Redux

Redux is most commonly used with React to manage application state, but can also be used with other libraries and frameworks, such as Angular.

CORE CONCEPTS IN REDUX

One-Way Data Flow

In most applications, there are three parts:

* State – the current data used in the app
* View – the user interface displayed to users
* Actions – events that a user can take to change the state

The flow of information would go like this:

* The state holds the current data used by the app’s components.
* The view components display that state data.
* When a user interacts with the view, like clicking a button, the state will be updated in some way.
* The view is updated to display the new state.
* Redux helps separate the state, the view, and actions by requiring that the state be managed by a single source.

State

*State* is the current information behind a web application.

With Redux, state can be any JavaScript type, including: number, string, boolean, array, and object.

Here’s an example state for a todo app:

const state = [ 'Print trail map', 'Pack snacks', 'Summit the mountain' ];

Each piece of information in this state—an array in this case—would inform some part of the user interface.

QUESTION: Define the state of a playlist application in a state array. It should represent this playlist:

1. Take Five
2. Claire de Lune
3. Respect

Answer:

Answer: const state = ['Take Five', 'Claire de Lune', 'Respect'];

Actions

 This entire interaction can be defined as an action. In Redux, actions are represented as plain JS objects.

Here’s what that action might look like:

const action = {  
  type: 'todos/addTodo',  
  payload: 'Take selfies'  
};

* Every action must have a type property with a string value. This describes the action.
* Typically, an action has a payload property with an object value. This includes any information related to the action. In this case, the payload is the todo text.
* When an action is generated and notifies other parts of the application, we say that the action is dispatched.

Here are two more example actions:

“Remove all todos”. This requires no payload because no additional information is needed:

const action = {  
  type: 'todos/removeAll'  
}

“Remove the ‘Pack snacks’ todo”:

const action = {   
  type: 'todos/removeTodo',  
  payload: 'Pack snacks'  
}

Define an action object named addNewSong that represents adding a new song to the playlist.

It should have the following information:

* A type of 'songs/addSong'
* A payload of 'Halo', the title of the song to add

const state = [ 'Take Five', 'Claire de Lune', 'Respect' ];

// create a action

const addNewSong = {

 type: 'songs/addSong',

 payload: 'Halo'

};

2. Define an action named removeSong that represents removing a song from the playlist.

It should have the following information:

* A type of 'songs/removeSong'
* A payload of 'Take Five', the title of the song to remove

// another action

 const removeSong = {

  type: 'songs/removeSong',

  payload: 'Take Five'

};

3. Define an action named removeAll that represents removing all songs from the playlist.

It should have the following information:

* A type of 'songs/removeAll'

//another action

const removeAll = {

  type: 'songs/removeAll'

}

Reducers

A reducer, or reducer function, is a plain JavaScript function that defines how the current state and an action are used in combination to create the new state.

Here’s an example of a reducer function for a todo app:

const initialState = [ 'Print trail map', 'Pack snacks', 'Summit the mountain' ];  
   
const todoReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'todos/addTodo': {  
      return [ ...state, action.payload];  
    }  
    case 'todos/removeAll': {  
      return [];  
    }  
    default: {  
      return state;  
    }  
  }  
}

There a few things about this reducer that are true for all reducers:

* It’s a plain JavaScript function
* It defines the application’s next state given a current state and a specific action
* It returns a default initial state if no action is provided
* It returns the current state if the action is not recognized

There are two intermediate JavaScript syntaxes used here:

1. We use the equals sign = to [supply a default value](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters) for the state parameter.
2. We use [the spread operator (...)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) to copy the current state and any changed values into a new object, not the existing state argument. We’ll explain why in the next exercise.
3. Let’s start building a reducer for our playlist application. For this first checkpoint, it should:

* Be named reducer
* Accept state and action arguments
* Default state to initialState if no value is provided
* Use a switch statement on the action.type property
* Always return state as the default case

1. Add a case for the 'songs/addSong' action type.

If the action.type is 'songs/addSong', return a copy of the state object with the new song added.

You can expect an action like this:

{   
  type: 'songs/addSong',   
  payload: 'Take Five'  
}

Checkpoint 3 Passed

3. Add a case for the 'songs/removeSong' action type.

If the action.type is 'songs/removeSong', return a copy of the state object with the specified song removed. Use the [array filter() method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

You can expect an action like this:

{   
  type: 'songs/removeSong',   
  payload: 'Respect'  
}

Checkpoint

Solution

// Define reducer here

const reducer = (state = initialState, action) => {

  switch (action.type) {

    case 'songs/addSong': {

      return [...state, action.payload]

    }

    case 'songs/removeSong': {

      return state.filter(song => song !== action.payload);

    }

    default: {

      return state;

    }

  }

}

const initialState = [ 'Take Five', 'Claire de Lune', 'Respect' ];

const addNewSong = {

  type: 'songs/addSong',

  payload: 'Halo'

};

const removeSong = {

  type: 'songs/removeSong',

  payload: 'Take Five'

};

const removeAll = {

  type: 'songs/removeAll'

}

Rules of Reducers

1. They should only calculate the new state value based on the state and action arguments.
2. They are not allowed to modify the existing state. Instead, they must copy the existing state and make changes to the copied values.
3. They must not do any asynchronous logic or have other “side effects”.

1.

We have some reducers here that are breaking the rules!

The reducer in app1.js violates the first rule of reducers: it calculates the new state based on something other than the current state and action arguments.

Fix this by assuming that the song being added will be passed into the reducer as the payload of the action object.

App.js

// Reducer violates rule 1:

// They should only calculate the new state value based on the state and action arguments.

const globalSong = 'We are the World';

const playlistReducer = (state = [], action) => {

 switch (action.type) {

   case 'songs/addGlobalSong': {

     return [...state, action.payload];

   }

   default:

     return state;

 }

}

// Example call to reducer

const state = [ 'Take Five', 'Claire de Lune', 'Respect' ];

const addAction = { type: 'songs/addGlobalSong', payload: 'We are the World' };

const newState = playlistReducer(state, addAction);

2. The reducer in app2.js violates the second rule of reducers: it modifies the existing state.

Fix this by using the spread operator ... within a new array instead of using push() on the existing state.

App2.js

// Reducer violates rule 2:

// They are not allowed to modify the existing state.

// Instead, they must copy the existing state and make changes to the copied values.

const todoReducer = (state = [], action) => {

 switch (action.type) {

   case 'todos/addTodo': {

     return [...state, action.payload];

   }

   case 'todos/removeAll': {

     return [];

   }

   default: {

     return state;

   }

 }

}

// Example call to reducer

const state = [ 'Print trail map', 'Pack snacks', 'Summit the mountain' ];

const addTodoAction = { type: 'todos/addTodo', payload: 'Descend' };

const newState = todoReducer(state, addTodoAction);

3. The reducer in app3.js violates the third rule of reducers: it has a side effect. The initial state will not be the same every time you call the reducer.

Fix this by assuming that the random value will be provided as the payload of the action object.

*Note that this reducer is called with undefined. In this case, the default parameter will be used to set state.*

App3.js

 // Reducer violates rule 3:

 // They must not do any asynchronous logic or have other “side effects”.

const initialState = [0, 1, 2];

const reducer = (state = initialState, action) => {

 switch (action.type) {

   case 'numbers/addRandom': {

     return [...state, action.payload];

   }

   default: {

     return state;

   }

 }

}

// Example call to reducer

const randomAction = { type: 'numbers/addRandom', payload: Math.random() };

const newState = reducer(undefined, randomAction);

IMMUTABLE UPDATES AND PURE FUNCTIONS

reducers must make immutable updates and be pure functions.

If a function makes immutable updates to its arguments, it does not change the argument but instead makes a copy and changes that copy. (Sounds similar to rule 2, no?) It’s called updating immutably because the function doesn’t change, or mutate, the arguments.

This function mutates its argument:

const mutableUpdater = (obj) => {  
  obj.completed = !obj.completed;  
  return obj;  
}  
   
const task = { text: 'do dishes', completed: false };  
const updatedTask = mutableUpdater(task);  
console.log(updatedTask);   
// Prints { text: 'do dishes', completed: true };  
   
console.log(task);   
// Prints { text: 'do dishes', completed: true };

Meanwhile, this function “immutably updates” its argument:

const immutableUpdater = (obj) => {  
  return {  
    ...obj,  
    completed: !obj.completed  
  }  
}  
   
const task = { text: 'iron clothes', completed: false };  
const updatedTask = immutableUpdater(task);  
console.log(updatedTask);   
// Prints { text: 'iron clothes', completed: true };  
   
console.log(task);   
// Prints { text: 'iron clothes', completed: false };

By copying the contents of the argument obj into a new object ({...obj}) and updating the completed property of the copy, the argument obj will remain unchanged.

Note that, plain strings, numbers, and booleans are immutable in JavaScript so we can just return them without making a copy:

const immutator = (num) => num + 1;  
const x = 5;  
const updatedX = immutator(x);  
   
console.log(x, updatedX); // Prints 5, 6

If a function is pure, then it will always have the same outputs given the same inputs.

This is a combination of rules 1 and 3:

* Reducers should only calculate the new state value based on the state and action arguments.
* Reducers must not do any asynchronous logic or other “side effects”.

In this example, the function is not a pure function because its returned value depends on the status of a remote endpoint.

const addItemToList = (list) => {  
  let item;  
  fetch('https://anything.com/endpoint')  
    .then(response => {  
      if (!response.ok) {  
        item = {};  
      }  
   
      item = response.json();  
   });  
   
   return [...list, item];    
};

The function can be made pure by pulling the fetch() statement outside of the function.

let item;  
  fetch('https://anything.com/endpoint')  
    .then(response => {  
      if (!response.ok) {  
        item = {};  
      }  
   
      item = response.json();  
   });  
   
const addItemToList = (list, item) => {  
    return [...list, item];  
};

1.

The function in immutable.js mutates its arguments because it uses the array splice() function. Rewrite it using the [slice() method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) and the [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax).

If done correctly, the output should still be:

[ 'a', 'c', 'd' ]

Checkpoint

Answer

const removeItemAtIndex = (list, index) => {

 return [

   ...list.slice(0,index),

   ...list.slice(index+1, list.length)

 ]

};

console.log(removeItemAtIndex(['a', 'b', 'c', 'd'], 1));

2.

The function capitalizeMessage() in pure.js is impure because it depends on an external file. Re-write it so that it is pure.

You will need to read the file outside of the function and pass in the resulting data.

Answer

const fs = require('fs');

const file = './data.txt';

const capitalizeMessage = (message) => {

  return message.toUpperCase();

}

const message = fs.readFileSync(file, 'utf8');

console.log(capitalizeMessage(message));

Store

Redux uses a special object called the store. The store acts as a container for state, it provides a way to dispatch actions, and it calls the reducer when actions are dispatched. In nearly every Redux application, there will only be one store.

We can rephrase our data flow using the new term:

1. The store initializes the state with a default value.
2. The view displays that state.
3. When a user interacts with the view, like clicking a button, an action is dispatched to the store.
4. The dispatched action and the current state are combined in the store’s reducer to determine the next state.
5. The view is updated to display the new state.

Review

* Redux is a library for managing and updating application state based on the Flux architecture
* Redux makes code more predictable, testable, and maintainable by consolidating state in a single object. Components are just given data to render and can request changes using events called actions.
* In a Redux application, data flows in one direction: from state to view to action back to state and so on.
* State is the current information behind a web application.
* An action is an object describing an event in the application. It must have a type property and it typically has a payload property as well.
* A reducer is a function that determines the application’s next state given a current state and a specific action. It returns a default initial state if none is provided and returns the current state if the action is not recognized
* A reducer must make follow these three rules:
  1. They should only calculate the new state value based on the existing state and action.
  2. They are not allowed to modify the existing state. Instead, they must copy the existing state and make changes to the copied values.
  3. They must not do any asynchronous logic or other “side effects”.
* In other words, a reducer must be a pure function and it must update the state immutably.
* The store is a container for state, it provides a way to dispatch actions, and it calls the reducer when actions are dispatched. Typically there is only one store in a Redux application.

What is the Redux API?

Redux applications are built upon a one-way flow of data model and are managed by the store:

* The state is the set of data values that describes the application. It is used to render the user interface (UI).
* Users interact with the UI which dispatch actions to the store. An action is an object that expresses a desired change to the state.
* The store generates its next state using a reducer function which receives the most recent action and the current state as inputs.
* Finally, the UI is re-rendered based on the new state of the store and the entire process can begin again.

This lesson will focus on creating a basic Redux application with the createStore() method from the Redux API and the following related store methods:

* store.getState()
* store.dispatch(action)
* store.subscribe(listener)

Install the Redux Library

Redux is a library that provides concrete methods to help implement the framework.

To make use of the Redux package, it can be installed using the Node Package Manager (npm). Then, its methods can be imported.

Let’s start by installing the redux package and importing its createStore() method. In the next exercise you’ll learn how to use this method.

1: In the bash terminal enter the following command for installing the redux package using NPM.

npm install redux

2.In the store.js file use [ES6 import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) syntax and import the createStore method from the redux package:

import { createStore } from 'redux';

Create a Redux Store

As you know, every Redux application uses a reducer function that describes which actions can update the state and how those actions lead to the next state.

For example, suppose you wanted to build an application for a light switch. Its reducer might look like this:

const initialState = 'on';  
const lightSwitchReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'toggle':  
      return state === 'on' ? 'off' : 'on';  
    default:  
      return state;  
  }  
}

This reducer handles a single action type 'toggle' and returns the next state of the store: 'on' if it had been 'off' and vice-versa. If an unrecognized action is received, the current state of the store is returned.

The programmer could manually execute the reducer with the current state of the store and the desired action to perform like so:

let state = 'on';  
state = lightSwitchReducer(state, { type: 'toggle' });  
console.log(state); // Prints 'off'

Redux exports a valuable helper function for creating this store object called createStore(). The createStore() helper function has a single argument, a reducer function.

To create a store with lightSwitchReducer, you could write:

import { createStore } from 'redux'  
   
const initialState = 'on';  
const lightSwitchReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'toggle':  
      return state === 'on' ? 'off' : 'on';  
    default:  
      return state;  
  }  
}  
   
const store = createStore(lightSwitchReducer);

1: First, import the createStore method from redux.

Answer

// Import createStore here

import { createStore } from 'redux';

const initialState = 0;

const countReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    default:

      return state;

  }

}

2: Now, below the countReducer function, declare a variable called store.

Then, call createStore() with countReducer as the argument. Assign the returned value to store.

Answer:

// Create the store here

const store = createStore(countReducer);

Dispatch Actions to the Store

The most commonly used method, store.dispatch(), can be used to dispatch an action to the store, indicating that you wish to update the state. Its only argument is an action object, which must have a type property describing the desired state change.

const action = { type: 'actionDescriptor' };   
store.dispatch(action);

Each time store.dispatch() is called with an action object, the store’s reducer function will be executed with the same action object. Assuming that the action.type is recognized by the reducer, the state will be updated and returned.

Let’s see how this works in the lightswitch application from the last exercise:

import { createStore } from 'redux';  
   
const initialState = 'on';  
const lightSwitchReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'toggle':  
      return state === 'on' ? 'off' : 'on';  
    default:  
      return state;  
  }  
}  
   
const store = createStore(lightSwitchReducer);  
   
console.log(store.getState()); // Prints 'on'  
   
store.dispatch({ type: 'toggle' });   
console.log(store.getState()); // Prints 'off'  
   
store.dispatch({ type: 'toggle' });  
console.log(store.getState()); // Prints 'on'

In this example, you can also see another store method, store.getState(), which returns the current value of the store’s state. Printing its value between each dispatched action allows us to see how the store’s state changes.

Internally, when the store executes its reducer, it uses store.getState() as the state argument. Though you won’t see it, you can imagine that, when an action is dispatched like this…

store.dispatch({ type: 'toggle'});

…the store calls the reducer like this:

lightSwitchReducer(store.getState(), { type: 'toggle' });

1: Let’s get back to our counter application. The count starts at 0 and we want to increment it up to 2.

At the bottom of store.js dispatch two actions with a type of 'increment' using the store.dispatch() method.

2.

At the bottom of store.js, confirm that the current state of the store is 2 by printing out the current value of the state to the console.

3.

Let’s modify the countReducer function so that it can handle 'decrement' actions as well.

Add an additional case to the countReducer function that handles 'decrement' action types and returns state - 1.

4.

Now, let’s dispatch some 'decrement' action to the store.

At the bottom of store.js , dispatch 3 actions with a type of 'decrement'.

5.

Finally, print to the console the final value of store.getState(). The final state should be -1.

Answer

import { createStore } from 'redux';

const initialState = 0;

const countReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    case 'decrement':

      return state - 1;

    default:

      return state;

  }

}

const store = createStore(countReducer);

// Dispatch your actions here.

store.dispatch({type:'increment'});

store.dispatch({type:'increment'});

console.log(store.getState());

store.dispatch({type:'decrement'});

store.dispatch({type:'decrement'});

store.dispatch({type:'decrement'});

console.log(store.getState());

Action Creators

Typing out the entire action object can be tedious and creates opportunities to make an error.

For example, in the light switch application, whose reducer accepts 'toggle' actions to turn the light 'on' or 'off', you might write:

store.dispatch({Type:'toggle'});  
store.dispatch({type:'toggel'});  
store.dispatch({typo:'toggle'});

Did you spot the errors?

In most Redux applications, *action creators* are used to reduce this repetition and to provide consistency. An action creator is simply a function that returns an action object with a type property. They are typically called and passed directly to the store.dispatch() method resulting in fewer errors and an easier-to-read dispatch statement.

