logo2

Content Authoring

Learning Management System

Revision History

|  |  |  |
| --- | --- | --- |
| Date | Author | Change Log |
| 11/07/2016 | Kevin Carroll | Initial Creation |
| 11/08/2016 | Kevin Carroll | Changes to content/lesson repository structuring and other minor edits on grading. |
| 11/14/2016 | Kevin Carroll | Added formats for “per answer” feedback on questions. |
| 12/01/2016 | Kevin Carroll | Coding Assignment definition completed. Added new activity type, web project…tbd. |
| 01/15/2017 | Kevin Carroll | Added “to-do” tag documentation |
| 3/22/2017 | Kevin Carroll | Updated max-attempts, added “note” tag |
| 6/21/2017 | Kevin Carroll | Updates for 1.4 restructure |

Contents

[Overview 3](#_Toc469318211)

[Building Your Lesson Content 3](#_Toc469318212)

[Why not Word? 3](#_Toc469318213)

[Why Markdown? 4](#_Toc469318214)

[Automagical Modular Lesson Structuring (AMLS) 4](#_Toc469318215)

[Content Types 6](#_Toc469318216)

[Corollary Blocks 6](#_Toc469318217)

[Block Opening Line 7](#_Toc469318218)

[Block Closing Line 7](#_Toc469318219)

[C-Block Keys 7](#_Toc469318220)

[.lms File Format 7](#_Toc469318221)

[Lesson Creation 8](#_Toc469318222)

[Instructions 8](#_Toc469318223)

[Instructional Topics 9](#_Toc469318224)

[Quizzes & Exams 9](#_Toc469318225)

[Projects 10](#_Toc469318226)

[Coding Assignments 10](#_Toc469318227)

[Special considerations for authoring a coding exercise 10](#_Toc469318228)

[Grading a coding exercise 11](#_Toc469318229)

[Web Exercise/Project 15](#_Toc469318230)

[activity-type: web-project 15](#_Toc469318231)

[What happens when a student “runs” their web project? 16](#_Toc469318232)

[How your content makes it to the Student 16](#_Toc469318233)

[Content Authoring Stylistic Guidelines 16](#_Toc469318234)

[Appendix 17](#_Toc469318235)

[Virtual Classrooms Layouts 17](#_Toc469318236)

[Activity C-Block Keys 17](#_Toc469318237)

# Overview

Codercamps’ LMS content authoring is based largely around the markdown content authoring standard and supports all tags and phrases codified in the [Common Mark](http://commonmark.org/help/) specification. The LMS supports any markdown editor on the market and does not contain any specific or special markdown extensions. This is not to say that there are not special call outs or corollary action definitions to instruction the LMS on how to navigate a lesson or activity.

All corollary actions are contained within a corollary action block (aka c-block), making using of the standard triple backtick syntax of markdown to delineate a code block. The rest of this document will focus on the definition of LMS content and how that content relates to the corollary action blocks when it is scanned by the LMS.

# Building Your Lesson Content

A lesson is a loosely associated collection of lectures (instructions) and a series of activities (quizzes­­, assignments etc.) related to a group of common topics. How you organize your lesson content is up to you. Every lesson, however; contains some common elements to allow you to control its presentation. You, the author, can specify everything about the layout of the information on the screen. Perhaps you want a video to show in the top left corner along with readable text on the right-hand side and a navigation pan in the lower left. Maybe you are writing a coding exercise that needs a preview pane and an edit window along with instructions on what to do. These actions and directions are coded in markdown in a special code section called a “c-block.” (The ‘c’ being short for corollary). Most of your content is going to be text based; be that the main text of the lecture topic, quiz questions or project definitions. Instead of using the overly verbose HTML standard for content the Coder Camps LMS uses mark down.

## Why not Word?

Put simply, not everyone has access to Microsoft Word. Some use libreoffice or open office and many mac users prefer Pages. The markdown specification is compatible with every operating system since it is purely text based doesn’t suffer from information loss inherit with converting between multiple versions of Word or between different office document editing programs. Also, you can easily edit a markdown file on a tablet or mobile device without the need of a complicated editor.

## Why Markdown?

Brett Terpstra, in one of his [blog post](http://brettterpstra.com/2011/08/31/why-markdown-a-two-minute-explanation/)s, provides the best explanation on why using Markdown is a better option than HTML.

**It’s easy:** the syntax is so simple you can barely call it “syntax.” If you can use an emoticon, you can write Markdown.

**It’s clean:** Markdown translates quickly to perfectly-formed HTML. No missing closing tags, no improperly nested tags, no blocks left without containers. You also get 100% less cruft than exporting HTML from Microsoft Word. There’s no styling inline, nothing that will otherwise break a site’s design or mess with the XSLT formatting for PDF output. In short, it’s foolproof.

**It’s portable:** your documents are cross-platform by nature. You can edit them in any text-capable application on any operating system. Transporting files requires no zipping or archiving, and the filesize is as small as it can possibly get.

