React – Flux

In this lab, you will modify a React app to use Flux and the Flux Dispatcher.

# Objectives

In this lab, you will

* Modify an existing React app
* Add a Flux Dispatcher
* Add a TodoStore which uses the todo-controller
* Add TodoStore CHANGE listenders
* Run the app

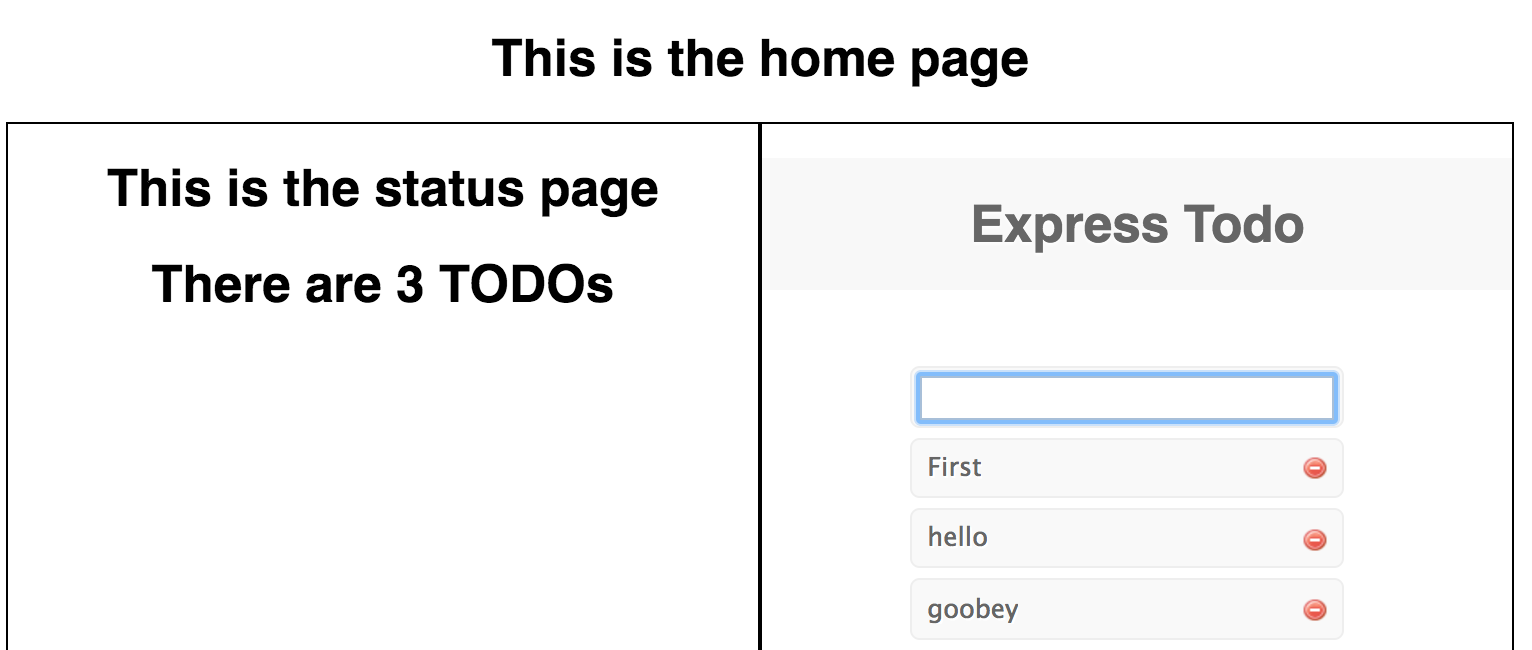
# Run the Solution

1. Go to the solution/react-app folder and start the application. It should display an app with several components. The <Status> component displays the number of TODO items while the <Todos> component displays the actual TODO application. Start the app and view the page at <http://localhost:8000>

yarn install

yarn start

1. In the application shown below, add and remove some TODO items and watch the status number change.



1. In the solution, Flux connects the <Status> and <TodoContainer>
2. Our task is to add Flux to the application so that the two components can access the same business logic.

