_NAME + "/" + TOPIC_NAME, TOPIC_NAME)
```

Step 4: Add a File

Finally, you will need to add files to the channel as learning resources. This can be accomplished using the `add_file` method, which accepts these arguments:

- `path` (str): Path in zip file to find folder
- `title` (str): Content's title
- `download_url` (str): Url or local path of file to download
- `license` (str): Content's license (use le_utils.constants.licenses)
- `license_description` (str): Description for content's license
- `copyright_holder` (str): Who owns the license to this content?
- `source_id` (str): Content's original ID (optional)
- `description` (str): Description of the content (optional)
- `author` (str): Author of content
- `language` (str): Language of content (optional)
- `thumbnail` (str): Path in zipfile to find thumbnail (optional)
- `write_data` (boolean): Indicate whether to make a node (optional)

For instance:

```
from le_utils.constants import licenses

# Assume writer is a DataWriter object
PATH = CHANNEL_NAME + "/" + TOPIC_NAME + "/filename.pdf"
writer.add_file(PATH, "Example PDF", "url/or/link/to/file.pdf", license=licenses.CC_
↪BY, copyright_holder="Somebody")
```

The `write_data` argument determines whether or not to make the file a node. This is espcially helpful for adding supplementary files such as thumbnails without making them separate resources. For example, adding a thumbnail to a folder might look like the following:

```
# Assume writer is a DataWriter object
TOPIC_PATH = CHANNEL_NAME + "/" + TOPIC_NAME
PATH = TOPIC_PATH + "/thumbnail.png"
```

(continues on next page)

(continued from previous page)

```
thumbnail = writer.add_file(PATH, "Thumbnail", "url/or/link/to/thumbnail.png", write_
↪data=False)
writer.add_folder(TOPIC_PATH, TOPIC_NAME, thumbnail=thumbnail)
```

Every content node must have a `license` and `copyright_holder`, otherwise the later stages of the content pipeline will reject. You can see the full list of allowed license codes by running `print(le_utils.constants.licenses.choices)`. Use the ALL_CAPS constants to obtain the appropriate string code for a license. For example, to set a file's license to the Creative Commons CC BY-NC-SA, get get the code from `licenses.CC_BY_NC_SA`.

Note: Files with `licenses.PUBLIC_DOMAIN` do not require a `copyright_holder`.

4.4.5 Extra Tools

PathBuilder (ricecooker.utils.path_builder.py)

The `PathBuilder` clas is a tool for tracking folder and file paths to write to the zip file. To initialize a PathBuilder object, you need to specify a channel name:

```
from ricecooker.utils.path_builder import PathBuilder

CHANNEL_NAME = "Channel"
PATH = PathBuilder(channel_name=CHANNEL_NAME)
```

You can now build this path using `open_folder`, which will append another item to the path:

```
...
PATH.open_folder('Topic')          # str(PATH): 'Channel/Topic'
```

You can also set a path from the root directory:

```
...
PATH.open_folder('Topic')          # str(PATH): 'Channel/Topic'
PATH.set('Topic 2', 'Topic 3')     # str(PATH): 'Channel/Topic 2/Topic 3'
```

If you'd like to go back one step back in the path:

```
...
PATH.set('Topic 1', 'Topic 2')     # str(PATH): 'Channel/Topic 1/Topic 2'
PATH.go_to_parent_folder()         # str(PATH): 'Channel/Topic 1'
PATH.go_to_parent_folder()         # str(PATH): 'Channel'
PATH.go_to_parent_folder()         # str(PATH): 'Channel' (Can't go past root level)
```

To clear the path:

```
...
PATH.set('Topic 1', 'Topic 2')     # str(PATH): 'Channel/Topic 1/Topic 2'
PATH.reset()                       # str(PATH): 'Channel'
```

Downloader (ricecooker.utils.downloader.py)

`downloader.py` has a `read` function that can read from both urls and file paths. To use:

```
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
js_content = read('https://example.com/loadpage', loadjs=True)   # Load js before␣
↪getting contents
```

The `loadjs` option will load any scripts before reading the contents of the page, which can be useful for web scraping.

If you need to use a custom session, you can also use the `session` option. This can be useful for sites that require login information.

HTMLWriter (ricecooker.utils.html_writer.py)

The HTMLWriter is a tool for generating zip files to be uploaded to Kolibri Studio

First, open an HTMLWriter context:

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

To write the main file, you will need to 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`:

```
# 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
```

If you need to check if a file exists in the zipfile, you can use the `contains` method:

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

See the above example on BeautifulSoup on how to parse html.

CHAPTER 5

Ricecooker developer docs

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

5.1 Notes for ricecooker library developers

5.1.1 Supported Python Versions for Chefs

All chefs written need to support either **Python 3.4 or 3.5**.

If you need a module or need to use syntax that is only available in newer Python versions, please get in touch.

5.1.2 Computed identifiers

Channel ID

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

- Is used in the wire formats used to communicate between `ricecooker` and Kolibri Studio
- Appears as part of URLs for on both Kolibri Studio and Kolibri
- Determines the filename for the channel sqlite3 database file that Kolibri imports from Kolibri Studio.

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
channel_id = uuid.uuid5(
    uuid.uuid5(uuid.NAMESPACE_DNS, source_domain),
    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`.

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.

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`.

5.2 Daemon mode

Running a chef script with the `--daemon` option will make it listen to remote commands: either from sushibar and/or from localhost cron jobs.

5.2.1 SushiBar control channel

To enable remote commands from sushibar, start the chef script using

```
./chef.py --daemon  <otherstuff>
```

5.2.2 Local control channel

To also enable local UNIX domain sockets commands, start the chef script using

```
./chef.py --daemon --cmdsock=/var/run/cmdsocks/channelA.sock  <otherstuff>
```

Once the chef is running, a chef run can be started by sending the appropriate json data to the UNIX domain socket `/var/run/cmdsocks/channelA.sock`. Use the `nc` command for this (install netcat using `apt-get install netcat-openbsd`).

```
/bin/echo '{"command":"start"}' | /bin/nc -UN /var/run/cmdsocks/channelA.sock
```

If you need to override chef run `args` or `options` use:

```
/bin/echo '{"command":"start", "args":{"publish":true}, "options":{"lang":"en"} }' | /
↪bin/nc -UN /var/run/cmdsocks/channelA.sock
```

The above command will run the chef, re-using the command line args and options, but setting `publish` to `True` and also providing the keyword option `lang=en`.

Chef runs can be scheduled by setting up cronjobs for the above commands.

5.3 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 on the Sushibar dashboard, in Kolibri Studio, and 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.

5.3.1 Project management and support

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

5.3.2 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`
- other chefs running on `cloud-kitchen`
- various other chefs running on partner orgs infrastructure

5.3.3 Scheduled runs

Chefs scripts can be scheduled to run automatically on a periodic basis, e.g., once a month. In between runs, chef scripts stay dormant (daemonized). Scheduled chefs run by default with the `--stage` argument in order not to accidentally overwrite the currently active content tree on Studio with a broken one. If the channel content is relatively unchanged and raises no flags for review, the staged tree will be ACTIVATED, and the channel PUBLISHed automatically as well.

5.3.4 Chef inventory

In order to keep track of all the sushi chefs (30+ and growing), SushOps people maintain this spreadsheet listing and keep it up-to-date for all chefs:

- chef_name, short, unique identified, e.g., `khan_academy_en`
- chef repo url
- command necessary to run this chef, e.g., `./kachef.py ... lang=en`
- scheduled run settings (crontab format)

This spreadsheet is used by humans as an inventory of the chef scripts currently in operation. The automation scripts use the same data to provision chef scripts environments, and setting up scheduling for them on the LE cheffing servers.

5.3.5 SushOps tooling and automation

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

```
fab -R cloud-kitchen   setup_chef:chef_name     # clones the chef_name repo and␣
↪installs requirements
fab -R cloud-kitchen   update:chef_name         # git fetch and git reset --hard to␣
↪get latest chef code
fab -R cloud-kitchen   run_chef:chef_name       # runs the chef
fab -R cloud-kitchen   schedule_chef:chef_name  # set up chef to run as cronjob
```

You can import the reusable fab commands from `ricecooker.utils.fabfile`. [WIP]

5.4 Command line interface

This document describes logic `ricecooker` uses to parse command line arguments. Under normal use cases you shouldn't need modify the command line parsing, but you need to understand how `argparse` works if you want to add new command line arguments for your chef script.

5.4.1 Summary

A sushi chef script using the new API looks like this:

```
#!/usr/bin/env python
...
...
class MySushiChef(BaseChef):     # or SushiChef to support remote monitoring
    def get_channel(**kwargs)` -> ChannelNode (bare channel, used just for info)
        ...
```

(continues on next page)

(continued from previous page)

```
    def construct_channel(**kwargs) -> ChannelNode (with populated Tree)
        ...
...
...
if __name__ == '__main__':
    chef = MySushiChef()
    chef.main()
```

5.4.2 Flow diagram

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

```
MySushiChef -----extends----> BaseChef                    commands.uploadchannel
---------------------------   ------------------------    ----------------------
                              1. main()
                              2. parse_args_and_options()
                              3. run(args, options)
                                                          4. uploadchannel(chef, *args,
↪ **options)
                                                          ...
5. get_channel(**kwargs)
                                                          ...
6. construct_channel(**kwargs)
                                                          ...
                                                          ...
                                                          DONE
```

5.4.3 Changes

Old `uploadchannel` API (a.k.a. compatibility mode)

- pass in chef script file as "`<file_path>`" to uploadchannel
- `uploadchannel` calls the function `contruct_channel` defined in the chef script

New `uploadchannel` API

- The chef script defines subclass of `riececooker.chefs.SushiChef` that implement the methods `get_channel` and `construct_channel`:

```
class MySushiChef(riececooker.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
...
...
```

(continues on next page)

(continued from previous page)

```
...
if __name__ == '__main__':
    chef = MySushiChef()
    chef.main()
```

- The `__init__` method of the sushi chef class configures an `argparse` parser (`BaseChef` creates `self.arg_parser` and each class adds to this shared parser its own command line arguments.)
- The `main` method of the class parses the command line arguments and calls the `run` method (or the `deamon_mode` method.)

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

- The chef's `run` method calls `uploadchannel` (or `uploadchannel_wrapper`)

```
class BaseChef():
    ...
    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

5.4.4 Compatibility mode

Calling ricecooker as a module (`python -m ricecooker uploadchannel oldchef.py ...`) will run the following code in `ricecooker.__main__.py`:

```
from ricecooker.chefs import BaseChef
if __name__ == '__main__':
    chef = BaseChef(compatibility_mode=True)
    chef.main()
```

The `BaseChef` class with `compatibility_mode=True` proxies call to its `construct_channel` method to the function `construct_channel` in `oldchef.py`. The call to `chef.main()` results in the following sequence of events:

```
oldchef.py                    BaseChef(compat mode)      commands.uploadchannel
---------------------------   ----------------------     ----------------------
                              1. main()
                              2. parse_args_and_options()
                              3. run(args, options)
                                                         4. uploadchannel(chef, *args,
↪ **options)
                                                         ...
                                                         ...
                              5. construct_channel(**kwargs)
5'. construct_channel(**kwargs)
                                                         ...
                                                         ...
                                                         DONE
```

Logging and progress reporting to SushiBar server is not supported in compatibility mode.

5.4.5 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
  - BaseChef: the method `BaseChef.__init__` configures argparse for the following:
    - `compress`, `download_attempts`, `prompt`, `publish`, `reset`, `resume`, `stage`, `step`, `thumbnails`, `token`, `update`, `verbose`, `warn`
    - in compatibility mode, also handles `uploadchannel` and `chef_script` positional arguments
  - SushiChef:
    - `daemon` = Runs in daemon mode
    - `nomonitor` = Disable SushiBar progress monitoring
  - MySushiChef: the chef's `__init__` method can define additional cli args
- `options` (dict): additional [OPTIONS...] passed at the end of the command line
  - used for compatibility mode with old rieceooker API (`python -m ricecooker uploadchannel ... key=value`)
- `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.

5.4.6 Daemon mode

In daemon mode, we open a `ControlWebSocket` connection with the SushiBar and wait for commands.

When a command comes in on the control channel, it looks like this:

```
message = {"command":"start", "args":{...}, "options":{...}}
```

Then the handler `ControlWebSocket.on_message` will start a new run:

```
args.update(message['args'])        # remote arguments overwrite ricecooker cli args
options.update(message['options'])  # remote options overwrite cli options
chef.run(args, options)
```

After finishing the run, a chef started with the `--daemon` option remains connected to the SushiBar server and listens for more commands.

5.5 Contributing

Contributions to this project are welcome and are in fact greatly appreciated! Every little bit helps and credit will always be given. Whether you're a junior Python programmer looking for a open source project to contribute to, an advanced programmer that can help us make `ricecooker` more efficient, we'd love to hear from you. We've outlined below some of the ways you can contribute.