**It’s flexible:** output your documents to a wide array of formats. Convert to HTML for posting on the web, rich text for sending emails or importing into a layout program for final arrangement or any number of other proprietary formats.

**It fits any workflow:** You can make Markdown work with any workflow. It can speed up just about any writing-related process with very little setup. It can also be scripted all to hell, if you want, because plain text is the most flexible of any format known to computer-kind.

# Modular Content Structuring

The LMS supports a multi-tier hierarchy to break-down a complex set of course work. You can divide your course into as many learning modules (a.k.a. Units) as you need, each with their own set of lessons. Unit names (for purposes of reporting) can be changed via an “.lms” file, which we’ll explain in more detail later in this document. The following example represents the typical layout of our content for all of the course work Coder Camps offers.

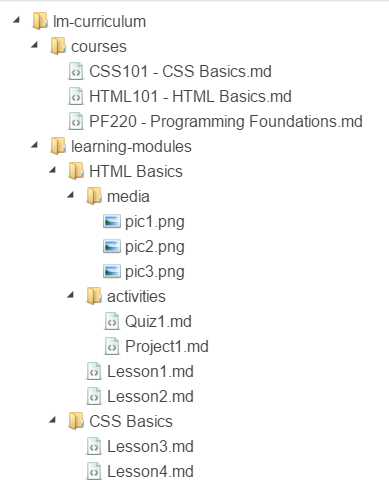


Fig. An example file structure of authored learning content

This file tree can be read as: There exists two modules named “HTML Basics”, “CSS Basics”. Within the HTML Basics Module, there exists two sets of lecture topics, called instructions; called “lesson1” and “Lesson2”. Those two instructions also reference three media files (the three pictures) within their content. There also exist 2 different activities: a quiz and a project. How those activities are presented is determined in the c-block sections of the instructions.

There also exist 3 courses named CSS101, HTML101, and PF220. Those courses define the modules (from the learning-modules folder) that make up each. For the most part a course is simply a collection of N modules of work in a given order.

#### On Content Ordering

A numeric order on activity files in a module can be used by convention, it’s not required (“01 – Lesson1.md”, “02 – Lesson2.md” etc.). You can specify content ordering in one or more “.lms” files in the module or via a rank key in the c-blocks if you desire, though said ordering would have to be updated every time you add or change a file name. If not explicitly declared the LMS will automatically use alphabetization as the content presentation order and will strip the numeric indicators so they are not displayed to the students or used in reports.

#### Activity and Media Folders

The folder names *Activities* and *Media* are reserved to allow for separation of interactive activities (non-instruction based activities) and media files used to deliver content. These are merely a convenience and are not required. All activities and media can be lumped together in the single module/unit folder if desired.

# Content Types

The contents of each unit tier can have the following types of content, determined by the file extensions. A content type determines how the LMS will handle the file; be it a markdown file (activities, projects or quizzes) or piece of content to serve as part of a lesson, as is the case with videos, pictures or another media.

|  |  |  |
| --- | --- | --- |
| Content Type | Supported Extensions | Description |
| Media | *Video*: .mp4, .ogg, .webm  *Images*: .jpg, .png, .gif | Media files are passed through the LMS and served as content to the browser. No processing is done to them. |
| Lesson Content | .md | Markdown files are processed by the LMS, line by line. C-blocks are converted into handling instructions and regular markdown (content) is converted into HTML and served to the browser. |
| Styles | .css | Advanced content authors may wish to include their own CSS for controlling the display of various elements. Custom CSS files can be added and applied to the HTML specified via the [commonmark specification](http://spec.commonmark.org/0.26/) (v0.26). |
| Configuration | .lms | .lms files contain processing instructions for building a module of a course from the content structure. These files are converted into handling instructions much like c-blocks. They are not copied to the content that is served to the student. |

Table: Supported content types and file extensions.

**Files with extensions not listed here are ignored.**

#### On Internally Hosted Videos

Any videos should be rendered in three different formats (mp4, ogg and webm). This is to support cross-browser video streaming standards implemented by the various browsers. Failure to supply all three formats could result in reduced availability of your lesson to some students. The LMS will give you a warning if you do not supply all three formats, but will not stop your lesson creation.

#### On Externally Hosted Media

Feel free to use any externally hosted media within your markdown. The LMS will happily render it per the rules for that specific content type. This is especially encouraged for video and may require some additional support for use of a content delivery network (CDN) for faster loading times for our students.

# Corollary Blocks

A corollary block is a series of keys and values to allow you to configure and script the delivery of an activity. These blocks will typically look like this:

```c-lms

activity-name: Intro to HTML

activity-type: instruction  
topic: Tags and Attributes  
classroom: three-panel-right

```

Fig. This is an example c-block, note the opening and closing triple-backtick syntax with the key phrase “c-lms” to specify the block. Other similar blocks not including the “c-lms” specifier will be ignored and parsed as standard markdown code segments.

