PubSweet 2.0 RFC

by Jure Triglav, July 2017
PubSweet Lead Developer and Architect

Many thanks to Adam Hyde, Alf Eaton, Richard Smith-Unna, Kristen Ratan, and Nicole Martinelli for helping improve this document.

The aim of this document is to briefly describe the current state of PubSweet 1.0, and outline, for discussion, a suggested path to PubSweet 2.0.

RFC discussion (please add all substantive comments here):
https://gitlab.coko.foundation/pubsweet/pubsweet/issues/16

1. Inception

PubSweet is a framework for developing publishing workflows. It was conceived by Adam Hyde in late 2015, his main objectives at the time were to design a framework to:

- Lessen the effort to build publishing platforms that **meet a wide range of use cases, including different content types, divergent workflows, etc.**
- Increase the reusability of code
- Leverage technologies that heighten the opportunities to build developer community
- Leverage technologies that are within the development budget and staffing reach of as many publishers as possible
- Support a cultural shift from workflow optimization to innovation
- Give publishers a choice in how they work

Adam mapped out some ideas for a highly decoupled system built with JavaScript. The initial architecture of the framework was then hashed out over a number of meetings between Adam, Michael and Oliver from Substance and myself in late 2015. Development started in earnest 18 months ago after a brief period of R&D. At the time, I was the sole developer but since then Richard Smith-Unna and Alf Eaton joined the PubSweet Core team.

Our aim in these early meetings was to lay the foundations for a framework that would eventually enable non-developers to assemble a publishing platform of their choice using a UI and existing components. It's a lofty, ambitious aim but not at all unrealistic -- although it may take us some time to get there. In the meantime (and for as long as PubSweet exists) we must focus on supporting developers, especially component developers, to easily develop publishing workflows with minimal effort. Consequently, further improving developer experience is the primary theme for PubSweet 2.0.

We also had many additional technical aspirations for PubSweet in those early meetings, most of which have been reached. A distilled list of those early goals include:

- Develop a custom component for PubSweet within a day
- Hello World in a couple of hours!
- Slim server API
- Easy install
- Easy access management
- Hard problems are solved by the library
- Natural, block-level configuration
- Client-side only SDK
- Follow SemVer

Eighteen months after those initial meetings, we now have a version 1.0 of PubSweet. This release is a coherent framework that faithfully represents our initial assumptions and ideas. Along the way, we've learned a great deal about how to build an app framework that supports a wide variety of publishing workflows and solves shared under-the-hood needs like flexible

authorization, file conversion, real-time updates, content formats of choice, publishing service integrations, etc.

On top of implementing what we initially designed, we've also successfully supported an innovative web-based scholarly monograph production platform built in collaboration with the University of California Press using PubSweet - Editoria (this week at its 1.0 release). We've also started development of a PubSweet-based Journal platform - xpub in collaboration with the Collabora Psychology journal. It's likely that in the near future many more journal solutions and additional publishing use cases such as micropubs and preprint workflows will be built with PubSweet.

So let's take a closer look at PubSweet 2.0. We aim to be as community-centric as possible, so we're opening this document up for your review and comments. We hope this will open the door for voices outside Coko to take ownership of, and help improve, the vision and implementation of PubSweet. We strongly believe the best road to changing publishing is through combined community effort.

You can reach out to us in Mattermost, or add comments to the Gitlab RFC issue, links below:

Chat:
https://mattermost.coko.foundation/

RFC (please add all substantive comments here):
https://gitlab.coko.foundation/pubsweet/pubsweet/issues/16

2. The Current State - PubSweet 1.0

Our PubSweet ecosystem currently consists of several logically separated parts - the PubSweet Command Line Interface, PubSweet Server, PubSweet Client and the PubSweet Components (whose lovely icons are the handiwork of by Henrik van Leeuwen.)

pubsweet-cli

The PubSweet command-line interface is the entry point into the PubSweet universe. It enables easy creation of new applications (comes with an initial app that prepares the basics for PubSweet development), with '`pubsweet new`', adding and removing components with '`pubsweet add`' and '`pubsweet remove`', managing databases with '`pubsweet setupdb`' and '`pubsweet adduser`', and running applications with '`pubsweet run`'. It's been thoroughly tested and every merge request goes through continuous integration.

pubsweet-server

PubSweet-server is a Node.js web server, using Express.js as the HTTP server, backed by either PouchDB or CouchDB document store, this is a solid and extendable (with components) server-side solution for publishing apps. The document store is abstracted away by a Model layer (Collection, Fragment, User, Team) with support for complex validations (extendable/configurable by apps) using JOI, https://github.com/hapijs/joi, and support for relations using relational-pouch. All of the server's API endpoints come with Authsome built in (an attributes based authorization system, more on this later), which allows for both simple and sophisticated configurable authorization within workflows. Additionally, key endpoints support live/real-time updates in coordination with pubsweet-client. It's been thoroughly tested and every merge request must pass continuous integration.