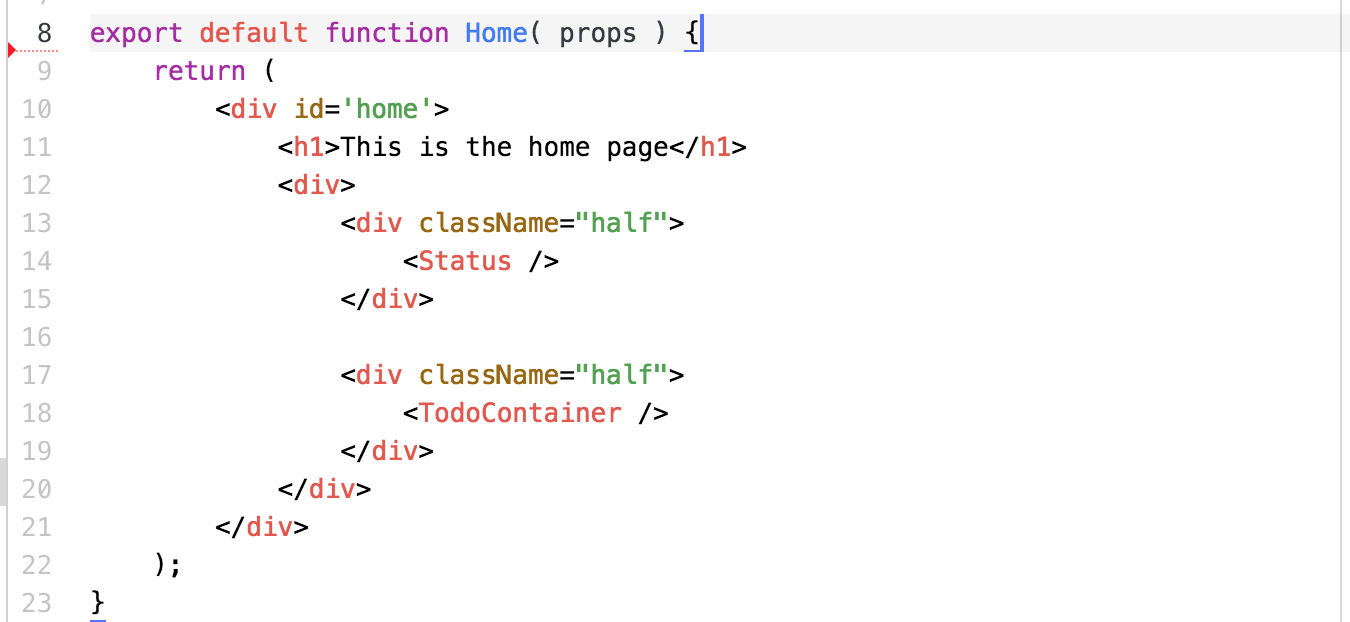
# Run the React app to verify correctness

1. Go to the lab/react-app folder and start the lab application. It should display an app with several components. The <Status> component displays the number of TODO items while the <Todos> component displays the actual TODO application. Start the app and view the page at <http://localhost:8000>

yarn install

yarn start

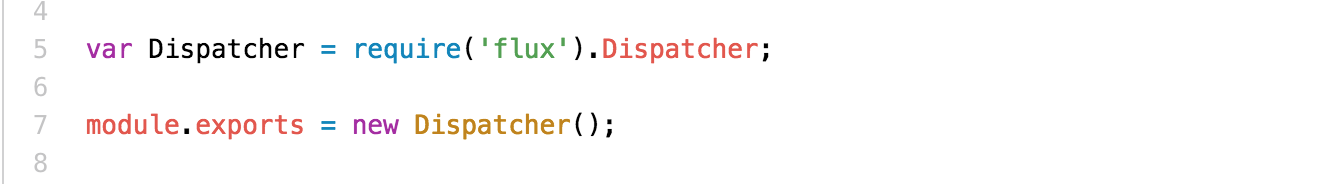
1. Examine the <Home> component as shown below. Notice it divides the screen into two halves. The left side contains the <Status> component and the right side contains the <TodoContainer> component.



