H5P Code Editor

Project Report

Max Iraci-Sareri
School of Computer Science
The University of Adelaide
a1771834@student.adelaide.edu.au

Supervisor: Dr Cheryl Pope
School of Computer Science
The University of Adelaide
cheryl.pope@adelaide.edu.au

2021/06/11

Abstract

H5P is an open-source online content project run by community members. Using HTML5 content amongst JavaScript and CSS, H5P provides interactive web content through an easy-to-use plugin that can be directly integrated into other web pages. While H5P has many different functions and elements, no code editor exists. This project investigates a means to create such an editor through the H5P platform, including research on existing JavaScript-based editors, accessibility, and compliance to H5P's standards and the W3C Web Content Accessibility Guidelines.

1 Introduction

The integration of code editors into existing online content can assist with education and experimentation when approaching any programming topic. To this end, many existing online platforms have included JavaScript-based code editors into relevant spaces, from the 'Tryit Editor' present on *W3Schools* to the JavaScript sandbox site *JSFiddle*.

H5P is an open-source community-developed project for the purpose of creating and sharing reusable HTML5 modules. Using JavaScript, they are often interactive, and can be featured on many existing webpages such as WordPress or Canvas. H5P plugins work to reliably and easily share content for the purpose of education or demonstration. To this end, it has many usable plugins, including interactive video, quizzes, and presentations. However, H5P's collection of interactive elements does not include a code editor yet.

2 Project Motivation

2.1 Implementation for H5P

Implementation of a code editor for H5P would allow educators to more efficiently teach programming online. From the way the editor is structured in comparison to other H5P content, new questions could be posed, including identification and fixing of errors in pre-written code, or full implementation questions where previously writing code in-browser could have led to small errors or issues.

A fully-functional code editor would also not be restricted in terms of programming language; allowing use of many popular languages would mean educators could tailor the editors to the course content at will.

2.2 Accessibility Issues

When approaching this problem from the surface level, a question may arise: 'why not use any existing open-source code editor to power the H5P plugin?' Unfortunately, most open-source online code editors aren't built to the same standards that H5P expects. Accessibility is a core feature of all H5P content, through the W3C Web Content Accessibility Guidelines (WCAG); however, most existing editors don't incorporate many elements targeted for accessibility. This project investigates several different existing editors to find solutions to the following issues.

2.2.1 Tab Functionality

The prime example of problematic accessibility in online code editors is in the functionality of the `Tab` key; when it is pressed, the common standard for code editors is to input a number of spaces or a Tab character. However, when the `Tab` key is pressed in a browser, the convention is to move focus to the next element on the page; this helps people who may have issues using a browser using standard methods. The difference in standardised functionality means that most browser-based code editors implement the more familiar 'add spaces or tab' method. However, this function breaks Criterion 2.1.2 ('no keyboard trap') of the WCAG, which states "If keyboard focus can be moved to a component of the page using a keyboard interface, then focus can be moved away from that component using only a keyboard interface"[1].

2.2.2 Screen Reader Support

Section 1.1 of the WCAG, entitled 'Text Alternatives', focuses on requiring text-based alternatives to content that may be purely visual. Criterion 1.1.1 ('Non-text Content') states that if any content "accepts user input, then it has a name that describes its purpose"[1]. This is normally achieved in HTML5 by the inclusion of `aria` tags; several different types of these tags exist to mark content for screen readers. Most online code editors do not include `aria` tags; this negligence will result in screen readers interpreting the content improperly and potentially reading out invisible elements or not reading out visible ones.

2.3 Research Questions

The project is guided by the following main research questions:
RQ1: To what extent can an existing code editor be integrated into a plugin for H5P using the H5P framework?
RQ2: What considerations must be taken to develop for the H5P platform, and how does that affect feature implementation?
RQ3: What features need to be implemented in an online code editor through H5P to make it accessible to all users?

3 Literature Review

3.1 Existing Code Editors

This section investigates individual existing online code editors to determine their implementation, accessibility, and suitability for use on the H5P platform.

3.1.1 CodeMirror

CodeMirror is a popular fully-featured code editor with support for over 100 languages alongside autocompletion, linting, and many other configurable features. According to CodeMirror's website, over 170 webpages utilise CodeMirror, including *GitHub*'s in-browser edit feature[2].