pubsweet-client

PubSweet-client is a React application with a central state store (using Redux) extendable with components. It supports the Authsome system (for visual indications of current user's permissions) and supports live/real-time updates. This application is built with 'webpack', and so comes with support for optimisation features (tree-shaking/dead code elimination) and features that make development easier and allow for faster iteration (hot module replacement). Additionally, it comes with a custom-built Webpack theme plugin (pubsweet-theme-plugin), allowing for easy SCSS styling. It's also been thoroughly tested and each merge request must complete continuous integration.

pubsweet-components

Components are at the core of the PubSweet developer experience and we've gone through a lot of iterations to arrive at the current system. We have server-side and client-side components. These are full members of a PubSweet app. Server-side components have the ability to add API endpoints (e.g. the INK server component adding '`/api/ink`'), have access to models and the database, and can add middleware (a component that runs for every server request, e.g. logging). Client-side components are simply React components (our client-side app is built with React) and can be used as views, or as sub-views in a web application (e.g. a `PostsManager` is a full view, `FormGroup` is a sub-view), and they have the ability to integrate with the central Redux store. What they do is completely up to the developer, in other words, the client-side components can do everything a React component can do, everywhere.

An ever-growing list of components is hosted in a Lerna-managed monorepo, for both server and client applications, these are reusable first-party components that provide INK (file conversion engine) interaction, OAuth functionality, text editors, dashboards and many more.
.

```
├── lerna-debug.log
├── lerna.json
├── package.json
└── packages
    ├── Blog     (CLIENT - landing page for a blog, listing summaries of blog posts)
    ├── Draft.js (CLIENT - Facebook's Draft.js in an editor component)
    ├── Epub     (SERVER - component for creating Epubs from collections & fragments)
    ├── FormGroup   (CLIENT - input field with automatic validations, based on config)
    ├── InkBackend  (SERVER - proxy to an instance of INK)
    ├── InkFrontend (CLIENT - use of the INK endpoint from a client-side application)
    ├── Login       (CLIENT - username and password login view)
    ├── Manage      (CLIENT - a host component for an application)
    ├── Navigation  (CLIENT - customizable top navigation bar)
    ├── PasswordResetBackend  (SERVER - password reset component, email notifications)
    ├── PasswordResetFrontend (CLIENT - password reset forms component)
    ├── PepperTheme   (CLIENT - styling/theme component)
    ├── PostsManager  (CLIENT - admin dashboard for blogposts)
    ├── ScienceReader (CLIENT - Substance-based reader component)
    ├── ScienceWriter (CLIENT - Substance-based editor)
    ├── Signup        (CLIENT - signup forms)
    ├── TeamsManager  (CLIENT - team management, creating and updating teams)
    └── UsersManager  (CLIENT - user management)
```

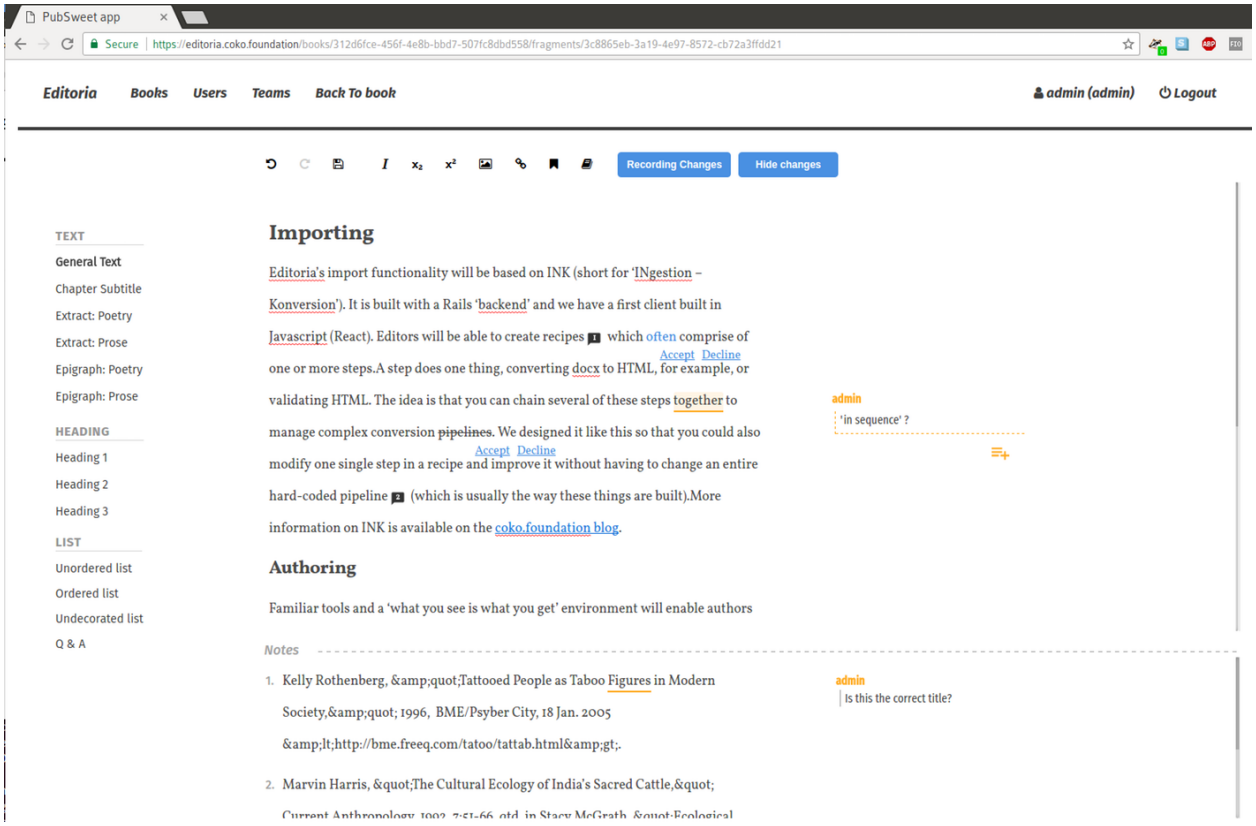
While the core PubSweet team is developing the above components (first-party components), there are also third-party components in development. Editoria's project, for example, has