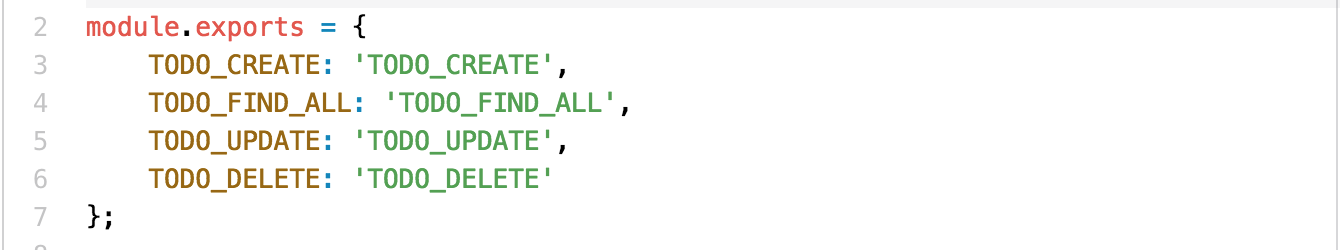
1. As indicated in the Best Practices section, the <TodoContainer> isolates the business logic from the <TodoList> component, a Presentation Component.
2. Sharing information between independent components is the problem with the current architecture. Flux allows us to share a store which then calls the todoController to do the work.
3. In our solution, the <TodoContainer> will go through the Flux dispatcher to delegate the work to the store. The store then delegates the CRUD work to the todoController. Every component that cares about the store can register a listener which the store calls each time the contents change. This arrangement allows the <Status> component to also use the store and be informed when some Flux dispatcher action changes the store.

# Examine the Dispatcher

1. In the src/components/Dispatcher.js file we use the Facebook Flux Dispatcher as shown below.



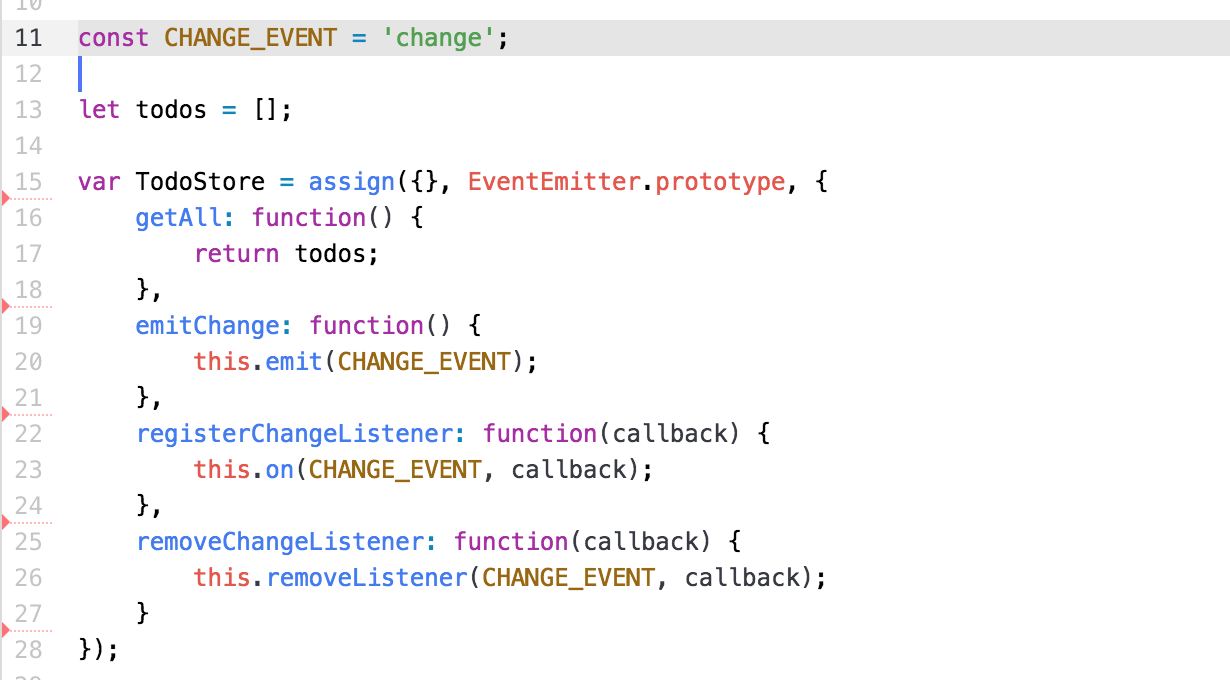
1. Notice we are using the Facebook Flux Dispatcher.
2. Examine the src/todos/todo-constants.js file as it defines the action types we use.