At the time of writing, the most recent stable version of CodeMirror is CodeMirror 5, which is the version most websites implementing CodeMirror use. CodeMirror 5 does not properly support pressing the `Tab` key to change focus as described in section 2.2.1. As of June 2020, the upcoming CodeMirror 6 entered open beta[3]; this version contains correct accessibility standards including for the `Tab` key, as it is "designed from the start with accessibility in mind"[4].

CodeMirror 6's documentation shows how extensively customisable it is, with the ability to add and remove individual features (such as text highlighting or autocompletion) at any time. This modularity alongside its support for bundling (see section 4.2) makes it a good fit for integration into a H5P module.

CodeMirror 6 was the code editor that was eventually chosen for use in this project. A potential downside when choosing CodeMirror 6 was that it is still in beta; any problems or errors could have been unfixable without further development. This led to some minor issues while developing for accessibility.

3.1.2 EditArea

EditArea is a free JavaScript editor designed to function around an HTML5 `textarea` tag. Some of the supported features include customisable real-time syntax highlighting, auto-indentation, and "possible plugin integration"[5]. However, it doesn't seem to support CSS customisation natively, and performs poorly on large amounts of text.

Furthermore, it does not have screen-reader support, nor does it support any method to shift focus away from the editor, so it cannot be considered accessible.

3.1.3 Ace Editor

The Ace Editor is the functional backbone of the "Cloud9" online collaborative IDE. It is fully-featured and contains support for large documents alongside multiple cursors and live syntax checking [6].

Unlike many other editors investigated in this project, Ace provides a 'built' version of the source code, preventing the user from having to bundle an `npm` package (see section 4.2). This saves on development time and allows for faster updates.

Ace appears to have some screen-reader support insofar as the line numbers and structure are marked not to be read aloud. It, however, does not support pressing `Tab` to change focus, instead adding space characters no matter what.

3.1.4 CodeFlask

CodeFlask is a minimal code editor designed for small code snippets. It is advertised as being good "as [a] code playground"[7], which could fit the premise of a H5P plugin well. Unfortunately, it seems that it has many more drawbacks than features; the site warns against using it for any large-scale purpose or on any older browsers, and suggests *CodeMirror* for "a robust solution".

CodeFlask does not support screen readers, and traps focus inside the editor element, preventing proper accessibility. Therefore, it remains unsuitable for an H5P plugin in its current state.

3.1.5 Monaco Editor

Microsoft's Monaco Editor is the code editor behind the *Visual Studio Code* IDE. Its main feature is its *Rich Intellisense* for several languages, which includes code auto-completion and parameter information, alongside automatic code syntax checking[8].

Monaco Editor has syntax-highlighting support for over 50 languages. It has fully-featured support for screen readers and accessibility, including screen-reader-compatible instructions on the current line and column, alongside information on how to change focus from the code editor to a different element.

For use in the H5P code editor, the Monaco editor appears only somewhat suitable. The process to create support for syntax-highlighting for a language is simple and straight-forward, and supports simple coloring and themes. However, it does not support mobile frameworks or *Rich Intellisense* for C-based languages or Python.

3.2 Designing for H5P

As no code editor plugin exists for H5P currently, this section focuses on approaches on generally producing content for the H5P platform, with the benefits and challenges that come with it.

3.2.1 Visual Design

Investigation and research has been performed to determine what works best aesthetically for H5P content. Wilkie et. al [9] determined that H5P activities should be "engaging, user friendly, [and] visually pleasant"; the information presented should be "easy to read and comprehend". To fulfil this in the H5P code editor, the overall structure would remain relatively simple; the editor itself would display in an easy-to-read monospaced font at a reasonable size. Since the editor would only have information as comprehensible as the code inside it, syntax highlighting would be implemented corresponding to the editor's designated programming language.

Furthermore, to ensure consistency, the editor would have to be styled as close to H5P's existing modules as possible, which should help lower cognitive overload.

3.2.2 Active Participation

In an article entitled *Encouraging students to take action in developing problem-solving competency*, Tran et. al state "Activities that are based on the principle of play can have great potential in supporting students' activity and motivation"[10]. To allow programmers to directly type their code with ease of syntax highlighting would facilitate active learning and participation, so the code editor would somewhat fulfil the principle of play. Alongside the editor, an optional field could be added for creators to add written instructions, to assist in guiding participants.

Ideally, to facilitate active learning, the code would be executable in-browser to show users the results of their programming. This is out of the scope of this project for most programming languages, and would work better in a separate package or module; the code editor should instead support exporting its text to a separate H5P module which may have such functionality.