developed the Wax editor (a componentized web-based Word Processor based on Substance) and the BookBuilder component (for managing workflows/interactions in book production). In addition, the early development of a set of journal components can be found in the xpub project (a PubSweet-based Manuscript Submission System):

.

```
├── components
│   ├── App.js
│   ├── DecisionForm.js
│   ├── DeclarationAnswers.js
│   ├── DeclarationQuestions.js
│   ├── EditorForm.js
│   ├── EditorList.js
│   ├── Navigation.js
│   ├── Project.js
│   ├── ProjectActions.js
│   ├── ProjectList.js
│   ├── RemoveProject.js
│   ├── ReviewForm.js
│   ├── ReviewerInvitationForm.js
│   ├── ReviewersForm.js
│   ├── ReviewersList.js
│   ├── RolesSummary.js
│   ├── RolesSummaryItem.js
│   ├── SnapshotActions.js
│   ├── Snapshots.js
│   └── UploadManuscript.js
└── containers
    ├── AuthenticatedContainer.js
    ├── DeclarationsContainer.js
    ├── EditorsContainer.js
    ├── NavigationContainer.js
    ├── ProjectContainer.js
    ├── ProjectListContainer.js
    ├── ReviewersContainer.js
    ├── RoleContainer.js
    ├── RolesSummaryItemContainer.js
    ├── SnapshotsContainer.js
    ├── UploadManuscriptContainer.js
    └── WaxContainer.js
```

Note that with the recent work on the Wax Editor and xpub, we're also breaking components themselves down into even smaller reusable pieces (components of components, if you will).



Wax Editor (PubSweet Component)

Consequently, there's already a wealth of components available and we're actively working on helping others build third-party components by creating a developer's resource at pubsweet.org (under construction). You can also find many of the components registered on npmjs.

All of the development for a PubSweet application (e.g. Editoria, xpub) happens at the component level. We encourage reuse of as many existing components as possible when building a publishing workflow with PubSweet.

The Collaborative Knowledge Foundation has also, in parallel, developed a methodology (collaborative design sessions or "The Cabbage Tree Method" CTM) that helps use-case specialists design systems. The way these sessions have gone so far maps incredibly well onto components.

In addition to the above parts of the PubSweet universe, to understand PubSweet we also need to look at Authorization, the basic architecture of a PubSweet app and PubSweets Model System.

Authsome (PubSweet authorization)

