ECE 458 Resource Manager: Evolution 2
By Davis Gossage, Emmanuel Shiferaw, Sam Ginsburg and Allan Kiplagat

Retrospective Analysis

For Evolution 1, we chose the MeteorJS framework and designed our application to be centered on a simple MongoDB data model (resources and reservations collections) and several meteor packages for different functionalities (resource searching, calendar display, etc).

We are happy with our MeteorJS framework choice since it accomodated all of the new requirements well. As anticipated, the simplicity of the data model added work for this evolution, with the new resource permissions requiring adjustment of the model and application logic, but making these adjustments was straight-forward given the initial model simplicity.

We had identified our heavy use of 3rd party packages as a vulnerability, given that requirement changes could force us to do major customizations for the packages or make design changes. Fortunately, the requirements tied to the main 3rd party packages - *filter-collections* for resource filtering and *fullcalendar* for the calendar UI - did not change for evolution 2. Using the *alanning:roles* package for Evolution 1 also simplified implementation of user permission requirements because the package supports permissions and groups out of the box.

Analysis of Current Design

As in Evolution 1, we are using the MeteorJS framework, along with a MongoDB data model and several meteor packages. Resource filtering and calendar UI requirements did not change, so the implementations using the *filter-collections* and *fullcalendar* packages stayed the same.

We use three other major packages to implement key features: *alanning:roles* for user permissions, *swagger* for the API and *http* for Duke NetId login. These packages speed up development and minimize complexity but as we mentioned in the evolution 1 writeup, they can be a liability if requirement changes are not supported thus forcing package editing or application redesign.

For the API, the pre-existing server methods were exposed using Meteor route configuration. Our server maintains a subset of methods that we expose to the API. This object contains the method names as well as the expected parameters for the method. When a POST request is made to the /api path, the router pulls the parameters and the requested method name. It then makes sure the user has authenticated with their API secret and that the requested method is publically exposed. A feature of JavaScript called function invocation with apply() is used to call the function with a string and pass parameters in as an array of variable length. This approach lets us easily add new API methods by simply writing them on the server and

exposing them. All we need to do to expose an existing server method is add the name of the method and the expected parameters to our list of exposed methods. We think this solution is flexible and abstract enough to handle any future requirement changes.

The swagger package is used to generate API documentation and test API methods. We chose swagger because of its simple setup and rich feature set. We add our exposed methods and parameters to a JSON file, specify the endpoints and what kind of security is required to run each method and then swagger shows all of the methods broken up by tags and includes a test bed feature where someone can enter their API secret and test every method. One weakness we see moving forward is that swagger is not capable of auto-generating JavaScript documentation, this means that we must keep the outward facing swagger docs up to date with the internal server method documentation. We think the features of swagger far outweigh this minor inconvenience.

For the Duke NetId login, we had originally envisioned using an oauth meteor package but it was undocumented and we decided to implement oauth ourselves. The most ideal approach would have been to build a Duke Oauth meteor package that integrates well with the existing *accounts-password* package used to manage user accounts. However, we received Duke oauth credentials & instructions from OIT later than expected and we chose to do a quick implementation in order to meet the evolution deadline. After receiving the user's netid email from the Duke Oauth server, we check whether the user has an existing account in *accounts-password* and create a new account if not. One limitation of this approach is that users cannot use the same email address to sign in regularly and also with Duke Oauth. We intend to build a separate duke oauth package to overcome this limitation and conform to meteor's oauth conventions.

For user permissions we use the *alanning:roles* package that supports both permissions and groups. Users can perform actions if they have the required permission themselves, or if their group has that permission. For resource permissions, each resource has lists of *view access* and *reserve access* permissions, and only users bearing those permissions can view and reserve the resource.

Contributions to this evolution:

Davis:

- API / swagger docs

Emmanuel:

- Shibboleth/Oauth

Sam:

- Permissions/Groups

Allan:

- Shibboleth/Oauth