The above code could be rewritten using an action creator called toggle() like so:

const toggle = () => {  
  return { type: "toggle" };  
}  
store.dispatch(toggle()); // Toggles the light to 'off'  
store.dispatch(toggle()); // Toggles the light back to 'on'  
store.dispatch(toggle()); // Toggles the light back to 'off'

1.

Let’s add some action creators into the counter application. The two actions that the countReducer can handle are 'increment' and 'decrement'.

First, at the top of the store.js file, create an action creator function called increment() that returns an object with a type: 'increment' property.

2.

Well done! Now, after the increment() action creator and before the countReducer, add in a second action creator named decrement() that returns an action object with the 'decrement' action type.

3. In store.js, actions are being dispatched to the store like so:

store.dispatch({ type: 'increment' });  
store.dispatch({ type: 'decrement' });

For each existing call to store.dispatch(), replace the typed-out action objects with function calls to the appropriate action creator.

Solution

import { createStore } from 'redux';

// Create your action creators here.

const increment = () => {

  return { type: 'increment' };

}

const decrement = () => {

  return { type: 'decrement' };

}

const initialState = 0;

const countReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    case 'decrement':

      return state - 1;

    default:

      return state;

  }

}

const store = createStore(countReducer);

// Modify the dispatches below.

store.dispatch(increment());

store.dispatch(increment());

console.log(store.getState());

store.dispatch(decrement());

store.dispatch(decrement());

store.dispatch(decrement());

console.log(store.getState());

Respond to State Changes

In a typical web application, user interactions that trigger [DOM events](https://developer.mozilla.org/en-US/docs/Web/Events) ("click", "keydown", etc…) can be listened for and responded to using an [event listener](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener).

Similarly, in Redux, actions dispatched to the store can be listened for and responded to using the store.subscribe() method. This method accepts one argument: a function, often called a *listener*, that is executed in response to changes to the store‘s state.

const reactToChange = () => console.log('change detected!');  
store.subscribe(reactToChange);

In this example, each time an action is dispatched to the store, and a change to the state occurs, the subscribed listener, reactToChange(), will be executed.

Sometimes it is useful to stop the listener from responding to changes to the store, so store.subscribe() returns an unsubscribe function.

We can see this in action in the light switch application:

// lightSwitchReducer(), toggle(), and store omitted...  
   
const reactToChange = () => {  
  console.log(`The light was switched ${store.getState()}!`);  
}  
const unsubscribe = store.subscribe(reactToChange);  
   
store.dispatch(toggle());  
// reactToChange() is called, printing:  
// 'The light was switched off!'  
   
store.dispatch(toggle());  
// reactToChange() is called, printing:  
// 'The light was switched on!'  
   
unsubscribe();   
// reactToChange() is now unsubscribed  
   
store.dispatch(toggle());  
// no print statement!  
   
console.log(store.getState()); // Prints 'off'

* In this example, the listener function reactToChange() is subscribed to the store
* Each time an action is dispatched, reactToChange() is called and prints the current value of the light switch. It is common for callbacks subscribed to the store to use store.getState() inside them.
* After the first two dispatched actions, unsubscribe() is called causing reactToChange() to no longer be exectued in response to further dispatches made to store.

Note: It is not always required to use the *unsubscribe()* function returned by *store.subscribe()*, though it is useful to know that it exists.

1.

The first thing to do is to define a state change listener.

Define a function called printCountStatus() with no arguments. It should print to the console the following message:

console.log(`The count is ${REPLACE\_WITH\_CURRENT\_STATE}`);

Make sure to replace REPLACE\_WITH\_CURRENT\_STATE with the proper code for getting the current state from the store.

2.

Now that you have a change listener function, subscribe it to the store so that it is called each time the state changes.

If done correctly, you should see this in the console output:

The count is -1  
The count is 0  
The count is 1

Answer

import { createStore } from 'redux';

const increment = () => {

  return { type: 'increment' }

}

const decrement = () => {

  return { type: 'decrement' }

}

const initialState = 0;

const countReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    case 'decrement':

      return state - 1;

    default:

      return state;

  }

}

const store = createStore(countReducer);

// Define your change listener function here.

function printCountStatus() {

  console.log(store.getState());

}

store.subscribe(printCountStatus);

store.dispatch(decrement()); // store.getState() === -1

store.dispatch(increment()); // store.getState() === 0

store.dispatch(increment()); // store.getState() === 1

Connect the Redux Store to a UI

The UI for this application should display the current count number and allow the user to increment or decrement this value using the buttons provided. Take a look at the connected web browser window and you can see that the elements for such an interface are present, but they haven’t been connected to the Redux store yet.

Connecting a Redux store with any UI requires a few consistent steps, regardless of how the UI is implemented:

* Create a Redux store
* Render the initial state of the application.
* Subscribe to updates. Inside the subscription callback:
  + Get the current store state
  + Select the data needed by this piece of UI
  + Update the UI with the data
* Respond to UI events by dispatching Redux actions

Open up the **index.html** file and you can see the three HTML elements that are currently being rendered:

<p id='counter'>Waiting for current state.</p>  
<button id='incrementer'>+</button>  
<button id='decrementer'>-</button>

Now, open up **store.js** where you will find the pieces of Redux code that you have built throughout this lesson: the action creators increment() and decrement(), the reducer countReducer, and the store that ties it all together. Additionally, the following values have been added:

* counterElement, incrementer, and decrementer: references to the HTML elements in **index.html**
* render: A state-change listener for responding to changes to the store‘s state.
* incrementerClicked and decrementerClicked: DOM event handlers for responding to the buttons being clicked by the user.

These new functions and elements will allow us to plug the Redux store into the UI

Questions:

1. The counterElement should display the current value of the store, but currently it is displaying the message “Waiting for current state.” You can change this text by assigning a new value to counterElement.innerHTML like so:

counterElement.innerHTML = "New text to display";

Within the render() function, reassign counterElement.innerHTML to display the current state of the store in the UI.

Then, after the definition of render(), call it once to render the initial state of the store.

2. The incrementerClicked() function will be called each time the incrementer button is clicked by the user. When this happens, the store should be notified and the state should be incremented by 1.

Inside incrementerClicked(), use store.dispatch() and and the appropriate action creator to tell the store to increase its state by 1.

*Note: completing this checkpoint will not cause the UI to change (you’ll see why soon)*.

3. Now, pressing the incrementer button will send a { type: 'increment' } action object to the store which increases the value of the state to 1. But, the UI doesn’t seem to be updating.

In order for the UI to react to changes to the state of the store, you have to subscribe a change listener to the store using store.subscribe()!

Below the declaration of the render() function, call store.subscribe() and pass in render as the argument so that the UI re-renders each time the state of the store changes. Then, try clicking on the “+” button.

4. Nicely done! Press the incrementer button and you should see the counter increase in the UI! All that’s left to do is get the decrementer button’s event handler to work.

Within decrementerClicked(), dispatch a decrement() action to the store.

Solutions : store.js

/\* Note to learners:

Normally, you would import redux like this:

  import { createStore } from 'redux';

Due to Codecademy's technical limitations

for testing this exercise, we are using

`require()`.

\*/

const { createStore } = require('redux');

// Action Creators

function increment() {

  return {type: 'increment'}

}

function decrement() {

  return {type: 'decrement'}

}

// Reducer / Store

const initialState = 0;

const countReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    case 'decrement':

      return state - 1;

    default:

      return state;

  }

};

const store = createStore(countReducer);

// HTML Elements

const counterElement = document.getElementById('counter');

const incrementer = document.getElementById('incrementer');

const decrementer = document.getElementById('decrementer');

// Store State Change Listener

const render = () => {

  counterElement.innerHTML = store.getState();

}

render();

store.subscribe(render);

// DOM Event Handlers

const incrementerClicked = () => {

  store.dispatch(increment());

}

incrementer.addEventListener('click', incrementerClicked);

const decrementerClicked = () => {

  store.dispatch(decrement());

}

decrementer.addEventListener('click', decrementerClicked);

index.html

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <link rel="stylesheet" href="./index.css">

  <title>Learn ReactJS</title>

</head>

<body>

  <main id="app">

    <p id='counter'>Waiting for current state.</p>

    <button id='incrementer'>+</button>

    <button id='decrementer'>-</button>

  </main>

</body>

<!-- Do Not Remove -->

<script src="https://content.codecademy.com/courses/React/react-16-redux-4-bundle.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.0.5/redux.min.js" integrity="sha512-P36ourTueX/PrXrD4Auc1kVLoTE7bkWrIrkaM0IG2X3Fd90LFgTRogpZzNBssay0XOXhrIgudf4wFeftdsPDiQ==" crossorigin="anonymous"></script>

<script src="./store.js"></script>

</html>

React and Redux

Redux can be used within the context of any UI framework, though it is most commonly paired with React.

We can be more specific about the common steps involved in connecting Redux to a React UI:

* A render() function will be subscribed to the store to re-render the top-level React Component.
* The top-level React component will receive the current value of store.getState() as a prop and use that data to render the UI.
* Event listeners attached to React components will dispatch actions to the store.

Take a look at **store.js** in the code editor. Here, you can see the completed light switch application following this pattern.

* The render() function is subscribed to the store.
* store.getState() is passed as a prop called state to the <LightSwitch /> component.
* The LightSwitch component displays the current state of the store, either 'on' or 'off', and adjusts the background colors accordingly.
* The LightSwitch component declares a click handler that dispatches a toggle() action to the store.

Note 1: The prop name *state* isn’t a special React name and can be customized as the programmer sees fit. For example, *lightSwitchState={store.getState()}* would also be valid

.

### Questions:

* Create a Redux store
* Render the initial state of the application.
* Subscribe to updates. Inside the subscription callback:
  + Get the current store state
  + Select the data needed by this piece of UI
  + Update the UI with the data
* Respond to UI events by dispatching Redux actions

Solution: store.js

import React from 'react';

import ReactDOM from 'react-dom';

import { createStore } from 'redux';

// REDUX CODE

///////////////////////////////////

const toggle = () => {

  return {type: 'toggle'}

}

const initialState = 'off';

const lightSwitchReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'toggle':

      return state === 'on' ? 'off' : 'on';

    default:

      return state;

  }

}

const store = createStore(lightSwitchReducer);

// REACT CODE

///////////////////////////////////

// Pass the store's current state as a prop to the LightSwitch component.

const render = () => {

  ReactDOM.render(

    <LightSwitch

      state={store.getState()}

    />,

    document.getElementById('root')

  )

}

render(); // Execute once to render with the initial state.

store.subscribe(render); // Re-render in response to state changes.

// Receive the store's state as a prop.

function LightSwitch(props) {

  const state = props.state;

  // Adjust the UI based on the store's current state.

  const bgColor = state === 'on' ? 'white' : 'black';

  const textColor = state === 'on' ? 'black' : 'white';

  // The click handler dispatches an action to the store.

  const handleLightSwitchClick = () => {

    store.dispatch(toggle());

  }

  return (

    <div style={{background : bgColor, color: textColor}}>

      <button onClick={handleLightSwitchClick}>

        {state}

      </button>

    </div>

  )

}

Index.html

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <link rel="stylesheet" href="./index.css">

  <title>Learn ReactJS</title>

</head>

<body>

  <div id="root">

  </div>

</body>

<!-- Do Not Remove -->

<script src="https://content.codecademy.com/courses/React/react-16-redux-4-bundle.min.js"></script>

<script src="./store.compiled.js"></script>

</html>

Implementing a React+Redux App

Now that you have implemented the counter app using the HTML DOM API, and have seen a working React+Redux application, it is time to implement it using React.

Take a look at the **store.js** file and you will find the following functions and values have been defined for you:

* The action creators increment() and decrement()
* The store and its reducer countReducer()
* A React component called CounterApp which declares two event handlers, onIncrementButtonClicked and onDecrementButtonClicked
* A render() function which renders CounterApp using ReactDOM.render()

The React component CounterApp and the render() function are entirely disconnected from the Redux store. Let’s change that!

1.

At this point, you should be familiar with the desired functionality of this UI. Notice that nothing is currently being rendered in the web browser.

First, below the render() function’s definition, call render() once to render the UI with the initial state.

2.

The CounterApp component should display the current state of the store. Before it can display the current state, it must be given the current state value.

Within the render() function, pass a prop value to CounterApp called state. Its value should be the current state of the store. Your render() function should look something like this:

const render = () => {  
  ReactDOM.render(  
    <CounterApp   
      state={currentStateValueGoesHere}   
    />,  
    document.getElementById('root')  
  )  
}

3.Now that the current state of the store is being passed to the CounterApp component, it can render that data in the UI.

First, at the top of the CounterApp() function, declare a variable called state. It should be assigned the value of props.state.

Then, modify the <h1> element inside the return statement of render() such that it displays the current state.

4.At this point, your user interface should be displaying the current state of the store, 0. The next step is to update the state when either of the buttons are pressed.

First, modify the onIncrementButtonClicked event handler such that it dispatches an increment() action to the store.

Then, modify the onDecrementButtonClicked event handler such that it dispatches a decrement() action to the store.

5.

Well done! So far we can display the current state and dispatch action from the <CounterApp /> - all that’s left is to re-render the component every time the state changes.

At the bottom of **store.js**, subscribe the render function to the store.

Solution: store.js

import React from 'react';

import ReactDOM from 'react-dom';

import { createStore } from 'redux';

// REDUX CODE

///////////////////////////////////

const increment = () => {

  return {type: 'increment'}

}

const decrement = () => {

  return {type: 'decrement'}

}

const initialState = 0;

const counterReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'increment':

      return state + 1;

    case 'decrement':

      return state - 1;

    default:

      return state;

  }

}

const store = createStore(counterReducer);

// REACT CODE

///////////////////////////////////

const render = () => {

  ReactDOM.render(

    <CounterApp

      state={store.getState()}

    />,

    document.getElementById('root')

  )

}

render();

// Render once with the initial state.

// Subscribe render to changes to the store's state.

function CounterApp(props) {

  const onIncrementButtonClicked = () => {

    store.dispatch(increment())

  }

  const onDecrementButtonClicked = () => {

    store.dispatch(decrement())

  }

  return (

    <div id='counter-app'>

      <h1> {props.state} </h1>

      <button onClick={onIncrementButtonClicked}>+</button>

      <button onClick={onDecrementButtonClicked}>-</button>

    </div>

  )

}

store.subscribe(render);

Review

* Install the redux library into your project using npm install redux.
* Import the createStore() helper function from the 'redux' library.
* Create a store object that holds the entire state of your Redux application using createStore().
* Get the current state of the store using store.getState().
* Dispatch actions to the store using store.dispatch(action).
* Create action creators to reduce the repetitive creation of action objects.
* Register a change listener function to respond to changes to the store using store.subscribe(listener).
* Recognize the pattern for connecting Redux to any user interface.
* Implement a Redux application using either the HTML DOM API or React.

**Introduction to Strategies for Complex State**

* React
  + How to create components
  + How to render components using ReactDOM.render()
  + How to nest components and pass data through props
* Redux
  + One-way data flow model: State → View → Actions → State → View …
  + How to create a reducer function: (state, action) => nextState
  + How to write action objects and action creators
  + How to create a store using createStore()
  + How to use the store methods getState(), dispatch(), and subscribe()

**Slices**

Redux is best suited for complex applications with many features that each have some state-related data to be managed. In these cases, objects are the go-to data type to represent the entire store’s state.

For example, consider a todo app that allows a user to:

* add to a list of todos
* mark individual todos as complete or incomplete
* apply a filter to show only the completed todos, only the incomplete todos, or all of the todos

After adding a few todos and setting the filter to show incomplete todos, the state might look like this:

state = {  
  todos: [  
    {  
      id: 0,   
      text: 'Complete the Learn Redux course',   
      isCompleted: false  
    },  
    {  
      id: 1,   
      text: 'Build a counter app',   
      isCompleted: true  
    },  
  ],  
  visibilityFilter: 'SHOW\_INCOMPLETE'

most Redux applications begin with an initialState that allows the programmer to do two key things:

1. Plan out the general structure of the state
2. Provide an initial state value to the reducer function

For the todo app, this may look like this:

const initialState = {  
  todos: [],  
  visibilityFilter: 'SHOW\_ALL'  
};  
const todosReducer = (state = initialState, action) => {  
  // rest of todosReducer logic omitted  
};

The Recipes application will have the following three slices:

1. allRecipes: an array of all recipe objects
2. favoriteRecipes: an array of recipe objects chosen by the user from state.allRecipes
3. searchTerm: a string that filters which recipes are displayed

An example of the store’s state may look like this:

state = {  
  allRecipes: [  
    {id: 0, name: 'Jjampong', img: 'img/jjampong.png' },  
    {id: 2, name: 'Cheeseburger', img: 'img/cheeseburger.png' },  
    //… more recipes omitted  
  ],  
  favoriteRecipes: [  
    {id: 1, name: 'Doro Wat', img: 'img/doro-wat.png' },  
  ],  
  searchTerm: 'Doro'  
};

Notice that each recipe is represented as an object with an id, name, and img property.

Now that you know what the state structure looks like, the first step is to create an initialState object.

Question: begin by declaring a new variable called initialState and assign to it an empty object.

Now let’s add slices to the initialState object.

First, add an allRecipes property to the initialState object with an initial value of an empty array.

This array will be filled once we fetch the data from a database.

**3.** Next, add a favoriteRecipes property to the initialState object, also with an initial value of an empty array.

The user will select which recipes to add to this slice as their favorites.

**4.** Finally, add a searchTerm property to the initialState object with an initial value of an empty string.

The user will change this value by using a search input field.

