Evolution 1 Written Analysis

Evaluation of Stack Choice

MySQL - Database

For this project we chose MySQL for our primary database. So far, our choice of MySQL has been a great decision in complementing our stack. In addition, it has worked well with respect to Evolution 1. We made this decision over nonrelational or schemaless databases (like Mongo) for several reasons:

Relational Data:
The data in this project is largely relational. The relationships between reservations, users, resources and tags are very often bidirectional. For example, given a reservation we need to be able to view the user who created it, and given a user we need to be able to view that user's reservations. This relation is made very easy and clean in a schema-ed database. This also makes it very easy to add any additional relationships (e.g. tags-users, tags-reservations, etc.) in the future.

The cascaded delete feature of SQL also makes it very easy to ensure that no orphans exist when deleting highly relational data. Where a database like Mongo would require several different queries to delete a resource with many relations (a resource with a tag or a reservation made by a user), SQL takes only one. This saves a lot of developer time when adding features, as well as keeping our code more simple as more relations are added.

Powerful Queries:
Because SQL has very powerful, brief queries in a standard language with lots of query generation libraries, it is quick to write a lot of business logic into SQL queries. At this point, the majority of our business logic (with reservations, tags etc.) is relatively simple -- and all of this logic is directly written into queries. This allows all of our services to have a very standard structure, and for a lot of our query generation code to be reused.

In addition, in terms of performance, because all of our data is baked into our SQL queries, we do not need to pull extra data from the database and then post-process the data. Instead, we only pull the data that is relevant under the given constraints. For now this isn't a huge issue, but with more data and more requests, this could provide strong benefits.

Strong Schemas:
The strong schemas of MySQL force us to create well-defined objects (users, resources, etc.) and explicitly defined relationships with relational tables. This is valuable when adding features because it makes the creation of 'special cases' very difficult. For example, if new features were

added to certain users, SQL would require the creation of a well-defined schema for these new features. With a schemaless database, developers are enabled to easily create ‘special’ user documents (with no well-defined schema of the new features), which can become extremely unwieldy as more features are added.

Node.js - Backend Framework

We chose to use Node.js as our server side framework. We chose Node because compared to other back end frameworks, we felt it was the easiest out of the choices (Rails, Django, Play, etc.). Node.js is incredibly powerful, works at a very high abstraction level, and makes life really simple for the server side developer. One of the useful features of Node is that it utilizes an event-based asynchronous execution of I/O operations from a thread pool. The developer uses Node as a single threaded framework, however Node actually acts as a single-threaded listener that then delegates work to a pool of threads.

Node.js only exposes a single thread to a user, making developing very easy since applications are coded as if they are single threaded. After a connection has been established, the single thread will do one of two things depending on the workload. For non I/O bound work, a single thread will simply execute and block during the request. However, for longer requests like accessing a database (which is very prevalent in our application), then the single thread will delegate the task to a pool of underlying native C++ threads in an asynchronous fashion. The main single thread chooses an I/O thread to do the work and kicks it off with a callback; the main thread then returns to listening for the next connection.

Another big positive for Node is the giant community that it comes with. Because Node has become so popular, there are many different frameworks and libraries that make our lives significantly easier as developers. There are a lot of things done for us, as well as many developers that have experienced issues similar to ones we have. The community on Stack Overflow is very useful, as well as the plethora of tutorials available online. We also make use of npm, the node package manager. This allows us to easily download and fix applications that are available.

One of Node’s biggest weaknesses, however, is that it is not very good at CPU-intensive work. If the connection required is very CPU-intensive, then this will block of the application and lead to bad performance. However, we concluded that our application would not be CPU-intensive enough to tie up the single thread, and therefore we chose Node because of how simple it makes developing and the power that is in the hands of a developer to write a single threaded program with good performance.

Angular.js - Front End Framework

We chose to use AngularJS as our frontend Javascript framework for a variety of reasons.

MVC Design:
AngularJS strongly encourages the use of an MVC design. Each HTML template represents a view and is paired with an associated controller. Writing a modular frontend is easy - components can be separated into different views, each with their own controller (which can comprise logic, API calls, and data manipulation).

The view and controller are bound together through the $scope (a special javascript object) which acts as the model. By means of the $digest loop, all changes to the model are immediately reflected in both the view and controller - the developer then does not need to propagate data back and forth.

Abstraction:
Factories, services, and providers take the form of javascript objects and can hide the implementation of various application logic. Within our implementation, we created services to help in executing resource based logic (client/services/resource.js) as well as construct a timeline based on input data (client/services/timeline.js). Each service can be injected into a controller that needs the appropriate functionality.

Incredible Online Community:
Despite being inexperienced front end developers, we have created the UI for our application relatively easily. This is due to the huge amount of information found at https://docs.angularjs.org/api as well as a large Stack Overflow presence.

Dependency Injection:
Because AngularJS handles dependency injection, we can instantiate any pre-existing or custom service/factory/provider without having to worry about anything that it depends on.

