H5P tutorial
Version 0.3

Sviatoslav Tugeev (sviatoslav.tugeev@thi.de)
Technische Hochschule Ingolstadt

December 17, 2024

Contents

1 Foreword

These instructions are written on 20.11.2024, please note that H5P plugin and format can change with time. Additionally, these instructions contain the correct information to the best of the author's knowledge, which does not make them definitely 100 percent correct.

These instructions were written for Windows 11, but most if not all information is applicable to other platforms as well.

2 What is H5P?

H5P itself is a "plugin for existing publishing systems" that enables the system to create and/or display H5P content. H5P provides a packaging format, that allows combining HTML, CSS and Javascript into a single platform-agnostic package, that can be used and displayed anywhere the H5P plugin is installed. Just as any other combination of these elements, this package can theoretically be anything, from simple text to interactive videos, games and control panels for other elements. H5P has several key benefits, compared to just implementing HTML, CSS and Javascript manually on the platform directly:

- **Ease of implementation**: many platforms, such as Moodle, are difficult to modify manually. H5P provides an interface that, after a few initial structural hurdles, allows easy plug and play use of the created packages.
- **Reusability**: H5P content is structured in a way that is supposed to be easy to reuse, especially for non-programmers. After the initial package defining type and mechanics of content is created, you can easily use H5P interface to upload or create different sets of content, thus quickly creating variations of the basic template. The more complex the template, the more variations can be created.
- **Existing templates:** There are multiple templates for H5P activities already created and available for free, meaning you don't have to reinvent the wheel every time.

3 What is an .h5p package and what should it contain?

An .h5p file is essentially a zip archive (with few important differences discussed later), containing everything your activity needs, as well as several instructions on how the H5P plugin should handle your package. What it should contain differs slightly, depending on what the package is supposed to do. Generally speaking, there are two types of h5p packages:

- **Template package**, used to define the template for creating an .h5p activity. This package is usually uploaded to the platform by an administrator,

allowing the use of this template to create the variations of activity with different content. An example of this would be a package, defining the mechanics of a crossword and interface for creating one.

- **Instance package**, used to define the specific variation of the template, described in the template package. It can usually be created or uploaded to the platform by different users. An example would be an actual crossword description, with specific words and their positions on the field.

What is often a point of confusion is that both of these packages have .h5p extension in the end and instance package actually has to include the same files as the template package, as well as a few additional ones. Package structure will be described in more detail further on.
Another term that will be used throughout this tutorial is "H5P library". This term can describe an H5P package (usually template one), a folder within this package (defining the key functionality of this template), or a similar folder on the platform that hosts H5P content. Functionally, it is basically the same as the template package, with the following differences:

- H5P Library is simply a folder with the files, not necessarily a zipped .h5p archive.
- H5P packages can contain several libraries inside them, separating functionality into clear modules, although this tutorial does not examine this further.

4 Prerequisites

Theoretically, all you need to develop an H5P activity is access to a code editor, such as VSCode. However, there are a few caveats and ways you might want to or have to make your life easier.

4.1 Packaging prerequisites

There are several ways to pachage H5P content. There is a dedicated command to do that, offered by H5P command line interface. You can read more about it here: `https://h5p.org/h5p-cli-guide`. However, many complain that this command often doesn't work, and there are instances where manual approach is more reliable and presents a clearer picture. This tutorial will focus on this manual approach specifically, but the rest of information can be used with the CLI command as well.
In order to package H5P activity manually, you need access to the "zip" command. If you are working on Linux or MacOS, that should not be a problem. On Windows you will need to install WSL (Windows Subsystem for Linux), as by default Windows does not have access to this command.
Additionally, it might be very useful to prepare all the commands you need to execute. If you are working on MacOS, you can just copy them into a .sh

file. On Windows and Linux you will need a *makefile* instead, like the one included in this example. However, on Windows its use requires WSL as well. You can find instructions for installing WSL here: `https://learn.microsoft.com/en-us/windows/wsl/install`. To install "zip" and "make" packages you will most likely need commands "sudo apt install zip" and "sudo apt install make", however they can be a little different, depending on your specific Linux distribution.
The packaging process itself, as well as what the final package needs to contain, is presented in a further section.