Solution:

const initialState = {

  allRecipes: [],

  favoriteRecipes: [],

  searchTerm: ''

};

**vActions and Payloads For Complex State**

The initialState structure has been defined and you know that the state of this application has 3 slices: allRecipes, favoriteRecipes, and searchTerm. *Remember, actions in Redux are represented by plain JavaScript objects that have a type property and are dispatched to the store using the store.dispatch() method.*

When an application state has multiple slices, individual actions typically only change one slice at a time. Therefore, it is recommended that each action’s type follows the pattern 'sliceName/actionDescriptor', to clarify which slice of state should be updated.

Write some of your ideas down before revealing the actions you will be using:

1. 'allRecipes/loadData': This action will be dispatched to fetch the needed data from an API right when the application starts.
2. 'favoriteRecipes/addRecipe': This action will be dispatched any time the user clicks on the ❤️ icon of a recipe from the full set of recipes.
3. 'favoriteRecipes/removeRecipe': This action will be dispatched any time the user clicks on the 💔 icon of a recipe from their list of favorites.
4. 'searchTerm/setSearchTerm': This action will be dispatched any time the user changes the text of the search input field to filter the full set of recipes.
5. 'searchTerm/clearSearchTerm': This action will be dispatched any time the user clicks on the “X” button next to the search input field.

It’s also important to consider which of these actions will have a payload — additional data passed to the reducer in order to carry out the desired change-of-state. For example, consider the actions for the searchTerm slice:

store.dispatch({   
  type: 'searchTerm/setSearchTerm',   
  payload: 'Spaghetti'   
});  
// The resulting state: { ..., searchTerm: 'Spaghetti' }  
   
store.dispatch({   
  type: 'searchTerm/clearSearchTerm'   
});  
// The resulting state: { ..., searchTerm: '' }

* When the learner types in a search term, that data needs to be sent to the store so that the React components know which recipes to render and which to hide.
* When the user clears the search field, no additional data needs to be sent because the store can simply set the search term to be an empty string again.
* *Remember, action creators are functions that return a formatted action object.*
* Action creators enable Redux programmers to re-use action object structures without typing them out by hand and they improve the readability of their code, particularly when dealing with bulky payloads.

**Take a look at store.js where you will find that action creators for the two actions above have been defined for you. Your job is to create the remaining three: loadData(), addRecipe(), and removeRecipe()**

Question: Open up **./data.js** and you will see an array of recipe objects called allRecipesData is exported. Back in **store.js**, at the top of the file, this array is imported (later on, you will fetch data from an API rather than importing from a local file).

This array needs to be sent to the store so that it can populate the state.allRecipes slice, which is initially empty. This can be done using the loadData() action creator.

Complete the function loadData() such that it returns an action object with the following properties:

* type: The slice being modified is state.allRecipes and the action name is 'loadData'
* payload: The allRecipesData array.

*Remember to use the ‘sliceName/actionName’ pattern for type.*

**2.** Next up is addRecipe() which should be dispatched when the user clicks on the ❤️ icon of a particular recipe.

Notice that this function accepts a recipe parameter. The recipe object then needs to be sent to the store to be added to the state.favoriteRecipes slice. For example, this action might be dispatched like so:

const exampleRecipe = {   
  id: 4,   
  name: 'Cheeseburger',   
  img: 'img/cheeseburger.jpg'  
}  
store.dispatch(addRecipe(exampleRecipe));

Complete the function called addRecipe() such that it returns an action object with the following properties:

* type: The slice being modified is state.favoriteRecipes and the action name is 'addRecipe'
* payload: The recipe object parameter.

**3.** The last action creator is removeRecipe() which should be dispatched when the user clicks on the 💔 icon of a favorited recipe.

removeRecipe() also accepts a recipe parameter. The recipe object needs to be sent to the store so it knows which recipe to remove from the state.favoriteRecipes slice.

Complete the function called removeRecipe() such that it returns an action object with the following properties:

* type: The slice being modified is state.favoriteRecipes and the action name is 'removeRecipe'
* payload: The recipe object parameter.
* Solution store.js

import allRecipesData from './data.js';

const initialState = {

  allRecipes: [],

  favoriteRecipes: [],

  searchTerm: ''

};

// Dispatched when the user types in the search input.

// Sends the search term to the store.

const setSearchTerm = (term) => {

  return {

    type: 'searchTerm/setSearchTerm',

    payload: term

  };

}

// Dispatched when the user presses the clear search button.

const clearSearchTerm = () => {

  return {

    type: 'searchTerm/clearSearchTerm'

  };

}

// Dispatched when the user first opens the application.

// Sends the allRecipesData array to the store.

const loadData = () => {

      return {

        type: 'allRecipes/loadData',

        payload: allRecipesData

      };

}

// Dispatched when the user clicks on the heart icon of

// a recipe in the "All Recipes" section.

// Sends the recipe object to the store.

const addRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  };

}

// Dispatched when the user clicks on the broken heart

// icon of a recipe in the "Favorite Recipes" section.

// Sends the recipe object to the store.

const removeRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  };

}

Data.js

const allRecipesData = [

  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},

  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},

  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},

  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},

  { id: 4, name: 'Cheeseburger', img: 'img/cheeseburger.jpg'},

  { id: 5, name: 'Churrasco', img: 'img/churrasco.jpg'},

  { id: 6, name: 'Dumplings', img: 'img/dumplings.jpg'},

  { id: 7, name: 'Fish & Chips', img: 'img/fishnchips.jpg'},

  { id: 8, name: 'Hummus', img: 'img/hummus.jpg'},

  { id: 9, name: 'Masala Dosa', img: 'img/masaladosa.jpg'},

  { id: 10, name: 'Pad Thai', img: 'img/padthai.jpg'},

];

export default allRecipesData;

**Immutable Updates & Complex State**

*Remember, the store‘s reducer function is called each time an action is dispatched. It is passed the action and the current state as arguments and returns the store‘s next state.*

The [second rule of reducers](https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers#rules-of-reducers) states that when the reducer is updating the state, it must make a copy and return the copy rather than directly mutating the incoming state. When the state is a mutable data type, like an array or object, this is typically done using the spread operator (...).

Below, the todosReducer for a todo app demonstrates this in action:

const initialState = {  
  filter: 'SHOW\_INCOMPLETE',  
  todos: [  
    { id: 0, text: 'learn redux', completed: false },  
    { id: 1, text: 'build a redux app', completed: true },  
    { id: 2, text: 'do a dance', completed: false },  
  ]  
};  
   
const todosReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'filter/setFilter':  
      return {  
        ...state,  
        filter: action.payload  
      };  
    case 'todos/addTodo':   
      return {  
        ...state,  
        todos: [...state.todos, action.payload]  
      } ;  
    case 'todos/toggleTodo':  
      return {  
        ...state,  
        todos: state.todos.map(todo => {  
          return (todo.id === action.payload.id) ?   
            { ...todo, completed: !todo.completed } :   
            todo;  
        })  
      }  
    default:  
      return state;  
  }  
};

* The todosReducer uses the initialState as the default state value.
* When a 'filter/setFilter' action is received, it spreads the old state‘s contents (...state) into a new object before updating the filter property with the new filter from action.payload.
* When a 'todos/addTodo' action is received, it does the same except this time, since state.todos is a mutable array, its contents are also spread into a new array, with the new todo from action.payload added to the end.
* When a 'todos/toggleTodo action is received, it uses the .map() method to create a copy of the state.todos array. Additionally, the todo being toggled is found using action.payload.id and it is spread into a new object and updated.

*It should be clarified that the state.todos.map() method only makes a “shallow” copy of the array, meaning the objects inside share the same references as the originals. Therefore, mutations to the objects within the copy will affect the objects within the original.*

**Question: 1.**

First up is the searchTerm/setSearchTerm action. This action will be dispatched with a payload whose value is the term to be set as the new value for state.searchTerm.

Within the switch statement of recipesReducer(), fix the case that handles the 'searchTerm/setSearchTerm' action type.

* For this case, the reducer should return a new state object with an updated searchTerm slice set to the new term provided by action.payload.

If done correctly, the second state printed to the console should show the search term set to "cheese".

**2.**Now, let’s fix the case for the favoriteRecipes/addRecipe action type. This action will be dispatched with a payload whose value is the recipe object to be added to the state.favoriteRecipes array.

* For this action type, the reducer should return a new state object with an updated favoriteRecipes slice.
* The new value should be a new array that includes all the previously added values in addition to the new recipe (from action.payload) added to the end.

Remember, you must not mutate the incoming state object or the original state.favoriteRecipes array!

**3.**The final case to fix is for the favoriteRecipes/removeRecipe action type. This action will be dispatched with a payload whose value is the recipe object to be removed from the state.favoriteRecipes array.

* For this case, the reducer should return a new state object with an updated favoriteRecipes slice.
* The favoriteRecipes slice should be a new array that includes all the existing values from state.favoriteRecipes except for the recipe from action.payload.

We recommend that you use the [.filter()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) array method and filter out the element whose 'id' matches the recipe from action.payload.

Solution: store.js

import { createStore } from 'redux';

import allRecipesData from './data.js';

const initialState = {

  allRecipes: [],

  favoriteRecipes: [],

  searchTerm: ''

};

const setSearchTerm = (term) => {

  return {

    type: 'searchTerm/setSearchTerm',

    payload: term

  };

}

const clearSearchTerm = () => {

  return {

    type: 'searchTerm/clearSearchTerm'

  };

};

const loadData = () => {

  return {

    type: 'allRecipes/loadData',

    payload: allRecipesData

  };

};

const addRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  };

};

const removeRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  };

};

/\* Complete this reducer \*/

const recipesReducer = (state = initialState, action) => {

  switch(action.type) {

    case 'allRecipes/loadData':

      return {

        ...state,

        allRecipes: action.payload

      }

    case 'searchTerm/clearSearchTerm':

      return {

        ...state,

        searchTerm: ''

      }

    case 'searchTerm/setSearchTerm':

      return {...state, searchTerm: action.payload};

    case 'favoriteRecipes/addRecipe':

      return {

        ...state,

        favoriteRecipes: [...state.favoriteRecipes, action.payload]

      };

    case 'favoriteRecipes/removeRecipe':

      return {

        ...state,

        favoriteRecipes: state.favoriteRecipes.filter(element => element.id !== action.payload.id)

      };

    default:

      return state;

  }

};

const store = createStore(recipesReducer);

/\* DO NOT DELETE \*/

printTests();

function printTests() {

  store.dispatch(loadData());

  console.log('Initial State after loading data');

  console.log(store.getState());

  console.log();

  store.dispatch(addRecipe(allRecipesData[0]));

  store.dispatch(addRecipe(allRecipesData[1]));

  store.dispatch(setSearchTerm('cheese'));

  console.log("After favoriting Biscuits and Bulgogi and setting the search term to 'cheese'")

  console.log(store.getState());

  console.log();

  store.dispatch(removeRecipe(allRecipesData[1]));

  store.dispatch(clearSearchTerm());

  console.log("After un-favoriting Bulgogi and clearing the search term:")

  console.log(store.getState());

}

**Data.js**

// The data has been reduced to make the output

// terminal easier to read.

const allRecipesData = [

  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},

  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},

  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},

  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},

];

export default allRecipesData;

**Reducer Composition**

The solution is to follow a pattern called *reducer composition*. In this pattern, individual *slice reducers* are responsible for updating only one slice of the application’s state, and their results are recombined by a rootReducer to form a single state object.

// Handles only `state.todos`.  
const initialTodos = [  
  { id: 0, text: 'learn redux', completed: false },  
  { id: 1, text: 'build a redux app', completed: true },  
  { id: 2, text: 'do a dance', completed: false },  
];  
const todosReducer = (todos = initialTodos, action) => {  
  switch (action.type) {  
    case 'todos/addTodo':   
      return [...todos, action.payload]  
    case 'todos/toggleTodo':  
      return todos.map(todo => {  
        return (todo.id === action.payload.id) ?   
          { ...todo, completed: !todo.completed } :   
          {...todo};  
      });  
    default:  
      return todos;  
  }  
};  
   
// Handles only `state.filter`  
const initialFilter = 'SHOW\_INCOMPLETE',  
const filterReducer = (filter = initialFilter, action) => {  
  switch (action.type) {  
    case 'filter/setFilter':  
      return action.payload;  
    default:  
      return filter;  
};  
   
const rootReducer = (state = {}, action) => {  
  const nextState = {  
    todos: todosReducer(state.todos, action),  
    filter: filterReducer(state.filter, action)  
  };  
  return nextState;  
};  
   
const store = createStore(rootReducer);

In the reducer composition pattern, when an action is dispatched to the store:

* The rootReducer calls each slice reducer, regardless of the action.type, with the incoming action and the appropriate slice of the state as arguments.
* The slice reducers each determine if they need to update their slice of state, or simply return their slice of state unchanged.
* The rootReducer reassembles the updated slice values in a new state object.

Take a look at **store.js** where you will find that the reducer for the Recipe app that you wrote in the last exercise (which can be found in **reducer-old.js**) has been partially rewritten to follow the reducer composition pattern:

* The initialState object has been replaced by individual initialSliceName variables which are used as default values for each slice reducer’s slice of state. This is another common feature of the reducer composition pattern.
* The allRecipesReducer and searchTermReducer slice reducers have been created for you. Notice that they each have a default case.
* Both slice reducers are called within the rootReducer to update their respective slices of state.

All that’s left is to complete the favoriteRecipesReducer() and include it in the rootReducer()!

**Question:1** Currently, the default favoriteRecipes value for favoriteRecipesReducer() is the string 'REPLACE\_ME'. Let’s fix that.

First, declare a variable named initialFavoriteRecipes and assign it to an empty array ([]).

Then, assign the default favoriteRecipes value for favoriteRecipesReducer() to initialFavoriteRecipes.

**2.** Next, complete the favoriteRecipesReducer such that it immutably updates the state.favoriteRecipes slice in response to the following action.type cases:

* 'favoriteRecipes/addRecipe': Return a new array with all of the prior values of favoriteRecipes with the action.payload value added to the end.
* 'favoriteRecipes/removeRecipe': Return a new array with all of the prior values of favoriteRecipes with the action.payload value removed.
* default: Return favoriteRecipes unchanged.

Refer to **reducer-old.js** for the solution code from the last exercise.

**3.** Well done! Now that you have the favoriteRecipesReducer() completed, you can use it within the rootReducer to update the state.favoriteRecipes slice.

Within rootReducer(), add a favoriteRecipes property to the nextState object.

Then, call favoriteRecipesReducer(), passing its slice of state and the action as arguments, and store the result as the value for nextState.favoriteRecipes.

Solution

Store.js

import { createStore } from 'redux';

import allRecipesData from './data.js';

// Action Creators

////////////////////////////////////////

const addRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  };

}

const removeRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  };

}

const setSearchTerm = (term) => {

  return {

    type: 'searchTerm/setSearchTerm',

    payload: term

  }

}

const clearSearchTerm = () => {

  return {

    type: 'searchTerm/clearSearchTerm'

  };

}

const loadData = () => {

  return {

    type: 'allRecipes/loadData',

    payload: allRecipesData

  };

}

// Reducers

////////////////////////////////////////

const initialAllRecipes = [];

const allRecipesReducer = (allRecipes = initialAllRecipes, action) => {

  switch(action.type) {

    case 'allRecipes/loadData':

      return action.payload

    default:

      return allRecipes;

  }

}

const initialSearchTerm = '';

const searchTermReducer = (searchTerm = initialSearchTerm, action) => {

  switch(action.type) {

    case 'searchTerm/setSearchTerm':

      return action.payload;

    case 'searchTerm/clearSearchTerm':

      return '';

    default:

      return searchTerm;

  }

}

// Create the initial state for this reducer.

var initialFavoriteRecipes = [];

const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes, action) => {

  switch(action.type) {

    // Add action.type cases here.

    case 'favoriteRecipes/addRecipe':

      return [...favoriteRecipes, action.payload];

    case 'favoriteRecipes/removeRecipe':

      return favoriteRecipes.filter(element => element.id !== action.payload.id);

    default:

      return favoriteRecipes;

  }

}

const rootReducer = (state = {}, action) => {

  const nextState = {

    allRecipes: allRecipesReducer(state.allRecipes, action),

    searchTerm: searchTermReducer(state.searchTerm, action),

    favoriteRecipes: favoriteRecipesReducer(state.favoriteRecipes, action)

  }

  return nextState;

}

const store = createStore(rootReducer);

**reducer-old.js**

// The data has been reduced to make the output

// terminal easier to read.

const allRecipesData = [

  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},

  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},

  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},

  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},

];

export default allRecipesData;

**combineReducers**

In the reducer composition pattern, the same steps are taken by the rootReducer for each slice reducer:

1. call the slice reducer with its slice of the state and the action as arguments
2. store the returned slice of state in a new object that is ultimately returned by the rootReducer().

import { createStore } from 'redux';  
   
// todosReducer and filterReducer omitted  
   
const rootReducer = (state = {}, action) => {  
  const nextState = {  
    todos: todosReducer(state.todos, action),  
    filter: filterReducer(state.filter, action)  
  };  
  return nextState;  
};  
   
const store = createStore(rootReducer);

The Redux package helps facilitate this pattern by providing a utility function called combineReducers() which handles this boilerplate for us:

import { createStore, combineReducers } from 'redux'  
   
// todosReducer and filterReducer omitted.  
   
const reducers = {  
    todos: todosReducer,  
    filter: filterReducer  
};  
const rootReducer = combineReducers(reducers);  
const store = createStore(rootReducer);

Let’s break this code down:

* The reducers object contains the slice reducers for the application. The keys of the object correspond to the name of the slice being managed by the reducer value.
* The combineReducers() function accepts this reducers object and returns a rootReducer function.
* The returned rootReducer is passed to createStore() to create a store object.

