ScrumSpace Database Submission (#6)

Overview

This report details our transition from a mock database to a local MongoDB instance. We'll be exploring in detail each individual route transformation, as well as any further modifications needed to successfully move our Scrumspace to a functioning database.

Running the Application

Just as before, our application is using Gulp to build and package our application. After dependencies are installed with `npm install`, running an `npm run serve` will trigger the Gulp task and then start the server synchronously.

Significant Changes/Improvements

Authentication

As mentioned in our last report, our implementation of protection/authentication is in its infancy, however it has expanded since our last report. The expansion can be broken down into a couple of different improvements. Firstly, a login screen was designed and built. This view is conditionally rendered on the client. Secondly, we have built an encrypted JWT exchange system (as detailed in class). This token is provided on POST to /login (with details from the login view), and stored in localstorage for the client to use as an API key. Thirdly, we have implemented general and project specific authentication as *middleware* in our express server. Our *general* loginAuth middleware protects the whole API against unauthorized access by validating the token sent by the client. Our *project specific* middleware protects all project API routes. It works by extracting the user's ID from the provided token and validating that it is a contributing member on the requested project.

Data Restructuring

Our data structure has gone through a significant *restructuring*. For a while, we were using a highly embedded structure for our data. This worked well for our team because it highlighted the embedded nature of the data itself. However, when deciding how to approach our implementation of an actual database, we more seriously considered issues of concurrency atomicity. We realized that while MongoDB can handle embedded data, writing atomic queries that supported modifying documents as *deeply nested* as ours would be an incredibly difficult task.

With a consensus from our group we decided to *flatten* our data structure. All nested documents were moved to the root level of our database. This means we have separate collections for users, projects, sprints, stories and tasks. While some routes grew in complexity due to this change, most became significantly simpler and allowed for atomic queries.

In order to cleanly support this restructure, we built a packaging service for our projects. The service is called when a client-side application requests an initial state, and packages a project into its original nested structure. We are considering refactoring the client-side to support flatted data natively in the future, but have not made any solid plans to do so as of yet.

Individual Contributions

Dylan Fischler

GET /api/user/search?searchStr=someString&key=someKey

For the search route, I wrote a generic collection search utility (src/server/api/shared/search) that receives a collection name, a search key and a results limiter. While the utility is currently only being used for user searching, it can be used elsewhere if needed in the future. The query runs a regex case-insensitive search across the specified collection on the specified key.

PUT /api/project/:project_id/story/:story_id/task/:task_id/assigned_to

This route assigns an array of user IDs to a task. I removed the ability to add/remove users from the task because while our client-side emulates adding/removing users from a task, behind the scenes it just *replaces* the assigned users. As a safety check, the route validates all user ID's supplied before modifying the task.

PUT /api/project/:project_id/story/:story_id/task/:task_id/blocked_by

This route is exactly identical to the assigned_to route specified directly above, however it deals with Task IDs instead of User IDs.

GET /api/init
While designing the project packaging mechanism, I updated this route (previously Abhay's responsibility) to support the new database. It's largely identical to Abhay's previous implementation, with added support for project packaging.

The packageProjects utility (src/server/api/shared/projectUtils), is a fairly complex service (I hope to simplify it somehow). As mentioned above, the utility was written to allow for an easy refactor to a flattened data structure, packaging the referenced entities from various collections into an embedded format for the client.

Note: *The following two routes were not completed by the assigned developers in time for an agreed upon "merge and fix" deadline, I wrote the routes optimistically in case deadlines were not met, and merged them in when the deadline had passed. This has been discussed with the assigned developers, and their work on the below tasks will exist in separate branches (specified in their writeup sections)*

PUT api/user/
This route allows the client to update their user settings. The route grabs the user_id passed through from the loginAuth middleware to ensure the user can only update their own settings. If password details are set in the body, the route validates the old password against the object in the database. Following, the route then updates the user object in the database and sends it back to the user (Socket support needs to be added). It's important to note that this route has not been properly hooked up to the UI (this was another founders responsibility from a couple submissions back).

PUT /api/project/:project_id/story/:story_id/task/:task_id
This route updates the top level fields on a Task object. If a task status is supplied, the route pushes a history record into the Task. Additionally, if the task is moving from Blocked to another status, all blocked tasks are cleared from the record. In order to enforce atomicity, the route constructs the full update object manually before running the query, due to conditionally applied changes. The logic is located in src/server/models/Task.js.

Socket.IO
While websockets were always on our todo list, we realized that it was more of a necessity than we initially thought. When thinking about how our application would actually be used, we realized that features like the Scrum Board and Project Planning would often be occurring simultaneously on multiple clients. In order to support this, we needed an always up-to-date design. In order to interface Socket.IO with our redux layer, we decided to send actual Actions

in our socket messages, and simply proxy these directly to our client side Reducer. Using Socket.IO with our already existing Actions allowed for a pretty simple drop-in implementation. Known issues are detailed below in the Lingering Bugs section.