4.2 Testing environment

While you can easily develop your H5P package without testing it, it is a good idea to have access to a simple testing environment. There are multiple ways to do this, but one of the simpler ones is installing a local H5P server. Detailed instructions on how to do this are available here: `https://github.com/h5p/h5p-cli`
To use this new server for testing your package, you have two choices. One is safer, takes longer and is more suited for testing your activity at rare specific points. The other is better if you want to test your package after every single change, but can make it more difficult to go back if you break something.

4.2.1 Clear way

Later on in this tutorial you will see the directory structure of your package, which includes a folder "H5P.LibraryName". In the folder where you installed your H5P server you should have the folder called "libraries". To test your package you will need to copy the "H5P.LibraryName" folder into the "libraries" folder and start/restart the H5P server. After that you can use your template by going to your server page and clicking "new". This method will require you to copy the "H5P.LibraryName" every time you make a change, but will ensure that the original package is not affected no matter what.

4.2.2 Faster way

Alternatively, you can merge the folder structures of the example described in this tutorial and the H5P development environment, such that the "H5P.LibraryName" is located in the "libraries" folder. That way whatever changes you make, you can see them in the server directly.

4.2.3 Limitations

Unfortunately, while this testing environment can be quite fast, it likely does not fully represent the actual platform you are developing for, such as Moodle. It is possible that if your package is relatively complex, it might have difficulties when transitioning from h5p testing server to the actual platform, which you will need to resolve. It might be a good idea to install a second level of testing environment to guarantee compatibility, for example the local Moodle installation.

5 Creating Template h5p package

First, I will describe creation of a template package, because files included there are mandatory, no matter which package you are creating. To create this package you need to create a set of mandatory files and your custom content in one folder, then actually package folder in a particular way.

5.1 Folder structure

There are several possible structure you could create, but I recommend the one below. Files are marked in *italics*:

```
MainFolder
├── makefile
└── WorkFolder
    └── H5P.LibraryName
        ├── style.css
        ├── code.js
        ├── library.json
        └── semantics.json
```

Please note, that files *library.json* and *semantics.json* have to be where they are and have to be named that way, otherwise the package will not work. The name of your library can vary, but the name of the folder that contains it has to have the "*H5P.*" in the beginning! Finally, the "*code.js*" can have a different name, but it necessary for your package to work, because it contains the code that actually defines the functionality of your library. It has to be located in the same place as *library.json*, i.e. not in a subfolder. The *makefile* does not have an extension and will be described below. *style.css* is not mandatory for a package to work, but is useful to illustrate how you would include your own CSS.
We will go through the files one by one

5.1.1 Makefile

This file contains the instructions for creating an actual package and currently seems necessary to package it correctly. The reason for that (and why you can't just compress the files into a zip archive via right click and rename it later) is that the package can't contain actual directories. I will not go into details what that means, but essentially, while the file structure should remain after packaging, the directories themselves are not allowed. Here is the example of the makefile:

```
all:
  rm -f -- PackageName.h5p
  cd WorkFolder && zip -r -D ../PackageName.h5p ./*
```

The first line instructs to remove the file PackageName.h5p if it already exists, to allow you to easily execute this file every time you change something, the second goes into the WorkFolder and zips everything inside recursively into a

package PackageName.h5p, while not including any actual folders. You can, of course, replace the “PackageName” and “WorkFolder” with your own names.

5.1.2 style.css

The content of this file is just an example, and in this case contains simple CSS definition for the container that will actually house the H5P content on the page, as well as image and text displayed inside. The ".h5p-" prefix is not necessary, but might be useful to avoid confusion and conflict between different CSS files.

```
.h5p-containerstyle {
    width: 400px;
    margin: 0 auto;
    border: 1px solid #ccc;
    box-shadow: 2px 2px 4px #ccc;
}

.h5p-containerstyle .imagestyle {
    width: 100%;
    height: auto;
}

.h5p-containerstyle .textstyle {
    width: auto;
    margin: 16px;
    font-family: fantasy;
    font-size: 20px;
    font-weight: bold;
    line-height: 24px;
    text-align: center;
}
```