### Block Opening Line

The opening tag for a c-block must be the exact phrase ```c-lms on a single line. That is three (3) backticks followed by the phrase “c-lms” with no whitespace or text preceding or in the middle of the phrase. The rest of the line must be empty or can be a variable length of whitespace if necessary. The line is terminated by a carriage return/new line. The “backtick” is the key in the upper left corner of the American standard qwerty keyboard, below the escape key. It is typically shared by the tilde (~). The “c-lms” phrase indicates this block to be a corollary block directive within the LMS. Failing to include the “c-lms” segment will cause this block to be treated like any other formatted code block that the markdown syntax can produce. It will be outputted to the screen verbatim.

### Block Closing Line

The closing tag for the c-block must be the exact phrase ``` on a single line. That is three (3) backticks with no text or whitespace preceding or in the middle of the phrase. The rest of the line must be empty or can be a variable length of whitespace if necessary. The line is terminated by a carriage return/new line.

## C-Block Keys

See Appendix B for a list of key names. See the section on Lesson Creation for usage.

## .lms File Format

Like c-blocks, .lms files contain key/value pairs identifying key meta-information about the module/unit where the file exists. The following keys exist for lms files.

* lesson-order**:** a comma separated list of instructional activities specifying the order in which the material should be presented. Activity naming rules apply and the final name of the instruction (be it file name or c-block key) should be used.
* unit-name: Purely cosmetic in nature, the unit name applies custom naming to the unit. If omitted the folder name is used as the Unit Name.
* description: Adds a quick blurb about this unit to the user. Generally, 500 characters or less, no markdown.
* activity-weight: determines what percentage each of the activity types counts toward the overall grade for the unit.

# Lesson Creation

A lesson is a nebulous collection of activities designed to work together to deliver related information of educational value. The combination of instructions and activities and how they are organized or presented is up to you as the author. Currently lessons can contain the following types of activities:

* **Instruction (required)**: A lecture style presentation where the information is given to the user with minimal interaction. The user will click an “ok” or “next” button at the end of, or in the middle of, the instruction to advance to the next activity. Lessons are constructed based off the instructional activities and as such all lessons are required to have at least one instructional component.
* **Coding Assignment**: A coding assignment allows the student to perform one or more live coding exercises in which feedback can be given, relative to their answer. This is for compiled and interpreted language exercises such as c#, java etc. Use the web project activity for a more robust “browser-esque” experience.
* **Web Exercise/Project:** A web project is a collection of files served across the internet…as a website. Files are authored and evaluated similarly to a coding assignment, though not nearly as deeply. They can only be checked syntactically via automated testing. Web projects can, optionally be persisted beyond the activity to have a lifespan if necessary.
* **Quiz**: A collection of multiple choice questions that receive a grade. You, as the content author, can specify a required grade to advance to the next activity.
* **Exam**: Like a quiz in functionality except it carries more weight and serves as a major metric for a module or unit. There can only be one exam per unit of content. Unlike quizzes, exams can contain coding exercises as graded questions. Via the “start-activity” key you can place your exam anywhere within your lesson structure, however; if not explicitly referenced, the exam will be given after the end of the last activity of the last lesson.
* **Project**: A project represents a completed body of work by the student. It is presented as an option to upload a zip file containing their work (typically source code). This zip file is stored for the instructor to review. Like an exam there can be only one project per module or unit of content.

## Instructions

An instruction is the “teaching” part of a lesson. Instructions are used to convey information in a one-way “teacher speaking to the student” delivery method. All instructions are delivered via a “virtual classroom”, represented by the browser. You, as the content author, can choose from a few virtual classroom layouts and change between them when you change instruction topics. (*see appendix A for available virtual classroom layouts*). Unless otherwise specified the title of the first instruction is used as the “lesson name” in reports, syllabi and other metrics. All lessons begin with an instruction and unless specified all instructions represent individual lessons within a module or unit.

### Instructional Topics

Topics are sub divisions of a lecture. If I am viewing the “HTML Page Structure” instruction, it’s likely that there are two or three topics that I care about being covered (overview, tags and attributes for example). It is reasonable to assume that there are different videos, text or slides for each topic or, if purely text based, that the user must interact with the screen to advance to the next topic. (i.e. “click next to continue”).

Sample markdown file for an instructional activity:



## Quizzes & Exams

A series of related questions the student must answer. These questions are presented one at a time with a running tally of questions completed vs. questions remaining. After answering the student is optionally shown a to congratulate them on being correct or offer feedback if they were incorrect. Feedback can be specified based on which question response they choose or a general “error” feedback if the right answer was not chosen. You may also choose to give no immediate feedback and present the grade and feedback at the end of the activity.

#### Grading

Each quiz or exam allows the author to specify a required score to continue, in the form of a simple percentage of correctness (correct divided by total).

In the event a student does not pass an exam or quiz on their first attempt, they are permitted to retry as many times as you specify. After they have exhausted their allowed attempts without passing the student can continue with the next activity unhindered.