Just as before, when an action is dispatched to the store, the rootReducer() is executed which then calls each slice reducer, passing along the action and the appropriate slice of state.

The last 6 lines of this example can be rewritten inline:

const store = createStore(combineReducers({  
    todos: todosReducer,  
    filter: filterReducer  
}));

Take a look at **store.js** where you will find the slice reducers that you created in the last exercise. Now, however, the rootReducer() is missing. Rather than creating this function by hand, you will use combineReducers().

Question: First, at the top of **store.js**, import combineReducers from the redux library.

**2.** combineReducers() accepts an object of reducers as its argument. Let’s create one!

At the bottom of **store.js**, create a variable called reducers. Assign to it an object with three properties: allRecipes, favoriteRecipes, searchTerm. Each property should be assigned its associated slice reducer.

**3.** Now, declare another variable called rootReducer. Call combineReducers() with the reducers object as an argument and assign the returned value to rootReducer.

**4.** Finally, pass the rootReducer to the createStore() function and save the returned value in a new variable called store.

Solution:store.js

// Import combineReducers from redux here.

import { combineReducers, createStore } from 'redux';

import allRecipesData from './data.js';

// Action Creators

////////////////////////////////////////

const addRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  };

}

const removeRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  };

}

const setSearchTerm = (term) => {

  return {

    type: 'searchTerm/setSearchTerm',

    payload: term

  }

}

const clearSearchTerm = () => {

  return {

    type: 'searchTerm/clearSearchTerm'

  };

}

const loadData = () => {

  return {

    type: 'allRecipes/loadData',

    payload: allRecipeData

  };

}

// Reducers

////////////////////////////////////////

const initialAllRecipes = [];

const allRecipesReducer = (allRecipes = initialAllRecipes, action) => {

  switch(action.type) {

    case 'allRecipes/loadData':

      return action.payload

    default:

      return allRecipes;

  }

}

const initialSearchTerm = '';

const searchTermReducer = (searchTerm = initialSearchTerm, action) => {

  switch(action.type) {

    case 'searchTerm/setSearchTerm':

      return action.payload

    case 'searchTerm/clearSearchTerm':

      return ''

    default:

      return searchTerm;

  }

}

const initialFavoriteRecipes = [];

const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes, action) => {

  switch(action.type) {

    case 'favoriteRecipes/addRecipe':

      return [...favoriteRecipes, action.payload]

    case 'favoriteRecipes/removeRecipe':

      return favoriteRecipes.filter(recipe => {

        return recipe.id !== action.payload.id

      });

    default:

      return favoriteRecipes;

  }

}

// Create your `rootReducer` here using combineReducers().

const reducers = {

  allRecipes: allRecipesReducer,

  favoriteRecipes: favoriteRecipesReducer,

  searchTerm: searchTermReducer

};

const rootReducer = combineReducers(reducers);

const store = createStore(rootReducer);

data.js

const allRecipesData = [

  { id: 0, name: 'Biscuits', img: 'img/biscuits.jpg'},

  { id: 1, name: 'Bulgogi', img: 'img/bulgogi.jpg'},

  { id: 2, name: 'Calamari', img: 'img/calamari.jpg'},

  { id: 3, name: 'Ceviche', img: 'img/ceviche.jpg'},

  { id: 4, name: 'Cheeseburger', img: 'img/cheeseburger.jpg'},

  { id: 5, name: 'Churrasco', img: 'img/churrasco.jpg'},

  { id: 6, name: 'Dumplings', img: 'img/dumplings.jpg'},

  { id: 7, name: 'Fish & Chips', img: 'img/fishnchips.jpg'},

  { id: 8, name: 'Hummus', img: 'img/hummus.jpg'},

  { id: 9, name: 'Masala Dosa', img: 'img/masaladosa.jpg'},

  { id: 10, name: 'Pad Thai', img: 'img/padthai.jpg'},

];

export default allRecipesData;

**A File Structure for Redux**

break up a Redux application using the [Redux Ducks pattern](https://github.com/erikras/ducks-modular-redux), like so:

src/  
|-- index.js  
|-- app/  
    |-- store.js  
|-- features/  
    |-- featureA/  
        |-- featureASlice.js  
    |-- featureB/  
        |-- featureBSlice.js

As you can see in your coding workspace, this file structure has already been set up for you.

All of the Redux logic lives within the top-level directory called **src/**. It contains:

* The entry point for the entire application, **index.js** (we will return to this file in the next exercise).
* The sub-directories **app/** and **features/**.

The **src/app/** directory has only one file (for now), **store.js**. As before, the ultimate purpose of this file is to create the rootReducer and the Redux store. Now, however, you’ll notice that the file is empty! So where did the reducers and action creators go?!

The **src/features/** directory, and its own **src/features/featureX/** sub-directories, contain all of the code relating to each individual slice of the store‘s state. For example, for the state.favoriteRecipes slice, its slice reducer and action creators can be found in the file called **src/features/favoriteRecipes/favoriteRecipesSlice.js**

**Question: 1** The reducers object passed to combineReducers() should contain the slice reducers responsible for updating the various slices of the store‘s state. In the prior lesson, those slice reducers all existed in the same file. Now, you need to import them.

At the top of the **store.js** file, import the following values from their respective files:

* allRecipesReducer
* favoriteRecipesReducer
* searchTermReducer

**2.** Excellent! Now that you have imported the slice reducers, you use them to construct the reducers object to be passed to combineReducers().

Within the reducers object, add three key:value pairs where each key is the name of a slice and each value is the slice reducer responsible for managing that slice’s state.

**3.**Now that you have the reducers object, you can create the store using a combination of the combineReducers() and createStore() Redux functions.

You are going to do this all in one line of code!

* First call combineReducers() with reducers as an argument.
* Then, pass the entire combineReducers(reducers) function call as an argument to createStore().
* Finally, store the value returned by createStore() in a new variable called store.

**4.**Well done! You’ve reconnected all of the slice reducers from separate files back into the store within **src/app/store.js**. In the next exercise, you’ll learn how to build on this application structure by incorporating React components and dispatching actions from them. To do this, the store needs to be available to other parts of the application.

Export the store value from **src/app/store.js**.

Solution: store.js

import { createStore, combineReducers } from 'redux';

// Import the slice reducers here.

import { favoriteRecipesReducer } from '../features/favoriteRecipes/favoriteRecipesSlice.js';

import { allRecipesReducer } from '../features/allRecipes/allRecipesSlice.js';

import { searchTermReducer } from '../features/searchTerm/searchTermSlice.js';

const reducers = {

  // Add the slice properties here

  favoriteRecipes: favoriteRecipesReducer,

  allRecipes: allRecipesReducer,

  searchTerm: searchTermReducer,

}

// Declare the store here.

export const store = createStore(combineReducers(reducers))

favoriteRecipeSlice.js

export const addRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  };

}

export const removeRecipe = (recipe) => {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  };

}

const initialFavoriteRecipes = [];

export const favoriteRecipesReducer = (favoriteRecipes = initialFavoriteRecipes, action) => {

  switch(action.type) {

    case 'favoriteRecipes/addRecipe':

      return [...favoriteRecipes, action.payload]

    case 'favoriteRecipes/removeRecipe':

      return favoriteRecipes.filter(recipe => {

        return recipe.id !== action.payload.id

      });

    default:

      return favoriteRecipes;

  }

}

**Passing Store Data Through the Top-Level React Component**

Take a look at the **src** folder in your workspace and you will find the following file structure (new files have a (+) next to their name):

src/  
|-- index.js  
|-- app/  
    |-- App.js (+)  
    |-- store.js  
|-- components/  
    |-- FavoriteButton.js (+)  
    |-- Recipe.js (+)  
|-- features/  
    |-- allRecipes/  
        |-- AllRecipes.js (+)  
        |-- allRecipesSlice.js  
    |-- favoriteRecipes/  
        |-- FavoriteRecipes.js (+)  
        |-- favoriteRecipesSlice.js  
    |-- searchTerm/  
        |-- SearchTerm.js (+)  
        |-- searchTermSlice.js

*If you look at the actual file structure in your code editor, you may notice a few unfamiliar files / directories not mentioned in the structure above. The* ***test/*** *directory and* ***index.compiled.js*** *file are used to test your code on Codecademy. You can ignore them.*

The new components are:

* <App />: The top-level component for the entire application.
* <AllRecipes />: The component for rendering the recipes loaded from the “database”.
* <FavoriteRecipes />: The component for rendering the recipes favorited by the user.
* <SearchTerm />: The component for rendering the search bar that filters the visible recipes.
* <Recipe /> and <FavoriteButton />: Generic components used by <AllRecipes /> and <FavoriteRecipes />

Aside from the generic components, each feature-related React component file is located in the same directory as the slice file that manages the data rendered by that component. For example, **FavoriteRecipes.js** and **favoriteRecipesSlice.js** are both in the **src/features/favoriteRecipes/** directory.

Open the **src/app/App.js** file where the top-level component, <App />, is stored. As in most React applications, this top-level component will render each feature-component and pass any data needed by those components as prop values. In Redux applications, the data passed to each feature-component includes:

1. The slice of the store‘s state to be rendered. For example, the state.searchTerm slice is passed to the <SearchTerm /> component.
2. The store.dispatch method to trigger state changes through user interactions within the component. For example, the <SearchTerm /> component will need to dispatch setSearchTerm() actions.

This distribution of the store.dispatch method and the slices of state to all of the feature-components, via the <App /> component, begins in the **index.js** file. Open up the **src/index.js** file where you will see some standard React code for rendering the top-level <App /> component. You’ll notice that the store is missing and the <App /> component isn’t receiving any props!

QUESTION: **1.** In order to pass the store‘s current state and its dispatch method to the <App /> component, the store must first be imported into the **index.js** file.

At the top of **index.js**, import the store from **store.js**.

**2.**Next, get the current state of the store and pass it to the <App /> component as a prop called state.

*Note: You won’t see anything rendered until the next checkpoint!*

**3.**The <App /> component isn’t rendering yet because it is expecting to receive a dispatch method.

Pass the store.dispatch method to the <App /> component as a prop called dispatch.

If done correctly, you should see the <FavoriteRecipes /> and <AllRecipes /> components rendered (without data, for now)!

**4.**Why is the recipe data not being rendered? Remember that the state.allRecipes slice begins as an empty array and the data is only loaded AFTER the user opens the page. This data fetch is happening but render isn’t subscribed to changes to the store yet!

At the bottom of **index.js**, use store.subscribe() to subscribe the render function to the store such that each time the store‘s state changes, the entire <App /> will be re-rendered.

SOLUTION: INDEX.JS

import React from 'react';

import ReactDOM from 'react-dom';

import { App } from './app/App.js';

// Import 'store' here.

import { store } from './app/store.js';

const render = () => {

  // Pass `state` and `dispatch` props to <App />

  ReactDOM.render(

    <App

  state={store.getState()}

  dispatch={store.dispatch}

    />,

    document.getElementById('root')

  )

}

render();

// Subscribe render to changes to the `store`

store.subscribe(render);

APP.JS

import React from 'react';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';

import { SearchTerm } from '../features/searchTerm/SearchTerm.js';

export function App(props) {

  const {state, dispatch} = props;

  const visibleAllRecipes = getFilteredRecipes(state.allRecipes, state.searchTerm);

  const visibleFavoriteRecipes = getFilteredRecipes(state.favoriteRecipes, state.searchTerm);

// You'll add the <FavoriteRecipes /> component in the next exercise!

  return (

    <main>

      <section>

        <SearchTerm

          searchTerm={state.searchTerm}

          dispatch={dispatch}

        />

      </section>

      <section>

        <h2>Favorite Recipes</h2>

      </section>

      <hr />

      <section>

        <h2>All Recipes</h2>

        <AllRecipes

          allRecipes={visibleAllRecipes}

          dispatch={dispatch}

        />

      </section>

    </main>

  )

}

/\* Utility Helpers \*/

function getFilteredRecipes(recipes, searchTerm) {

  return recipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.toLowerCase()));

}

**Using Store Data Within Feature Components**

you were able to pass the current state of the store and its store.dispatch method to the top-level component, <App />. This allowed the <App /> component to distribute the dispatch method and the slices of the store‘s state to each feature-component.

So it looks like you’re done, right? Not quite. Try adding a favorite recipe and you’ll see that it just disappears! Take a closer look at **App.js** and you’ll notice that the <FavoriteRecipes /> component is missing. Then, open up **FavoriteRecipes.js** and you’ll see that it is also incomplete. Let’s fix that.

Plugging in a feature-component to a Redux application involves the following steps:

* Import the React feature-components into the top-level **App.js** file.
* Render each feature-component and pass along the slice of state and the dispatch method as props.
* Within each feature-component:
  + Extract the slice of state and dispatch from props.
  + Render the component using data from the slice of state.
  + Import any action creators from the associated slice file.
  + Dispatch actions in response to user inputs within the component.

This process is not different from how you implemented a React + Redux application in the past. Now, however, you must consider that the slices of the store‘s state and the dispatch method must be passed through props.

**Question: 1.**

Open up the **App.js** file.

First, import the FavoriteRecipes component from the **FavoriteRecipes.js** file.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now, you can add in the <FavoriteRecipes /> component to the <App /> component’s structure. Like the other two components, you will need to pass the dispatch method to the component as a prop.

The slice data passed to <FavoriteRecipes /> will need to be filtered first based on the value of state.searchTerm. The filtered version of state.favoriteRecipes has been created for you and stored in the variable visibleFavoriteRecipes.

Within the return statement of the <App /> component, in the space below the <h2>Favorite Recipes</h2> element, add in a <FavoriteRecipes /> component. You should then pass along the following props:

* favoriteRecipes: the visibleFavoriteRecipes value
* dispatch: the dispatch method from the store.

If you complete this step correctly, you should see a blank square rendered under the “Favorite Recipes” header.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Open up the **FavoriteRecipes.js** file. The job of any presentational component in a Redux app is twofold:

1. Render the data for their associated slice of state.
2. Dispatch actions in response to user interaction within the component.

To do these two things, <FavoriteRecipes /> was given two props: favoriteRecipes and dispatch.

At the top of FavoriteRecipes(), extract these two values from the props parameter.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Now that the FavoriteRecipes() component has access to the favoriteRecipes slice of state, you can render its data instead of the blank box! Take a look at the return statement:

return (  
  <div className="recipes-container">  
    {['REPLACE\_ME'].map(createRecipeComponent)}  
  </div>  
);

Replace the entire ['REPLACE\_ME'] array with the favoriteRecipes prop value.

If done correctly, every recipe object within favoriteRecipes will be mapped to a <Recipe /> component and be rendered (try it out!).

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

The <FavoriteRecipes /> component wants to dispatch an action to the store within onRemoveRecipeHandler(), but where are the action creators to help create those actions?

Remember, they have been moved to, and exported from, the **favoriteRecipesSlice.js** file!

At the top of **FavoriteRecipes.js**, import the action creator function, removeRecipe.

Checkpoint 6 Passed

Stuck? Get a hint

**6.**

Finally, the removeRecipe() action creator accepts a recipe argument.

Within onRemoveRecipeHandler(), which receives a recipe parameter, dispatch a removeRecipe() action with recipe as an argument.

Checkpoint

**Solution app.js**

import React from 'react';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';

import { SearchTerm } from '../features/searchTerm/SearchTerm.js';

// Import the FavoriteRecipes component here.

import {FavoriteRecipes} from '../features/favoriteRecipes/FavoriteRecipes.js'

export function App(props) {

  const {state, dispatch} = props;

  const visibleAllRecipes = getFilteredRecipes(state.allRecipes, state.searchTerm);

  const visibleFavoriteRecipes = getFilteredRecipes(state.favoriteRecipes, state.searchTerm);

  // Render the <FavoriteRecipes /> component.

  // Pass `dispatch` and `favoriteRecipes` props.

  return (

    <main>

      <section>

        <SearchTerm

          searchTerm={state.searchTerm}

          dispatch={dispatch}

        />

      </section>

      <section>

        <h2>Favorite Recipes</h2>

        <FavoriteRecipes

        dispatch={dispatch}

          favoriteRecipes={visibleFavoriteRecipes}

        />

      </section>

      <hr />

      <section>

        <h2>All Recipes</h2>

        <AllRecipes

          allRecipes={visibleAllRecipes}

          dispatch={dispatch}

        />

      </section>

    </main>

  )

}

/\* Utility Helpers \*/

function getFilteredRecipes(recipes, searchTerm) {

  return recipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.toLowerCase()));

}

**favoriteRecipeSlice.js**

import React from 'react';

import FavoriteButton from "../../components/FavoriteButton";

import Recipe from "../../components/Recipe";

const unfavoriteIconUrl = 'https://static-assets.codecademy.com/Courses/Learn-Redux/Recipes-App/icons/unfavorite.svg'

// Import removeRecipe from favoriteRecipesSlice.js

import {removeRecipe} from './favoriteRecipesSlice.js'

export const FavoriteRecipes = (props) =>{

  const favoriteRecipes = props.favoriteRecipes;

  const dispatch = props.dispatch;

  const onRemoveRecipeHandler = (recipe) => {

    // Dispatch a removeRecipe() action.

    dispatch(removeRecipe(recipe))

  };

  // Map the recipe objects in favoriteRecipes to render <Recipe /> components.

  return (

    <div id='favorite-recipes' className="recipes-container">

      {favoriteRecipes.map(createRecipeComponent)}

    </div>

  );

  // Helper Function

  function createRecipeComponent(recipe) {

    return (

      <Recipe recipe={recipe} key={recipe.id}>

        <FavoriteButton

          onClickHandler={() => onRemoveRecipeHandler(recipe)}

          icon={unfavoriteIconUrl}

        >

          Remove Favorite

        </FavoriteButton>

      </Recipe>

    )

  }

};

**Review**

* The action.payload property is used to hold additional data that the reducer might need to carry out a given action. The name payload is simply a convention and its value can be anything!
* The spread syntax (...) and array methods such as .map(), .slice(), and .filter() can be used to immutably update the state of a complex app.
* Reducer composition is a design pattern for managing a Redux store with multiple slices.
* The root reducer delegates actions to slice reducers that are responsible for updating only their assigned slice of the store’s state. The root reducer then reassembles the slices into a new state object.
* combineReducers() is a method provided by the redux library that accepts a collection of reducer functions and returns a rootReducer that implements the reducer composition pattern.
* In a Redux application, slice reducers are often written in separate files. This pattern is known as [Redux Ducks](https://github.com/erikras/ducks-modular-redux).