5.1.3 code.js

Again, this is just an example for the Javascript code you can include in your package. This code defines what your library actually does. The code is based on the "greeting card" example from the official github repository and is well commented, however, several things should be noted:
1) At line 1 we create an H5P object to actually house our library.
2)At line 3 we define the name of our library - the "LibraryName" should always match, wherever you write it! The folder "H5P.LibraryName" should always have the same name as in this file!
3) At lines 10 and 11 we define what kind of input our library can expect, i.e. what content will be accessible in the actual code. These are the default values, and they can be be overridden if, for example, the user uploads their own content in these fields.
4) Finally, as seen on line 28, to access the files used during the creation of the activity, the H5P.getPath method should be used. You can read more about this method in the official documentation.

```
var H5P = H5P || {};

H5P.LibraryName = (function ($) {
  /**
   * Constructor function.
   */
  function C(options, id) {
    // Extend defaults with provided options
    this.options = $.extend(true, {}, {
      test_text: 'Hello world!',
      test_image: null
    }, options);
    // Keep provided id.
    this.id = id;
  };

  /**
   * Attach function called by H5P framework to insert H5P content
    into
   * page
   *
   * @param {jQuery} $container
   */
  C.prototype.attach = function ($container) {
    // Add css class to the container that houses all futher H5P
    content
    $container.addClass("h5p-containerstyle");
    // Add image if provided.
    if (this.options.test_image && this.options.test_image.path) {
      $container.append('<img class="imagestyle" src="' + H5P.
    getPath(this.options.test_image.path, this.id) + '">');
    }
    // Add text.
    $container.append('<div class="textstyle">' + this.options.
    test_text + '</div>');
  };

  return C;
})(H5P.jQuery);
```

5.1.4 library.json

This file describes the library you have created and provides information to the H5P plugin about how it should be handled. It is a snadart json file, with several mandatory and multiple optional fields. You can read the full explanation of the mandatory fields here: https://h5p.org/library-definition
Note the preloadedJs and PreloadedCss sections - the path to the files is described relatively to the location of the library.json and you can easily include multiple files in both lists. The value is always provided in standard key-value pairs, surrounded by curly braces, with each element separated by a comma.

```
{
    "title": "Example template",
    "description": "Displays an image and a text",
    "majorVersion": 1,
    "minorVersion": 0,
    "patchVersion": 1,
    "runnable": 1,
    "author": "Author name",
    "license": "cc-by-sa",
    "machineName": "H5P.LibraryName",
    "coreApi": {
        "majorVersion": 1,
        "minorVersion": 0
    },
    "preloadedJs": [
        {
            "path": "code.js"
        }
    ],
    "preloadedCss": [
        {
            "path": "style.css"
        }
    ]
}
```

5.1.5 semantics.json

When a user on the platform uses an h5p template package to create a template instance, they are presented with an interface that prompts them to upload files or fill out some fields. This file describes the structure of this interface and can be quite complex. You can read much more about its structure here: https://h5p.org/semantics
In the provided example, the user will be prompted to write some text and upload an image. The image is optional and is marked as such.

```
[
  {
    "label": "Text",
    "name": "test_text",
    "type": "text",
    "default": "Hello world!",
    "description": "This text will be shown when the package is
    displayed."
  },
  {
    "label": "image",
    "name": "test_image",
    "type": "image",
    "optional": true,
    "description": "Image, optional."
  }
]
```

5.2 Packaging a template h5p package

To package this example, follow these steps:

1. Go to your MainFolder directory and open a terminal there. You can use an actual terminal, a terminal in your code editor, or any other type.

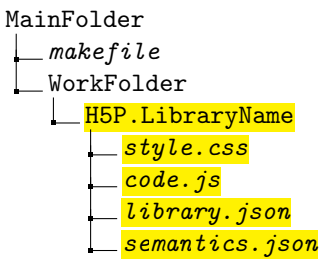
2. If you are working on Windows, you should have already installed a windows subsystem for linux, so now execute "wsl" command to launch your linux subsystem and get access to the "make" and "zip" commands.

3. Simply execute "make" to launch your *makefile*, located in the MainFolder. It will go through all the commands written in it and create a PackageName.h5p file in your MainFolder that you can now use.

To use this type of package on, for example, Moodle, you will need to upload it in the website administration section, in H5P - Manage H5P content types. After you do that, depending on the Moodle version, you will be able to either choose the template directly when creating an H5P activity, or in the content bank section.

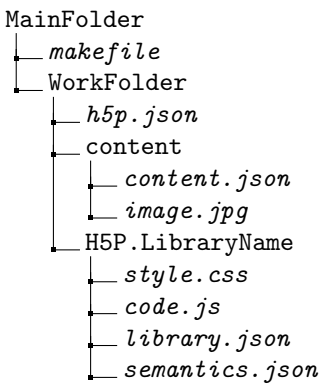
5.3 What should the final package look like?

