

Adapt Authoring Tool

Server Refactor Proposal

06.02.2018

prepared by Tom Taylor

# Overview

The aim of this document is to outline an approach to improving the structure of the node-based server component of the Adapt authoring tool.

# Goals

1. To lower the barrier to entry/reduce the overhead of working with core code for new and existing developers.
2. To facilitate a plug-in based architecture, and therefore reduce the need for major core changes in the future.
3. To expose a consistent and reliable public API for both internal use by the application and to third-party code.
4. To improve the stability of core code, and allow more effective automated testing.

# Proposed changes

## Folder structure

The backend/server code is currently very difficult to navigate and digest - particularly for newcomers. Better structuring the files and folders will be a major first step away from this. I propose the following:

* **Add a backend folder**. Move all server files/folders into a nested ‘backend’ folder, as is the case with the front-end code. The only files/folders left in the root should be relevant to the application as a whole (e.g. package.json, README.md etc.).
* **Refactor the lib managers**. These currently contain a lot more than just management code, usually at the very least data models and REST API definitions. These should all be moved out into separate files, with all models perhaps going in a lib/models folder.
* **Replace the routes folder with an API folder**. I propose we create a new folder at the root/server level called api to contain the public-facing routing/API endpoints for the application. The majority of the code in here should be concerned with routing requests, with any business logic moved to the relevant lib file (it may also be necessary to add controller files to each API/router to act as a halfway-house between the API and lib). API See **III. Routing/API** below for more on the API changes.

Example folder structure:

conf

lib

models

modules

auth-local

filestorage-localfs

api

download

export

preview

loading

test

## Plugin architecture

The existing architecture already allows for some degree of modularity, but this is restricted to specific types of functionality, and quite verbose in practise. The following plugin types are currently supported:

* auth: methods for users to authenticate with the server
* content: represent the Adapt content objects
* filestorage: used for asset management ???
* output: methods for outputting courses
* Non-standard plug-ins: database

Problems with this strategy:

* **Confusing nomenclature**. Naming these pieces of functionality ‘plugins’ can cause some confusion with framework plugins. Additionally, naming this folder ‘plugins’ suggests that it contains add-on functionality; everything in here is in fact core and required. Using the name ‘modules’ may be more self-explanatory (and fits with the frontend folder).
* **Flexibility for the sake of it**. The current system is confusing at best; it’s not at all obvious how plugins work, how to add new plugins, how to access plugins, or even what plugins are. This is exacerbated by the fact that different types of plugin use completely different code, and so a knowledge of each type’s quirks is essential. Simplicity is key here - we need to remove any complexity where it isn’t needed. In particular, we need to re-evaluate what we want to make ‘pluggable’ - the output for example will likely never change from Adapt (nor should it) — on the other hand, it may become necessary to to write alternative output plugins to support future releases of Adapt.
* **Lack of flexibility**. In a contradiction to the previous point, the authoring tool is both too flexible in some areas, while being completely inflexible in others; it is quite difficult to add new pieces of standalone functionality which don’t fall into the above plugin types. While it’s completely possible *technically*, in practise any custom plugins usually requires the developer to add files in various areas/folders across the backend. It should be possible to drop a folder somewhere on the server, and for that to be picked up and run as necessary.
* **Opinionated manager classes**. The top-level plugins (i.e. OutputPlugin, ContentPlugin) and their managers are very opinionated towards the current system and structure of the Adapt framework. This is particularly the case with output plugins; the OutputManager code is far too specific to Adapt, and were any other plugins developed, considerable rework would be required. At. The very least, all plugin code needs to be moved from the manager classes into their own files.
* **Hidden API**. This is discussed at various points in this document, and is a particular issue here; in addition to various bits of the API in the lib/manager files, there is also a lot hidden away in the various plugin subfolders.

## Routing/API

The entire routing strategy needs to be overhauled to allow better readability, easier extension and more complex function. Express 4 brought a host of enhancements which we haven’t yet utilised, due to the authoring tool originally being built for Express 3 and retrofitted to work with v4 later.

