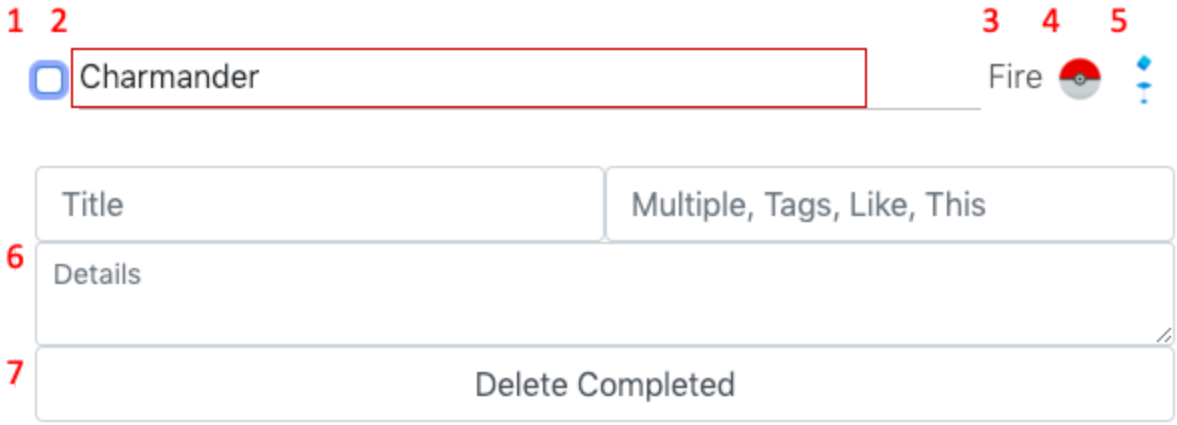
CVWO Final Submission

Larry Law, A0189883A
https://young-reef-14225.herokuapp.com/



User Manual

No	Use Case	Description
1	Check Boxes	Toggle to mark completed Status is preserved upon refreshing Data is not deleted
2	Quick-Edit	Click to trigger “Edit” action
3	Tag Filter	Toggles between index and tagged task **If user clicks on another tag while the view is not index, app will toggle back to index first before toggling to that tag.*
4	Show Details	Toggles between index and details
5	Edit	Same as Quick-Edit. Added as the Pokestop looked cool.
6	Quick Submit	Submits with “ENTER” key Line break with “SHIFT + ENTER” Title must have >0 words Form auto refreshes & focuses **On mobile, submit button will be displayed*
7	Delete	Deletes all completed tasks. Original and associated data are also deleted. (Explained further in “Accomplishments” section) ** In edit mode, delete completed button will not be displayed.*

*: Corner Cases. These cases are explicitly stated in bugtest.md

Accomplishments

My greatest accomplishments are the knowledge picked up. Thus, the focus of this section will be on my learning. Doing this assignment allowed me to understand...

1. Request-Response Cycle through implementing CRUD Operations

What happens when we type "google.com" into our browser? Our browser first goes to the DNS server to retrieve the real address of the website. Thereafter, it sends a HTTP request to the web server. The server then determines if the requested site is static or dynamic. If it's static, the server simply retrieves the files (HTML, CSS, and JS) and sends it back to the client. If it's dynamic, the server will forward this request to the web application. This is where Rails come to play.

The web application will look at the HTTP verb and URL of the request, and route it to the right controller. At the controller, we access the Models, which serve two main purposes. First, Models allow us to perform database operations, as it references to a table in the database. Second, Models establishes the association between each other, which streamlines database operations. (Will be explained further later) After performing the desired database operations, the controller will pass the fetched data to the view. The view weaves these data into its HTML/JS templates, before sending the generated HTML back as a HTTP response.

2. Associations with Tagging System

I made a mistake by first implementing the Tagging System with the popular "acts-as-taggable-on" gem. Where's the learning when the whole task is spoonfed to me? This realisation pivoted my focus from flashy features to deliberate learning.

I implemented the tagging system with the `has_many :through` association using three models: "Task", "Tag", and "Tagging". While `has_and_belongs_to_many` would have been a better choice as I did not use the connecting model, I stuck with the former choice as it was more commonly used.

Two features propelled my understanding of associations. Firstly, I implemented the Task class method `not_tagged_with` which returns an array of *tasks* that does not have the associated *tags*. Secondly, I saved database space by *destroying the associated Tag and Tagging objects of a Task object*, when that Task object is destroyed. Since every Tagging object belongs to that Task, they can be automatically destroyed by simply making them dependent on the Task. Tag objects are tricker as they might still contain other tasks. Thus I implemented a `destroy_empty_tags` function that only destroys the tag when it has only one task left (the task that will be destroyed)

3. Views with AJAX
Interestingly, I spent 2 whole weeks implementing AJAX. That's how much I learnt (read: struggled) from AJAX.

AJAX forced me to *master partials*. Given that I had to respond to AJAX request with JS, I have to write HTML somewhere. Enter partials. Dealing with (nested) partials made me realise the importance of keeping them *independent*, by passing in local variables instead of instance variables.

In using local variables, I deciphered the magic of View helpers such as `form_with`, `link_to`, checkboxes etc. The forms that never rendered taught me how the model parameter for `form_with` generates the scope, url, and method just from the variable you passed into it. Perhaps more importantly, I learnt how all these helper functions simply turn ruby code into HTML code. (Most interesting of which are checkboxes)

Other struggles include: *mastering CSS selectors through jQuery* (which in turn allowed me to understand the importance of a well structured DOM) and using *ERB* (understanding raw, html_safe, j).

4. And here are other accomplishments which I am happy to discuss in person

- **Abstraction** with Asset Pipeline for my JS (12*) and CSS assets, Private methods (3*), Ruby helper (1*), Partials (3). If you chance upon "magical" functions, their implementation reside in the above mentioned locations.
    - Notably, the Asset Pipeline made for faster webpage loading by caching the concatenated assets.
    - (*: Number of functions abstracted)
- **Debugging** with byebug, Chrome DevTools, and `alert`
- **Mobile Friendly webpage** with Media Queries, relative CSS values (rem, %), minimal hover effects
- **Flex boxes** by implementing Quick-Edit
- How Rails handle **checkboxes**

Why CVWO?
CVWO's aim of *serving* strikes a chord with me - I chose CS to improve the lives of those around me, not to make another shopping cart. And it is precisely this alignment in belief, that you can expect me to give my 110% on the job.

With that, my To Do List journey on Rails has come to an end. Thank you CVWO team for this opportunity to learn web dev :)