As stated before, the final package is essentially a zip archive with an .h5p extention. The key detail is that the archive can not actually contain the folders themselves. If it does, when uploading the package to your platform you will get an error "content type is not allowed". This is not something you can check just by renaming the archive back to .zip and opening it - you will see folders there wherever the archive contains them or not. This is because the folders can be included as objects, or simply as part of paths to the files inside them, and they both look the same when opening or unpacking an archive. To avoid packaging them, make sure to use the -D parameter for the zip command, as stated in this tutorial.
The final package, if you open it, should contain everything inside the WorkFolder folder, but NOT the WorkFolder itself! When you open your archive, if you see this WorkFolder inside - you need to adjust your packaging script to start one level deeper. What the package needs to contain is highlighted on the graph below:



6 Creating an instance h5p package

Unlike the template package, you can upload this package directly when creating a new h5p activity on the platform of your choice. It already includes the content that is used by the library. This requires to make a few alterations to the process described previously. You will need all the files described above, in the same structure, but before packaging it you will need to add one new folder and several new files. The resulting structure should look like this:



As you can see, compared to the template package, the instance package has not just the library folder, a content folder with a *content.json* file and an image inside, as well as an *h5p.json* file. Content folder, *h5p.json* and *content.json* files need to have these exact names and be located exactly where they are, otherwise the package will not work. Let's go through the new elements again, one by one.

6.1 New elements for instance h5p package

6.1.1 h5p.json

This file describes the details that are specific to your instance of the template. It does not contain any vital information, but it tells the H5P plugin how to process your specific instance. You can read more about the structure and the meaning of various fields, some of which are mandatory and some are optional, here: https://h5p.org/documentation/developers/json-file-definitions

Here is the example of an *h5p.json* for this package:

```
{
"title": "Example instance package",
"language": "en",
"author": "Tutorial user",
"license": "CC BY-SA",
"licenseVersion": "4.0",
"preloadedDependencies": [{
  "machineName": "H5P.LibraryName",
  "majorVersion": 1,
  "minorVersion": 0
}],
"mainLibrary": "H5P.LibraryName",
"embedTypes": ["div"]
}
```

Note the lines 8 and 12! The library names again have to match everywhere - the name of your library folder, the name of the library in code.js, the name of the libraries in *h5p.json* - all of them need to be identical!

6.1.2 Content folder

Content folder is for exactly that - content. It stores the files that your template will use to create an instance. In this case it is just an image that will be displayed when our package is used.

6.1.3 content.json

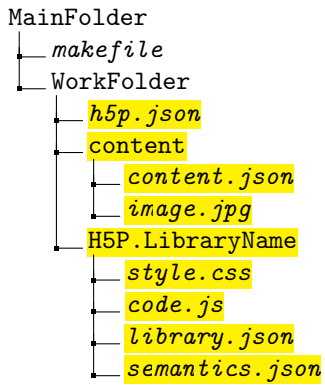
This file is mandatory and has to be present in your instance package even if no actual content is used - in that case it can even stay empty, as long as it is there. *content.json* stores information about the data described in *semantics.json*, which the user usually provides when creating the instance of the template. Essentially, it is a list of variables and their values that will are passed to your library in *code.js*, in lines 10 and 11, overriding the default.
Here is the content.json for this project:

```
{
    "test_text": "Hello world!",
    "test_image": {
      "path": "earth.jpg",
      "width": 300,
      "height": 300
    }
}
```

6.2 Packaging instance h5p package

Packging an instance package is exactly the same - the makefile packages everything inside the WorkingFolder, so it packages the new elements just the same. You can use the resulting .h5p package simply by uploading it to your platform of choice, without administrative priviliges. The final content of an instance

H5P package from this tutorial should include the elements, highlighted on the graph below:



7 Converting existing projects into H5P packages

To convert an existing project into an H5P package, you need to essentially hook up the files you already have into the H5P library you are creating.