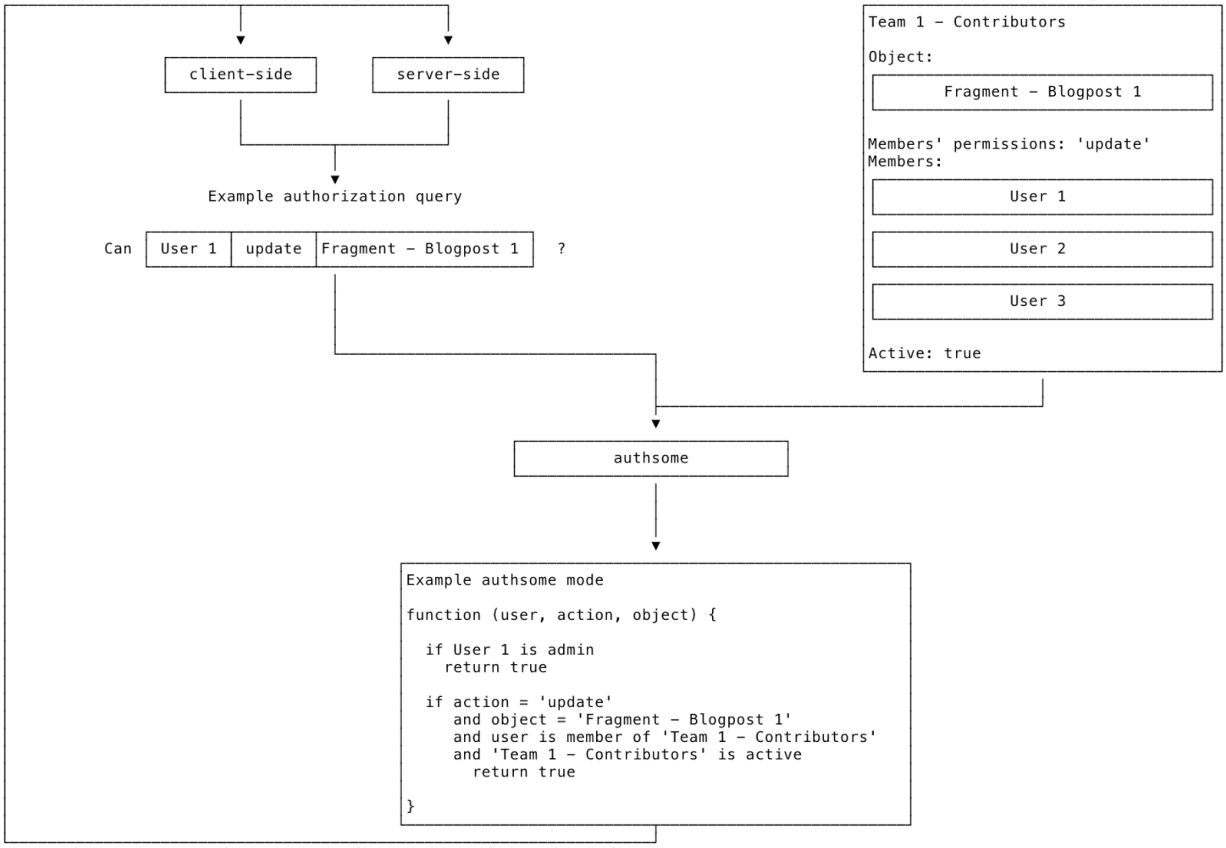
Authsome is an Attributes-Based Access Control (ABAC) system that we built to seamlessly manage authorization for both simple and complex publishing workflows. Publishing workflows require that various users have been granted access to specific resources dependent on a number of variables, for example:

1. A copy-editor can edit a paper only when that paper is in a copy-editing state
2. A reviewer can read a paper if that paper is in a review state, can add comments to that paper, but cannot edit the paper
3. A co-author can edit a paper if the author added them to the team of contributors and the paper has not yet been submitted
4. A technical check contractor can edit a paper's metadata (and no other properties), only when the paper is in a TC state, and not placed on hold by a senior editor
5. A book author can respond to a copy editor comments when the book is in the review state, can show/hide track changes (but can not turn track changes off), but is not permitted to edit the content

You can see the number of conditional variables for access control can balloon and while system designers need to aim to keep authorization variables to a minimum it is not always possible. Fortunately, Authsome can manage both simple and complex scenarios and enables the more complex *modes* (see below) to be easily optimized over time.

Authsome manages a huge variety of different -- publisher specific -- access requirements by connecting users, teams of users and objects (such as fragments and collections) with what we call *Authsome Modes* - custom authorization modes written for a specific use case (e.g. a scientific blog mode, a book production mode, Editoria's mode, etc). Importantly, it's also easy to model a simpler role-based authorization system using Authsome if required, by naming teams according to roles. But since Teams can also be conditionally active (e.g. only active before 3 p.m., only active if the object's state is 'reviewing'), more complex authorization approaches can also be achieved.