Built in Services:
Various services are built into angular and easy to use. Commonly used services within our app are

$http - handles all HTTP requests to our Node backend
$scope - allows for a model to connect the view and controller
$q - helps in resolving the outcome asynchronous events (promises)
$location - allows us to easily change between views within our app

Unit Testing Framework:

Angular provides a unit testing framework (Jasmine) as well as a means of running tests in the browser (Karma). Though we did not have enough time to construct a front end testing suite, we hope to do so in the future.

However, AngularJS does have a few weaknesses that are worth note in future evolutions.

Handling Large Amounts of Data:
Angular is slow when handling applications with large data sets. This is primarily due to the dirty checking feature (which must check all model data on a $digest loop for possible changes). If our application begins to slow down, we must consider the amount of data being passed to the front end via backend routes.

Prototypical Inheritance of $scope:
The angular $scope object inherited prototypically from parental $scope objects. This essentially means that controllers can refer to objects in parent controllers. This can cause issues if a sub controller is paired with a view that does not have a parent view/controller.

Evaluation of Current Design

API/Services:

Modularity/Separation:
The API which connects our Node backend to the Angular frontend is designed to be as modular as possible. Endpoints serve as methods which do one thing (to one type of object). This means that our API is very simple -- little more than CRUD endpoints exist for each entity in the database. There are exceptions, such as a filter tags method (which looks at both tags and resources) and an endpoint that will create, and optionally add tags to that resource. However, our current design makes it easy to extend the database schema. For example, because tags are only associated with resources in a relational SQL table (tag_resource) -- and not in any services -- it would be trivial to add tags to any other object (reservations, users etc.).

This also lets us minimize the amount of side effects that modifying one service or schema could have. If everything is separate, changing something in one schema should not change anything in others. This saves a ton of time in debugging, especially when multiple are working on similar parts of the project. This also keeps models and services very simple, short, and easy to integrate with the frontend.

The extent of the modularity in our code was largely made possible by the strong and well-defined schemas in SQL. In effect, we did not have to put any data from another object in the schema for any other object (e.g. resources contain no reservation data, resources contain no tag data, etc.) and we can build services interacting with one table and one table only.

Routes vs. Services:
In addition to modularity, we divided most of our API logic into two layers: services and routes.

Routes provide the request and response processing logic for a given endpoint. They cover all of the logic behind validating sessions, authorizing admin and users, extracting and validating information from request bodies or queries and formatting data/errors to be sent in the response.

Services deal with all of the internal logic for queries. A service contains all of the business logic and SQL queries necessary to respond meaningfully to the query. The services are called from the routes (e.g. the create_resource service is called from the PUT method at /resource).

Having these two logic layers allows for a separation of policy and mechanism. This means that if we needed to change the method we used to collect data (say we changed our table structure or the node middleware we are using) nothing would need to be modified in the receiving and sending of the data in the route. Only the respective service would need to be modified -- and there would be no respective change in the view. Alternatively, a change in how data is sent or received would only touch the route and not the service.

Another benefit of this structure is code reusability. Because the services are largely independent of the route itself, the same service can be used many times over. For example, the service that queries the database to check if reservations conflict (*get_conflicting_reservations*), is used to process GET, PUT and POST methods in the reservation routes. Additionally, if a large change occurred required reorganizing the API and database, the logic in many services could still be used with very minor modifications -- allowing the change to be made in the routes and tables, without touching the logic.

Maximize returned data:
For every endpoint which returns data, we made a decision to return as much data about a given object and its relations as possible. This is especially relevant when dealing with very relational data (resources, for example, have associated tags and reservations). In the resource GET method the names of the tags and the reservation information would be returned.

As the project scales and relations between objects become more complex, this approach could become unwieldy. However, at the current scale of the project, this approach allows the view to add features without needing to modify the API.

This is another area where our choice to use SQL has become very valuable. Adding extra relational data to GET queries requires only additional table joins, with no change to the underlying schema. This allows us to very quickly change what relational data is returned with very little change to the query, no need for redundant data in the database and no change to the underlying schema.

Scheduled events:
For our scheduled emails, we use a scheduler library for Node (Agenda). Whenever a reservation is created or updated, a callback is scheduled for the start-time of the reservation. At that time, a SQL query runs that ensures that the reservation still exists, and that the start time is still the same. If not, the job dies and does nothing. If yes, the email is sent. These callbacks are backed by a Mongo DB, so they persist even if the app itself dies.

This provided significant advantages to MySQL's built-in scheduler. Primarily, this scheduler allowed us to trigger functions on certain events, rather than to poll for database changes. This is a much more scalable option; if thousands of resources exist in a database, thousands of queries would have to be executed every few seconds.

However, when triggering events, these queries only happen once -- at the time the reservation was originally scheduled to start. These scheduled functions can run arbitrary javascript code, it is also much easier to add arbitrary scheduled jobs if they arise, whether they require database queries, pure javascript or a mix of both.