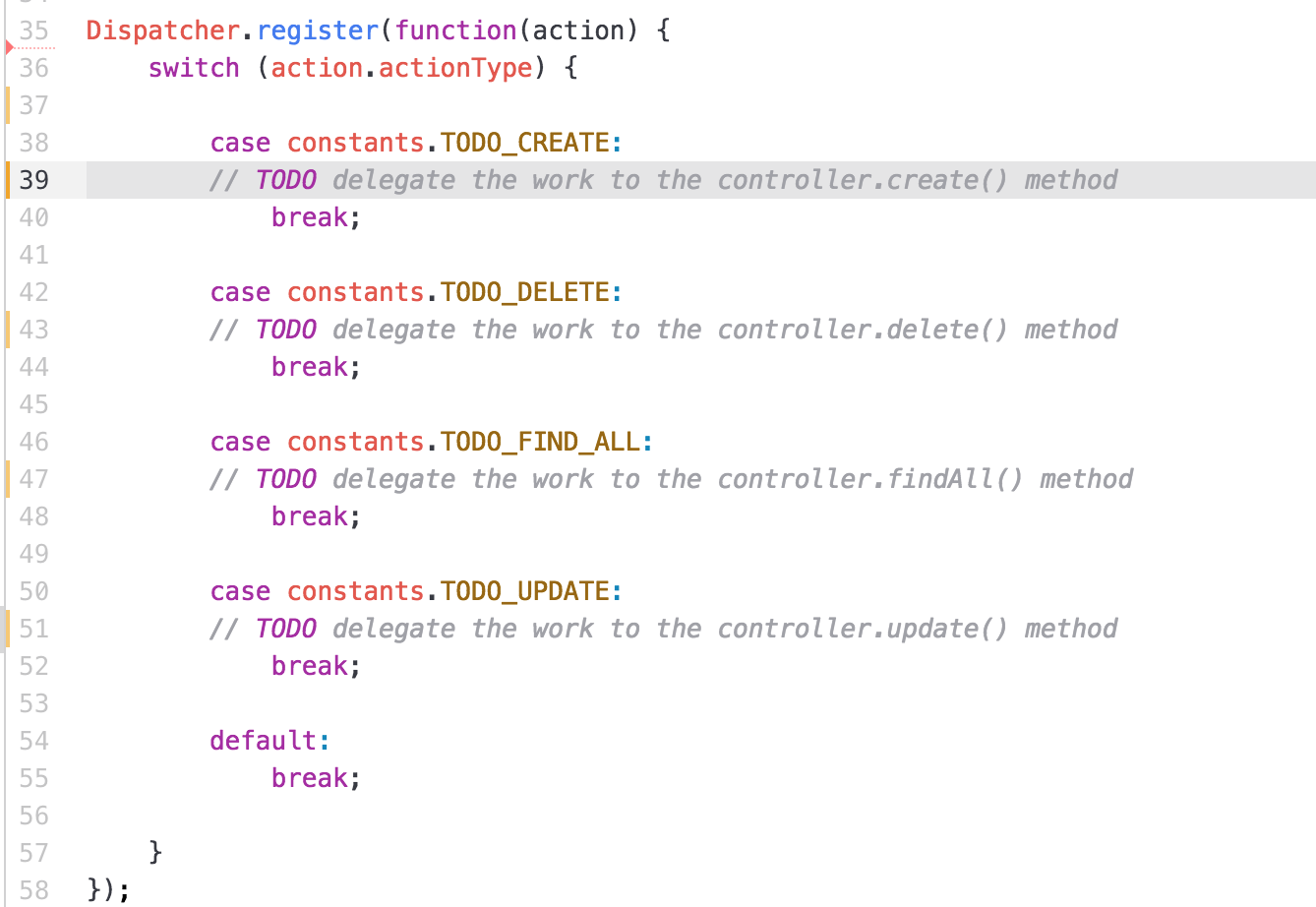
1. These are the actionType values we use in the TodoStore.