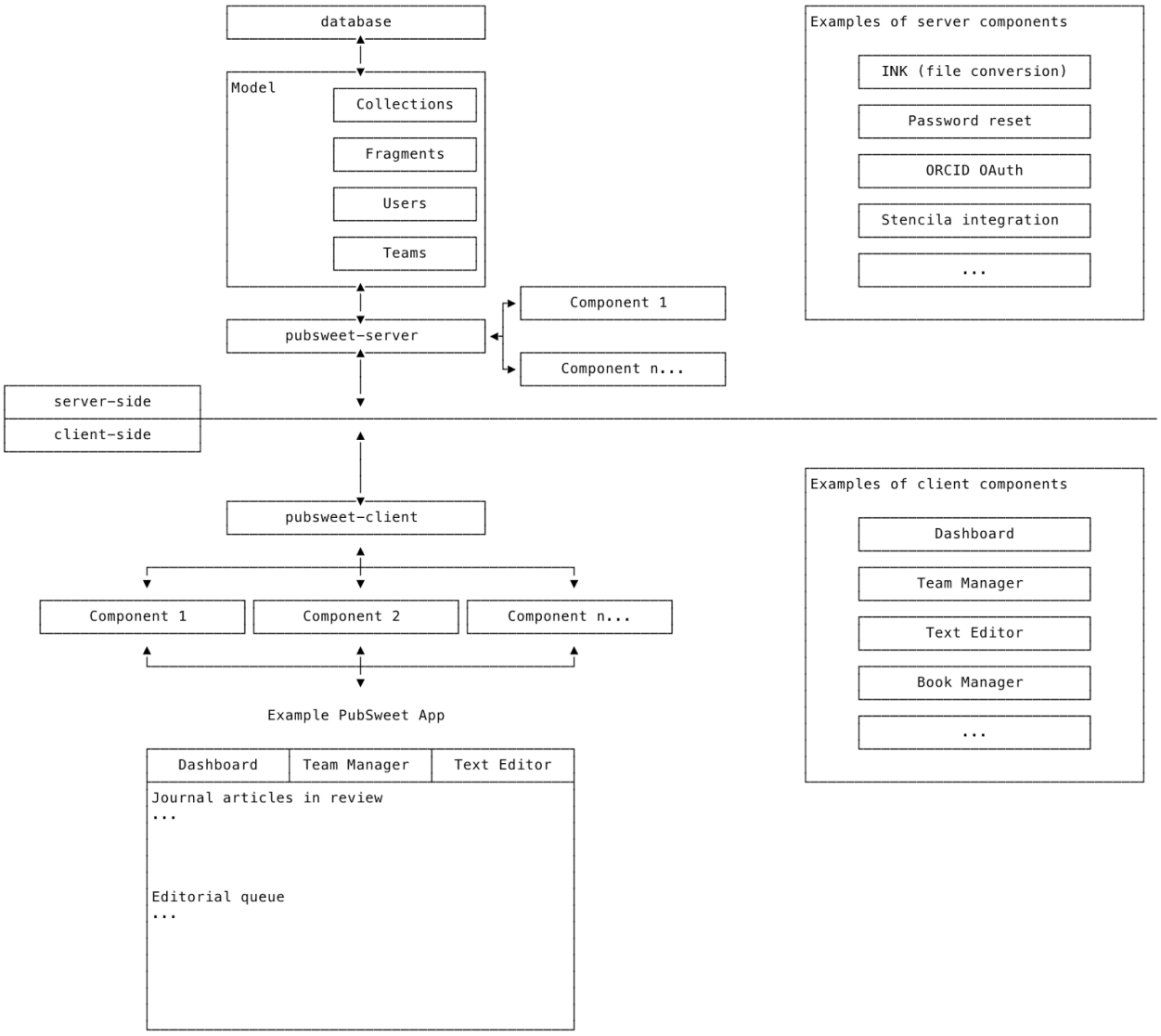
Below is an example implementation of the Authsome with an example mode.


client-side
server-side
Example authorization query
Can
User 1
update
Fragment - Blogpost 1
?
Team 1 - Contributors
Object:
Fragment - Blogpost 1
Members' permissions: 'update'
Members:
User 1
User 2
User 3
Active: true
authsome
Example authsome mode
function (user, action, object) {
if User 1 is admin
return true
if action = 'update'
and object = 'Fragment - Blogpost 1'
and user is member of 'Team 1 - Contributors'
and 'Team 1 - Contributors' is active
return true
}

The basic architecture of a PubSweet app

When all of the moving parts above play together, they form a PubSweet application. Getting from scratch to something you can start developing with is easy using the '`pubsweet new`' command. This creates a minimal scaffolding in a new Git repository. You can start developing immediately by running the app in a development environment using '`pubsweet run –dev`'. This will start a pubsweet-server instance, compile and serve the client-side app using 'webpack', and monitor your files for changes (automatically reloading or rebuilding when appropriate).

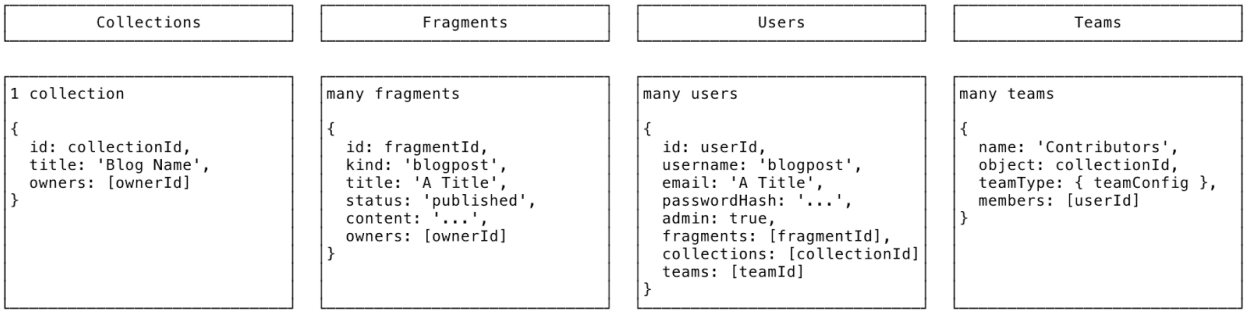
Applications built with PubSweet have an easy-to-read architecture, shown here:



High-level architecture overview of a PubSweet app

PubSweet models

We've designed a model system that can account for a wide variety of publishing workflows. It achieves this by being quite generic while also allowing for sub-typing. The example below illustrates this with an example PubSweet 'scientific blog' application.