[(Jake, UI/UX) Picture of a quiz question]

#### Multiple Choice Question Format

A multiple-choice question should be bounded by a blank line on the top/bottom of the question body and begin with no special formatting (headers, quotes, bolded text etc.). Once began, the question text may contain any markdown necessary ending with an unordered list (using a dash, ‘-‘) representing the multiple-choice options. A question should contain no blank lines and may optionally begin with a parenthetical number representing the points assigned to the question. When omitted a question is defaulted to one point. Surround the answer(s) in a double star/asterix (\*\*) to indicate it is the correct one.

Optional, customized feedback is provided on a “per answer” basis. A sublist, using a single asterix and a space(‘\* ‘), should be created for the feedback to be shown on each answer. The feedback will be displayed if they submit the answer as their chosen selection. When the feedback is shown to the user is based on the c-block declarations of the quiz itself (see ‘show-feedback’ and ‘immediate-feedback’ tags in the appendix).

Example Quiz Markdown File:

#### 

#### Coding Assignment Questions

Exams may include coding assignments. Given the complexity of a coding assignment a c-block call out using the “start-activity” key should be injected into the exam document at the point the coding assignment should appear.

## Projects

A project is simply a place where a student can upload one or more files for their instructor to grade. No grading is done online (for now) and you, the author, can choose to allow the student to skip the project or require at least one file be uploaded before continuing. Once graded, the results become part of the students completed curriculum and final grade. A project can include a lengthy detail of media and instructions detailing the assignment. It cannot, however; contain multiple instruction topics or video related timing marks.

[(Jake, UI/UX) Picture of typical Project Assignment]

## Coding Assignments

These live exercises allow students to code online in the browser without the need for a separate IDE. A coding assignment is displayed in three sections: the editor, compilation results/output and instructional prompt(s). The editor can be prepopulated with multiple files or delivered blank (depending on the assignment) to allow the student to write code from scratch. You can configure it to allow augmentation of existing code with new functionality or correct a bug. Theoretically, instructional prompts can contain any markdown text of any length.

A coding assignment must contain at least one exercise but can contain more. Multiple exercises are independent and unrelated but will be grouped together in a “wizard” style format advancing from exercise to exercise in a progressive order as they are answered correctly.

Each coding exercise in an assignment is delimited by a change in topic, much like an instruction. (due to the complexity of coding exercises the system is unable to automatically delineate topics like it can with multiple choice quizzes). The content between the topic declaration and the first file declaration is considered the instructional prompt. Once the first file declaration is made (a c-block with a file-name and file-lanaguage tag) all subsequent content sections up to the next topic are considered either files to be displayed in a tabbed document viewer for editing or feedback and grading criteria in response to an output.

### Special considerations for authoring a coding exercise

As an author of the assignment, it’s your responsibility to decide what areas of the code are editable and what needs to be graded (these are not necessarily related). This might include prompting the student with some pre-existing code (perhaps an outline of a class or a program.cs with a void main()) and allowing them to fill out the body of a function, or giving them a class and forcing them to implement an interface or give them an entirely written program with full reign to change anything to produce an expected output.

Imagine a tabbed editor in an IDE, you may have a single code file or multiple files open at any given time. It makes sense to provide the same type of experience to the student. After the prompt for the activity begin defining your “files” that should be included in the project one by one using a c-block tag:

```c-lms

file-name: program.cs

file-language: csharp

```

The ‘file-name’ is what will be shown to the student in the tabbed editor (exercises with one file will not display the file name directly). It is also sometimes used by the compiler when returning compilation results, so make sure the names are meaningful. The rest of the content should be the code of the file that you want to present to the student. By default the student is unable to edit a line of code; precede the line with an octothorpe (#) to allow editing. The following defines a c# file with a class and function definition that gives the students 3 editable lines to fill in their solution. If you wish the student to be able to edit a line of code you supply (to correct a bug for instance), include the octothorpe on that line.

```c-lms

file-name: greeting.cs

file-language: csharp

```

namespace CC.LMS.Exercise1

{

using System;

public class Greeting

{

public void SayHello()

{

#

#

#

}

}

}

### Grading a coding exercise

Grading a coding assignment can be managed in multiple ways depending on your goals for the assignment. You may only care of the output of the application or you may care, syntactically, about the user’s input. The level of detail in grading/acceptance criteria is up to you. Let’s assume the above file has a prompt as follows:

“Write some code so that the function SayHello writes ‘Hello World!’ to the console.”

#### Syntax checking

Sometimes, especially early in the learning process, it makes sense to check the actual syntax and lettering of a student’s work. This can be difficult to accurately grade and is often pedantic. Its best for some small exercises where the student’s written code is, at most, one or two lines. Using the above assignment, you can define a syntax test in two ways:

* **Hash** – A hash perfect representation of the provided answer code, any deviations will fail the test case. The hash test *is case sensitive* but is *not whitespace sensitive.* The hashing algorithm will remove all whitespace (tabs, carriage returns, spaces, new lines etc.) from both the student’s answer and your provided standard before calculating the hash result.
* **Regex –** A regular expression against the student’s completed code (edited lines and supplied lines) to provide more targeted feedback. This is useful for checking variable naming, use of correct functions or casing of function names.