4 Methodology

4.1 Tools and Software Used

4.1.1 Development Environments

As a fast and simple development environment, Visual Studio Code was used to program and iterate the plugin, but most IDEs could have been used to achieve a similar result.

As H5P exists as dynamic content for learning management solutions, it does not exist in a static context. This means a database of some kind is required to iterate and develop for H5P content. The guides and tutorials for H5P recommend using *Drupal* for development as it allows for faster updates and is less restrictive regarding plugin version numbers; it was therefore selected for use in this project.

4.1.2 Code Editor Base

The fundamental online code editor used for this plugin was *CodeMirror 6*, due to the reasoning presented in section 3.1.1. The most useful feature from the use of CodeMirror was its modularity; the ability to load only syntax highlighting for the plugin's language options as opposed to a larger collection of languages at once drastically lowered the file size of the bundled JavaScript.

4.1.3 H5P Framework

H5P has its own requirements for development; working within these restraints allows for easier modularity for all plugins within the system. Plugins built in H5P do not have their own existing HTML files to serve as a framework; instead, all elements and JavaScript have to be injected into the page later through use of *jQuery* and H5P's internal functions. Development was conducted for this project with this in mind only using the framework provided by H5P. This made testing more complicated as it had to exist in the context of another plugin (see section 4.3).

4.2 Bundling the Code

CodeMirror 6, like most `npm` packages, is a collection of ECMAScript modules. These modules are designed to work as multiple JavaScript files that all depend on each other, making code modular, easy to use, and clearer to read and debug. However, modern browser support for ECMAScript modules is lacking at best; while some modern browsers have theoretical support for modular systems, their support for nested dependencies is questionable. This issue is compounded by the fact that older browsers such as Internet Explorer or old Safari versions have no support for modules entirely.

Bundling describes the process of taking a script and combining it with its dependencies (and, recursively, their dependencies) until it is entirely operable from one individual script that can be loaded by most browsers. This process can be performed by several different bundling software, each with their own advantages. *CodeMirror* recommends use of *Rollup*, and most existing H5P plugins use *Webpack*, but the bundler used in this project was *ParcelJS* due to its simplicity and efficiency compared to other bundlers[11]. To make iteration as smooth as possible, an automated script was set up to copy the bundled code into the correct location whenever changes were made.

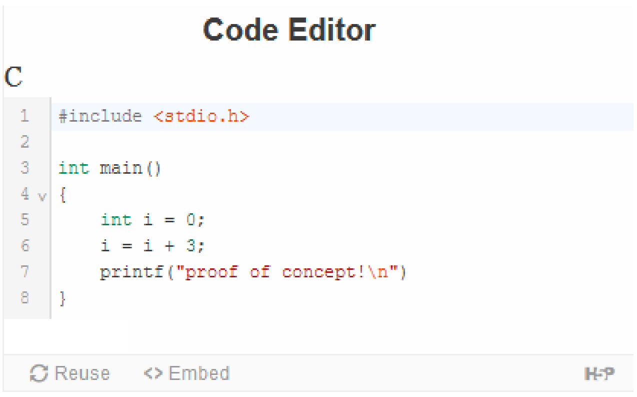
4.3 H5P Widgets

Within the framework of H5P, two layers of content tools are available: widgets and plugins. While similar in appearance, they exist in two different contexts. Plugins exist for content authors to configure and end-users to interact with; they dictate how the content is displayed and what interactions users can provide to it. Widgets, however, exist in the context of content authors only; any widgets themselves are not visible in the finished content, instead being used to assist in its creation. Examples of widgets include color pickers or question editors. Commonly widgets are not fully required for creation of content, but instead make it easier for authors to quickly iterate and create what they need.

The distinction between widgets and plugins is not very clear in H5P's documentation for newer developers, so it can be hard to decide which is better to create for the platform. This problem is amplified by the fact that plugins can also exist as dependencies for other plugins without being widgets; for example, the 'H5P Branching Scenario' plugin creates 'a dynamic course made from other H5P libraries that changes depending on your users' answers.'

When deciding what form this project's code editor should take, the choice was difficult as a code editor could see use as both. Plugin form would allow for authors to combine the editor into a compiler plugin to see and grade the results of users' typed code. However, widget form would allow authors to more easily type code into existing plugins, saving time over a traditional HTML `textarea` element. This project focuses on the creation of a widget rather than a plugin to attempt to make content creation more simple; this was partially how it was eventually tested (see section 5.1).