# Modify the <TodoStore> Component

1. Examine the src/todos/todo-store.js file



1. In the above, line 11 defines the event the store emits when the model changes. The model is the array, todos, in line 13.
2. Line 15 defines the TodoStore as an object that extends the EventEmitter and adds the four methods shown. The getAll() method allows the components to get all the todos (duh…). The registerChangeListener() and removeChangeListener() methods manages an array of listeners for the CHANGE\_EVENT. Each component registers itself on componentDidMount() and unregisters itself on componentWillUnmount(). The TodoStore invokes the listener callbacks when the todo array changes.
3. The TodoController has not changed.
4. The above is the store. It only allows the clients to get all of the TODO list and register event listeners. When the TODO list changes (via the Dispatcher shown later), the store will call all the registered event listeners. The event listener will usually call getAll() to grab the current list of TODO items.
5. Examine the rest of the file below:



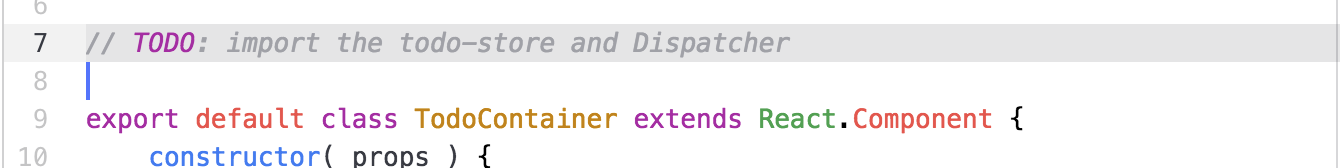
1. The above registers the TodoStore with the Dispatcher. The client invokes the Dispatcher with the appropriate action object using Dispatcher.dispatch( ACTION ). The Dispatcher uses the switch statement to invoke the appropriate controller method.
2. For each action type, enter the code to delegate the work to the controller. It should save the results into the global variable, todos and fire the event using the function, emitChangeEvent() provided. NOTE: each action object has a different set of properties based on the requirements of the controller. Some require just the id, some require the entire TODO item, etc.

# Integrate Flux with the Components

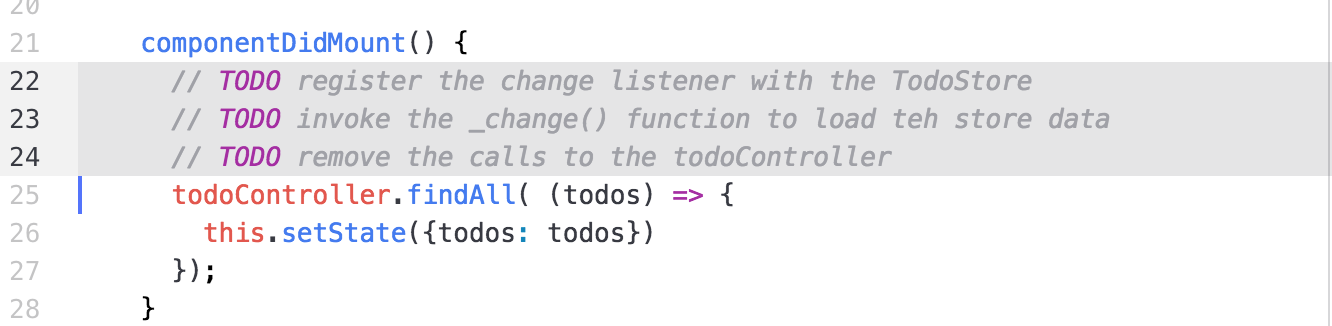
Of the four components, two of them (Home.js and Layout.js) do not change. The <Status> component ( src/components/Status.js) now registers an event listener so that it knows when the TODO list changes, but does NOT directly change the list. The <TodoContainer> component (src/todos/TodoContainer.js) now registers an event listener AND refactors all controller calls to go through the Dispatcher.