Defining syntax tests looks like the following:

```c-lms

test-id: HelloWorldSyntax

test-type: syntax

test-file: greeting.cs

``

namespace CC.LMS.Exercise1

{

using System;

public class Greeting

{

public void SayHello()

{

Console.WriteLine("Hello World!");

}

}

}

```c-lms

test: HelloWorldSyntax

test-method: ^.\*Console.WriteLine.\*$

test-passed: true

```

Good use of Console.WriteLine

```c-lms

test: HelloWorldSyntax

test-method: ^.\*[Cc]onsole.[Ww]rite[Ll]ine.\*$

test-passed: false

```

Double check your capitalziation, it matters a lot!

```c-lms

test: HelloWorldSyntax

test-method: ^.\*cout.\*$

test-passed: false

```

CSharp is different than C++, cout is not a real thing, we use Console.WriteLine instead.

```c-lms

test: HelloWorldSyntax

test-method: hash-no-match

test-passed: false

```

Woops, You dun screwed up.

The test starts with a declaration of the test-id, which is then referenced for the conditions in the test and a test-file to indicate which of the student’s source files is being tested. You then, optionally, declare an exact representation of the file if you wish to perform letter perfect hash matching. After the declaration comes the test conditions. Each condition’s test-method attribute is evaluated, in order, until a match is found. If a match is found, the test is considered passed or failed per the test-passed tag. Only one test condition, the first match found, is evaluated per unit test.

The test-method has two special cases for hash matching (hash-match, hash-no-match). Hashing to check a student’s answer is always a binary operation, either it matches or it doesn’t. Thus, you need to decide on what your test case is accomplishing. If the answer they provided is just like your provided standard does that mean they did it correctly or are you testing for an invalid condition?

#### Unit Testing (Code Execution)

You can unit test the student’s code at runtime by supplying the outlines of a program that will execute the code they wrote. Using the same exercise defined earlier, you can develop a unit test that looks like the following:

```c-lms

test-id: HelloWorldOutput

test-type: unit

file-name: program.cs

file-language: csharp

```

namespace CC.LMS.Exercise1

{

using System;

public class Program

{

public static void Main(string[] args)

{

var quickGreeting = new Greeting();

quickGreeting.SayHello();

}

}

}

```c-lms

test: HelloWorldOutput

test-stream: stdout

test-method: ^[Hh]ello World\n$

test-passed: false

```

Close, double check the instructions. Programing is all about details, check your casing? Did you include a captial H? Does your output

end in an exclamation point?

```c-lms

test: HelloWorldOutput

test-stream: compiler

test-method: ^.\*CS1002.\*$

test-passed: false

```

Ouch, looks like your program failed to compile. Double check that each line of code ends with a semi-colon.