**THE REACT REDUX LIBRARY**

Introduction

[React Redux](https://react-redux.js.org/) is the official Redux-UI binding package for React. This means React Redux handles the interactions between React’s optimized UI rendering and Redux’s state management.

Most importantly you should be familiar with Redux’s one-way data flow:

* Starting with the application state
* Rendering components based on that state
* Dispatching an action to trigger state changes
* Re-rendering any component affected by the new state

React Redux provides tools that will help you implement each stage of the data flow with a React UI.

### Instructions

Before continuing you should note the application’s current functionality that will be replaced in the following exercises:

1. Calling ReactDOM.render() using render().
2. Passing store.getState() through <App> component props.
3. Passing store.dispatch through <App> component props.
4. Subscribing render() to the Redux store so it is called after state updates.
5. Using the props parameter in **App.js** to pass data through the component, also known as props drilling.

**Why React Redux**

React is handling the UI rendering through ReactDOM.render() which is inside the render() function. Redux is managing the state with the store and passing the state and dispatch references through props. Redux also triggers the UI to re-render with store.subscribe(render).

The first issue with this implementation is passing the state and dispatch reference through props. Using props to access the state or to dispatch actions adds unneeded complexity. Also, the <App> component uses props drilling, which means it is passing props to child components without using them. The last issue is subscribing render() to changes in the state. This creates more overhead by repeatedly calling ReactDOM.render(), which is not the intended implementation for rendering React components.

With React Redux you will learn how to solve these issues by:

* Giving the entire application access to the Redux store without using props and props drilling.
* Subscribing individual components to specific pieces of the application state for optimized rendering.
* Easily dispatching actions within components.

**Installing react-redux**

To take advantage of React Redux within your application, you must install the react-redux package using npm, the Node Package Manager. If you’re not familiar with npm, you can [learn more in the documentation](https://docs.npmjs.com/about-npm). This includes understanding the directory structure, confirming installation and versions of your packages.

To install React Redux using npm, type the following command into your terminal and hit the “enter” key:

npm install react-redux

After installation, your application will have access to the tools provided by the React Redux package.

**THE REACT REDUX LIBRARY**

The <Provider> Component

The <Provider> component from the react-redux library gives the components of an application access to the Redux store without the need to pass the store directly to the React components through props. To implement this, wrap the <Provider> component around the top-level component and pass store through the store prop of the <Provider>:

import React from 'react';  
import ReactDOM from 'react-dom';  
import { App } from './app/App.js';  
import { store } from './app/store.js';  
   
import { Provider } from 'react-redux'  
   
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById("root")  
);

This example demonstrates:

* Importing Provider from react-redux.
* Wrapping the <Provider> component around the root <App /> component.
* Passing the store through the <Provider> component’s store prop.

Before moving on to the instructions, it is important to note that ReactDOM.render() is no longer inside a render() function and therefore nothing is subscribed to changes in the Redux store. This is something you will address in the next few exercises. Like many improvements in the world, you sometimes have to break the process to make it better.

Question: **1.**

Before you can use the <Provider> component you must import it.

In the **index.js** file, create an import statement that retrieves the Provider component from react-redux.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

The next step is to insert the <Provider> component into the recipe application.

Inside ReactDOM.render(), wrap the top level component, <App> with the <Provider> component.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

With the <App> component now nested inside the <Provider> component, pass the imported Redux store through the <Provider> component store prop.

This begins the process of accessing data throughout the application components but you must do a couple more things to see any progress.

Checkpoint

Solution: index.js

import React from 'react';

import ReactDOM from 'react-dom';

// Import Provider component below.

import { Provider } from 'react-redux';

import { App } from './app/App.js';

import { store } from './app/store.js';

ReactDOM.render(

  // Wrap root application with Provider component below.

  <Provider store={store}>

    <App />

    </Provider>,

  document.getElementById('root')

);

App.js

import React from 'react';

import { loadData } from '../features/allRecipes/allRecipesSlice.js';

import { AllRecipes } from '../features/allRecipes/AllRecipes.js';

import { SearchTerm } from '../features/searchTerm/SearchTerm.js';

import { FavoriteRecipes } from '../features/favoriteRecipes/FavoriteRecipes.js';

export function App() {

  return (

    <main>

      <section>

        <SearchTerm />

      </section>

      <section>

        <h2>Favorite Recipes</h2>

        <FavoriteRecipes />

      </section>

      <hr />

      <section>

        <h2>All Recipes</h2>

        <AllRecipes />

      </section>

    </main>

  )

}

Selectors

A selector function, or selector, is a pure function that selects data from the Redux store’s state. Each component in an application that needs access to the state will have one or more selectors that extract only the necessary data for that component.

As pure functions, selectors should output the same data given the same input. This means that no randomness or side effects can occur inside the function.

A selector:

* Takes state as an argument.
* Returns what is needed by the component from state.

/\*   
Given a state with an array of objects, labeled 'todos', and a string, labeled 'filter':  
   
state = {  
  todos: [  
    {id: 1, isComplete: true, text: 'Go shopping'}  
    {id: 2, isComplete: false, text: 'Call home'}   
  ],  
  filter: 'SHOW\_ALL'  
}  
\*/  
   
// Select the current filter  
export const selectFilter = state => state.filter;  
   
// Select the `text` from each todo in an array.  
export const selectTodoText = state => state.todos.map(  
  todo => todo.text);  
   
// Select the id values of completed todos in an array.  
export const selectIsCompleteIDs = state => state.todos.filter(  
  todo => todo.isComplete).map(todo => todo.id)

1. The first selector selectFilter returns the string state.filter.
2. selectTodoText returns an array of the .text value for each todo object .
3. selectIsCompleteIDs returns an array of the id values from the todo objects where isCompleted is true.

Question: First, you need to implement the selectAllRecipes selector.

Towards the bottom of **allRecipesSlice.js**, implement selectAllRecipes and test the code:

1. Start with an export statement
2. Define selectAllRecipes with state as the only argument
3. Return the allRecipes piece of the state
4. To test the selector, uncomment the call to testSelectAllRecipes() at the bottom of the file

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

You can now implement a new selector that retrieves the recipes based on the searchTerm in the state.

Below the selectAllRecipes selector:

1. Start with an export statement
2. Define selectFilteredAllRecipes with state as the only argument
3. For now, leave the function body empty ({})

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

You will use the selectAllRecipes and selectSearchTerm to retrieve data for selectFilteredAllRecipes.

Inside the function body of selectFilteredAllRecipes, create two variables:

* allRecipes and assign the return value of selectAllRecipes(state)
* searchTerm and assign the return value of selectSearchTerm(state)

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Now you have an array of the recipes in allrecipes and the current search term string in searchTerm. To get a filtered list of recipes based on the search term you will use the JavaScript array [filter()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) method.

Inside selectFilteredAllRecipes, call allRecipes.filter() with the following callback function as the argument:

recipe => recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

Then, return the result.

Uncomment the test function testSelectFilteredAllRecipes() to see the filtered recipe names printed in the output console.

Selectors can be simple or complex. All are an important step to accessing the Redux store data within components.

Checkpoint

**Solution: allRecipeSlice.js**

import allRecipesData from '../../../data.js'

import { selectSearchTerm } from '../searchTerm/searchTermSlice.js';

export const loadData = () => {

  return {

    type: 'allRecipes/loadData',

    payload: allRecipesData

  }

}

const initialState = [];

export const allRecipesReducer = (allRecipes = initialState, action) => {

  switch (action.type) {

    case 'allRecipes/loadData':

      return action.payload;

    case 'favoriteRecipes/addRecipe':

      return allRecipes.filter(recipe => recipe.id !== action.payload.id);

    case 'favoriteRecipes/removeRecipe':

      return [...allRecipes, action.payload]

    default:

      return allRecipes;

  }

}

// Implement the selectors below.

export const selectAllRecipes = state => state.allRecipes;

export const selectFilteredAllRecipes = state => {

  const allRecipes = selectAllRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return allRecipes.filter(recipe => recipe.name.toLowerCase().includes(searchTerm.toLowerCase()))

};

// This code is for testing the seletors only.

const testState = {

  allRecipes: allRecipesData,

  searchTerm: 'ch'

}

const testSelectAllRecipes = () => {

  console.log('All Recipes')

  console.log(selectAllRecipes(testState));

}

const testSelectFilteredAllRecipes = () => {

  console.log('\nRecipes filtered by searchTerm')

  console.log(selectFilteredAllRecipes(testState));

}

// Uncomment these to test each selector.

testSelectAllRecipes();

testSelectFilteredAllRecipes();

**searchTeamSlice.js**

const initialState = ''

export const searchTermReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'searchTerm/setSearchTerm':

      return action.payload;

    case 'searchTerm/clearSearchTerm':

      return '';

    default:

      return state;

  }

}

export function setSearchTerm(term) {

  return {

    type: 'searchTerm/setSearchTerm',

    payload: term

  }

}

export function clearSearchTerm() {

  return {

    type: 'searchTerm/clearSearchTerm'

  }

}

export const selectSearchTerm = (state) => state.searchTerm;

**The useSelector() Hook**

To use these instructions the useSelector() hook is provided by react-redux. useSelector() accomplishes two things:

* It returns data from the Redux store using selectors
* It subscribes a child component of <Provider /> to changes in the store. React, not Redux, will re-render the component if the data from the selector changes.

These tasks are both accomplished by calling useSelector() inside a component definition and assigning its returned value to a variable.

// Todos.js  
import { useSelector } from 'react-redux';  
import { selectTodos } from 'todosSlice.js';  
   
export const Todos = () => {  
  const todos = useSelector(selectTodos);  
   
  return (  
    <p>{todos}</p>  
  )  
};

In the above example, useSelector() takes the imported selector function selectTodos as an argument. The returned value is the selected data from the Redux store and is assigned to todos.

Calling useSelector()inside the component definition also subscribes the Todos component to re-render if any changes occur in the todos portion of the Redux store. This optimizes the performance of the application by only re-rendering components that have had their data change and not the entire application.

useSelector() can also use an inline selector as an argument:

const todos = useSelector(state => state.todos);

Inline selectors can be useful if you need to use props for data retrieval.

export const Todo = (props) => {  
  const todo = useSelector(state => state.todos[props.id]);

This final example uses props.id to extract a single element from an array or object in the Redux store.

useSelector() completes the 3 step process for accessing data from the Redux store using react-redux.

1. The <Provider> component is used to provide the Redux store to the nested application.
2. Selectors are created to give instructions on retrieving data from the store.
3. useSelector() is called within a child component of <Provider> for executing selector instructions to retrieve data and subscribe to re-rendering.

### Instructions

**1.**

To access Redux store data with useSelector(), you first need to import it from react-redux.

In **AllRecipes.js** import useSelector from react-redux.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Along with useSelector() you need access to the selectFilteredAllRecipes selector defined in the previous exercise.

In **AllRecipes.js** add selectFilteredAllRecipes to the **allRecipesSlice.js** import statement.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

With both import statements complete, you are now able to access the data using the selector function and useSelector().

Inside the AllRecipes() component function:

* Define a variable allRecipes.
* Assign it the value returned by useSelector().
* Pass selectFilteredAllRecipes to useSelector().

In this exercise, the data was initialized with recipes so when you run the code you should see the recipe data rendered in the browser.

Checkpoint

**Solution**

**AllRecipes.js**

import React, { useEffect } from 'react';

// Implement the import statements below.

import{ useSelector } from'react-redux';

// Add the selector to the below import statement

import { selectFilteredAllRecipes, loadData } from './allRecipesSlice.js';

import { addRecipe } from '../favoriteRecipes/favoriteRecipesSlice.js';

import FavoriteButton from "../../components/FavoriteButton";

import Recipe from "../../components/Recipe";

const favoriteIconURL = 'https://static-assets.codecademy.com/Courses/Learn-Redux/Recipes-App/icons/favorite.svg';

export const AllRecipes = () => {

  // Implement allRecipes variable below.

  const allRecipes = useSelector(selectFilteredAllRecipes);

  const onFirstRender = () => {

    // dispatch(loadData());

  }

  useEffect(onFirstRender, []);

  const onAddRecipeHandler = (recipe) => {

    // dispatch(addRecipe(recipe));

  };

  return (

    <div className="recipes-container">

      {allRecipes.map((recipe) => (

        <Recipe recipe={recipe} key={recipe.id}>

          <FavoriteButton

            onClickHandler={() => onAddRecipeHandler(recipe)}

            icon={favoriteIconURL}

          >

            Add to Favorites

          </FavoriteButton>

        </Recipe>

      ))}

    </div>

  );

};

**Review**

* React and Redux work well together but need more to support React’s UI optimization and Redux’s one-way data flow.
* The react-redux library provides React application components access to the Redux store
* The <Provider> component wraps around the root component to give its descendants access to the - Redux store without props drilling
* Selectors are pure function used to access all or part of the state in the Redux store
* useSelector() retrieves the application state through selectors. It must be called from within a component
* useSelector() subscribes components to data retrieved from the selectors. React, not Redux, re-renders those components when the selected data changes
* useDispatch() returns a reference to Redux store dispatch() function

**THE REDUX TOOLKIT**

Intro to Redux Toolkit

Redux Toolkit contains packages and functions that are essential for building a Redux app.

**Installing Redux Toolkit**

you must first install the @reduxjs/toolkit package into your application. You can do this with the Node Package Manager.

While in the root directory of the application, you’ll first need to enter the following command:

npm install @reduxjs/toolkit

**"Slices" of State**

A normal Redux application has a JS object at the top of its state tree. We refer to one key/value section of that object as a “slice”. In the following example, state.todos and state.visibilityFilter are slices.

const state = {  
  todos: [  
    {  
      id: 0,  
      text: "Learn Redux-React",  
      completed: true,  
    },  
    {  
      id: 1,  
      text: "Learn Redux Toolkit",  
      completed: false,  
    }  
  ],   
  visibilityFilter: "SHOW\_ALL"  
}

We typically define one reducer for each slice of the state. Those are called “slice reducers”. Let’s take a look at the slice reducer for the state.todos slice:

/\* todosSlice.js  \*/  
const addTodo = (todo) => {  
  return {  
    type: 'todos/addTodo',  
    payload: todo  
  }  
}  
   
const toggleTodo = (todo) => {  
  return {  
    type: 'todos/toggleTodo',  
    payload: todo  
  }  
}  
   
const todos = (state = [], action) => {  
 switch (action.type) {  
   case 'todos/addTodo':  
     return [  
       ...state,  
       {  
         id: action.payload.id,  
         text: action.payload.text,  
         completed: false  
       }  
     ]  
   case 'todos/toggleTodo':  
     return state.map(todo =>  
       todo.id === action.payload.id ? { ...todo, completed: !todo.completed } : todo  
     )  
   default:  
     return state  
 }  
}

### Instructions

**1.**

At the top of **favoriteRecipesSlice.js** in the code editor, import createSlice() from the '@reduxjs/toolkit' library.

**favoriteRecipeSlice.js**

import { createSlice } from '@reduxjs/toolkit';

Refactoring with createSlice()

In the last exercise, we looked at one way to define a slice reducer and the associated action creators.

/\* todosSlice.js  \*/  
const addTodo = (todo) => {  
 // logic omitted...  
}  
   
const toggleTodo = (todo) => {  
  // logic omitted...  
}  
   
const todos = (state = [], action) => {  
  // logic omitted...  
}

We can do the same work, but more simply, with createSlice()! createSlice() has one parameter, options, which is an object with the following properties

* name: a string that is used as the prefix for generated action types
* initialState: the initial state value for the reducer
* reducers: an object of methods, where the keys determine the action type strings that can update the state, and whose methods are reducers that will be executed when that action type is dispatched. These are sometimes referred to as “case reducers”, because they’re similar to a case in a switch statement.

/\* todosSlice.js \*/  
const options = {  
 name: 'todos',  
 initialState: [],  
 reducers: {  
   addTodo: (state, action) => {  
     return [  
       ...state,  
       {  
         id: action.payload.id,  
         text: action.payload.text,  
         completed: false  
       }  
     ]  
   },  
   toggleTodo: (state, action) => {  
     return state.map(todo =>  
       (todo.id === action.payload.id) ? { ...todo, completed: !todo.completed } : todo  
     )  
   }  
 }  
}  
   
const todosSlice = createSlice(options);

In the options object passed to createSlice() in the snippet above, name is set to 'todos', initialState is set to an empty array, and we have two case reducers: addTodo and toggleTodo. Note that the names of the case reducer functions are conventionally written in lowerCamelCase.