Model structure for a 'scientific blog'

In this example using PubSweet 1.0, there's one **Collection**, it represents the entire blog and has connections to all existing blogposts, represented by **Fragments**. In the Fragment example, you can see the subtyping, '`kind = blogpost`'. Both Collections and Fragments have owners, one of only two fixed, baked-in/hard-coded, authorization based ideas in PubSweet; the other is the admin. Each collection or fragment has at least one owner and each PubSweet app has at least one admin. The extent of permitted actions for these two types of users is defined elsewhere using Authsome Modes.

There are many **Users** and those users can belong to multiple **Teams**. Teams are formed around either a Collection or a Fragment and the interaction between these forms the backbone for our authorization system (who can do what to an article/book/chapter etc, and when). Again, the limits of permitted actions for users and team members are defined in the configured Authsome Mode.

The model system talks to the current database of choice, PouchDB, which can also be replaced with CouchDB in a production/high-volume setting.

On the client-side, a central store/layer enables interactions with all of the above models, as long as the current user has the permissions to do so.

Let's look at the individual parts of the data model in detail.

Collections

A Collection is a versatile model/data type that represents groups of fragments. It has its own metadata and links to the collection's fragments.

To expand a bit on Collections, here are few more examples:

- In the case of a journal application, a Collection can represent the entirety of a journal, a volume of the journal, an issue of the journal, or individual articles
- In the case of a book production application, a Collection can represent the entire corpus of books published, a single book or a series of books.
- In the case of a collaborative editor application, a Collection can represent the list of documents

Fragments

A Fragment is a model representing a single item. It has its metadata and the content of the item it represents. Again, let's look at a few examples:

- In a journal application, a Fragment can represent an article, a section of the article, a graph or illustration in the article, or supplemental information.
- In a book production application, a Fragment can represent a chapter, a table of contents, a footnote, or a preface.
- In a collaborative editor application, a Fragment can represent a single document, a comment on that document or a drawing in that document.

How users actually end up modelling the data is not prescribed and a system of Collections and Fragments provides a lot of flexibility, and supports a huge variety of publishing use cases.

Users

A User is a model representing a registered user in a PubSweet application. It contains its required data, the email, username, password (stored as a strong bcrypt hash) and admin status, metadata (e.g. OAuth information) and links to the user's teams and the collections and fragments it owns.

Teams

Teams form an integral part of the authorization system PubSweet uses. All teams have a team type, available types are defined in the configuration, a team name, an object the team is based around (e.g. a collection or a fragment) and its members.

A team type is defined like this:

```
teamContributors: {
  name: 'Contributors',
  permissions: 'create'
  active: function (paper) {
    return paper.status === 'writing'
  }
}
```

The above team grants the ‘create’ permission (what that means is defined in the Authsome mode) on the team’s object (either a fragment or a collection) for all members of teams of this `‘teamContributor’` type, if the object’s status is equal to ‘writing’.

The team activity is a crucial piece of the authorization puzzle, since it allows us do object state-based authorization (who can do what if a paper is in ‘review’, or a paper is in ‘copy-editing’ mode, etc.), but also other, wider state-based functionality (e.g. time-based authorization control for shifts). Authsome modes have access to users, teams, collections and fragments, and so can base a permission decision on all of the relevant states.

3. Lessons learned

Building a system with this many moving parts involves a lot of small iterations on small parts of the code and while each individual change is small, they add up over time. We've dedicated a lot of time to make each subsystem as stable and as well-tested as possible, yet there are subsystems that stand out in terms of how much resources we've spent to keep them spotless.

To be more specific: We've dedicated a lot of resources to 'pubsweet-server', but a disproportionate amount of that time has been spent on managing relations using an object store within pubsweet-server. This is surprising, because the object relations manager (ORM) is a small portion of the overall code and functionality, but it had the most critical bugs of the subsystems (we had a similar issue with the scaffolding code in pubsweet-cli). It's even more surprising because object relations are a generally solved problem in the JS ecosystem (Bookshelf.js, Sequelize, Objection and friends). However, at the time we looked at all of the projects listed above, but each had a flaw or two (too many open issues, too few recent contributions, bugs) and each initially looked somewhat like a solution that meant starting over. Whereas, our own (yet to be built) wheel, would of course be much better, not have those flaws, and generally be a lot easier maintain and extend! (That wasn't the case in the long run, see reasons above.) It turns out that a much better way to approach this would be to pick an existing, even if somewhat broken wheel, fix it then contribute the fixes to that project's community. Which is, of course, what open source is all about.

This brings forward our first guiding principle for PubSweet 2.0 :

> *Be more rigorous in finding and using existing working wheels and provide improvements to those, if needed, instead of building our own.*

This approach fits very nicely with the way that the Collaborative Knowledge Foundation works across all projects - focusing on using existing open source projects as much as possible and contributing back upstream with fixes and comments as much as we can. Of course we *have* followed this principle in most cases with PubSweet 1.0 and have improved a lot of the existing projects out there by commits to tools such as 'yarn', 'webpack', 'prompt', 'node-config', 'lodash', 'substance' and others. On top of that, we've provided functionality feedback and bug reports to many more. Yet we could do better, particularly with the ORM code and pubsweet-cli.

