

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**

To improve readability, the code needs to be better structured to allow newcomers to navigate the file structure quickly and easily. I propose the following:

* Move all server files into a separate ‘backend’ folder (all similar), as is the case with the front end code. The only files/folders left in the root should be applicable to the whole application (e.g. package.json, README.md etc.), everything else should be relocated into either the frontend or backend folders.

## **Plugin architecture**

Something something something.

## **Routing**

The entire routing strategy needs to be overhauled to allow better readability and more complex function. Additionally, as the authoring tool was originally built during the time of Express 3, it doesn’t make the most of the enhancements which are available as of 4+.

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.

In terms of functionality, there are a few enhancements we should make to the existing server:

### **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, 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’)

# **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.

# **root**

Everything contained directly in this folder should be relevant to the entire application. Anything specific to the front-end/back-end should now be found in their respective folders.

Files remaining in this folder are:

* various config files (.editorconfig, .travis.yml etc.)
* install.js
* upgrade.js
* server.js
* package.json
* README.md
* LICENSE
* ... you get the idea

Note: I'd like to look into moving the install/upgrade/server commands to the task-runner we choose.

# **backend**

This folder now contains everything related to the back-end server application.

### **api**

This folder will contain the public-facing API of the application. The majority of the code in here will be concerned with routing requests, the business logic will still be handled by the 'manager'/controller files in lib (to make sure we separate concerns properly, it may make sense to separate out the 'controller' code from lib into individual controller files, which could go into here too). I intend to use express 4 sub-routers to do this.

This folder will replace the existing routes folder. I think it's also important that we make the application API flexible enough to allow these API 'plugins' to self-register/initialise, as the current solution of handling the loading/preloading/initialisation/whatever in the main application itself (e.g. application.js) is difficult to follow, and not in any way self-documenting (which is something it always helps to aim for).

Depending on how much stuff we have related to individual APIs, the folder may contain sub-folders, or just a .js file for each api router.

e.g.  
// just files  
assetRoutes.js  
userRoutes.js  
// folders  
asset/  
  assetRoutes.js  
  assetModel.js  
  assetSchema.js  
  assetController.js  
  ... blah  
user/  
  userRoutes.js  
...etc.

### **conf**

I'd like to do more with this folder, as it's not all that useful at the moment. I think it would be useful to make a bit more of this, and create multiple levels of config (at least for each type of environment: dev, prod, test), and store all options in here. As it is we already have a testConfig.json in test/ which should probably be in here.

We could also hook into process.env.NODE\_ENV or something to set/determine this, and load the suitable file.

I.e. require('./' + env);

### **lib**

This folder contains reusable (and unit-testable!) libraries that are shared and used throughout the app. This folder will largely remain the same is it is now. The main change here will be that all REST/API related code will be moved out into the api folder.

### **models**

*Big question mark on this one, as not sure it's needed.*

Contains all model data.

### **plugins**

I'm still pondering what to do with this folder. I'm not too fond of the plugin architecture we use, as I found it pretty impenetrable coming on to the project, and still feel that it's more complex than it needs to be without being flexible enough. I'm also aware of the potential avalanche of extra work/regression issues that will come with completely rearchitecting this.

Particular things I don't like:

* **The name:** For me, being called 'plugins' confuses things for two reasons:
  1. The framework uses plugins; some things in here relate to framework plugins, most things don't.
  2. It gives the sense that this folder contains add-on/additional content/functionality. Everything in here is in fact core and required. With the front-end, I went with the name 'modules' -- this may work here.

1. **The API endpoints being hidden away deep in the folder:** Although I understand the need for this, it makes working with the API very cumbersome and difficult to follow. Considering everything in here is 'core', this seems slightly unnecessary.

### **test**

This folder will contain all of the tests for the back-end application, and will likely be the same as now.

# **tasks**

This folder will contain tasks related to the building/running of the application. Grunt is the chosen task runner currently, so these may all be grunt tasks.

# **temp**

This folder needs to be looked at. Current issues I see:

* Name: temp is a complete misnomer; delete this folder at your peril. Needs to be renamed, or rearchitected so that this is in fact temporary.