Authentication
Our authentication (located at /src/server/api/shared/authentication), is separated into two pieces of middleware. In order of use, requests first hit loginAuth. The middleware extracts a user_id from the token (using jwt verify) and passes it to our isUserValid function, which then verifies if the specified user exists in our database. If all is good, it calls the next middleware function, otherwise returning a 401. For project routes, we have additional middleware (the default export in authentication.js). This function assumes a valid user (from loginAuth), and validates that the user is a member of the project in the request. This project authentication middleware is very strongly based on a previous authentication middleware that Abhay wrote (just modified to support our new loginAuth middleware).

The benefit of moving authentication handling into middleware, is that it allows us to force consistent handling of authentication across all routes, multi-level authentication and limited code duplication.

Abhay Vatsa

All these routes are authenticated by middleware

POST /api/:project_id/story/
This route is what creates a story in the sprint planning view.

It will allow the user to create stories. By default, these stories have a null sprint_id. When viewed in the application, this is visible in backlog. These stories each contain a title, detail and the foreign keys to tasks.

After the database saves this new story, it will emit a redux action via socket.io.

PUT /api/:project_id/story/:story_id

Put stories will allow users to update the story matching *story_id* in the sprint planning and scrum board views. In sprint planning it will let you move stories to different sprints. In scrum board, it allows updating the title and description.

After the database saves this new story, it will emit a redux action via socket.io.

In the future, it will also handle adding and removing tasks. We moved from a hierarchical data model to a flat data model. This specifically made updating tasks attached to a story more complex. Previously we would just update the array of tasks, but with foreign key references this becomes more challenging.

Another advanced issue that exists is that we need to send nested data back to the client. This causes some issues what application data is expected. On put, I need to nest all tasks data into the story.

This entire interaction is quite fragile and needs to be redesigned. Perhaps as a sequence of modifications that are sent to the server. I would like to design a client on the client that caches data, invalidates old data and sends new data to redux.

DELETE /api/:project_id/story/:story_id

Delete is used in sprint planning to remove a story matching the story_id from the backlog or from sprints. If a story is removed from a sprint, it will move to a backlog. If a story is removed from the backlog, it is deleted from the project.

When the database operation is done, a socket.io event is sent to the client.

Ryan Jerue

POST /api/project/:project_id/sprint

This route is what creates a sprint object at the top level. It pushes the objectID of the newly created sprint to the project object that has the id matching project_id. The route undergoes authentication via middleware by checking to see if the user creating the sprint is a member of the project. Validation is done via a schema file. The route then uses sockets to return a new sprint object from the server to place into the redux store.

PUT /api/project/:project_id/sprint/:sprint_id

This route is similar to that of the POST, however it instead edits a sprint of the given sprint_id that matches the objectID in the database. The route also undergoes authentication via middleware by checking to see if the user creating the sprint is a member of the project. Validation is done via a schema file. The route then uses sockets to return an edited sprint object from the server to change what is in the redux store.

DELETE /api/project/:project_id/sprint/:sprint_id

This route deletes a sprint that has an objectID that matches the given sprint_id. A sprint will not be deleted if the user tries to delete the current_sprint. Additionally, all stories that are in the deleted sprint have their upreferenced sprint_id set to null. Finally, the ObjectID of the Sprint is also removed from the project that the route is associated with. The route also undergoes authentication via middleware by checking to see if the user creating the sprint is a member of the project. The route does not require validation. The route then returns an updated project object with everything embedded from the server using a function called projectFromID. projectFromID works with the packageProject function by filtering the projects that are packaged to find the project of the correct ID.

PUT /api/project/:project_id/sprint/:sprint_id/start

This route is a put route that changes a project's current_sprint to the given sprint's ObjectID. Additionally, it also set's the start date of the given sprint to the current datetime. The route also undergoes authentication via middleware by checking to see if the user creating the sprint is a member of the project. The route does not require validation. The route then returns a project object of the given project via projectFromID as described above.

PUT /api/project/:project_id/sprint/:sprint_id/stop

This route is a put route that changes a project's current_sprint to null. Additionally, it creates an end_date of the given sprint to the current datetime. The route also undergoes authentication via middleware by checking to see if the user creating the sprint is a member of the project. The route does not require validation. The route then returns a project object of the given project via projectFromID as described above.

Supriya Kankure

PUT /api/user/

This route updates the user settings. When the user visits this page, the user has the option to choose what fields (display name, first name, last name, and email) to update. Additionally if the user wants to change their password, they must type their old password, and a new

password. If the old password entered matches what is in the database, then the password will be reset to the new password the user types in. For this assigment, due to several technical issues, I could not complete my route on time, and another founder fixed it. To view the code I was able to get done, please check the branch usersetings and please see src /server /api/user/index.js. There are several issues with this code being that it does not correctly update the database, and causes errors in postman.

