ECE 458 Resource Manager: Evolution 4
By Davis Gossage, Emmanuel Shiferaw, Sam Ginsburg and Allan Kiplagat

This is the fourth and final write-up for the Resource Manager that we have been building throughout this semester. It contains a retrospective analysis, an analysis of the current design and team member contributions.

Retrospective Analysis

After all four evolutions, we remain pleased with the choice of Meteor as our web framework. In past writeups we have discussed the packaging system as one of the main reasons for choosing Meteor, and it continued to be helpful in this evolution. We decide to use a package when a certain problem has been solved before and we don't see any major value-add by creating our own solution. There have been situations when requirements change and leave the scope of what the package is capable of, but in those cases we have been able to modify and extend the package to suit our needs. We did this with the 'Accounts' package when NetID support was required. We liked the idea of keeping our existing user authentication system so we chose to build NetID support on top of that system.

The biggest difficulty we've found using packages relates to them being black boxes, code that we did not write and are initially unfamiliar with. We use the alanning:roles package which manages our system-wide permissions by maintaining a key-value store of permission strings and users which belong to those permissions. We were noticing crash reports (**Figure 1 in Supplementary Material section**) in production caused by the Roles package creating duplicate keys in Mongo. It took us a while to determine that this issue was caused by attempting to add duplicate permissions with the same string in our application. This behavior makes sense but wasn't documented, and since we didn't write the package it took us longer to figure out what exactly was going on.

While Meteor blurs the line between server and client by allowing development of both in the same project directory, one of the great advantages of Meteor is that it forces explicit data security by using the publish and subscribe model. We are able to selectively publish data related to the current user, and that data represents the universe of data the the client is able to see and access. For example, when publishing resources we ensure that a user has view permissions for a given resource before exposing it to the client.

One potentially poor design decision that hurt us a little in this evolution relates to the modularity of our HTML code. We use templates which are reusable HTML and JavaScript components. For example, we have a template which displays a resource's information. We use these templates throughout the app to stay consistent with the way things are displayed. In this evolution particularly, we realized we could have been more explicit about deciding which templates should be created and where they should appear. Instead we ran into

some repeated code in the create resource and edit resource pages. These two pages do very similar things, but because they were created at different times there was not an appropriate template to use for both. To avoid this problem we should have been more generic when creating the edit resource page, keeping in mind that it might be necessary to have the page handle creating a resource from scratch.

Analysis of Current Design

The two main feature additions in this evolution were resource sharing and resource hierarchies. The resource sharing required significant changes to the 'restricted resources' feature as explained in the *sharing* section below. Implementing the resource hierarchies required minimal modifications of the existing code and we used the jqtree package for hierarchy display as explained in the *hierarchy system* section below.

As explained in the retrospective analysis, meteor packages have been very helpful and Table 1 in the Supplementary Material section summarizes the major 3rd party meteor packages in use. While this is the last evolution, if new features were to be added to the project, we can envision using other suitable packages provided that they are well documented and tested. This reliance on 3rd party packages speeds up development and minimizes complexity, but is also risky since future requirements might not be compatible with the packages and force us to edit, replace or remove the packages (all work-heavy and potentially complicated actions). This risk has proven worth taking so far.

Sharing:

To implement the new ‘resource sharing’ feature, we had to disobey the open-closed principle by going into our methods/implementation of the previous ‘restricted resources’ feature. We could not see a way to avoid this, as the two features interact deeply, and we wrote our previous code with the (false) assumption that the nature of ‘conflicting’ reservations would not change. The required behavior changes with regards to restricted resources for *2* of the sharing cases: if a restricted resource has “limited” sharing, then an approval on a reservation will only deny other concurrent reservations if the share limit has been reached. If a restricted resource has unlimited sharing, then an approval on a reservation should not deny any other resources. In the final case (restricted resource has exclusive/no sharing), then the behavior is the same as before - an approval on a reservation denies all concurrent reservations containing the resource. Given this required behavior, there was no way to make the Approval process work for restricted resources without changing the code that runs upon approval - the server-side **approveOrDenyReservation()** method. During each approval of an incomplete reservation, all ‘conflicting’ reservations must have *all* of their resources checked for sharing limits (just as they would be if being approved). This means querying the entire database once for each *resource* on each conflicting *reservation* during every attempted approval or denial. This works perfectly for the

purposes of our project, but we can imagine that in a larger system with more resources per reservation, and many conflicting reservations, that this could be a significant performance bottleneck.

Although we had to alter the old implementation at a low level, the number of changes that needed to be made overall was small. The aforementioned approval method needed to be altered. In the database, each resource now has fields that dictate how much it can be shared: *share_level* and *share_amount*. The *share_level* is exactly as described in the specifications: Exclusive, Limited, or Unlimited. *share_amount* only applies in the case of Limited, and describes how many reservations can concurrently hold the resource.