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/learningequality/ricecooker/issues

If you are reporting a bug, please include:

- Which version of `ricecooker` you're using.
- Which operating system you're using (name and version).
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open game for community contributors.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

The `ricecooker` library can always use more documentation. You can contribute fixes and improvements to the official `ricecooker` docs, add docstrings to code, or write a blog post or article and share your experience using `ricecooker`.

Submit Feedback

The best way to send us your feedback is to file an issue at https://github.com/learningequality/ricecooker/issues.

If you are proposing a new feature:

- Explain in detail how it would work.
- Try to keep the scope as narrow as possible to make it easier to implement.
- Remember this is a volunteer-driven project, and contributions are welcome :)

5.5.2 Getting Started!

Ready to contribute? In order to work on the `ricecooker` code you'll first need to make you have Python 3 on your computer. You'll also need to install the Python package pip if you don't have it already.

Here are the steps for setting up `ricecooker` for local development:

1. Fork the `ricecooker` repo on GitHub.
2. Clone your fork of the repository locally, and go into the `ricecooker` directory::

```
git clone git@github.com:<your-github-username>/ricecooker.git
cd ricecooker/
```

3. Create a Python virtual environment for this project (optional, but recommended):
   - Install the `virtualenv` package using the command

```
pip install virtualenv
```

   - The next steps depends if you're using a UNIX system (Mac/Linux) or Windows:
     - For UNIX operating systems:
       * Create a virtual env called `venv` in the current directory using the command:

```
virtualenv -p python3 venv
```

       * Activate the virtualenv called `venv` by running:

```
source venv/bin/activate
```

         Your command prompt will change to indicate you're working inside `venv`.
     - For Windows systems:
       * Create a virtual env called `venv` in the current directory using the following command:

```
virtualenv -p C:/Python36/python.exe venv
```

         You may need to adjust the `-p` argument depending on where your version of Python is located. Note you'll need Python version 3.4 or higher.

       * Activate the virtualenv called `venv` by running:

```
.\venv\Scripts\activate
```

4. Install the `ricecooker` code in the virtual environment using these commands::

```
pip install -e .
```

5. Create a branch for local development::

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

6. When you're done making changes, check that your changes pass flake8 linter rules and the `ricecooker` test suite, including testing other Python versions with tox::

```
flake8 ricecooker tests
pytest
tox
```

   To get `flake8` and `tox`, just `pip install` them into your virtualenv.

7. Commit your changes and push your branch to GitHub::

```
git add .
git commit -m "A detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

8. Open a pull request through the GitHub web interface.

5.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in `README.md`.
3. The pull request should work for Python 3.4, 3.5. Check https://travis-ci.com/learningequality/ricecooker/pull_requests and make sure that the tests pass for all supported Python versions.

5.5.4 Tips

To run a subset of tests, you can specify a particular module name::