Of course, this goal must be balanced with avoiding straying too close to the bleeding edge. We've been very lucky that almost all of our choices (e.g. webpack, jest, and yarn) involving relatively new libraries have paid off pretty well in the long term. In the short term, there's a little pain, but it's worked out well as the libraries improve. It's a tricky line to walk and we continue to weigh the costs and benefits for each library we adopt while bearing in mind we can not always be right. Periodic review and an openness to improved choices is something we will continue to abide by.

Looking back, it's also clear that PubSweet's vision is strongest with the components. If PubSweet is doing its job, then developers can focus purely on building and innovating on the

component level. The more components exist, the more powerful the PubSweet universe becomes and the quicker it is to solve existing publishing problems and foster innovations.

This brings us to our second important focus for PubSweet 2.0

> *Focus on improving component developer experience.*

That's not to say the current experience is bad -- far from it. It's pretty easy to develop a PubSweet component, but we should aim for the highest bar possible when it comes to improving component developer experience. There are many things we can do towards this from the obvious - better documentation and a clearer project structure - to the more sophisticated such as improving the utility of pubsweet-cli.

Finally, I believe we need to improve flexibility in the components data model. Currently, all components are bound to our data model consisting of Collections, Fragments, Users and Teams. An editor for a journal paper works on a fragment, a dashboard component works on a collection. But it a new PubSweet developer might rightly ask, for example, what happens when we have a component that deals with reviews, are reviews also fragments? Which collection do they belong to? What about files, are they also fragments? The answer: they could be. Fragments are so general that they can describe anything, but it can get confusing for the uninitiated and may cause more cognitive drag than necessary when developing larger applications.

Which brings us to my recommended third and (possibly) final aim for PubSweet 2.0:

> *Enable PubSweet components to extend the PubSweet Model System.*

This enables component developers to design their own models if they wish to. This also, of course, makes components, as first-class citizens of the PubSweet ecosystem, extremely powerful. Which is what we want!

With these thoughts in mind, I've put together a list of recommendations for PubSweet 2.0. The intention of this document is to generate discussion around these issues so we can make the best decisions possible and ensure we meet the needs of our growing community.

4. PubSweet 2.0 proposals

The following are the proposals for PubSweet 2.0. They're all up for discussion!

They grow out of our three guiding principles from the lessons learned above:

1. Be more rigorous in finding and using existing projects and provide improvements to those, if needed, instead of building our own.
2. Focus on improving component developer experience.
3. Enable PubSweet components to extend the PubSweet model system.

A couple of quick notes before we dive into details. In parallel to these proposals it's important that we improve, and maintain, developer resources. This will be achieved with dedicated resources provided by Coko outside the developer team. Any additional help here from documentation writers, community experts, workshop leaders, designers etc. is very welcome. Additionally, I believe we need to normalize some of the language around PubSweet. Component scope, for example, varies from forms to views, but we have multiple ways of talking about it. Any comments focused on improving and stabilizing the PubSweet lexicon will be very much appreciated.

I put the following changes up for discussion for PubSweet 2.0.

A. Replace ORM code with an existing library

Following the rationale and example above, replacing our own code for object relationships with an existing library is a necessary first step. We've looked again at the number of options available and Bookshelf.js looks to be the strongest contender in terms of its API and features, but, again, it's up for discussion.

B. Replace scaffolding code with an existing library

While PubSweet CLI does many things one of its most important functions is to create the scaffolding for an initial app. Since there are very good existing projects that create scaffolds, such as Facebook's `create-react-app` (client-side), Express.js' `express-generator` (server-side), `yeoman` generators (client and server-side) and others, the bespoke scaffolding code could be replaced with one or more of the above generators.

C. Extend the PubSweet CLI to improve utility for developers

PubSweet CLI does a lot, however it could do more to support developers, namely more options in the context of ops (e.g. backup database, restore from backup), and more features to support development and introspection. At the very least, the importing of components through the CLI and creation of routes would be beneficial but this topic is wide open for suggestions.

D. Simplify the project structure

The current project structure has a fairly well organized file structure but it is worth revisiting this periodically. If you can see any ways to simplify what we currently have, this is a great time to speak up.

E. Extending component models

This item possibly requires the most explanation. There's a lot of value in a clean and understandable data model, for example: a paper is a Paper and a review is a Review, and a Paper can have many Reviews and Files. But if we define a data model like that from the start (currently the case with the generic collections and fragments), then we're limiting the use-cases of PubSweet to applications described by that model (in the above case, journals), and that's something we don't want. So how do we get clean, accurate, specific *and* flexible data models? I propose we extend the component system to allow a component to first, bring its own schema and second, to extend the existing components' schemas.

