Reflection
Through this project, I have learnt how to write a frontend and backend application and how to make them communicate. Although this seemed to be a basic application at first, the actual implementation turned out to be surprisingly complex.

The most difficult part of this project was learning Ruby on Rails' various domain specific languages. DSLs are extremely powerful as they allow us to write "code that writes code". For example the use of ActiveRecord Associations allows so many internal operations to be abstracted into merely a few statements. However, the process of learning a DSL is almost harder than learning a new general purpose programming language, because each individual DSL has very specific applications and hence the commands of different DSLs can differ greatly.

I am perhaps most proud of the fact that my application is able to handle a large number of entities (User, Task, Project, Subtask, Comment, Filter, Label, Tag). The object-oriented approach I took allows the project to be readily extended to add new features. For example, I can easily add a new color field to the Tag entity. Of course, this meant that the redux store is rather massive, but it was worth the effort.

The only incomplete feature in this app regards shared projects. Previously working functionalities like adding, deleting, editing tasks within a project all required significant refactoring from both the frontend and the backend, and I have say, my code looked quite a bit uglier. Hence, I decided to stop at making things work for creation and deletion.

There are also some things that I have yet to try, and I will read up on them in future when I have the time.

1. Greater separation between front end application and back end application. Instead of relying react-rails' integration of React and Rails, I would like to make an application where these are completely separate application.
I believe cross-origin resource sharing is a concept relevant to this (?).

2. More advanced usages of Redux. Right now my code still has plenty of repetitive code. I would like to learn more about higher level abstractions (e.g. higher order reducers) with respect to action creators and reducers.

3a. More advanced usages of React. Currently, I have only used the basic hooks offered by React. I would like to learn more about memoization, useCallback, custom hooks.

3b. While my application is a working one, there are still some parts I have yet to fully understand. A major one is avoiding null reference errors when dispatching async thunks. E.g. If I have the following series of dispatches:

```
thunk {
      dispatch(actionA) // creates new reference in Store.allObjA.objA.relatedobjB
      dispatch(actionB) // updates Store.allObjB and adds the new objB
}
```

Dispatching actionA creates a reference to a new objB that has yet to be added to the store. The react application then tries to render in the objB that has yet to be added by actionB. Of course, a simple way to avoid this in the case (which I used) was to switch the order of dispatching actionA and actionB. However, this definitely would not work for a more general case. I wonder if there is a better way to do this.

Optional objectives:
Cron scheduler

During development, since I was changing my schema rather regularly, coupled with logical errors in my code that did not manipulate the database correctly, rails db:schema:load + rails db:migrate was ran very often. So, I did not find any use in doing database backups.

TypeScript
I wrote my application originally in Javascript, as I am much more familiar with JavaScript and I decided to stick with what I know best.
After I finished implementing most of the functionality of my app, I decided to learn some TypeScript, and have converted most of my .js files to .ts/.tsx files.
Benefits
1. TypeScript makes auto-imports and code auto-completion a lot easier.
2. Also, there were a number of minor bugs in my code like passing the wrong parameters to a function that I would not have spotted with vanilla Javascript. TypeScript's stricter requirements highlighted those errors to me.

In retrospect, I believe I made the right choice using JavaScript first. The conversion from JavaScript to TypeScript was very tedious. If I had started with TypeScript, the learning curve of TypeScript, combined with the learning curves of Ruby, Redux and React may have been too much to handle. After all, one of the advantages of JavaScript's ducktyping is that it is very easy to prototype code. And for smaller projects, debugging with runtime errors is rather viable as well. For a single-person project like the TodoApp, JavaScript serves its purpose very well.

Hosting
My application has been deployed to Heroku. The deployment link is in my project description on GitHub.

Redux
Redux was crucial to the development of my project for the following reasons:
1. I had an object-oriented mindset when initially designing my database structure. Due to the large number of object types (Project, Task, Label, Subtask etc.) in my project, redux was needed to compartmentalize data fetched from the backend. (i.e. creation of rootReducer from projectReducer, taskReducer etc).
2. I have a large number of React Components, some of which are deeply nested. A central source of information is needed to reduce prop drilling and unnecessary fetches from the backend.
3. All communication (HTTP Requests) between the frontend and the backend lies within the ActionCreators files (xxxAction.tsx) in my application. This further improved the organisation of my code.