# Modify the <TodoContainer> Component

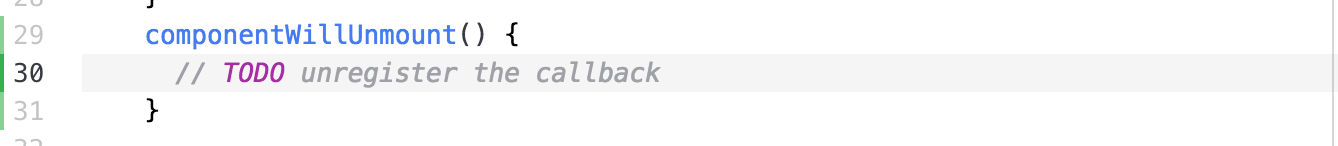
1. Examine each section of the < TodoContainer > component defined in src/todos/TodoContainer.js.



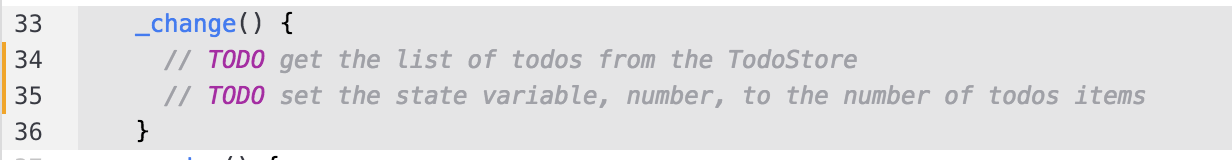
1. In the above, import the TodoStore and the Dispatcher



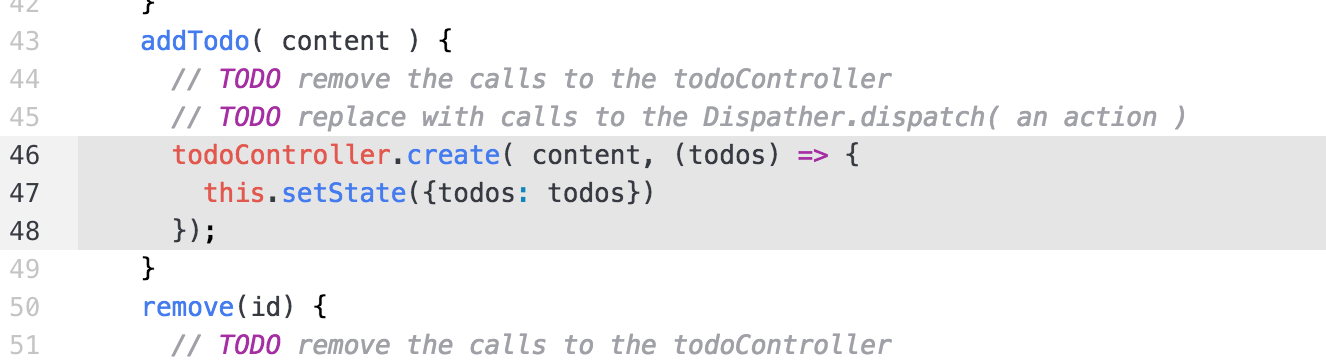
1. In the above, remove the direct calls to the controller. Insert a call to the TodoStore to register the method ( this.\_change ) to the store’s change listener. Invoke the method to get all the todo items.



1. In the above, unregister the listener from the store before the component unmounts. Use TodoStore.removeChangeListener().