```c-lms

test: HelloWorldOutput

test-stream: stdout

test-method: ^Hello World\n$

test-passed: true

```

Perfect! You wrote the code that produced the correct answer. Continue on to the next question.

In the unit test itself, we instantiate the students code and call the function they wrote. The unit test file is shipped, along with the student’s greeting.cs file to the compiler for compilation and execution. The compiler returns both the compilation errors, if any, as well as the stdout stream. In this c# example that’s anything written to Console.

Once executed, the results of the unit test are evaluated against N number of test cases. test-stream indicates which of the two response streams we want to test against. Just like with a syntax test, test-method represents a regex declaration used to evaluate against the chosen stream. Typically, you will inspect one or other as a program that doesn’t compile typically won’t print anything to stdout. There are potentially more complicated scenarios where an unhandled exception or a stack overflow are found at runtime resulting in some potential output to stdout as well as system messages such as invalid termination codes.

In the above unit test we offer 3 different targeted feedback options, one congratulates them when the output is exactly correct to what we want, one warns them of the importance of detail and that their output is close but not quite right and the last looks for a specific compiler error ([CS1002](https://msdn.microsoft.com/en-us/library/8b47kkwz.aspx)) and gives them targeted feedback to make sure they include semi-colons on their lines of code.

The LMS will make use of [repl.it](https://repl.it/) to compile and execute compiled and interpreted languages. You can preview the potential output passed to the LMS on their website. [Here is the implementation of the C# compiler](https://repl.it/languages/csharp) we’ll be using.

#### Execution and Grading through Unit Tests

Unit and syntax tests are engineered in an “assumed success” manner, meaning anything that is not a direct failure is considered passed and correct. If the student’s input (or output) does not match any of your test methods in your written test case, the test is considered passed and the student can advance. Each unit test will be allowed to run and the results of each test are aggregated via a logical AND operation. For each test that does provide feedback, that feedback is shown to the student. This means some tests could show “successful” feedback and others show “failure” feedback and that feedback shown together.

A completed coding activity with one exercise could be implemented as follows:



## Web Exercise/Project

Web projects are nebulous arrangements of files served across the internet from a storage medium. Implementation wise this is URL within the coder camps ecosystem and publicly available (i.e. http://{subdomain}.codercamps.com/{username}/{project}/). Where {project} is the activity-name of the authored web project. Students will edit their files in the browser, using the same IDE/editor as the coding assignments. When a student “runs” their website project you may execute syntax tests before allowing the files to be accepted. Once accepted these files are persisted to the web server and served back to the student, the output pane in this context is an iframe pointing to the file location for the activity.

A web project is two distinct activity types, the difference lies in the intent:

* **web-exercise** – A short, isolated set of exercises, used for teaching html or CSS basics. The contents of the exercise are hosted for usage but once the student exits the exercise the contents are immediately deleted. This is the “web equivalent” of a coding assignment from the student perspective.
* **web-project** –A more permanent project able to be carried over multiple exercises or iterations. This project will persist while the student is active in the lms. It can be incrementally increased in complexity over time or multiple lessons.

### activity-type: web-project

A web project is intended to be worked on over multiple lessons and iterations. When you define a web project, the name of the project (activity-name) becomes a global identifier to identify the project from anywhere the student is working or taking lessons, regardless of course or learning module. Any files you define in the assignment definition (just like with a coding exercise) inherit those properties for the current lesson but those files remain part of the project and will show up as editable and changeable in the future if the project is reloaded for additional work in a later lesson. The following rules apply to a web-project’s file set whenever it is loaded:

1. If the student has worked on this project before, their existing files are loaded to the editor first.
2. The editor is then loaded with the files defined in this assignment’s definition that might be missing from the student’s existing file set.
   * Any student files with the same name as those defined in this assignment definition are skipped in favor of the student’s existing file.
3. An attempt is made to map the code of the student’s files to editable and uneditable line definitions in the current assignment’s file template in the editor on a line by line basis.
   * Unmatchable lines are left editable by default. A match is determined by a non-case-sensitive, none-white-space sensitive hash of the line.
4. Any files defined in the student’s project, but not part of the current assignment shown for reference but fixed and unchangeable for the duration of the exercise.

**Warning: The name of a web-project is globally scoped, if a project name is used elsewhere as a web-exercise the contents of the project will be replaced on the first submission of the web-exercise assignment and all previous data will be lost.**

### What happens when a student “runs” their web project?

When a student runs their project, or exercise the individual files are syntax checked against any syntax tests you provide (on a per file basis) and feedback is given. Assuming all the tests are passed those files are shipped to the student project web space and then served, via iframe, to the preview pane portion of the editor. This occurs every time the student clicks to run or execute their work.

In addition to publishing the files, the student’s submission is also recorded to a persistent storage medium for later review.



# How your content makes it to the Student

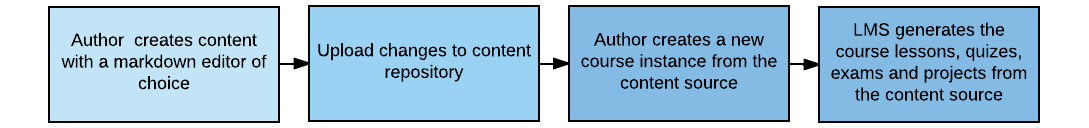


Fig. An author’s interaction with the content authoring system.