Let's try out an example with a scientific blog, breaking it down into components and schemas:

Dashboard (admin)

A dashboard for a scientific blog deals with `BlogPosts`, it lists them and shows actions one can take (create a new blog post, edit or delete it). The data model it needs to operate is a table of blog posts, something like this:

```
type BlogPost {
  title: String
  createdAt: DateTime
  id: ID! @isUnique
  updatedAt: DateTime
}
```

If we build an application with just this dashboard component, our final schema would be equivalent to the above and we'd be able to add or remove blog posts (storing only the title, since the component doesn't need other information). To make it useful, we'd need to add an editor.

Editor

A simple editor deals with a document, so the editor's component specifies this requirement by saying:

```
type Document {
  title: String
  createdAt: DateTime
  id: ID! @isUnique
  updatedAt: DateTime
}
```

If configured to, it can also extend a host component's type with a relation to the document:

```
extend type HostComponentType {
  document: Document @relation
}
```

Where, in our scientific blog application, the `HostComponentType` is somehow either inferred or configured to be BlogPost. You could literally drag the Editor onto the Dashboard (or perhaps a job for pubsweet-cli), to signify `Dashboard(Editor)`, and the Dashboard's type would be extended, in addition to the following client-side routes being automatically generated:

```
/dashboard
/dashboard/:blogPostId/editor
```

Now we need to display these posts to complete the blog.

Blog landing page

The blog's blogroll or landing page needs a list of blog posts, same as the dashboard. Since the blog landing page doesn't write to this table, if the schema doesn't already exist because of the Dashboard and Editor component, that would mean that there isn't any component in the application to populate it. We could still manually fill in the data (not through the application layer, but through the database layer), so it's perhaps not a reason to error, but a warning is warranted ("Component `BlogLandingPage` depends on a schema that doesn't exist. Creating missing schema.").

```
type BlogPost {
  title: String
  createdAt: DateTime
  id: ID! @isUnique
  updatedAt: DateTime
  document: Document @relation
}
```

Notice how it expects the document field too, which comes from the Editor's schema. This means that the Blog landing page component will give a warning, if either the Dashboard or Editor component is missing. It should be noted that both Dashboard and Editor components

would also raise a warning about missing schema dependencies, but the developer would understand the meaning of it as a mere notice, since those two components are meant to create schemas. We could potentially be more strict and express the schema needs of this component, and bail/error if they aren't met (but that prevents using the component with manually filling data on the database layer).

Given a good admin interface (separate db layer, Graph.cool example), it's not hard to imagine a situation where someone merely uses the blog landing page component and manages the data (copying and pasting data) in the admin interface (and not in the application layer's dashboard and editor).

And now we have a blog. It has a clear data model and all of the above components can be used in isolation, but they can behave differently depending on wired up to other components. Let's look at the final example: What if we want to add comments to the blog?

Comments

A comments component wants a table to store comments:

```
type Comment {
  title: String
  createdAt: DateTime
  id: ID! @isUnique
  updatedAt: DateTime
}
```

Additionally and optionally, it wants to extend the schema of a `HostComponentType` (adding to it a list of comments):

```
extend type HostComponentType {
  comments: [Comment] @relation
}
```

Adding comments to a blog can then be as simple as connecting it with the blog landing page component, signifying `BlogLandingPage(Comments)`.

F. Utilize GraphQL for API queries

You may have noticed that the above descriptions for extending models are GraphQL schemas with minor additions (such as the extend syntax). I put those in there deliberately because I believe we should consider including GraphQL in PubSweet 2.0. Leveraging GraphQL would mean we could provide all of the necessary API capabilities to create, read, update and delete component defined models through a single endpoint -- i.e. it wouldn't cause an explosion of the number of API endpoints. Additionally, this single endpoint is incredibly flexible and things

like filtering for a specific view, and getting only the minimum required data, are supported out of the box.

5. Suggested roadmap

The following is a very brief outline for a possible PubSweet 2.0 roadmap. This is a pretty rough outline because I expect it to be shaped by your comments and our conversations, but I've broken it down into logical order. Developers should not need to modify existing component code when upgrading unless they wish to leverage new functionality (e.g. extend the models with component code, or use GraphQL queries) since we will maintain the existing API and there are no major *required* changes to how PubSweet 1.0 components interact with a PubSweet 1.x or 2.0 system.

1.1 Simplify Project Structure (if necessary)
1.2 Replace bespoke CLI code
1.3 Extend CLI
1.4 Replace bespoke ORM Code
1.5 Introduce component models
1.6 Implement GraphQL
1.7 Update CLI to meet the needs of 1.5 and 1.6

Being ambitious but relatively realistic, I'd like to see us arrive at PubSweet 2.0 within 4-6 months. This might be a useful scoping mechanism when planning the roadmap, as well as a nudge for frequent reviews and community check-ins.

Of course, you may well have your own ideas that aren't listed here, or have strong opinions about the suggested approaches. We'd love to hear these and invite you to discuss them with us. We're active on the Coko Foundation Mattermost channel and there's also a RFC issue created in gitlab for discussion of your PubSweet 2.0 ideas and this document.
Let's move forward together!