Rachana Lingutla

POST /api/project/

This route creates a new project. Despite the data restructuring, my routes have not undergone much change, as projects was a top level collection to begin with. The functionality basically creates a new project object based on the parameters passed. The project is assigned a random project ID number and then pushed to the database. The route undergoes a twisted variation of authentication. Anyone can create a project, but if the logged in user doesn't assign themselves to the project, then the new project won't appear in their projects page as they haven't been assigned to it! The route then uses sockets to return a new project object from the server to place into the redux store.

PUT /api/project/:project_id/

This route edits a project based on the settings passed on the changes passed into the title and users field. The 'update project settings' functionality undergoes authentication via middleware by checking to see if the member attempting to edit the project is actually on the project. The route then uses sockets to return an edited project from the server and also changes the data in the redux store.

DELETE /api/project/:project_id/

This route deletes the project with the ID that matches the id of the project passed in as a parameter. This undergoes authentication so the project can only be deleted by members on the project. The route does not require validation. The database function returns the id of the deleted project so that the reducers can remove the project id from our redux store of data.

Migrating from Mock DataBase to MongoDB

I also aided the transition of our project from the old database to the MongoDb one. It took some time to to get mongo express to work and to figure out where to place MongoClient, but in the end we got it working.

Niharika Venkatathri

PUT /api/project/:project_id/story/:story_id/task/:task_id
This route updates the status and description of a task when edited. It calls the Task.update function in src/server/models/Task.js. This function was worked on by two individuals separately. My version was not chosen for the final submission in master. My version of the function which interacts and works with MongoDB can be found in taskDetailServerRoutes. It works exactly like the one in master does. The database function returns an updated Task object upon calling this function. This function uses find to find a task object and once found, calls findOneAndUpdate to update the particular Task object. The Task object to be updated is constructed once the Task is found. This is done manually and not with the help of $set because some of the fields only need to be updated conditionally.

Migrating from Mock DataBase to MongoDB
I aided in the initial setup and transition of the data from Mock DataBase to MongoDB. I specifically went through the data structure and updated each id to an ObjectID. I combed through each file in the architecture to make sure that we didn't miss any small details. Additionally, I helped to get Mongo express working.

Lingering Bugs

Sprint Review and Sprint Ending (Dylan Fischler)
We've made progress on the work needed to complete this feature. A sprint now supports transitioning to a Review state, and Ryan has built the routes required to end a sprint. What's left is building the ability to end a sprint from the client side. We have future plans to build a more helpful and in depth review phase, where a team can actually review everything completed during the sprint period.

Sockets
Implementing Socket.io in our project was a huge step forward, however there are a couple lingering bugs left to fix. Firstly, sockets are not encrypted, we need to research how to protect our sockets from malicious intent. Secondly, all changes are broadcasted to all clients. This is an obvious security concern and we have plans to fix it (with Socket.IO Rooms), however it does not break anything on the client-side.

User Settings (Supriya Kankure)

The save buttons in the user settings page should trigger the server routes, but as of now are not connected. This is an issue with the client side, which will be fixed, so the user can interact with the page. I will be working on fixing this.

Need to refresh page in order to use project functionalities (Rachana Lingutla)

For all project functionalities, I am finding that I can only manipulate project settings (updating and deleting) on new projects after I have refreshed the project page after the project has been created. The same goes for updating the title or users in a project. The changes can only be seen once you refresh the page. However, delete project seems to work fine, so I'm guessing that it's an issue with the reducers. If I get a chance to look at this assignment in the future, I'll probably fix this.

Awkward issue with creating a new project (Rachana Lingutla)

Currently, when creating a new project, the currently logged in user does not get automatically added to any project they create. So if they forget to include themselves, they will notice that the project doesn't appear after a page refresh in the projects page because the logged in user isn't authorized to work on the project. Again, if I get a chance to look at this in the future, I'd ideally like to automatically add the logged in user to any projects created and then remove the logged-in user as an option in the multi-select feature in the modal. This way users will be able to create new projects without too much hassle.

Project and User Avatars

Both Project and User entities have avatars to aid in the visual identification of each. Currently our mock data contains these avatars, however we do not yet have the functionality for a user to upload these avatars for new entities.

Resolving Flat Data on client

With our new flat data model, we need to have a layer that handles invalidating old client data and minimizing data sent across the wire.

Removed Features

We have decided to remove Task Deletion and Task Creation due to the fact that there is no place in the UI where the user can do this individually. We made this decision to enforce the practices of Scrum, motivating teams to stick to the sprints planned before sprint commencement. If a user needs to remove or add a task to a sprint, they can do so in the Sprint Planning page.

Rachana Honor's Feature

This will follow in a separate report by the extension deadline.