7.1 Javascript and CSS

Hooking up Javascript and CSS files into your library is the easiest - you just need to place them into your H5P.LibraryName folder and edit the *library.json* file to load them. *library.json* defines what .js files it loads in the *preloadedJs* section, same goes for the .css files and the *preloadedCss*. If you want to add new files - just include them via a simple key-value pair. For example, if you placed another file called "Code2.js" in your "H5P.LibraryName" folder and want to load it, your *library.json* needs to look like this:

```
{
    "title": "Example template",
    "description": "Displays an image and a text",
    "majorVersion": 1,
    "minorVersion": 0,
    "patchVersion": 1,
    "runnable": 1,
    "author": "Amendor AS",
    "license": "cc-by-sa",
    "machineName": "H5P.LibraryName",
    "coreApi": {
        "majorVersion": 1,
        "minorVersion": 0
    },
    "preloadedJs": [
        {
            "path": "code.js"
        },
        {
            "path": "code2.js"
        }
    ],
    "preloadedCss": [
        {
            "path": "style.css"
        }
    ]
}
```

7.2 HTML

Including HTML is a bit trickier. For security reasons, H5P does not allow loading HTML files, both directly or via the *semantics.json*. And while there are multiple ways you could solve this issue, here are the two main ones I would recommend:

7.2.1 Loading via append

Just like in the example in this tutorial, you can use the *append* method to manually edit the H5P container. This is useful if you want direct and nuanced control over every element and don't have too much HTML code. To add another div segment, for example, just add the line

```
    $container.append('<div> new block with text </div>');
```

to the *C.prototype.attach* function.

7.2.2 Loading via separate .js file

This method is much more useful when you have a lot of HTML and would prefer just copying it all directly, instead of rewriting it. To do this you can create another file in the "H5P.MyLibrary" folder, called, for example, *htmlLoader.js*. In it you can write the following:

```
H5P.LibraryName.HTMLContent = `
<div>This is a pre-written div from a separate file!</div>
<div>And this is a second pre-written div!</div>
`;
```

Note the backticks - from ES6 they allow you to store your string on multiple lines for easier edit. Make sure that the name of the library - "H5P.LibraryName" is the same as everywhere else! You will need to load this file in your *library.json*, just like you did the *code.js*. Essentially, this script will create a new variable that your library can use, which will contain the HTML you want to add. Functionally it is not really different, and you will still need to write some code to actually insert it into the page, but that way you can easily separate blocks of your HTML into separate file or files. To load this HTML, simply add the following line to the *C.prototype.attach* function in your *code.js*:

```
$container.append(H5P.LibraryName.HTMLContent);
```

7.3 Linking to files

Another issue you might encounter when converting existing projects to work with H5P is accessing files. If you are fetching files dynamically from an outside address, it should function as it has. However, if you were storing relative path to the file and packaged it with your project, you will probably need to change a few things.
To package your files with H5P the best way is to place them inside the *content* folder of your package - that way they are always handled correctly by the H5P plugin. The "*image.jpg*" in the example project is an example of the correct location. To load it later, you will have to use the following command:

```
var filePath = H5P.getPath('PathToFile', contentId);
```

The *PathToFile* should describe the relative path to the file inside the *content* folder. ContentID is trickier, as to access it you will need to reference the instance

of the library that you have created (see *this.id* in code.js). There are several ways to get access to this id, but here I will describe just one available through the standardized H5P toolset.
If you want to access the instance ID from another file, one of the methods you can use is checking the H5P instances.

```
H5P.instances.forEach((instance) => {
    if (instance.contentId) {
    console.log('Content ID:', instance.contentId);
  }
});
```

Each h5p instance on the page gets its own ID and this script simply iterates through them and prints out these IDs into the console. If there is only one H5P instance launched on the page, this will only produce a single result. If there are several, you might need to add a check for the instance library name. Simply plug the ID into the previous script as *contentId* and you will get the correct path to the file you want to access, which you can use, for example, for the "*fetch*" command.

8 Interaction between H5P and Moodle

This section is currently under development, but several key aspects need to be presented regardless.

8.1 Automatic template package installation

Even if you do not upload the template package to the Moodle H5P administrative page, when you upload the instance package, the system automatically extracts the template package from it - if you remember, the instance package has to contain all the files that the template one has. That is why - the template package is extracted automatically and installed into the system.

8.2 Version iteration

When creating an H5P package, be it instance or template, you always have to note its version inside the *library.json*. The role of these parameters is explained in the official documentation, but it is important to state the following:

- Only one version of the library should be installed at the same time - if you want several versions to be installed at the same time, make sure to change the *machineName* parameter in one of them to essentially create a separate type of library.
- If you install a library with the same major and minor version, but with a higher patch version than the existing one, it will override the existing library in all instances. If you implemented some changes in your library but do not see them when uploading the library to Moodle - check if you have iterated the patch number!