Adapt Authoring Tool

Server Refactor Prototype Breakdown

15.01.2019

prepared by Tom Taylor

Overview/Disclaimer

This prototype is intended to showcase a modular architectural approach for the next iteration of the authoring tool server application. It is intended to be a starting point for discussion, and not a final approach.

No promises made to the quality of any code you may encounter..!

Areas to pay particular attention to:

- General modular approach
- Module lifecycle (instantiation, preload, boot)
- Inter-module communication (events, hooks)

*Also see the **Problems/Discussion Points** section for more areas for discussion.*

Running the prototype

Installation

To run the prototype, download the main repo from GitHub:

https://github.com/taylortom/adapt-authoring

Once you've got this, install all NPM dependencies with:

```
npm install
```

...and start the application with:

```
npm start
```

Testing

You can view a page grouping the usable functionality for the prototype at the **`/prototype`** endpoint.

Documentation

The documentation is auto-generated from code comments using ESDoc. You can generate the docs for the app using the NPM `docs` script:

```
npm run docs
```

The main entry point for the documentation is `/docs/index.html` in the **adapt-authoring** root.

You can also view a copy of the hosted docs here (not necessarily up-to-the-minute).

Execution flow

1. The NPM start script simply instantiates the core **App**, and calls **App#start**.

2. The App instance then starts all the required Adapt modules by doing the following:
   ***Note**: each module must include Adapt-specific config in its **package.json** and export a function, see here for more info.*
   a. **Initialise** (*sync*): sets config, creates the module instance, and performs any synchronous start-up tasks, e.g. setting defaults.
   b. **Preload** (*async*): performs any tasks required prior to the module actually starting. Anything that doesn't fit into the initialise phase (most notably anything async) should go here, e.g. loading a config file.
   c. **Boot** (*async*): actions to actually *start* the module, e.g. connecting to a database.

   … That's it!

Worked example

For modules which are dependent on other modules, it may be necessary to perform certain tasks during specific stages of the app life-cycle.

The **Users** module has two dependencies:

Server

The handling of **/users** routes is delegated to the **Server** module by creating a sub-router. Sub-routers will not be added once the server's started (which occurs during the server's boot phase), so this can only be done during preload.

MongoDB

The Users module can only start performing DB operations once we're connected to the DB, which happens during the MongoDB module's boot phase, so it waits for MongoDB's 'boot' event to be fired which signifies the Module has finished booting.

Structure

adapt-authoring

This is the main entry point of the application, defining the module dependencies, and providing a CLI interface to the main application (via NPM scripts).

Included modules

adapt-authoring-core: Reusable 'core' functionality: base classes, interfaces, data types, utilities etc. The only standalone functionality in this module is the main **App** module.

adapt-authoring-helloworld: Trivial example module which creates a sub-router to handle **GET** requests to **/helloworld,** renders **Hello World!** in response. Includes pre-route middleware.

adapt-authoring-logger: Standalone logging module to log messages to the console.

adapt-authoring-middleware: Bundles some Express middleware to allow for override/replacement by third parties.

adapt-authoring-missingroute: Gives the user feedback when trying to access any unhandled routes.

adapt-authoring-mongodb: An implementation of a 'DataStore' module for storing data in a MongoDB database. Uses Mongoose as the adapter.

adapt-authoring-oauth: module which will provide authentication/authorisation via Express middleware. Not implemented in the prototype.

adapt-authoring-prototypeui: Provides a basic UI for testing the prototype. Accessible from http://localhost:5000/prototype

adapt-authoring-server: Creates an Express 4 server with a main router and an API router. Provides mechanisms for creating sub-routers and adding middleware to both existing and custom routers.

adapt-authoring-theme: Serves as a base for the front-end theme. In its current state, just statically serves a public folder which contains CSS and assets.

adapt-authoring-ui: Example entrypoint for the front-end app. Displays a page at the server root (/).

adapt-authoring-users: Handles CRUD of system users.

Application anatomy

Core

App

Module sub-class, handles the loading of all sub-modules, and the starting of the application.

DataStore

An abstract class for storing data. Potential sub-class candidates are MongoDB, local file-storage and Amazon S3/other cloud service.

DataStoreQuery

Abstract object to represent a data query in a uniform way, which is used to run queries on any DataStore module. Each DataStore module is responsible for consuming and transforming DataStoreQuery objects into a format compatible with the persistence layer being used.

Events

Uses Node events to add event emitting/listening.

Hooks

Allows objects to interrupt the execution flow, and execute custom code at predefined points. Hooks can execute in parallel or series, and allow data mutation (if allowed by the hook instance).

Loadable

Adds defined life-cycle stages to a module to allow easier preloading/loading of multiple modules.

Module

Base class for all modules. Implements Loadable, Hookable and Events.

Moduleloader

A utility for the bulk loading of multiple modules.

Requester

A wrapper to handle HTTP requests consistently.

Responder

A wrapper to handle Express responses consistently (e.g. formats the response in a standard way, sends correct HTTP status codes).

Utils

Various utility functions.