4.4 Styling


Code Editor

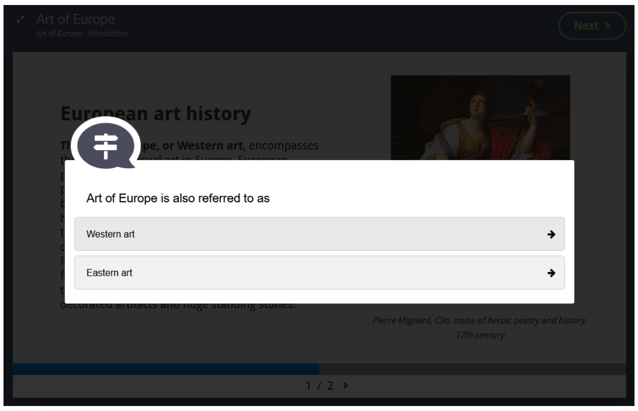
C

```c
#include <stdio.h>

int main()
{
    int i = 0;
    i = i + 3;
    printf("proof of concept!\n")
}
```

Figure 1: The proof of concept used as a baseline for the project.

CSS styling of the code editor began as an iterative process. Initially a proof of concept was derived based simply on addition of CodeMirror to the H5P workspace (see figure 1). As an initial iteration the base CodeMirror editor font was changed to the sans-serif monospaced font Consolas for readability. From there, as the code editor was initially intended to be a full plugin, experiments involved attempting to match the style of H5P's full-plugin elements such as the Branching Scenario (figure 2) or Drag and Drop (figure 3) plugins. These plugins attempted a smoother look, with rounded edges and box shadows for most elements.


Art of Europe
Art of Europe - Introduction
Next »
European art history
Pierre Mignard, Clio, muse of heroic poetry and history, 17th century
1 / 2

Art of Europe is also referred to as

Western art

Eastern art

Figure 2: An example use of the H5P Branching Scenario plugin. All elements in the foreground have some level of rounded corners.

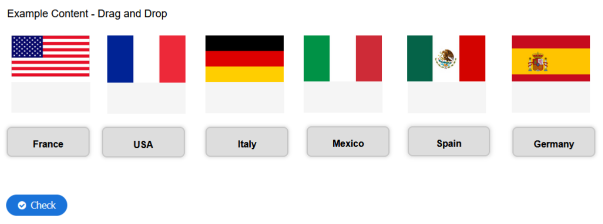


Figure 3: An example use of the H5P Drag and Drop plugin. Note the rounded corners on most elements, and the subtle box shadows on the draggable elements.

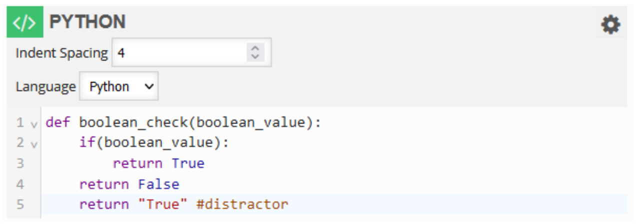
As the scope changed to become a widget (see section 4.3), the styling decisions adapted alongside it, moving towards the style of code blocks on the `H5P.org` website in tutorials and on the forum. This style involved simplistic solid-color design alongside an icon indicating the section contained code. It also displayed the programming language for the code block prominently in capital letters above the code. Recreating this style involved adding a dependency to the code editor widget in the form of the `FontAwesome` plugin, which provided the aforementioned icon. `FontAwesome` was also used to add a settings gear on the top-right of the widget, used to configure language and spacing settings.

With these styling decisions, the baseline style of the editor was finalised (figure 4); additional auxiliary styling involved adjustment of selection dropdowns in the settings menu for ease of use and font size for the CodeMirror editor for readability.

4.5 Challenges

4.5.1 Caching

While developing H5P content, rapid iteration of files is to be expected, updating functionality or visuals. This continuous development works well to ensure that changes do not have to be compiled, and can be apparent visually. However, to be more efficient with



Indent Spacing 4

Language Python

```
def boolean_check(boolean_value):
    if(boolean_value):
        return True
    return False
    return "True" #distractor
```

Figure 4: The final styling for the H5P.CodeEditor widget.

data transfer, most modern browsers cache content such as scripts or stylesheets, which can be detrimental towards the development process. When working on script functionality, it can be hard to determine when caching has occurred, as it is not immediately apparent visually. This can lead to the code in a developer's IDE mismatching what appears on the website. In extreme cases, HTTP redirects can be preserved, locking out access from a website until the browser's cache is cleared.