RESTful consistency:
Although not as big a deal without a public API, we have made an effort (and still are) to return meaningful error information and codes at every endpoint for most possible errors. Ideally, for every endpoint and error, both a meaningful HTTP status and error message would be returned. While still a work in progress, this descriptiveness will allow both tests and the view to distinguish between different types of errors, and make life easier for both developers and clients.

Session persistence, authentication, and encryption:
In order to implement session persistence and authorization, we used a node module connected to a redis store, to store database backed session cookies. In the vein of our above dedication to maximize returned data, the session store contained all user information (except for the password), and was easily accessible at every route. This ease-of-access allowed us to be very flexible in creating relational data between objects and users.

The persistent session cookie also made it very easy to share authentication information between the front-end and the API. We used another node module to set permission limits for each endpoint. Although we only have user and admin permission levels at the moment, this node module makes it very easy to extend that to provide any number of permission levels with minimal developer effort.

We obtained SSL certificates to enable encryption on our servers and support the HTTPS protocol. Our servers are:

Development Servers
Ashwin -> http://colab-sbx-366.oit.duke.edu

Chris -> http://colab-sbx-202.oit.duke.edu

Test Server

Rahul -> http://colab-sbx-212.oit.duke.edu

Production Server

Stephen -> http://colab-sbx-123.oit.duke.edu

On launch, an admin user is created with the username “admin” and password “Treeadmin”. For future evolutions, we will try and move these configurations to a config file to allow the sysadmin to customize this behavior.

Complicated callback structures:
Because of the asynchronous and event based execution of MySQL in Node, every SQL query is followed by a callback. While this is an efficient way for the server to handle requests, it also makes the flow of the routes and services difficult to follow -- especially when multiple services cross-over in an endpoint. As our code gets more complicated and more services are created, this could seriously slow things down and introduce many unforeseen bugs.

In the next evolution we are going to look into libraries like *async.js*, and other callback managers to reduce the complexity of multi-callback chains. This will hopefully make it much easier to manage and develop new services as we go forward.

Lack of testing
Our lack of any kind of consistent testing framework has been a huge hindrance to us so far. On more than one occasion, a broken version of the API has been pushed -- and the bug was not necessarily caught until a few days later. This is definitely not scalable as we start to add more and more endpoints and services with potential side effects in other parts of the app.

Towards the end of this evolution, we began developing some makeshift testing in manual test scripts and a python script that hits and checks the results from most API endpoints. Going into this next evolution, we will be looking to build unit tests with Mocha.js, and hopefully get a CI system set up on top of those tests. Having consistent, function level tests to ensure that broken code is not pushed to GitHub will be a huge plus in our ability to spend time developing rather than debugging.

Database Schemas:

Normalized schemas:
For our database schema, we used very normalized tables with each discrete entity (resources, users, reservations, tags) represented as a table.The relations between each table are represented by relational tables (reource_tag, user_resource, etc.). The references in the database are also closely managed to allow for very easy cascade deletion of all children.

As mentioned in the framework section, this design allows us to add relations to data without changing the existing schema at all -- all we need to do is add a relational table, and slightly modify our services to create these relations. This design should dramatically improve extensibility and issues with side effects as the project goes on.

Event based SQL pool:
The Node module we are using for the Node interface with MySQL provides a very convenient event based response system. In this system we can trigger a function whenever a table row is returned, whenever an error is returned, and whenever the query is finished. The event based system is derived from this stack overflow post:
http://stackoverflow.com/questions/17015590/node-js-mysql-needing-persistent-connection

Our choice to use this module has proved to be hugely useful. The different events allow for a huge amount of customization and reuse in our services. For example, because we could trigger different callbacks when checking for reservation conflicts, we were able to reuse the same service in three or four different endpoints, with different behaviors on different events. As we start to build even more services and business logic, the ability to reuse our services for different purposes will become incredibly valuable.

Front End Evaluation:

Modularity:

One strength of the current design is the manner in which different components are kept modular. This is primarily due to the way AngularJS handles the MVC relationship, however our design focused on subdividing the system into smaller, more manageable components. For example, the functionality that allows users to filter from reservations (with included/excluded tags, start time, end time) is handled through the view filter_reservation.html, the controller filter_reservation.js, and the service timeline.js.

Future evolutions that require additional functionality can solved either by

- Adding a new view and controller (for truly new functionality)
- Creating or refactoring functionality into a service (for functionality that needs to be shared)
- Modifying existing controllers (for changes to current functionality)

Maintainability:

At its current state, we believe the front end to be maintainable. We tried to keep components simple and readable, such that any developer could help extend and maintain the UI.

However, AngularJS does have a learning curve, meaning that unfamiliar built in services may be difficult to use at first for beginners.

Robustness:

As a group, we feel that we have implemented a robust frontend. We have gone above the official Evolution 1 requirements, and hope that by validating all user submitted data and providing success / alert messages, we will have an easier time with future requirements.

Testing:

Unfortunately, we do not have a testing suite implemented for the front end. It proved difficult to balance finishing and improving upon the Evolution 1 requirements and creating thorough tests. In the long term, we hope to have an extensive test suite which would give us more confidence when refactoring code.