Modules

Server

A wrapper for an Express 4 server, which exposes various functions to allow routes to be added to the server, as well as middleware.

All modules must do their routing via this module. Sub-routers can be added either to the root router (i.e. `http:/localhost/custom-endpoint`) or to the API router (i.e. `http:/localhost/api/custom-endpoint`). This differentiation allows us to treat endpoints differently according to their function (e.g. applying auth middleware to any API-level router).

Constructor

Express server created, delegate functions added to module to enable various Express functions.

Preload

N/A

Boot

- ‘Pre’ middleware is added
- Sub-routers are added

- Server is started
- ‘Post’ middleware is added

Mongodb

Extends DataStore.

Boot

- Connects to the MongoDB

Users

This is a more complete example of a typical module, and includes an **API** for use by third-parties/the UI, **controller** to format the API input, a **lib** file which does the heavy lifting (in this case just handles CRUD to the DB module).

Note: the contents of lib should be easily unit-testable, and shouldn’t be concerned with an IO with regards to HTTP requests etc.

OAuth

A middleware module which adds two major middleware functions to the API stack (note these only apply to API routes, and are not fully implemented in the prototype):

Authentication: verifies the user’s identity and allows/denies access to the API.

Authorisation: verifies the user’s permissions, and allows access to specific resources accordingly.

Middleware

A module packaging up a bundle of Express middleware to be added to the stack (in the case of the prototype, this is simply body-parser). The idea with this module being that third-parties are likely to want to customise the middleware stack with their own.

Note: we will need to be careful with any assumptions made by core code as a result of middleware which may/may not exist (e.g. in the case of body-parser which handles incoming `req.body` data).

Missing Routes

Module which handles unhandled routes on both the main and API routers, sending an appropriate response according to which router is hit. In the case of the main router, an HTML 404-type page is rendered, whereas a 404 API request will return JSON.

Theme

An example of how theming could work in the future. Makes CSS files and assets available statically (using `express.static`), which are then included by the various `.hbs` templates across the other modules.

Proposed Module Definition

For maintainability, it would be preferable to choose a consistent approach to developing modules such that any developer can tackle a problem with limited knowledge of a specific module.

With this in mind, a ‘module’ in the prototype typically consists of the following elements:

- A module subclass
- An API definition
- Arbitrary code to perform the tasks required of the API & interact with the rest of the system
- Language files
- Assets
- Views
- Unit tests

The following structure worked well in the prototype:

`lang/`	Language files
`lib/`	Code files
`api.js`	API definition. Contains no implementation code, but acts as an 'index' for developers reading the code.
`controller.is`	Provides a link between the API definition and the lib code Should perform any relevant IO functions, most commonly things like processing and validating the incoming request and sending a response.
`lib.js`	Meat and potatoes of the module; contains the module-specific code, and should be easily unit testable
`middleware.js`	Middleware functions
`module.js`	Main module definition; acts as the link between the above files
`public/`	Files which are statically served
`views/`	Handlebars views
`test/`	Unit tests

A structure like this serves to separate a module into separate logical parts to aid with maintenance. It should also help with unit testing.

Problems/Discussion Points

Design

CommonJS vs ES6 Modules

Which system makes most sense for this work?

```
const a = require('a'); // CommonJS
import b from 'b'; // ES6 modules
```

CommonJS: still the standard, more mature, better support, more familiarity.

ES6 Modules: the future standard (many areas still a work-in-progress), still experimental in Node.js v12, currently requires a flag and for JS files to either have a custom file extension or for modules to include custom config in package.json[1].

There is also the option to work with both types of modules, but wouldn't be an ideal approach. *Has potential as means of an upgrade path.*

Avoiding modular hard dependencies

There are several common contact points needed by sub-modules. From my initial research, I've identified these as:

- Auth (authentication + authorisation)
- File storage
- Data storage
- Configuration
- Logging

It would be useful to define some abstraction 'layers'/common APIs to allow devs to interact with the above without needing to tie code to a specific module. This approach also opens up the possibility to allow multiple modules of the same type to operate simultaneously.

A disadvantage to this design pattern is that you can lose a level of detail/accessibility to the core technologies underneath. I think this will be most noticeable with the DB - it would be a shame to lose the power and simplicity of working with Mongoose directly when it comes to querying etc.

Authentication/authorisation can (and should) be taken care of via middleware without any input needed from module developers (besides ensuring their endpoints are covered by whatever role/permissions definitions are put in place). It should be expected that once an HTTP request has been passed through the relevant auth middleware there will be no further permissions/access checks required by any module code.

> ***Discussion point:*** *should we allow for multiple modules within a layer to work simultaneously?*

[1] Node.js: Announcing a new --experimental-modules

Example: multiple DB modules saving data simultaneously.

This introduces consistency issues: if the data in each store isn't the same: which module should be used as the source of truth? In the case of a logging layer, multiple modules makes sense; for a database, it may cause more issues than it solves.

