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 improve the stability of core code, and allow more effective automated testing.
4. To expose a consistent and reliable public API for both internal use by the application and to third-party code.

Problem areas

Folder structure

The backend/server code is currently very difficult to navigate and digest - particularly for newcomers. It's not immediately obvious what files can be found where, even after working with the code for some time. Additionally, many key files/bits of code are obfuscated in long files, or hidden away in nested folders.

Plugin architecture

The existing architecture is quite verbose in practise, and doesn't allow the flexibility it initially suggests.. 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 the plugins for the courses themselves (framework). Additionally, naming this folder 'plugins' suggests that it contains add-on functionality; everything in here is in fact core and required.
- **A false sense of flexibility**. In the current system, it's not obvious how plugins work, how to add new ones, or even what plugins really are. Additionally, different types of plugin use different 'manager' code, and so a knowledge of each type's quirks is essential. We need to remove any complexity where it isn't needed here, and introduce a global (and simple) API that plugins can use to make consuming them more straightforward and transparent.. Currently, adding new pieces of standalone functionality which don't fall into one of the plugin types listed above usually requires the developer to add files in various 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.
- **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

This is one of the key areas in need of an overhaul, as it's the first point-of-contact for new developers. Some of the main problems with the current set-up are:

- **Inconsistent responses**: a developer can't rely on what's returned by the API either in the HTTP status codes it returns or the way data (or errors) are formed.
- **Inconsistent API**: the different API routes in the system are often slightly different from one another (even when using the same methods) which causes confusion for developers, and in general don't follow REST conventions.
- **Inconsistent error handling**: as mentioned above, the responses sent by the API are not consistent.
- **Inconsistent permissions checks**: each part of the API must manually check permissions in each request. This is very time-consuming and error-prone,. In various cases, checks within the same API route differ from each other, which can raise security concerns..
- **Poor readability**: this is particularly a problem in the more full-featured parts of the system (i.e. any of the manager files). Files are often far too long, and don't separate concerns where needed.
- **Difficult to unit test**: for all of the above reasons.
- **Documentation:** The api could be a lot more easily understood with the addition of docbloc's and perhaps some auto generated documentation

Proposed changes

Folder structure

With regards to the structuring of files, 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 be refactored into separate files where appropriate.

Modularise all functionality. I propose we merge the `plugin` and `routes` folders into a single folder (perhaps just called `modules`). The modules should all be invoked/loaded from the main application. These modules will contain/define:

- Third-party dependencies/libraries
- Public-facing routing/API endpoints
- Models
- Controllers
- Express middleware
- Configuration
- Unit tests

The system must be able to handle all of these parts without any complex coding from the side of the developer.

Example folder structure:

- conf/
- lib/
    - models/
- modules/
    - auth-local/
    - filestorage-localfs/
    - download/
    - export/
    - preview/
    - loading/
- test/

Plugin architecture

The best solution here is to completely rewrite the plugin architecture with the assumption that **everything is a module**.

To do this, we will need to define the core behaviours and utilities which will be needed by any module in the application, and make these available by extension. Existing core

functionality should be reworked into more granular modules with a 'single responsibility' approach,

The key point to make with this approach is that all modules will be treated as first-class citizens and have access to the same core code features, irrespective of whether core or third-party. This being said, it may be useful to make some distinction between core modules, and third-party plugin code, even if just from a conceptual view. For example, we may want to ensure that 'plugins' can make the assumption that all core modules are already loaded by the time they themselves are loaded. Additionally, for reasons of transparency, I think it makes sense to physically separate core and third-party code.

Routing/API

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

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

Improve consistency

This affects a number of key areas:

Routes: the routes themselves need to be looked at and brought in-line with a consistent style; some routes use unexpected HTTP methods, and some have very odd naming. Using Express 4's sub-routers is an excellent way to do this (see below).

Response data: we need to adopt a consistent strategy when it comes to how we return data to the client. This applies to specific data types (e.g. how should we return arrays consistently), as well as the data set as a whole (should we return any metadata along with the response data, as is common with RESTful implementations).

Response HTTP codes: these need to be made consistent, and to fit in with HTTP standards. In the very worst cases, there are routes which return a 200 status on error.

Error handling: as mentioned above, errors are rarely handled in an expected way, which results in the client receiving unexpected HTTP status codes, and the errors themselves formatted in a variety of ways. This ultimately results in unnecessarily complex response handling code on the client application.

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

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.

One of the results of this is that the application currently still 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. Additionally, modules should define and check their own permissions.

'Action' hooks

To allow for truly pluggable code, we need to implement a consistent hook-based system to allow 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], but this interface should extended 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 (Using some form of composition)

[1] See `ContentManager#addContentHook` and `ContentManager#processContentHooks`

- 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 minimum 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.
- Introducing a package.lock will help alleviate some of the issues we have seen with newer packages breaking things.

Obstacles

- Will break the 'session' code in the app. Which is used for encapsulating request data[2].

Adoption of ES6

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

Benefits

- Readability: 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 dependencies completely)..

Obstacles

- Has a relatively marginal benefit when considering the amount of work required (although this can be achieved through gradual improvements).
- For consistency, it would also make sense to switch to ES6 for the front-end.

Remove duplicated framework schemas

As of release v2.0.14[3], the framework source includes copies of the core content schemas, removing the need for us to bundle these with the server code. In addition to removing unnecessary duplication, uncoupling these schemas from the core code will allow for installs to more easily update framework versions and (in theory) reduce the work for the authoring tool core team.

[2] See lib/application.js.
[3] See commit release notes: 34bfa2bfd9faf4c2ea1fa465576fa1d9d97ec985

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[4]. 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 re-evaluated. 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.

[4] See the adapt_framework repository for more information.

Next steps

Moving forward from this document will involve the following processes:

1. Collating of feedback
2. Definition of final requirements
3. Generation of a technical specification
4. Writing of code!

If you have any questions, feel free to email me directly at thomas.taylor@kineo.com, or get in touch via Gitter using **@taylortom**.

Resources

Links to useful articles/products with similar approaches for research.

Google Course Builder

Uses a modular approach.

https://github.com/google/coursebuilder-core/tree/master/coursebuilder

NodeBB forum software

Uses a modular approach.

https://github.com/NodeBB/NodeBB

Possible fix for process.domain.session

Referencing our use of process.domain to capture a user's identity for the duration of the async request.

https://www.npmjs.com/package/continuation-local-storage