Caching issues were numerous and caused many issues while developing the H5P Code Editor widget. Most browsers have workarounds and toggles to prevent caching while developing, but these features are usually not very user-friendly or not typically enabled by default; for example, to prevent caching in Firefox, it has to be enabled in the developer console, and then subsequent refreshes must have the console open to prevent a cache for that specific site.

4.5.2 Documentation / Creation Guides

While H5P systems are simple to understand superficially from the tutorials provided on the H5P.org website, the creation of more complex content can be difficult for developers as the necessary guides are usually hard to find or missing. This caused issues with the creation of the code editor as guides for the creation of widgets only extend to a simple text output, exposing only a singular parameter rather than the three required for this project.

Furthermore, there seems to be very little in terms of accessible documentation. While the H5P developers have an automatic generator working to create some minimal documentation based on code comments, it is not only hard to use due to its lack of detail but also nearly unnavigable as there is no search function.

Some functionality can be gleaned from elsewhere, however; H5P hosts most of its existing plugins' code on their public GitHub organisation. This allows newer developers to look through the code present in other widgets and plugins and discern the functionality required for their own contributions. Discovery and experimentation through these existing repositories is effective for some code questions, but has its own limitations; many of the plugins present are not consistent with each other, having only small programming similarities.

5 Experimental Setup

5.1 Testing

5.1.1 Validation Testing

As part of the widget creation process, the core H5P code requires any input to be validated before being passed through to a plugin. For some widget instances this can mean a range of validation; for example, checking the input is in a correct range, testing if a color string is valid, or ensuring an email address has the correct format. For this project, there was very little required in terms of validation testing; the only main requirement to pass was that the code formatted was output as a string. Any further validation becomes unnecessary; for example, testing for code compilation success would be not only out of this scope in terms of its widget nature but also not helpful in comparison to the amount of effort undertaken to add it. Furthermore, any validation that requires testing of a programming language would add vast overheads in the form of generalisation for every language included in the widget itself.

5.1.2 Individual Testing

To test the widget, a small sample plugin based on the H5P tutorial 'Greeting card' plugin was created. The sample plugin would take input from the code editor widget and output it to the end-user. Despite having no core interactivity, it was useful in testing whether the parameters were being communicated from the widget to the main content.

5.1.3 Testing in Other Plugins

The H5P.CodeEditor widget created in this project was implemented in a way that makes it reasonably open for extension and use in larger H5P plugins. To this end, three parameters were exposed; the code itself, the language selected in the widget, and the indentation style in terms of spacing. This allowed any plugin using the widget to use these parameters if necessary.

An example of the code editor widget being implemented into a plugin is `H5P.ParsonsPuzzle`, which uses it to help content authors create interactive Parsons puzzles for students. The plugin uses the code editor as an input method, which allows it to be more flexible and extensible than a simple HTML5 `textarea`. Integration towards this plugin was an effective way of testing what the code editor itself needed, and this drove the need to expose the further parameters as required. At time of writing, the plugin has successfully integrated use of the widget, demonstrating successful development [12].

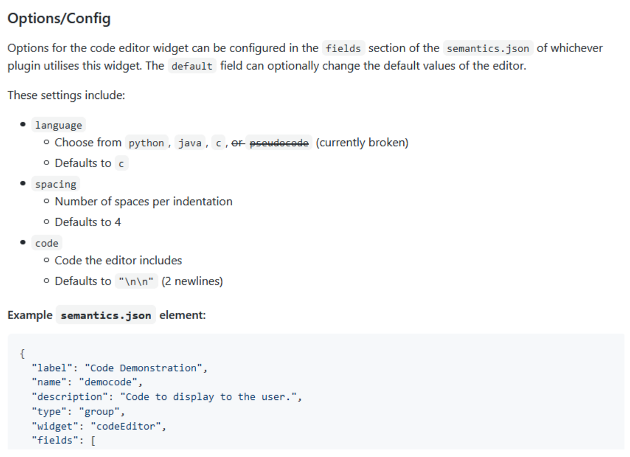
6 Results

6.1 Functionality

Based on the testing outlined, the code editor widget produced is functional, and passes the simple validation testing. It successfully matches the styling of similar H5P content in the correct context of a widget and is extensible enough to be a contribution towards the creation of any other H5P content if desired.