Rethinking authentication and authorisation

The current system doesn't work for the following reasons:

- Resource identifiers used aren't flexible enough to provide the fine-grained permissions; we need custom JS to handle edge-cases (of which there are many)
- Permissions/roles are too complex for anyone not familiar with the system
- Permissions are too granular to be useful for UI customisation (e.g. **READ /api/TENANT/content/course**) -- look into an alternative approach (e.g. permissions 'scopes'/groups of granular permissions)

Any auth system we implement should accomplish the following:

- Use a mainstream authentication mechanism for familiarity (e.g. OAuth)
- Allow direct API communication (i.e. not require log-in from the UI)
- Allow both API-level authorisation, and lower-level sub-router authorisation

Example authorisation scenarios:

- User has access to view all shared courses
- User has access to preview all shared courses
- User can delete users in their tenant

In the case of course content, we need to be able to grant access based on two things: the course resource itself, and the action on the resource (e.g. export). This must be possible using a combination of sensible URIs and Express routers.

How RESTful?

Due to it being a methodology rather than a standard, being 'RESTful' means different things to different people. Adapt needs to adopt something that both follows the REST methodology whilst also remaining easy to consume (some of the REST principles may be impractical in our context).

Definite yes

- Stateless (i.e. all information required for the server to process a request is contained within that single request)
- Consistent URIs
- Consistent return data format & structure
- Use standardised mapping of HTTP to REST methods (i.e. PATCH for update, PUT for replace, POST as a 'catch-all')
- Sensible use of status codes (**DEFINITELY NO** 200 code for failures)

Questions

- Links vs. object representation: returning links to other resources rather than the object representation itself, e.g.

```
{
  "href" : "https://root/api/users/USER_ID",
  "email" : "testuser@mail.com",
  "firstName" : "Test",
  "lastName" : "User",
  "roles" : [{ "href" : "https://root/api/roles/ROLE_ID" }]
}
```

Configuration

Our choice of app configuration interface will need to allow for multiple different scenarios:

- Ability for a module to expose configuration options
- Ability for a module to define default settings
- Ability for implementations to override defaults
- Different sets of options (and files?) according to the environment (dev, prod, test etc.)

Questions:

? Should these be split by environment or module

? Physical file separation? e.g.

```
conf/[modulename].json => {
  [env]: { ... }
}
conf/[env].json {
   [modulename]: { ... }
}
```

Alternative tech

Koa.js

Alternative to using Express if we're keen to kill callbacks completely. Much more bare-bones to Express.

GraphQL

Alternative to REST. Produces very nice streamlined responses, but would be a big unknown for most of the team. Maybe one for the future. Could be added at a later date.

Implementation

API definition

Would be useful to create an API class to allow devs to set up new APIs more easily, and allow a standardised approach.

For starters, something like this should do the following

- Create routers
- Have shortcuts for standard HTTP methods
- Handle/format requests (input) consistently
- Format responses consistently
- Handle errors consistently

It would also be useful to be able to link the API definition to other API documentation tools such as Swagger. Depending on how the API is defined, it may be possible to generate an OpenAPI file along with the other auto-docs process.

Hooks/Events

These are provided via two core files: hooks.js and events.js. Hooks take a lot of inspiration from WebPack's Tapable (and a prototype from Tom B).

Multi-tenancy

Although we may not release this fully alongside the server rewrite, it isn't something that can be plumbed in very easily later, so will need some thought in the initial design/implementation.

? Are we still intending to provide this - why
? What areas of the application does this affect
  - Data
    - Assets
    - Content
    - Framework/content plugins
    - Users
    - ???
  - UI
    - ???

? What separation are we looking to achieve for files & data (physical or meta)
? Can our goals be achieved any other way
  - Customise visibility of content
  - One database, many UIs

Code style

A code style guide will need to be defined to ensure we're as consistent as possible across the whole 'core' module bundle. As well as the usual specific syntax-type rules, this should also include best practices for developing authoring tool modules.

The documentation generator that we choose will need to be a consideration when it comes to code style, particularly with regards to which language features it works with, as there may be some coding styles which don't play nicely.

See Adapt Developer Guide (work-in-progress).

NPM

In this prototype, I've chosen to use NPM to manage the dependencies. This asks an additional question of how these packages should be versioned in the future, and whether they should be added to the public NPM registry.

Here are two options (note that both of these are interchangeable)

Publish to public registry

Makes the module publically available & searchable via NPM. We would probably use the **adaptlearning** organisation to publish, potentially using scoped package names.

Use git URL + tag

Means the module isn't published to NPM, and therefore isn't searchable. Keeps the ability to lock dependencies by tag/version (and use a semver range as with standard dependencies). It's also possible to add private git repositories in the same way (using **git+ssh** is probably the most straightforward way to do this).

Resources

- Prototype ESDoc documentation
- Current API reference
- JSON API Specification
- Tapable
- Boom: Consistent HTTP error handling

Modules

adapt-authoring

adapt-authoring-core

adapt-authoring-helloworld

adapt-authoring-logger

adapt-authoring-middleware

adapt-authoring-missingroute

adapt-authoring-mongodb

adapt-authoring-oauth

adapt-authoring-prototypeui

adapt-authoring-server

adapt-authoring-theme

adapt-authoring-ui

adapt-authoring-users