With createSlice()…

* We can write the case reducers as functions inside of an object, instead of having to write a switch/case statement.
* Action creators that correspond to each case reducer function we provide will be automatically generated, so we don’t need to worry about defining those ourselves.
* No default handler needs to be written. The reducer generated by createSlice() will automatically handle all other action types by returning the current state, so we don’t have to list that ourselves.

For now, let’s practice calling createSlice(). In the next exercise, we’ll take a look at the object that is returned by createSlice().

### Instructions

**1.**

Take a look at **oldFavoriteRecipesSlice.js** to see the old way of creating a reducer and action creators for the state.favoriteRecipes slice. Your job is to rewrite this code using createSlice within **favoriteRecipesSlice.js**.

First, createSlice() will need an options object to be passed in as an argument. At the top of the file and below the import statements, declare a variable called options. For now, assign to it an empty object.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Next, add the following three properties and corresponding values to the options object:

* name: 'favoriteRecipes'
* initialState: an empty array
* reducers: an empty object (for now).

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The options.reducers property should hold an object containing the case reducers for the slice.

* Each value in the options.reducers object should be a function whose name corresponds to an action type that the slice can handle.
* Each case reducer should have two parameters, state and action, and return the next state.

Using the logic defined in the **oldFavoriteRecipesSlice.js** file to guide you, add the two methods below to the options.reducers object:

* addRecipe
* removeRecipe

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Well done! The final step is to call createSlice() with the options object as an argument and export the result.

Below the options object,

* Declare a new variable called favoriteRecipesSlice
* Call createSlice() with options as the only argument and assign the result to favoriteRecipesSlice
* Export favoriteRecipesSlice

**favoriteRecipeSlice.js**

import { createSlice } from '@reduxjs/toolkit';

import { selectSearchTerm } from '../searchTerm/searchTermSlice.js';

/\* Create your Slice object here. \*/

const options = {

  name: "favoriteRecipes",

  initialState: [],

  reducers: {

    addRecipe: (state, action) => {

      return [...state, action.payload]

    },

    removeRecipe: (state, action) => {

      return state.filter(recipe => recipe.id !== action.payload.id)

    },

  },

}

 export const favoriteRecipesSlice = createSlice(options);

/\* Do not delete the code below...\*/

export const selectFavoriteRecipes = (state) => state.favoriteRecipes;

export const selectFilteredFavoriteRecipes = (state) => {

  const favoriteRecipes = selectFavoriteRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return favoriteRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

**oldFavoriteRecipeSlice.js**

import { selectSearchTerm } from '../searchTerm/searchTermSlice.js';

// Reducer

///////////////////////////////////////

const initialState = [];

export const favoriteRecipesReducer = (state = initialState, action) => {

  switch (action.type) {

    case 'favoriteRecipes/addRecipe':

      return [...state, action.payload]

    case 'favoriteRecipes/removeRecipe':

      return state.filter(recipe => recipe.id !== action.payload.id)

    default:

      return state;

  }

}

// Action Creators

///////////////////////////////////////

export function addRecipe(recipe) {

  return {

    type: 'favoriteRecipes/addRecipe',

    payload: recipe

  }

}

export function removeRecipe(recipe) {

  return {

    type: 'favoriteRecipes/removeRecipe',

    payload: recipe

  }

}

// Selectors

///////////////////////////////////////

export const selectFavoriteRecipes = (state) => state.favoriteRecipes;

export const selectFilteredFavoriteRecipes = (state) => {

  const favoriteRecipes = selectFavoriteRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return favoriteRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

**Writing "Mutable" Code with Immer**

Redux Toolkit’s createSlice() function uses a library called [Immer](https://immerjs.github.io/immer/) inside of it which helps avoid this mistake.

Immer uses a special JS object called a Proxy to wrap the data you provide and lets you write code that “mutates” that wrapped data. Immer does this by tracking all the changes you’ve made and then uses that list of changes to return an immutably updated value as if you’d written all the immutable update logic by hand.

So, instead of this:

const todosSlice = createSlice({  
  name: 'todos',  
  initialState: [],  
  reducers: {  
    addTodo: (state, action) => {  
      return [  
        ...state,  
        {  
          ...action.payload,  
          completed: false  
        }  
      ]  
    },  
    toggleTodo: (state, action) => {  
      return state.map(todo =>  
        todo.id === action.payload.id ? { ...todo, completed: !todo.completed } : todo  
      )  
    }  
  }  
})

You can write code that looks like this:

const todosSlice = createSlice({  
  name: 'todos',  
  initialState: [],  
  reducers: {  
    addTodo: (state, action) => {  
      state.push({   
        ...action.payload,   
        completed: false   
      })  
    },  
    toggleTodo: (state, action) => {  
      const todo = state.find(todo => todo.id === action.payload.id)  
      if (todo) {  
        todo.completed = !todo.completed  
      }  
    }  
  }  
})

addTodo is calling state.push() here, which is normally bad because the array.push() function mutates the existing array. Similarly, toggleTodo is simply finding the matching todo object, and then mutating it by reassigning its value.

Thanks to Immer, however, this code will work just fine!

### Instructions

**1.**

Inside of the options object, rewrite the logic for the addRecipe case reducer so that it uses .push() instead of the spread operator to add a new recipe.

**favoriteRecipeSlice.js**

import { createSlice } from '@reduxjs/toolkit';

import { selectSearchTerm } from '../searchTerm/searchTermSlice.js';

/\* Modify the options.reducers.addRecipe method. \*/

const options = {

  name: "favoriteRecipes",

  initialState: [],

  reducers: {

    addRecipe: (state, action) => {

      state.push(action.payload);

      },

    removeRecipe: (state, action) => {

      return state.filter(recipe => recipe.id !== action.payload.id)

    },

  },

}

export const favoriteRecipesSlice = createSlice(options);

export const selectFavoriteRecipes = (state) => state.favoriteRecipes;

export const selectFilteredFavoriteRecipes = (state) => {

  const favoriteRecipes = selectFavoriteRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return favoriteRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

**Return Object – Actions**

createSlice() would return an object that looks like this:

const todosSlice = createSlice({  
 name: 'todos',  
 initialState: [],  
 reducers: {  
   addTodo(state, action) {  
     const { id, text } = action.payload  
     state.push({ id, text, completed: false })  
   },  
   toggleTodo(state, action) {  
     const todo = state.find(todo => todo.id === action.payload)  
     if (todo) {  
       todo.completed = !todo.completed  
     }  
   }  
 }  
})  
   
/\* Object returned by todosSlice \*/  
{  
 name: 'todos',  
 reducer: (state, action) => newState,  
 actions: {  
   addTodo: (payload) => ({type: 'todos/addTodo', payload}),  
   toggleTodo: (payload) => ({type: 'todos/toggleTodo', payload})  
 },  
 // case reducers field omitted  
}

Let’s break this down:

* name holds the value of the string that is used as the prefix for the generated action types.
* reducer is the complete reducer function (we’ll take a closer look at this in the next exercise).
* actions holds the the auto-generated action creators.

So, what do these auto-generated action objects look like?

By default, the action creator accepts one argument, which it puts into the action object as action.payload. The action.type string is generated for us by combining the slice’s name field with the name of the case reducer function.

console.log(todosSlice.actions.addTodo('walk dog'))  
// {type: 'todos/addTodo', payload: 'walk dog'}

You’ll need to use the action creators in other files, so at a minimum you could export the entire slice object returned by createSlice().

However, we’ll use a Redux community code convention called the [“ducks” pattern](https://redux.js.org/style-guide/style-guide#structure-files-as-feature-folders-or-ducks), which suggests that we use named exports for the action creators and export them separately from the reducer.

export const { addTodo, toggleTodo } = todosSlice.actions

### Instructions

**1.**

In **favoriteRecipesSlice.js** in the code editor, print the name of the slice to the console.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Using a [for…in loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in), print out the actions in the actions object.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Export the actions. Remember to use named exports for these.

**Solution:**

**favoriteRecipeSlice.js**

import { createSlice } from '@reduxjs/toolkit';

import { selectSearchTerm } from './searchTermSlice.js';

export const favoriteRecipesSlice = createSlice({

  name: "favoriteRecipes",

  initialState: [],

  reducers: {

    addRecipe: (state, action) => {

      state.push(action.payload);

    },

    removeRecipe: (state, action) => {

      return state.filter(recipe => recipe.id !== action.payload.id)

    },

  },

});

export const selectFavoriteRecipes = (state) => state.favoriteRecipes;

export const selectFilteredFavoriteRecipes = (state) => {

  const favoriteRecipes = selectFavoriteRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return favoriteRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

/\* Begin coding below here. \*/

console.log(favoriteRecipesSlice.name)

for (const action in favoriteRecipesSlice.actions) {

  console.log(action)

}

export const {

  addRecipe, removeRecipe,

} = favoriteRecipesSlice.actions;

**Return Object - Reducers**

Let’s now take a closer look at reducer in the return object of createSlice().

const options = {  
  // options fields omitted.  
}  
const todosSlice = createSlice(options);  
   
/\* Object returned by todosSlice \*/  
{  
 name: 'todos',  
 reducer: (state, action) => newState,  
 actions: {  
   addTodo: (payload) => ({type: 'todos/addTodo', payload}),  
   toggleTodo: (payload) => ({type: 'todos/toggleTodo', payload})  
 },  
 // case reducers field omitted  
}

todosSlice.reducer is the complete reducer function, a.k.a the “slice reducer”.

When an action with the type 'todos/addTodo' is dispatched, todosSlice will execute todosSlice.reducer() to check if the dispatched action’s type matches one of todos.actions case reducers. If so, it will run the matching case reducer function and if not, it will return the current state. This is exactly the same pattern that we had previously implemented with switch/case statements!

Finally, todosSlice.reducer needs to be exported so that it can be passed to the store and be used as the todos slice of state. While the todosSlice.actions are exported as named exports, the todosSlice.reducer value is used as the default export.

export const { addTodo, toggleTodo } = todosSlice.actions;  
export default todosSlice.reducer

### Instructions

**1.**

In the code editor, print the entire object returned by createSlice(). Note how each action type corresponds to the name of a case reducer.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

At the bottom of **favoriteRecipesSlice.js**, export the reducer as the default export.

Checkpoint 3 Passed

Solution : favoriteRecipeSlice.js

import { createSlice } from '@reduxjs/toolkit';

import { selectSearchTerm } from './searchTermSlice.js';

export const favoriteRecipesSlice = createSlice({

  name: "favoriteRecipes",

  initialState: [],

  reducers: {

    addRecipe: (state, action) => {

      state.push(action.payload);

    },

    removeRecipe: (state, action) => {

      return state.filter(recipe => recipe.id !== action.payload.id)

    },

  },

});

export const selectFavoriteRecipes = (state) => state.favoriteRecipes;

export const selectFilteredFavoriteRecipes = (state) => {

  const favoriteRecipes = selectFavoriteRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return favoriteRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

export const {addRecipe, removeRecipe} = favoriteRecipesSlice.actions;

// Begin writing code here.

console.log(favoriteRecipesSlice);

export default favoriteRecipesSlice.reducer

**Converting the Store to Use `configureStore()`**

Redux Toolkit has a configureStore() method that simplifies the store setup process. configureStore() wraps around the Redux library’s createStore() method and the combineReducers() method, and handles most of the store setup for us automatically.

For example, take a look at this file which creates and exports a rootReducer…

// rootReducer.js  
   
import { combineReducers } from 'redux'  
   
import todosReducer from './features/todos/todosSlice'  
import filtersReducer from './features/filters/filtersSlice'  
   
const rootReducer = combineReducers({  
 // Define a top-level state field named `todos`, handled by `todosReducer`  
 todos: todosReducer,  
 visibilityFilter: visibilityFilterReducer  
})  
   
export default rootReducer

… and this file which creates and exports the store.

// store.js  
   
import { createStore, applyMiddleware } from 'redux'  
import thunkMiddleware from 'redux-thunk'  
import { composeWithDevTools } from 'redux-devtools-extension'  
import rootReducer from './reducer'  
   
const composedEnhancer = composeWithDevTools(applyMiddleware(thunkMiddleware))  
   
const store = createStore(rootReducer, composedEnhancer)  
export default store

Now, let’s take a look at how we can refactor these two files using configureStore(). configureStore() accepts a single configuration object parameter. The input object should have a reducer property that defines either a function to be used as the root reducer, or an object of slice reducers which will be combined to create a root reducer.

[There are many properties available in this object](https://redux-toolkit.js.org/api/configureStore), but for the purposes of this lesson, just the reducer property will be sufficient.

import { configureStore } from '@reduxjs/toolkit'  
   
import todosReducer from './features/todos/todosSlice'  
import filtersReducer from './features/filters/filtersSlice'  
   
const store = configureStore({  
 reducer: {  
   // Define a top-level state field named `todos`, handled by `todosReducer`  
   todos: todosReducer,  
   filters: filtersReducer  
 }  
})  
   
export default store

Note all the work that this one call to configureStore() does for us:

* It combines todosReducer and filtersReducer into the root reducer function, which will handle a root state that looks like {todos, filters}, removing the need to call combineReducers()
* It creates a Redux store using that root reducer, removing the need to call createStore()
* It automatically adds the thunk middleware (which you will learn about in the next lesson!)
* It automatically adds more middleware to check for common mistakes like accidentally mutating the state
* It automatically sets up the Redux DevTools Extension connection

Because of how much boilerplate code we’re able to bypass with configureStore(), we can just import the individual slice reducers straight into this file instead of creating a separate file for the root reducer and having to export/import it.

Since this is as simple as switching out the store setup code, all of the application’s existing feature code will work just fine!

Let’s confirm this in the instructions below.

### Instructions

**1.**

In the code editor, import configureStore at the top of **store.js**.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Change the ‘favoriteRecipesReducer’ import so that it’s being imported from **../features/favoriteRecipes/favoriteRecipesSlice.js**.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Rewrite the default export so that it uses configureStore() instead of createStore() to create the store with the given reducers.

When you’re done, save the file and click around the application in the browser to confirm that everything still works!

Checkpoint

**Solution: store.js**

import { createStore, combineReducers } from 'redux';

import favoriteRecipesReducer from '../features/favoriteRecipes/favoriteRecipesSlice.js';

import searchTermReducer from '../features/searchTerm/searchTermSlice.js';

import allRecipesReducer from '../features/allRecipes/allRecipesSlice.js';

import { configureStore } from '@reduxjs/toolkit';

export default configureStore({

  reducer: {

  favoriteRecipes: favoriteRecipesReducer,

  searchTerm: searchTermReducer,

  allRecipes: allRecipesReducer

  }

});

**Review**

* **R**edux **T**ool**k**it (RTK) contains packages and functions that build in suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.
* RTK has a createSlice() function that will help us simplify our Redux reducer logic and actions.
* createSlice() has one parameter, options. In this lesson, we covered three of option‘s properties: name, initialState, and reducers. options has more properties which will be covered in the next lessons.
* A case reducer is a method that can update the state, and will be executed when the corresponding action type is dispatched. This is similar to a case in a switch statement.
* You can write code that “mutates” the state inside the case reducers passed to createSlice(), and Immer will safely and accurately return an immutably updated state.
* createSlice() returns an object with the following properties: name, reducer, actions, and caseReducers.
* createSlice() has one parameter, options. In this lesson, we covered the option properties: name, initialState, and reducers. options has more properties which will be covered in the next lessons.
* We typically use a Redux community code convention called the “ducks” pattern when exporting the action creators and the reducer.
* RTK has a configureStore() function that simplifies the store setup process. configureStore() wraps around the Redux core createStore() function and the combineReducers() function, and handles most of the store setup for us automatically.

**REDUX MIDDLEWARE**

Middleware in Redux: One of the ways we can customize Redux is by adding middleware. middleware is the code that runs in the middle—usually between a framework receiving a request and producing a response. Middleware is a powerful tool for extending, modifying, or customizing a framework or library’s default behavior to meet an application’s specific needs.

In Redux, middleware runs between the moment when an action is dispatched and the moment when that action is passed along to the reducer. Middleware intercepts actions after they are dispatched and before they are passed along to the reducer. Some common tasks that middleware perform include logging, caching, adding auth tokens to request headers, crash reporting, routing, and making asynchronous requests for data. You can add any of these functionalities to your apps by using popular open-source middleware.

**Write Your Own Middleware**

To add a middleware to our project, we use Redux’s applyMiddleware function like so.

import { createStore, applyMiddleware } from 'redux';  
import { middleware1, middleware2, middleware3 } from './exampleMiddlewares';  
import { exampleReducer } from './exampleReducer';  
import { initialState} from './initialState';  
   
const store = createStore(  
  exampleReducer,   
  initialState,   
  applyMiddleware(  
    middleware1,   
    middleware2,   
    middleware3  
  )  
);

Middlewares must conform to a specific, nested function structure in order to work as part of the pipeline (this nested structure is also called a [higher-order function](https://eloquentjavascript.net/05_higher_order.html), if you’d like to read more). That structure looks like this:

const exampleMiddleware = storeAPI => next => action => {  
  // do stuff here  
  return next(action);  // pass the action on to the next middleware in the pipeline  
}

Each middleware has access to the storeAPI (which consists of the dispatch and getState functions), as well as the next middleware in the pipeline, and the action that is to be dispatched.

### Instructions

**1.**

In the code editor, you’ll notice we’ve created a simple reducer for you, and taken care of importing Redux’s createStore and applyMiddleware functions. We’ve created a store by calling createStore and passing it the reducer. Since all Redux middleware have the same basic structure, you can start by copying this snippet:

const logger = storeAPI => next => action => {  
  // do stuff here  
   
  return next(action);  
};

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Replace the comment // do stuff here, with a line of code that logs the contents of the store to the console. Remember, you can access the store’s state with storeAPI.getState().