```
$ py.test tests.test_licenses
```

5.6 Credits

- Jordan Yoshihara <jordan@learningequality.org>
- Aron Asor <aron@learningequality.org>
- Jamie Alexandre <jamie@learningequality.org>
- Benjamin Bach <ben@learningequality.org>
- Ivan Savov <ivan@learningequality.org>

- David Hu <davidhu@learningequality.org>
- Kevin Ollivier <kevin@learningequality.org>
- Alejandro Martinez Romero <mara80@gmail.com>

5.7 Ricecooker Python API

5.7.1 ricecooker package

ricecooker.chefs module

ricecooker.classes module

ricecooker.commands module

ricecooker.config module

ricecooker.exceptions module

exception ricecooker.exceptions.**FileNotFoundException**(**args*, ***kwargs*)
Bases: Exception

FileNotFoundException: raised when file path is not found

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**InvalidCommandException**(**args*, ***kwargs*)
Bases: Exception

InvalidCommandException: raised when unrecognized command is entered

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**InvalidFormatException**(**args*, ***kwargs*)
Bases: Exception

InvalidFormatException: raised when file format is unrecognized

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**InvalidNodeException**(**args*, ***kwargs*)
Bases: Exception

InvalidNodeException: raised when node is improperly formatted

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**InvalidQuestionException**(**args*, ***kwargs*)
Bases: Exception

InvalidQuestionException: raised when question is improperly formatted

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**InvalidUsageException**(**args*, ***kwargs*)
Bases: Exception

InvalidUsageException: raised when command line syntax is invalid

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**UnknownContentKindError**(**args*, ***kwargs*)
Bases: Exception

UnknownContentKindError: raised when content kind is unrecognized

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**UnknownFileTypeError**(**args*, ***kwargs*)
Bases: Exception

UnknownFileTypeError: raised when file type is unrecognized

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**UnknownLicenseError**(**args*, ***kwargs*)
Bases: Exception

UnknownLicenseError: raised when license is unrecognized

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception ricecooker.exceptions.**UnknownQuestionTypeError**(**args*, ***kwargs*)
Bases: Exception

UnknownQuestionTypeError: raised when question type is unrecognized

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

ricecooker.exceptions.**raise_for_invalid_channel**(*channel*)

ricecooker.managers module

ricecooker.sushi_bar_client module

ricecooker.utils module

There is *separate documentation* for the utilities.

Module contents

5.7.2 utils package

ricecooker.utils.browser module

ricecooker.utils.browser.**preview_in_browser**(*directory*, *filename='index.html'*, *port=8282*)

ricecooker.utils.caching module

ricecooker.utils.data_writer module

ricecooker.utils.downloader module

ricecooker.utils.html module

ricecooker.utils.html_writer module

ricecooker.utils.jsontrees module

ricecooker.utils.libstudio module

ricecooker.utils.linecook module

ricecooker.utils.metadata_provider module

ricecooker.utils.path_builder module

class ricecooker.utils.path_builder.**PathBuilder**(*channel_name=None*)

Bases: object

Class for formatting paths to write to DataWriter.

channel_name = None

go_to_parent_folder()

Go back one level in path Args: None Returns: last item in path

open_folder(*path_item*)

Add item to path Args: path_item: (str) item to add to path Returns: None

path = None

reset()

Clear path Args: None Returns: None

set (**path*)
Set path from root Args: path: (str) items to add to path Returns: None

ricecooker.utils.paths module

ricecooker.utils.paths.**build_path** (*levels*)
make a linear directory structure from a list of path levels names levels = ["chefdir", "trees", "test"] builds ./chefdir/trees/test/

ricecooker.utils.paths.**dir_exists** (*filepath*)

ricecooker.utils.paths.**file_exists** (*filepath*)

ricecooker.utils.paths.**get_name_from_url** (*url*)
get the filename from a url url = http://abc.com/xyz.txt get_name_from_url(url) -> xyz.txt

ricecooker.utils.paths.**get_name_from_url_no_ext** (*url*)
get the filename without the extension name from a url url = http://abc.com/xyz.txt get_name_from_url(url) -> xyz

ricecooker.utils.pdf module

There is *detailed documentation* available on the PDF tool.

PDF Utils

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

PDF splitter

When importing source PDFs like books that is a very long documents (100+ pages), it is better for 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 the information available in the PDF's table of contents.

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'},
  ...
]
```

Use this information to create individual `DocumentNodes` 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 `get_toc` method internally to obtain the 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)
```

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()` mothod 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': [
        {'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 `DocmentNodes` for each of the children of that chapter.

Accessibility notes

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

ricecooker.utils.tokens module

ricecooker.utils.zip module

ricecooker.utils.zip.**create_predictable_zip**(*path*)

Create a zip file with predictable sort order and metadata so that MD5 will stay consistent if zipping the same content twice. :param path: absolute path either to a directory to zip up, or an existing zip file to convert. :type path: str

Returns: path (str) to the output zip file

ricecooker.utils.zip.**write_file_to_zip_with_neutral_metadata**(*zfile*, *filename*, *content*)

Write the string *content* to *filename* in the open ZipFile *zfile*. :param zfile: open ZipFile to write the content into :type zfile: ZipFile :param filename: the file path within the zip file to write into :type filename: str :param content: the content to write into the zip :type content: str

Returns: None

Module contents

5.8 History

5.8.1 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`

5.8.2 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

5.8.3 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.

5.8.4 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)

5.8.5 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

5.8.6 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

5.8.7 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

5.8.8 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

5.8.9 0.6.2 (2017-07-07)

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

5.8.10 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

5.8.11 0.5.13 (2017-06-15)

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

5.8.12 0.1.0 (2016-09-30)

- First release on PyPI.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

S

U

W