So, for this feature, we needed to add 2 fields to one table, and alter 1 public-facing method (not including extra parameters in methods like add/modify resource with the 2 fields). We feel that this was a reasonably small amount of effort to add this functionality.

Hierarchy System:

To implement the hierarchy system where resources can be ‘parents’ or ‘children’ of other resources, we did not need to make any significant changes to the architecture of our code or data. We decided to use the simplest option we came up with, which was to simply add one additional field, ‘children_ids’, to our collection of resources. This field is a list of ids of all resources who are direct children of a resource. We chose not to keep track of an additional ‘parent’ field, since this would only have added additional overhead each time the positions of resources in the hierarchy changed, which would have increased the chance of bugs in that functionality. Also, it wasn’t necessary because the entire hierarchy can be created simply from the roots, or top-most parent resources.

To generate views for this new hierarchy structure, we used a package, ‘jqtree’. This package made it easy to generate interactive trees which can be rearranged or edited. We chose to use this package because a draggable tree view is a solved problem which would’ve taken an unnecessarily long time to replicate. The downsides of this would have been an increase in difficulty in the future if we wanted to add custom features to these hierarchy views, but at this time, the extra work saved was worth it. Alternatively, if we needed additional functionality in the future, we could move away from this package and instead build our own hierarchy view system. Another weakness, however, was that since jqtree was not built with our reservation management system in mind, we had to do some additional work to make our data compatible with the format in which jqtree required its data to be in. This involved new, recursive functions which created a separate, different copy of our resources which would be usable by jqtree. These functions renamed keys, generated additional lists of children which had to be searched for from our mongo collections, and filtered out unneeded resources. Even with this extra work, we believe using jqtree was still a good choice because the time saved outweighed the time added by this compatibility work. Making jqtree’s interactive features tie in to real changes in our data ended up being extremely easy to do because of the way jqtree allowed for writing custom

callback functions which would be run whenever something in the UI was changed. Therefore, we simply needed to tell the package to run our server methods which modified the hierarchy data when a user dragged a resource to a new position. This is an advantage which made some potentially complicated features very easy.

Reservations:

To create reservations, we decided to reuse the exisiting compound resource reservation system since that was the simplest option. We edited the 'createReservation' method to have it recursively get all of the descendants for the chosen resources and add them to the reservation. The rest of the reservation code then worked as before.

Supplementary Material

Table 1: Key meteor 3rd party packages

Package	**Functionality**
filter-collections	Resource filtering
fullcalendar	Calendar UI
alanning:roles	User permissions
swagger	Generating API documentation and test API methods
http	Duke NetId Login
collection2	Enforcing schema for Mongo Database
jqtree	Visualizing/editing resource hierarchy

Figure 1: Crash log from Roles package

```
1  MongoError: insertDocument :: caused by :: 11000 E11000 duplicate key
   error index: ResourceManager.roles.$name_1  dup key: { :
   "view_permission" }
2      at Object.Future.wait
   (/opt/ResourceManager/app/programs/server/node_modules/fibers/future.j
   s:398:15)
3      at [object Object].<anonymous> (packages/meteor/helpers.js:119:1)
4      at [object Object].MongoConnection.(anonymous function)
   (packages/mongo/mongo_driver.js:736:1)
5      at [object Object].Mongo.Collection.(anonymous function)
   (packages/mongo/collection.js:590:1)
6      at [object Object].Mongo.Collection.(anonymous function) [as
   insert] (packages/aldeed_collection2-core/lib/collection2.js:101:1)
7      at Object._.extend.createRole
   (packages/alanning_roles/roles_common.js:73:1)
```

Conclusion

Building this Resource Manager for the past semester has provided engaging practical experience on real-world software development. We have two main take-aways from our experience. First, we have a greater appreciation for good design since it has a significant effect on the ease/difficulty of future requirements/changes. Second, we consider framework choice very important since it has massive implications on what can be done and also influences design decisions.

Contributions

To this evolution:

Davis: Testing improvements, permissions UI improvements
Allan: Child reservation
Emmanuel: Resource Sharing
Sam: Hierarchy system and views

To project as a whole:

Davis:

- reservation calendar display, reservation detail view, server methods for managing reservations
- API / swagger docs
- compound reservations

Allan:

- Search/filter, Tags filters, Filter-collections bugfix
- Shibboleth/Oauth
- edit reservations

Emmanuel:

- Search/filter
- Shibboleth/Oauth
- Restricted resources
- Shared resources

Sam:

- login system, base meteor functionality, admin creating/editing resources, admin enrolling new users
- Permissions/Groups
- approval/denying, new overall UI
- Hierarchy system