6.2 Documentation

To assist with ease of use, documentation for the widget was written covering the syntax, setup, and dependencies required to incorporate the code editor into an existing plugin's creation system. Specifically, the configuration options regarding placeholder code, indent spacing, and default programming language are all displayed. This documentation is saved as a `README` file in the GitHub repository for the widget (see figure 5).



Options/Config

Options for the code editor widget can be configured in the `fields` section of the `semantics.json` of whichever plugin utilises this widget. The `default` field can optionally change the default values of the editor.

These settings include:

- `language`
  - Choose from `python`, `java`, `c`, ~~or~~ ~~`pseudocode`~~ (currently broken)
  - Defaults to `c`
- `spacing`
  - Number of spaces per indentation
  - Defaults to 4
- `code`
  - Code the editor includes
  - Defaults to `"\n\n"` (2 newlines)

Example `semantics.json` element:

```
{
  "label": "Code Demonstration",
  "name": "democode",
  "description": "Code to display to the user.",
  "type": "group",
  "widget": "codeEditor",
  "fields": [
```

Figure 5: A portion of the README documentation available on the GitHub repository for this project.

6.3 Research Questions

The results from this project somewhat successfully answer the research questions presented in section 2.3. This section answers each individual question formally.

In terms of integrating an existing code editor into the H5P framework, the process is straightforward and simple. H5P accepts integration of other NPM code through the use of bundling, so through CodeMirror 6 the integration experience was rather simple. Through the process undertaken in this project, where the bundled code and main integration were performed in two different contexts, the process was made more difficult through the need to include functions in the global namespace; this is not mandatory, however, as both contexts can and should be combined for a smoother and cleaner programming experience (see section 8.2.1).

Considerations made as part of the development process for the H5P platform primarily comprised of the decision between widget and core plugin functionality. This affected feature implementation through scope, users' input methods in terms of configuration, core target audience, and main styling and aesthetics. Through this singular decision, while the results may have performed a similar purpose, they would have been different in their functionality as part of the H5P context.

Accessibility was a core feature of this project as mentioned in the third research question; important steps were taken throughout the process of creating the widget to keep features accessible where possible. These features aimed to comply with the Web Content Accessibility Guidelines, mainly in terms of keyboard control and screen reader access. While these accessibility ideals were not fully realised through this project, the core functionalities of them still exist, and the main research undertaken as part of this project's motivation demonstrates the need for more accessible features throughout all Web content in future.

6.4 Limitations/Issues

6.4.1 Scope

As the code editor was developed with a small scope, it is missing many critical features that may make it extensible to more than just the small number of H5P plugins it was tested in. Primarily, there are only three programming languages present: Java, Python, and C; while these languages may cover a wide range of required needs, it is far from acceptable to call the widget generalised in this case.

Furthermore, while the code editor does expose parameters for use in other plugins, it only exposes three; more information as part of the widget such as bracket styling or total line and character count is not accessible and would have to be discovered by analysing the code block output itself.

6.4.2 Accessibility

One of this project's core research questions involves accessibility for all users, and while the final result technically fits this description, further development is required to fully achieve accessibility. Currently no dark theme is implemented, which can cause eye strain and other sight-related issues with black-on-white text. Furthermore, while Tab support is functional, the functionality is not described anywhere in a way that screen readers could access; this severely limits the accessibility since it is still possible for users to get stuck in the code editor element without any understanding on how to perform the escape sequence.

6.4.3 CodeMirror 6

CodeMirror 6's beta status as outlined in section 3.1.1 caused some limitations during this project. These limitations included inconsistency with some accessibility features; in particular, the Tab escape would rarely inconsistently cease functionality over different sessions for inexplicable reasons. For the majority of the H5P userbase this should not cause major issues as it was incredibly infrequent, but further investigation will be required into the exact cause of the bug.

7 GitHub Repository Access

The GitHub repository for the project can be found at `https://github.com/uofa-Parsons-hub/H5P.CodeEditor`.

8 Conclusion

8.1 Project Changes

When attempting a project similar to this, the first consideration taken should be consideration of target audience; whether to create for content developers or end-users will determine whether a plugin or widget is created. In this project this decision was delayed due to lack of clarification regarding the benefits and differences between widgets and core plugins; this indecision lasted until approximately midway through the project's timeframe, which led to wasted development time and effort. In future, this decision should be made first and foremost, as it shapes what form of development will be undertaken. Steps that can be taken to answer the 'widget or plugin' question could include investigation as to what the actual target audience of the content would be, whether that is content authors or the users themselves.