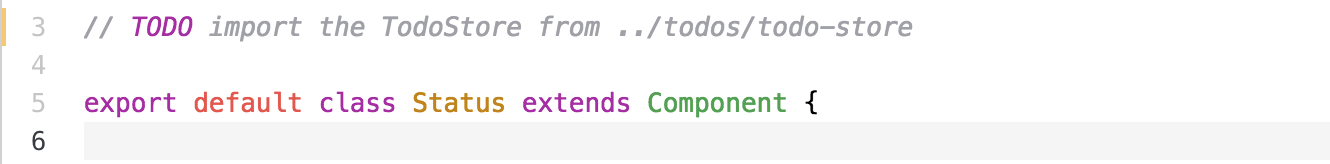
1. In the above, implement the \_change() method. It should get the current list of todo items from the store, and set the number of items into the state.



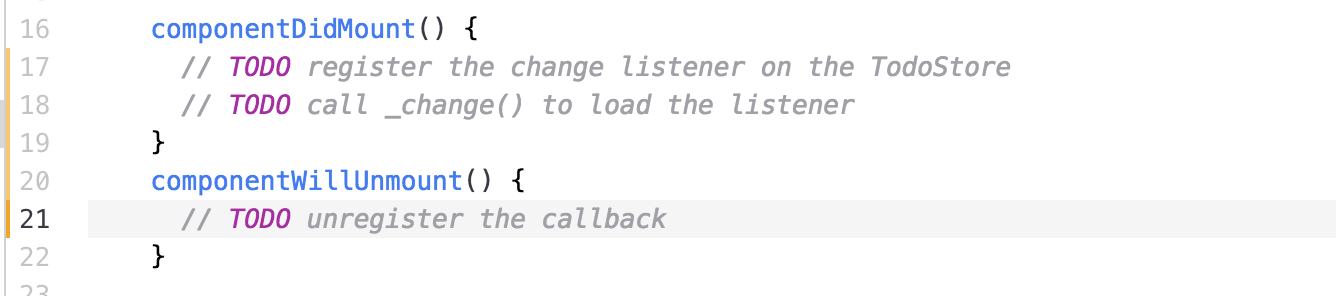
1. In the above, remove the direct calls to the controller and replace them with calls to the Dispatcher. Notice the calls do not directly update the state. The change listener, \_change(), does this automatically when the todo list changes.
2. Restart the server and the Todo application should continue to work.

# Add the Store Listener to the <Status> Component

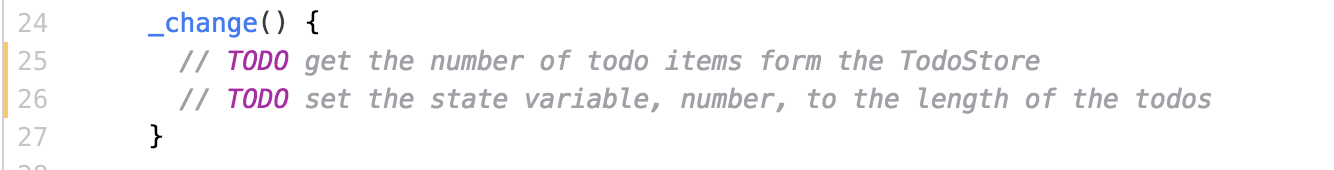
1. Examine the <Status> component from src/components/Status.js below.



1. In the above, import the todo-store



1. In the above, add and remove the event listener.



1. In the above, implement the change listener. The store calls this method when the todo list changes. Get the todo list, calculate the length, and set the state variable, number, to this value.
2. Restart the server and open the browser to <http://localhost:8000> to see the page.
3. Now, each time you add or remove a TODO item, the status component changes its number and the TODO app works normally.

# Food for Thought

1. This seems like a lot of extra work surrounding the todo-controller with the Flux wrapper. However, Flux provides a single entry point to the store. So any component can modify the store and the store informs all components when the store changes. Hmmmm… seems like the Observer pattern.
2. What happens when we have more than one store? The Redux pattern helps with that.
3. Could we have more than one event? Perhaps an event that triggers when the number of events changes? Perhaps the event could send the data to the event handler?

Congratulations. You have completed this lab.