An author’s interaction with the content workflow is intentionally minimalistic. The intended audience for content authoring is students, administrators and teachers. They should be able to use a content editor of their choice (that supports markdown) to style and layout their documents with minimal intrusion. It is expected that content authors are familiar with and have access to whatever location the master copies are stored. This will likely be github, gitlab or some other flavor of a git repository. The LMS will then use this file storage location for building courses. By using and integrating with git, the LMS gains version and author change tracking as well as some nice online markdown editing capabilities without having it build it internally.

**FUTURE**: In the future, it is likely that a custom markdown editor will exist to edit content and construct valid c-blocks with a point and click or drag/drop interface. This editor will ingrate with the content file management system and is likely to be implemented in version 2 or 3 of the LMS, late 2017.

# Content Authoring Stylistic Guidelines

[(Chris/Josh/JT) Content Authoring Guidelines]

# Appendix

## Virtual Classrooms Layouts

[(Jake, UI/UX)]

## Activity C-Block Keys

|  |  |  |  |
| --- | --- | --- | --- |
| Key Name | Description | Data Type | Notes on Implementation |
| rank | (optional) indicates the relative rank of this activity to others in the same module/unit. | Number | At most, once per activity. All activities within a module/unit are given an internal ranking based on their alphabetic position within the module. This internal ranking is used as the activity ranking when an explicit rank is not supplied. The user specified rank is chosen first in the event a tie occurs between a user ranked activity and another, internally ranked activity. |
| difficulty | (optional, instruction only) If included a difficulty rating bar is shown indicating how hard this lesson and its content will be | Number | 1,2,3,4 or 5 |
| activity-name | (optional) Indicates the name to use for this activity. Is used as the lesson name when used in an instruction activity. | 500 characters | At most, once per file. The file name is used if this tag is omitted. Multiple files with the same activity name will be joined when the course is generated and treated as if they were one file. The rank key will determine the order in which the final activity is structured. Standard ranking rules and defaults apply. |
| activity-type | (situational) Indicates the type of content. Values are limited to “instruction”,” quiz”,” exam”, ”project”, ”coding”, ”web-exercise”, “web-project”, "Course" | 15 characters | At most, once per file. If omitted, files in a module/unit directory are automatically treated as instructions. Required for any files in an “Activity” folder.  Note, the course (root level) .md file should have an activity-type of "Course". |
| topic | (optional) Indicates the name of the topic. If omitted, all the content encountered up until a topic is defined is considered part of the same topic. | 500 characters | Each topic declaration will begin a new topic of instruction or a break in content of other activities. |
| start-activity | (optional) Interrupts the current activity and starts a new one. The current activity will resume when the started activity completes. | 500 characters | Must match an activity-name found within the same unit. |
| topic-transition | (optional) indicates with preset transition is used when switching between topics. | 25 characters | *Pending such transitions get created and scripted by ui/ux team.* |
| classroom | (optional) Used in conjunction with the **topic** key. Specifies the virtual class room template to use for this topic. Defaults to “onePanel” if omitted. | 15 characters | Once per topic, must be a valid virtual classroom key (see appendix A). |
| panel | The name of a panel within a classroom. The content below the c-block will be placed into this panel. | 15 characters | Multiple content sections can be placed in a single panel if desired. Content will be “stacked” with the first encountered section at the top of the panel. If omitted, the topic will be placed into the panel identified as “main” on the virtual classroom. |
| video-url-ogg video-url-webm video-url-mp4 | A url to a video to be displayed. | 8000 characters | The url (relative from the file location or absolute to a location on the web) of a video to play. Should be combined with a **panel.**  If no other content is scheduled to be shown in the panel for the topic, the video will consume and be centered in the entire panel. Videos may be linked differently via applicable markdown, however; only videos added via a c-block are actionable and used to control content flow. |
| video-id | A custom id you give to a video. Strictly for referencing said video elsewhere in your activity. Videos are unique per topic. | 25 characters | Paired with video-url-\* to create a new video reference or alone as a reference to a video previously declared in the file. Used with other video-\* and panel keys to control user interaction. |
| video-show-at | A timing mark and optional **content-id** declaration in the format of mm:ss indicating when this content should be shown.  Example: “05:13, headers” | Formatted Timespan (mm:ss) | Must be paired with a **video-id** within scope of the current topic. The content below the **video-show-at** will be shown at the specific playback point in the referenced video.  Delayed till v1.1 |
| video-hide-at | A timing mark and optional **contentid** declaration in the format of mm:ss indicating when this content should be hideen.  Example: “05:13, headers” | Formatted Timespan (mm:ss) | Must be paired with a **video-id** within scope of the current topic. If no **content-id** is given, the content below the **video-hide-at** will be hidden at the specific playback point in the referenced video.  Delayed till v1.1 |
| video-show-content | (optional) A timing mark and **contentid** declarations to indicate that other content sections for the same topic are automatically shown.  Example: “05:13, headers, footers, functions” | Formatted Timespan (mm:ss) | Used in conjunction with **video-hide-at** to control the showing of content. This tag serves as a shortcut alternative to adding multiple **video-show-at** tags to other content sections while a video is playing.  Delayed till v1.1 |
| video-hide-content | (optional) A timing mark and **contentid** declarations to indicate that other content sections in the same topic are automatically hidden.  Example: “05:13, headers, footers, functions” | Formatted Timespan (mm:ss) | Used in conjunction with **video-show-at** to control the showing of content. This tag serves as a shortcut alternative to adding multiple **video-hide-at** tags in the need of hiding multiple sections of content at single timing mark.  Delayed till v1.1 |
| points | (optional, coding assignments, projects, instructions, quizzes, exams) used for designating point values for different activities. | Number | If omitted the activity is worth 1 point for a correct answer, with the exception of instructions, which are not scored. When included in the header of a quiz or exam, acts a default value for all questions in the quiz/exam unless overridden on the specific question. Can be included on a “per topic” basis for coding and project-upload exercises. |