8.2 Future Work

8.2.1 Bundling Process

While developing the plugin, the bundling process was performed separately, and the bundled code would be copied over through an automated script into the repository whenever updated. This process made development smoother and more reliable; however, this process is normally not performed ahead of time for most H5P projects. Instead, configuration and package files are added to the repository so that any developer branching the code can build and bundle the code locally. The process used for this project is not necessarily wrong, as the license for CodeMirror permits distribution under the same license, but in future this process should be changed to conform to the H5P standard.

8.2.2 Accessibility

As mentioned in section 6.4.2, the current version of the widget contains subpar optimisation in terms of dark theming or screen reader explanations. To fix the theme issue, an option could potentially be included in the settings menu to change the colouring; CodeMirror supports changing colour themes dynamically so this potentially would not result in much more development time. To fix the poor explanation of the Tab functionality, a small label readable by screen readers could be added which describes the correct escape method.

8.2.3 Documentation Contribution

To alleviate some of the challenges in terms of documentation and guides mentioned in section 4.5.2, future work could entail writing more expansive guides for the development of H5P content. Primarily this could include widgets that do not return the 'text' type; changing the return type of a widget is simple once understood but hard to discern without prior knowledge. To this end, a lot of H5P's existing documentation is superficial at best; while the entry-level creation tutorials work well for their purpose, any aspiring developer seeking to create more complex projects have to resort to reverse engineering. Because of this, more work could be undertaken to add to the existing guide content, potentially preventing the same pitfalls that arose during this project.

8.3 Reflection

While there are many more additions that could be made towards the widget as a whole, in general the project provided experience into the creation and implementation of open-source content as a piece of a larger codebase. Personally the process taught a lot about how H5P works and why it as a system is so beneficial to learning management solutions, but also how H5P suffers from a lack of full documentation. Future attempts to create H5P content will retain full context of the process and decisions required for successful development, including the choice for widget versus plugin creation.

References

[1] Michael Cooper et al. *Web Content Accessibility Guidelines (WCAG) 2.1.* W3C Recommendation. W3C, 2018. URL: `https://www.w3.org/TR/2018/REC-WCAG21-20180605/` (visited on 2021-03-23).

[2] Marijn Haverbeke. *CodeMirror: Real-world Uses.* 2021. URL: `https://codemirror.net/doc/realworld.html` (visited on 2021-03-21).

[3] Marijn Haverbeke. *CodeMirror 6 Enters Beta.* June 29, 2020. URL: `https://marijnhaverbeke.nl/blog/codemirror-6-beta.html`.

[4] Marijn Haverbeke. *CodeMirror 6.* 2021. URL: `https://codemirror.net/6/` (visited on 2021-03-20).

[5] Christophe Dolivet. *EditArea.* 2010. URL: `https://www.cdolivet.com/editarea/` (visited on 2021-03-21).

[6] ajax.org. *Ace - The High Perfomance Code Editor for the Web.* 2021. URL: `https://ace.c9.io/` (visited on 2021-03-23).

[7] Claudio Holanda. *CodeFlask - A micro code-editor for awesome web pages.* URL: `https://kazzkiq.github.io/CodeFlask/` (visited on 2021-03-21).

[8] Microsoft. *Monaco Editor.* 2021. URL: `https://microsoft.github.io/monaco-editor/` (visited on 2021-03-23).

[9] Sonia Wilkie et al. "Considerations for designing H5P online interactive activities". In: *Open Oceans: Learning without borders. Proceedings ASCILITE* (2018), pp. 543–549.

[10] D. Tran, T. Havlásková, and Z. Homanová. “Encouraging students to take action in developing problem-solving competency”. In: *2019 17th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2019, pp. 770–776. DOI: 10.1109/ICETA48886.2019.9039967.

[11] Devon Govett. *Parcel*. 2021. URL: https://parceljs.org/ (visited on 2021-03-23).

[12] Cheryl Pope, zhaoia, and wxw-matt. *H5P.ParsonsPuzzle*. 2021. URL: https://github.com/uofa-Parsons-hub/H5P.ParsonsPuzzle (visited on 2021-06-11).