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Instead of returning next(action), store the result of that function call in a const called nextState. Next, log nextState to the console. Finally, return nextState.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Apply your custom middleware to your store by adding a third argument to the call to createStore. This argument should be the result of calling applyMiddleware with the logger middleware you’ve written.

Checkpoint 5 Passed

Stuck? Get a hint

**5.**

Dispatch the following action to your store:

{  
  type: 'NEW\_MESSAGE',   
  payload: 'I WROTE A MIDDLEWARE'  
}

Note that the store’s new state was logged to the console. Congrats – you just wrote your first middleware!

Checkpoint 6 Passed

**SOLUTION**

import { createStore, applyMiddleware } from 'redux';

const messageReducer = (state = '', action) => {

  if (action.type === 'NEW\_MESSAGE') {

    return action.payload;

  } else {

    return state;

  }

}

const logger = storeAPI => next => action => {

    console.log(storeAPI.getState())

    const nextState = next(action);

    console.log(nextState)

    return nextState;  // pass the action on to the next middleware in the pipeline

};

const store = createStore(messageReducer, "", applyMiddleware(logger));

store.dispatch({

  type: "NEW\_MESSAGE",

  payload: "I WROTE A MIDDLEWARE"

})

**Introduction to Thunks**

One of the most flexible and popular ways to add asynchronous functionality to Redux involves using thunks. A thunk is a higher-order function that wraps computation we want to perform later. For example, this add function returns a thunk that will perform x+y when called.

const add = (x,y) => {  
  return () => {  
    return x + y;   
  }   
}

Thunks are helpful because they allow us to bundle up bits of computation we want to delay into packages that can be passed around in code. Consider these two function calls, which rely on the add function above:

const delayedAddition = add(2,2)  
delayedAddition() // => 4

Note that calling add does not cause the addition to happen – it merely returns a function that will perform the addition when called. To perform the addition, we must called delayedAddition.

### Instructions

**1.**

Consider the function remindMeTo, which we’ve defined for you in the code editor.

What do you think will happen if you run remindMeTo('call mom')? Call console.log(remindMeTo('call mom')) in the code editor to test your suspicion.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Logging remindMeTo('call mom') caused “Remember to call mom!!!” to appear in the console. Now write a function, remindMeLater, that takes a string, task, and returns a thunk that returns the result of calling remindMeTo with the argument task.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Call remindMeLater with a task you need to complete later and store the result in a variable reminder.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

What do you think will happen when you call reminder? Test your hunch by calling reminder in your code editor and logging the result to the console.

Checkpoint

**SOLUTION:**

const remindMeTo = task => {

  return `Remember to ${task}!!!`

}

console.log(remindMeTo('call mom'))

const remindMeLater = task => {

  return () => {

    return remindMeTo(task)

  }

}

const reminder = remindMeLater('buy groceries')

console.log(reminder())

**`redux-thunk`**

redux-thunk is a middleware that lets you do exactly that. redux-thunk makes it simple for you to write asynchronous logic that interacts with the store by allowing you to write action creators that return thunks instead of objects. These thunks can perform asynchronous operations, and per [the redux-thunk documentation](https://github.com/reduxjs/redux-thunk#motivation), “can be used to delay dispatching an action” (for example, until after an API response is received), or “to dispatch an action only if certain conditions are met”.

For example, imagine we’ve written a simple counter whose reducer contains a single value, which is updated by a single reducer. Without redux-thunk we are limited to writing synchronous action creators like this one:

const increment = () => {  
  return {  
    type: 'counter/increment',  
  }  
}

When we call dispatch(increment()), the value in our store immediately increases. With redux-thunk, we can extend our counter app to accommodate asynchronous action creators, like asyncIncrement, in addition to synchronous ones.

const incrementLater = async () => {  
  setTimeout(() => {  
    dispatch(increment())      
  }, 1000)      
};  
   
const asyncIncrement = () => {  
  return incrementLater;  
}

redux-thunk is such a popular solution for handling asynchronous logic that it is included in Redux Toolkit. It also exists as a standalone package, but you won’t need to install redux-thunk separately if you use Redux Toolkit. This is because Redux Toolkit’s configureStore function, which you learned about in a previous lesson, will apply redux-thunk to the store by default.

### Instructions

**1.**

Import the configureStore method from the @reduxjs/toolkit module.

Checkpoint

**SOLUTION:**

import { configureStore } from '@reduxjs/toolkit';

**Writing Thunks in Redux**

redux-thunk allows us to write action creators that return thunks, within which we can perform any asynchronous operations we like. Consider the following asynchronous action creator:

import { fetchUser } from './api'  
const getUser = (id) => {  
  return async (dispatch, getState) => {  
    const payload = await fetchUser(id);  
    dispatch({type: 'users/addUser', payload: payload});  
  }  
}

getUser has two key parts: the synchronous outer function (otherwise known as the thunk action creator) which returns the inner, asynchronous thunk. The thunk receives dispatch and getState as arguments, and dispatches a synchronous action after the asynchronous operation (fetchUser) completes.

To get the user with id = 32, we can call dispatch(getUser(32)). Note that the argument to dispatch is not an object, but an asynchronous function that will first fetch the user’s data and then dispatch a synchronous action once the user’s information has been retrieved.

### Instructions

**1.**

In your code editor, we’ve imported the function fetchRecipes, which makes an asynchronous request to fetch all the recipes to be displayed in our familiar app. Write a thunk action creator called loadRecipes that asynchronously fetches the recipes and dispatches a synchronous action with type = allRecipes/addRecipes and payload equal to the payload you get when the asynchronous request completes.

Checkpoint

**Solution: allRecipesSlice.js**

import { fetchRecipes } from '../../app/api'

import { createSlice } from "@reduxjs/toolkit";

// TO DO: write loadRecipes here!

const loadRecipes = () => {

  return async (dispatch) => {

    const recipes = await fetchRecipes()

    dispatch({type: 'allRecipes/addRecipes', payload: recipes})

  }

};

export const allRecipesSlice = createSlice({

  name: "allRecipes",

  initialState: {

    recipes: [],

    isLoading: false,

    hasError: false,

  },

  reducers: {

    addRecipes(state, action) {

      state.recipes = action.payload

    }

  },

});

export default allRecipesSlice.reducer;

**`redux-thunk` Source Code**

### Instructions

Make sure you understand how redux-thunk works by reviewing the redux-thunk source code. The key step happens on line 3, where the middleware checks whether or not the action is a function.

If the action is a function (as happens when we dispatch thunks returned by thunk action creators), then redux-thunk invokes that function. If the action is not a function (as happens when we dispatch plain objects), redux-thunk passes it through to the next step in the middleware pipeline.

**Solution:**

function createThunkMiddleware(extraArgument) {

  return ({ dispatch, getState }) => (next) => (action) => {

    if (typeof action === 'function') {

      return action(dispatch, getState, extraArgument);

    }

    return next(action);

  };

}

const thunk = createThunkMiddleware();

thunk.withExtraArgument = createThunkMiddleware;

export default thunk;

Conclusion

In this lesson you:

* Learned about Redux middleware and wrote your own simple logging middleware
* Encountered thunks and learned about how valuable thunks are for deferring computation

const remindMeLater = task => {   
  return () => {  
    remindMeTo(task)  
  }   
}

* Discovered redux-thunk, a middleware that allows you to write asynchronous action creators that return thunks instead of objects
* Automatically enabled redux-thunk by using configureStore

import { configureStore } from '@reduxjs/toolkit;

* Took a deep dive into the middleware’s source code, in order to understand how the middleware actually works
* Wrote your own asynchronous action creators in the format that redux-thunk expects

const getUser = (id) => {  
  return async (dispatch, getState) => {  
    const payload = await fetchUser(id)  
    dispatch({type: 'users/addUser', payload: payload})  
  }  
}

If you’d like to learn more, you can read the [redux-thunk documentation](https://github.com/reduxjs/redux-thunk) and [visit the Redux Toolkit site to see how configureStore includes redux-thunk by default](https://redux-toolkit.js.org/api/getDefaultMiddleware#getdefaultmiddleware).

**MANAGING PROMISE LIFECYCLE ACTIONS**

Promise Lifecycle Actions

It is common to dispatch a “pending” action right before performing an asynchronous operation, and “fulfilled” or “rejected” actions depending on the results of the completed operation. Take this simple thunk action creator, fetchUserById.

import { fetchUser } from './api';  
   
const fetchUserById = (id) => {  
  return async (dispatch, getState) => {  
    const payload = await fetchUser(id);  
    dispatch({type: 'users/addUser', payload: payload});  
  }  
}

Rewritten to include pending and rejected actions, it might look like this:

import { fetchUser } from './api'  
const fetchUserById = (id) => {  
  return async (dispatch, getState) => {  
    dispatch({type: 'users/requestPending'})  
    try {  
      const payload = await fetchUser(id)  
      dispatch({type: 'users/addUser', payload: payload})  
    } catch(err) {  
      dispatch({type: 'users/error', payload: err})  
    }  
  }  
}

We call these pending/fulfilled/rejected actions *promise lifecycle actions*. This pattern is so common that Redux Toolkit provides a neat abstraction, createAsyncThunk, for including promise lifecycle actions in your Redux apps.

Success

Data

Rejected

Failure

Pending

Fullfilled

Promise

Error

**createAsyncThunk**()

createAsyncThunk is a function with two parameters—an action type string and an asynchronous callback—that generates a thunk action creator that will run the provided callback and automatically dispatch promise lifecycle actions as appropriate so that you don’t have to dispatch pending/fulfilled/rejected actions by hand.

To use createAsyncThunk, you’ll first need to import it from Redux Toolkit like so:

import { createAsyncThunk } from '@reduxjs/toolkit';

Next, you’ll need to call createAsyncThunk, passing two arguments. The first is a string representing the asynchronous action’s type. Conventionally, type strings take the form "resourceType/actionName". In this case, since we are getting an individual user by their id, our action type will be users/fetchUserById. The second argument to createAsyncThunk is the payload creator: an asynchronous function that returns a promise resolving to the result of an asynchronous operation. Here is fetchUserById rewritten using createAsyncThunk:

import { createAsyncThunk } from '@reduxjs/toolkit'  
import { fetchUser } from './api'  
const fetchUserById = createAsyncThunk(  
  'users/fetchUserById', // action type  
  async (arg, thunkAPI) => { // payload creator  
    const response = await fetchUser(arg);  
    return response.json();  
  }  
)

There are a few things worth highlighting here. First, observe that the payload creator receives two arguments—arg and thunkAPI. We will elaborate on those in the next exercise. Second, note that the payload creator we provided doesn’t dispatch any actions at all. It just returns the result of an asynchronous operation.

As you can see, createAsyncThunk makes defining thunk action creators more concise. All you have to write is an asynchronous thunk function; createAsyncThunk takes care of the rest, returning an action creator that will dispatch pending/fulfilled/rejected actions as appropriate.

### Instructions

**1.**

In the code editor, we’ve provided loadRecipes, the asynchronous action creator you wrote in the last lesson. Now we’re going to refactor it using createAsyncThunk. To start, import createAsyncThunk from Redux toolkit (make sure you continue to import createSlice as well).

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Refactor loadRecipes using createAsyncThunk. Remember, createAsyncThunk takes two arguments: an action type string, and a payload creator function. Your action type string should be 'allRecipes/loadRecipes'. Your payload creator should retrieve the recipes by calling fetchRecipes, which we’ve imported for you. Once the recipes are fetched, you should return their json data, which you can access by calling .json() on the response to your call to fetchRecipes. Note: .json() is asynchronous, so you’ll want to await the result of that call.

Checkpoint 3 Passed

**Solution:**

import { fetchRecipes } from '../../app/api'

import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

const loadRecipes = createAsyncThunk(

  'allRecipes/loadRecipes',

  async (arg, thunkAPI) => {

    const data = await fetchRecipes()

    const json = await data.json()

    return json

  }

)

export const allRecipesSlice = createSlice({

  name: 'allRecipes',

  initialState: {

    recipes: [],

    isLoading: false,

    hasError: false,

  },

  reducers: {

    addRecipes(state, action) {

      state.recipes = action.payload

    }

  },

});

export default allRecipesSlice.reducer;

**Passing Arguments to Thunks**

If the thunk action creator is called searchUsers, we would call it like this: searchUsers({firstName: 'Ada', lastName: 'Lovelace'}).

If you need to access these variables individually, you can use ES6 destructuring assignment to unpack the object when you declare the payload creator and pass it to createAsyncThunk, like this :

const searchUsers = createAsyncThunk(  
    'users/searchUsers',  
    async ({ firstName, lastName}, thunkAPI) => {  
        // perform the asynchronous search request here      
    }  
)

If your thunk requires no arguments, you can just call your thunk action creator without, and the arg argument will be undefined. In the event the thunk requires only one param (for example, fetching a specific resource by id) you should name that first param semantically. Here’s the fetchUserById example from the last exercise, with the arg parameter semantically renamed to userId.

import { createAsyncThunk } from '@reduxjs/toolkit'  
import { fetchUser } from './api'  
const fetchUserById = createAsyncThunk(  
    'users/fetchUserById', // action type  
    async (userId, thunkAPI) => { // payload creator  
        const response = await fetchUser(userId)  
        return response.data  
    }  
)

The payload creator’s second argument, thunkAPI, is an object containing several useful methods, including the store’s dispatch and getState. [For an exhaustive list of methods available in the thunkAPI object, you can read the documentation](https://redux-toolkit.js.org/api/createAsyncThunk#payloadcreator).

### Instructions

**1.**

In the code editor, we’ve defined a thunk action creator searchRecipesByName. Rename arg to the semantically appropriate variable name recipeName.

**Solution:**

import { createAsyncThunk } from "@reduxjs/toolkit"

import { searchRecipes } from './api'

const searchRecipesByName = createAsyncThunk(

  'recipes/searchRecipesByName',

  (recipeName, thunkAPI) => {

    const response = await searchRecipes(recipeName)

    return response.data

  }

)

**Actions Generated by createAsyncThunk**

As you know, createAsyncThunk takes care of dispatching actions for each of the promise lifecycle states: pending, fulfilled, and rejected. But what exactly do these actions look like?

Building off the action type string you pass to it, createAsyncThunk produces an action type for each promise lifecycle states. If you pass the action type string 'resourceType/actionType' to createAsyncThunk, it will produce these three action types:

* 'resourceType/actionType/pending'
* 'resourceType/actionType/fulfilled'
* 'resourceType/actionType/rejected'

To use our earlier example:

import { createAsyncThunk } from '@reduxjs/toolkit'  
import { fetchUser } from './api'  
   
const fetchUserById = createAsyncThunk(  
  'users/fetchUserById', // action type  
  async (userId, thunkAPI) => { // payload creator  
    const response = await fetchUser(userId)  
    return response.data  
  }  
)

When you pass createAsyncThunk the action type string 'users/fetchUserById', createAsyncThunk producers these three actions types:

* 'users/fetchUserById/pending'
* 'users/fetchUserById/fulfilled'
* 'users/fetchUserById/rejected'

If you need to access the individual pending/fulfilled/rejected action creators, you can reference them like this:

* fetchUserById.pending
* fetchUserById.fulfilled
* fetchUserById.rejected

You will have to handle these action types in your reducers if you want to reflect these promise lifecycle states in your app. In the next exercise, we will show you how to do that.

### Instructions

**1.**

In the code editor, we’ve used createAsyncThunk to define a thunk action creator, loadRecipes. What three action type strings are generated by the call to createAsyncThunk? Write out the three strings in your code editor in the comments below the call to loadRecipes.

Checkpoint 2 Passed

**solution: allRecipeSlice.js**

import { fetchRecipes } from '../../app/api'

import { createAsyncThunk } from "@reduxjs/toolkit";

const loadRecipes = createAsyncThunk(

  'allRecipes/loadRecipes',

  (arg, thunkAPI) => {

    const response = await fetchRecipes();

    return response.data

  }

)

// The above call to createAsyncThunk will generate what three action types?

// 1. allRecipes/loadRecipes/pending

// 2. allRecipes/loadRecipes/fulfilled

// 3. allRecipes/loadRecipes/rejected

**Using createSlice() with Async Action Creators**

createSlice accepts a single argument, options, which is an object containing configuration parameters including a name, some initial state, and reducers. createSlice then uses these configuration parameters to generate a slice of the store, including action creators and action types for updating the state contained in that slice. Consider the following example:

const usersSlice = createSlice({  
  name: 'users',  
  initialState: { users:  [] },  
  reducers: {  
    addUser: (state, action) => {   
      state.users.push(action.payload)   
    }          
  },  
})

This call to createSlice, generates a slice of the store that responds to the action creator usersSlice.actions.addUser. But what if we’ve generated our action creators via calls to createAsyncThunk? Consider fetchUserById, the asynchronous action creator from earlier in this lesson:

const fetchUserById = createAsyncThunk(  
  'users/fetchUserById', // action type  
  async (userId, thunkAPI) => { // payload creator  
    const response = await fetchUser(arg)  
    return response.data  
  }  
)

This asynchronous action creator will generate three action types: 'users/fetchUserById/pending', 'users/fetchUserById/fulfilled', and 'users/fetchUserById/rejected'. Currently, these action types have no effect on our users slice, which only responds to the users/addUser action type generated by createSlice. How can we account for these promise lifecycle action types in our user slice? This is exactly the problem that extraReducers, an optional property on the configuration object passed to createSlice, was designed to solve.

extraReducers allows createSlice to respond to action types generated elsewhere. To make the users slice respond to promise lifecycle action types, we pass them to createSlice in the extraReducers property. Open usersSlice.js in your code editor to see an example of the extraReducers property in context.