| required-grade | (optional) Used on quiz and exam activities to indicate the required grade, expressed as a percentage to proceed past the activity.  Example: “78%”, “85.24%” | Percentage in the format of ##.## | If omitted, the student’s grade will be calculated from the multiple-choice questions and they will be given the option to retake the test up to the **max-attempt** limit if they desire. |
| max-attempts | (optional) Used to indicate the number of times the user can attempt the activity. (quizzes, exams, coding exercises, web exercises) | Number | If included, once the max attempts are reached if a required grade is specified, the student is locked out of proceeding forward until intervention by an instructor.  If omitted, indicates infinite attempts for all but exams…which are defaulted to 1 attempt. |
| allow-retry | (optional) Used on a quiz or exam in conjunction with **required-grade** and **max-attempts** to indicate if the student can continue to try for a better grade after they have satisfied the grading criteria but still have attempts left. | true/false | If the student takes a quiz/exam multiple times, even after receiving a passing grade, the highest grade achieved will be recorded in the student’s permanent record. |
| shuffle-questions | (optional) Used on a quiz or exam to determine if the questions should be presented in a random order. | true/false | (false if omitted) If not supplied, questions are presented to the student in the same order on each attempt (the order they are created). |
| show-feedback | (optional) Used on a quiz or exam to determine if feedback should be shown on a while the student is taking it. | true/false | (true if omitted) If set to false, the student is never shown any feedback on the quiz/exam. They are simply shown their grade at the end. The instructor can release the quiz feedback to the class at a time of their choosing. |
| show-points | (optional) Used on a quiz or exam to determine if the number of points each question is worth is indicated with the question. | true/false | True if omitted. |
| immediate-feedback | (optional) Used on a quiz or exam to determine if feedback responses (and correctness of answer) are immediately shown to the student after each question. | true/false | (true if omitted) If set to false, the question, students choice, correct choice and targeted feedback will be shown in a report style after the activity has concluded. |
| allowed-files | (optional) Used on a project to restrict the files to specific file types a student can upload. File types should be indicates in a comma separated “dot three” pattern.  Example: “.jpg, .zip, .html, .cshtml” | 8000 characters | If omitted, the allowed-files defaults to only “.zip” |
| max-file-count | (optional) Used on a project to indicate the maximum number of files a student can upload. | Number | If omitted, the file count on a project activity defaults to 1. (Anticipating a single zip file being uploaded for the project). |
| compiler | (required) the expected language of the compiler to use for the student is writing in a coding activity. | Csharp,  Java,  Javascript,  Ruby | Omitting this for a coding activity will result in a parsing error. |
| file-name | (required) Used once per file declaration in a coding assignment or unit test | 500 characters | Identifies the default file name to show in a tabbed view for a coding exercise. Also used for compiler output messages in the event a program fails to compile. |
| file-language | (required) Used once per file declaration in a coding assignment, web-exercise or unit test | Html,  Css,  JavaScript,  CSharp,  Java,  Ruby | Identifies the type of file in a coding exercise. Used for formatting, color coding and intellisense. |
| test-id | (required) Used to define a unit or syntax test in a coding exercise, this sets the definition of the test | 500 characters |  |
| test-file | (required for a syntax test) the name of the student’s file you wish to test against. | 500 characters |  |
| test-type | (required) the type declares how the test will be executed  *Limited to: “syntax”, “unit”* | 25 characters |  |
| test | (required) A reference to a previously declared test-id and defines a test case for the results of the test | 500 characters |  |
| test-method | (required) Regex or constant value used to compare a dataset. Limited to: hash-match, hash-no-match or a valid regex. | 2000 characters | Represents what the student’s data will be compared against (see coding exercises for more detail) |
| test-stream | (required for a unit test) Indicates the stream of content from the REPL execution.  Limited to: “stdout”, “compiler” | stdout,  compiler |  |
| test-passed | (required) a flag indicating that if this test case is matched to the output does that indicate that the test has passed or failed | true/false | The content following the test case block is always shown if the test case matches the response output. The passed flag indicates if the test is considered successful and allows the student to continue. |
| unit-order | (Course Only) Specifies the units of a course and the order they are presented | Comma separated list | The current behavior is a standard “prerequisite chain” such that The second listed unit will not be available until the first is completed and the third listed unit is unavailable until the second unit is complete etc. |
| description | (course, activity) a short description to use on transactional reports. | 500 characters | Text only, no mark down. |
| to-do | (any block, any file)A short note of remaining work to do. | 2000 characters | This is strictly an “author-time” tag. Its ignored by the system and will not appear in a course template. |
| sub-unit-concurrency | Specifies rules between units  **Excludes** – makes two units mutually exclusive (you can't take both)  **Limit** – allows you to specify that student may choose n number of units from a list of units  **After** – allows you to specify pre-requisites | Multiple single-line commands separated by | | Syntax is as follows:  excludes {unit1}, {unit2}  Ex. excludes Angular, Angular2  limit [n] {unit1},{unit2},...{unitN}  Not specifying a value for n defaults to limit 1  Ex. limit 2 Angular, Angular2, React  after {unit1}, {unit2}  Ex. after Scripting Basics, React   * Makes Scripting Basics a pre-requisite to React   after {unit1}, {unit2}, {unit3}, …{unitN}  Ex. after Scripting Basics, JavaScript, React   * Makes scripting basics a pre-requisite to JavaScript and JavaScript a pre-requisite to React |
| email-template | Specifies the name of an email template, such as CourseWelcome or CourseComplete. The body of the email template should immediately follow such c block. | 2000 characters | ```c-lms  email-template: CourseWelcome  ```  Course welcome email goes here |