I propose that we do the following with regards to the routing:

1. Separate all API code into individual files
2. Add API controller files for code which isn’t suitable for the lib/manager code
3. Introduce specific routers
4. Make better use of middleware (particularly in conjunction with the above point)

### Separation of concerns

With regards to readability, the biggest improvement we can make is to separate the public API from the controller/manager code that uses it. As well as making the files slightly shorter, it will also allow us to unit-test the code more effectively.

Ideally, the API/routing code itself should be simple enough to act as self-documentation, allowing newcomers to scan a file and quickly take in the public API endpoints.

### Sub-routers

### The application currently uses a single router for the entire application. Splitting this up into multiple routers grouped by function will allow us to modularise the code more easily. Express 4 also introduced various shortcuts to specify routes which we can use:

// only user-specific routes here:  
app.route(‘/users’)

.post(function() { … })

.get(function() { ... });

// ...and so on

Another big benefit of splitting up the routers is that it allows us to use separate middleware for each router, allowing us more flexibility. A few possibilities:

* Consistent permissions checks
* Consistent error handling

### Middleware

Following on from the previous point, it’s become necessary for us to reassess the server functionality, as it’s become apparent that there’s a lot of inconsistency in some of the core mechanisms, The following areas would benefit from a rework to Express middleware:

* + **Permissions checking**. Our current method to check whether users have permission for any given resource is very inconsistent and ad-hoc, but is something that could quite easily be automated using middleware.

## ‘Action’ hooks

Implement a consistent hook-based system to allow any part of the application (and indeed third-party code) to easily react to system actions. These actions are completely arbitrary, but are mostly likely to be related to CRUD actions. Such a system already exists for content plugins[[1]](#footnote-0), but this interface should also be easily added to other functions and objects.

Potential use-cases besides content plugins:

* + User: CRUD
  + Output: publish, preview

**Specification**

The interface should:

* Be easily used to ‘decorate’ existing objects and functions (possibly using Node.js’ utils.inherit function in the first instance, and ES6 class inheritance later)
* Allow for arbitrary events
* Take a callback function
* Allow the listener to specify the execution time of the callback (this will likely just be ‘pre’ and ‘post’)

## More powerful config

Our current configuration setup isn’t particularly flexible; I’d like to see us at least add the ability to specify different config setups (for each type of environment - dev, prod, test - for example). We already have a separate config for the backend tests, but the core configuration code doesn’t allow these to be specified and loaded on the fly. We could also use the process.env.NODE\_ENV to set/determine this, and load the suitable file at runtime (i.e. require('./' + env)).

# Other areas for consideration

## Node version support

In order to take advantage of the latest Node features (including various ES6 language features), we will need to look at upgrading the supported Node version (likely to v8).

**Benefits**

* Better ES6 language support, removing the need for some third-party modules.
* Node performance improvements.
* The NPM install process alone is much quicker.

**Obstacles**

* Will break the ‘session’ code in the app.

## Adoption of ES6

As briefly hinted at above, it would be beneficial to upgrade the core code to ES6.

**Benefits**

* Code will be neater and better structured if we switch to the OO features.
* Code can be optimised to use core language features, rather than possibly more inefficient third-party libraries (and potentially removes the need for some completely)..

**Obstacles**

* Has a more marginal benefit when considering the amount of work required.

## Tidy up automated tasks

This applies to both the Grunt tasks, and the install/upgrade/index scripts. I propose using a structure similar to the adapt\_framework, whereby all Grunt tasks and config files are separated and stored in a nested folder[[2]](#footnote-1). I also suggest we integrate our custom scripts with Grunt, so we have a unified/consistent way to run all tasks.

## Tidy up temp

This folder needs to be reevaluated. Current issues with the current setup:

* The name of this folder is a complete misnomer; tampering with any of its contents will cause fatal errors across the application. It either needs to be renamed, or the application re-architected so that it is in fact temporary.

1. See [ContentManager#addContentHook](https://github.com/adaptlearning/adapt_authoring/blob/master/lib/contentmanager.js#L417) and [ContentManager#processContentHooks](https://github.com/adaptlearning/adapt_authoring/blob/master/lib/contentmanager.js#L448) [↑](#footnote-ref-0)
2. See the [adapt\_framework repository](https://github.com/adaptlearning/adapt_framework/tree/master/grunt) for more information. [↑](#footnote-ref-1)