Note that in addition to using the extraReducers property, we also added some extra fields to our state object: a boolean, isLoading, which will be true when a request is pending, and otherwise false, and a boolean hasError, which we will set to true if our request to fetch a user is rejected. These additions will allow us to simply track promise lifecycle states so that we can create satisfying and informative user interfaces.

### Instructions

**1.**

In allRecipesSlice.js, we’ve used createAsyncThunk to define loadRecipes, an asynchronous action creator that fetches all our app’s recipes, and createSlice to define a slice of recipes in our app’s store.

Add two booleans — isLoading and hasError — to the initialState property passed to createSlice. What should their initial values be?

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Using the extraReducers property, add reducers for each of the promise lifecycle action types generated by createAsyncThunk.

What about the app’s behavior has changed? While the recipes are being fetched, the app displays a loading spinner. And if the recipes fail to fetch, the app displays an error message.

Why does the app behave differently when you pass extra Reducers to createSlice? Adding the extra reducers to the recipes slice causes the store to update in response to each of the pending/fulfilled/rejected actions dispatched by loadRecipes. These changes are reflected in the app’s UI.

**Solution: allRecipeSlice.js**

import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";

import {

  addFavoriteRecipe,

  removeFavoriteRecipe,

} from "../favoriteRecipes/favoriteRecipesSlice";

import { selectSearchTerm } from "../search/searchSlice";

export const loadRecipes = createAsyncThunk(

  "allRecipes/getAllRecipes",

  async () => {

    const data = await fetch("api/recipes?limit=10");

    const json = await data.json();

    return json;

  }

);

const sliceOptions = {

  name: "allRecipes",

  initialState: {

    recipes: [],

    isLoading: false,

    hasError: false

  },

  reducers: {},

  extraReducers: {

    [loadRecipes.pending]: (state, action) => {

      state.isLoading = true;

      state.hasError = false;

    },

    [loadRecipes.fulfilled]: (state, action) => {

      state.recipes = action.payload;

      state.isLoading = false;

      state.hasError = false;

    },

    [loadRecipes.rejected]: (state, action) => {

      state.isLoading = false;

      state.hasError = true;

    }

  }

}

export const allRecipesSlice = createSlice(sliceOptions);

export const selectAllRecipes = (state) => state.allRecipes.recipes;

export const selectFilteredAllRecipes = (state) => {

  const allRecipes = selectAllRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return allRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

export default allRecipesSlice.reducer;

**userSlice.js**

const fetchUserById = createAsyncThunk(

  'users/fetchUserById',

  async (userId) => {

    const users = await fetch(`api/users${userId}`)

    const data = await users.json()

    return data

  }

)

const usersSlice = createSlice({

  name: 'users',

  initialState: {

    users:  [],

    isLoading: false,

    hasError: false

  },

  reducers: {

    addUser: (state, action) => {

      state.users.push(action.payload)

    }

  },

},

extraReducers: {

  [fetchUserById.pending]: (state, action) => ({

    state.isLoading = true;

    state.hasError = false;

   }),

  [fetchUserById.fulfilled]: (state, action) => ({

    state.users.push(action.payload);

    state.isLoading = false;

    state.hasError = false;

  }),

  [fetchUserById.rejected]: (state, action) => ({

    state.isLoading = false;

    state.hasError = true;

  })

})

Conclusion

* Learned the three *promise lifecycle actions*: pending, fulfilled, and rejected
* Learned how to use createAsyncThunk, which abstracts the process of handling promise lifecycle states according to best practices/common design paradigms
* Imported createAsyncThunk from the Redux Toolkit:

import { createAsyncThunk } from '@reduxjs/toolkit';

* Refactored existing asynchronous action creators using createAsyncThunk.
* Made your reducers respond to pending/fulfilled/rejected promise lifecycle actions by supplying the extraReducers property to createSlice.

**olecipesSlice.js**

import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";

import {

  addFavoriteRecipe,

  removeFavoriteRecipe,

} from "../favoriteRecipes/favoriteRecipesSlice";

import { selectSearchTerm } from "../search/searchSlice";

// createAsyncThunk simplifies our Redux app by returning an action creator that dispatches promise lifecycle actions for us so we don't have to dispatch them ourselves.

export const loadRecipes = createAsyncThunk(

  "allRecipes/getAllRecipes",

  async () => {

    const data = await fetch("api/recipes?limit=10");

    const json = await data.json();

    return json;

  }

);

const sliceOptions = {

  name: "allRecipes",

  initialState: {

    recipes: [],

    isLoading: false,

    hasError: false

  },

  reducers: {},

  extraReducers: {

    [loadRecipes.pending]: (state, action) => {

      state.isLoading = true;

      state.hasError = false;

    },

    [loadRecipes.fulfilled]: (state, action) => {

      state.recipes = action.payload;

      state.isLoading = false;

      state.hasError = false;

    },

    [loadRecipes.rejected]: (state, action) => {

      state.isLoading = false;

      state.hasError = true;

    }

  }

}

export const allRecipesSlice = createSlice(sliceOptions);

export const selectAllRecipes = (state) => state.allRecipes.recipes;

export const selectFilteredAllRecipes = (state) => {

  const allRecipes = selectAllRecipes(state);

  const searchTerm = selectSearchTerm(state);

  return allRecipes.filter((recipe) =>

    recipe.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

};

export default allRecipesSlice.reducer;

**oldAllRecipeSlice.js**

// Without createAsyncThunk, we have to dispatch pending/fulfilled/rejected actions ourself.

export const loadRecipes = () => {

  return async (dispatch, getState) => {

    dispatch({type: "allRecipes/getAllRecipes/pending"})

    try {

      const data = await fetch("api/recipes?limit=10");

      const json = await data.json();

      dispatch({type: "allRecipes/getAllRecipes/fulfilled", payload: json})

    } catch (err) {

      dispatch({type: "allRecipes/getAllRecipes/rejected", payload: err})

    }

  }

}

Which Redux component can you skip writing tests for?

* Reducers
* Selectors
* Thunks

Yes! Tests should be written for all three components to make sure no regressions go unnoticed. ALL of these are neccesary)

What is code coverage used for?

Correct! Code coverage is useful to see what additional tests we should write to make sure everything is covered. Code coverage used as a tool to see which line of code are covered vy tests.

**Chapter: 22 GitHub Part II**

**GIT BRANCHING**

git branch

Up to this point, you’ve worked in a single Git branch called master. Git allows us to create *branches* to experiment with versions of a project. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the master branch until you’re ready to merge the happy ending to the master branch. You can use the command below to answer the question: “which branch am I on?”

git branch

Check what branch you are currently on.

In the output, the \* (asterisk) is showing you what branch you’re on. The project only has one branch at this time.

Solution:

$git branch

Fencing

\*master

**branching overview**

The diagram to the right illustrates branching.

* The circles are commits, and together form the Git project’s commit history.
* *New Branch* is a different *version* of the Git project. It contains commits from *Master* but also has commits that *Master* does not have.

New branch

**master**

**Git branching**

**Git branch 2:**

Right now, the Git project has only one branch: master.

To create a new branch, use:

git branch new\_branch

Here new\_branch would be the name of the new branch you create, like photos or blurb. Be sure to name your branch something that describes the purpose of the branch. Also, branch names can’t contain whitespaces: new-branch and new\_branch are valid branch names, but new branch is not.

### Instructions

**1.**

Let’s create a new version of a resumé to apply for a fencing instructor role.

Create a new branch called fencing.

Remember to spell the word “fencing” correctly.

Next, view your branches as you did in the previous exercise.

Notice in the output there now appear two branches: master and fencing.

Checkpoint

**Solution:**

**$git branch fencing**

**git checkout**

Great! You just created a new branch.

The master and fencing branches are identical: they share the same exact commit history. You can switch to the new branch with

git checkout branch\_name

Here, branch\_name is the name of the branch. If the branch’s name is skill

git checkout skill

Once you switch branches, you will now be able to make commits on the branch that have no impact on master.

You can continue your workflow, while master stays intact!

### Instructions

**1.**

Switch to the fencing branch from the master branch.

Checkpoint 2 Passed

**2.**

Use git branch to verify that you have switched branches.

In the output, notice the \* is now over the fencing branch.

**git checkout fencing**

**3. commit on a new branch**

Congratulations! You have switched to a new branch. All the commands you do on master, you can also do on this branch.

For example, to add files to the staging area, use:

git add filename

And to commit, use:

git commit -m "Commit message"

In a moment, you will make a commit on the fencing branch. On the far right, the diagram shows what will happen to the Git project.

### Instructions

**1.**

Print the Git commit log.

Notice the output:

* The commits you see were all made in the master branch. fencing inherited them.
* This means that every commit master has, fencing also has.

Note: if you find that your cursor is stuck in Git log, press q to escape.

Checkpoint 2 Passed

**2.**

In **resume.txt**, replace your skill at scheming against Hook with your experience in sword-fights.

Delete this line:

-Scheme against Captain Hook

and type this line in its place:

-Engage in swordfights with pirates

Checkpoint 3 Passed

**3.**

Add **resume.txt** into the staging area.

Checkpoint 4 Passed

**4.**

Commit the changes to the repository with a commit message.

**4. git merge**

What if you wanted include all the changes made to the fencing branch on the master branch? We can easily accomplish this by merging the branch into master with:

git merge branch\_name

For example, if I wanted to merge the skills branch to master, I would enter

git merge skills

In a moment, you’ll merge branches. Keep in mind:

* Your goal is to update master with changes you made to fencing.
* fencing is the giver branch, since it provides the changes.
* master is the receiver branch, since it accepts those changes.

### Instructions

**1.**

You are currently on the fencing branch. Switch over to the master branch.

Checkpoint 2 Passed

**2.**

Your sword-fighting experience is so impressive that it belongs on the master version of your resumé.

From the terminal, merge the fencing branch into the master branch.

Notice the output: The merge is a “fast forward” because Git recognizes that fencing contains the most recent commit. Git fast forwards master to be up to date with fencing.

**git merge fencing**

**5. merge conflict I**

The merge was successful because master had not changed since we made a commit on fencing. Git knew to simply update master with changes on fencing.

What would happen if you made a commit on master *before* you merged the two branches? Furthermore, what if the commit you made on master altered the same exact text you worked on in fencing? When you switch back to master and ask Git to merge the two branches, Git doesn’t know which changes you want to keep. This is called a *merge conflict*.

**6. merge conflict II**

Let’s say you decide you’d like to merge the changes from fencing into master.

Here’s where the trouble begins!

You’ve made commits on separate branches that alter the same line in conflicting ways. Now, when you try to merge fencing into master, Git will not know which version of the file to keep.

**7. delete branch**

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the master branch. After the branch has been integrated into master, it has served its purpose and can be deleted.

The command

git branch -d branch\_name

will delete the specified branch from your Git project.

Now that master contains all the file changes that were in fencing, let’s delete fencing.

8. **generalizations**

Let’s take a moment to review the main concepts and commands from the lesson before moving on.

* Git *branching* allows users to experiment with different versions of a project by checking out separate *branches* to work on.

The following commands are useful in the Git branch workflow.

* git branch: Lists all a Git project’s branches.
* git branch branch\_name: Creates a new branch.
* git checkout branch\_name: Used to switch from one branch to another.
* git merge branch\_name: Used to join file changes from one branch to another.
* git branch -d branch\_name: Deletes the branch specified.

**Git teamwork**

**git clone**

Sally has created the remote repository, **science-quizzes** in the directory **curriculum**, which teachers on the school’s shared network have access to. In order to get your own replica of **science-quizzes**, you’ll need to *clone* it with:

git clone remote\_location clone\_name

In this command:

* remote\_location tells Git where to go to find the remote. This could be a web address, or a filepath, such as:

/Users/teachers/Documents/some-remote

* clone\_name is the name you give to the directory in which Git will clone the repository.

**git remote -v**

* Nice work! We have a clone of Sally’s remote on our computer. One thing that Git does behind the scenes when you clone **science-quizzes** is give the remote address the name *origin*, so that you can refer to it more conveniently. In this case, Sally’s remote is *origin*.
* You can see a list of a Git project’s remotes with the command:
* git remote -v

**git fetch**

* After you cloned **science-quizzes**, you had to run off to teach a class. Now that you’re back at your computer, there’s a problem: what if, while you were teaching, Sally changed the **science-quizzes** Git project in some way. If so, your clone will no longer be up-to-date.
* An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with:

git fetch

* This command will not *merge* changes from the remote into your local repository. It brings those changes onto what’s called a *remote branch*. Learn more about how this works below.

**git merge**

* Even though Sally’s new commits have been fetched to your local copy of the Git project, those commits are on the origin/master branch. Your *local* master branch has not been updated yet, so you can’t view or make changes to any of the work she has added.
* In *Lesson III, Git Branching* we learned how to merge branches. Now we’ll use the git merge command to integrate origin/master into your local master branch. The command:
* git merge origin/master
* will accomplish this for us.

**Git workflow**

Now that you’ve merged origin/master into your local master branch, you’re ready to contribute some work of your own. The workflow for Git collaborations typically follows this order:

1. Fetch and merge changes from the remote
2. Create a branch to work on a new project feature
3. Develop the feature on your branch and commit your work
4. Fetch and merge from the remote again (in case new commits were made while you were working)
5. *Push* your branch up to the remote for review

Steps 1 and 4 are a safeguard against *merge conflicts*, which occur when two branches contain file changes that cannot be merged with the git merge command. Step 5 involves git push, a command you will learn in the next exercise.

**git push**

Now it’s time to share our work with Sally.

The command:

git push origin your\_branch\_name

will push your branch up to the remote, origin. From there, Sally can review your branch and merge your work into the master branch, making it part of the definitive project version.

**generalizations**

Congratulations, you now know enough to start collaborating on Git projects! Let’s review.

* A *remote* is a Git repository that lives *outside* your Git project folder. Remotes can live on the web, on a shared network or even in a separate folder on your local computer.
* The *Git Collaborative Workflow* are steps that enable smooth project development when multiple collaborators are working on the same Git project.

We also learned the following commands

* git clone: Creates a local copy of a remote.
* git remote -v: Lists a Git project’s remotes.
* git fetch: Fetches work from the remote into the local copy.
* git merge origin/master: Merges origin/master into your local branch.
* git push origin <branch\_name>: Pushes a local branch to the origin remote.

Git projects are usually managed on Github, a website that hosts Git projects for millions of users. With Github you can access your projects from anywhere in the world by using the basic workflow you learned here.

Chapter 24:

Basics of back-ends

Review: In order to deliver the front-end of a website or web application to a user, a lot needs to happen behind the scenes on the back-end! Understanding what makes up the back-end can be overwhelming because the back-end has a lot of different parts, and different websites or web applications can have dramatically different back-ends. We covered a lot in this lesson, so let’s review what we learned:

* The front-end of a website or application consists of the HTML, CSS, JavaScript, and static assets sent to a client, like a web browser.
* A web server is a process running on a computer somewhere that listens for incoming requests for information over the internet and sends back responses.
* Storing, accessing, and manipulating data is a large part of a web application’s back-end
* Data is stored in databases which can be relational databases or NoSQL databases.
* The server-side of a web application, sometimes called the application server, handles important tasks such as authorization and authentication.
* The back-end of web application often has a web API which is a way of interacting with an application’s data through HTTP requests and responses.
* Together the technologies used to build the front-end and back-end of a web application are known as the stack, and many different languages and frameworks can be used to build a robust back-end.

Now that you have a sense for server-side web development and what the back-end is, you’re ready to dive in and learn about the different parts in more depth! order to deliver the frond of a website or web application to a user, a l

Chapter: 25

Node

To see the version of node type: node -v

**LEARN NODE.JS**

The Node REPL

[REPL](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop) is an abbreviation for **r**ead–**e**val–**p**rint **l**oop. It’s a program that **l**oops, or repeatedly cycles, through three different states: a **r**ead state where the program **r**eads input from a user, the **e**val state where the program **e**valuates the user’s input, and the **p**rint state where the program **p**rints out its evaluation to a console. Then it **l**oops through these states again.

When you install Node, it comes with a built-in JavaScript REPL. You can access the REPL by typing the command node (with nothing after it) into the terminal and hitting enter. A > character will show up in the terminal indicating the REPL is running and prompting your input. The Node REPL will evaluate your input line by line.

If you’d like to type multiple lines and then have them evaluated at once you can type .editor while in the REPL. Once in “editor” mode, you can type control + d when you’re ready for the input to be evaluated. Each session of the REPL has a single shared memory; you can access any variables or functions you define until you exit the REPL.

A REPL can be extremely useful for performing calculations, learning a language, and developing code.

**Running a Program with Node**

We’ll need to create a file with a .js extension. We’ll call ours **myProgram.js**. Next, we’ll open that file with a text editor and add our code:

// Inside myProgram.js  
console.log('Hello World');

Our code is complete! Now, we want to execute it. We’ll open our terminal and navigate to the directory that contains **myProgram.js**. Finally, we’ll type the command node myProgram.js into our terminal.

$ node myProgram.js

The results of our program will print to the terminal.

Hello World

**Accessing the Process Object**

Node has a global process object with useful methods and information about the current process.

The process.env property is an object which stores and controls information about the environment in which the process is currently running. For example, the process.env object contains a PWD property which holds a string with the directory in which the current process is located. It can be useful to have some if/else logic in a program depending on the current environment. We could store this information on the process.env. One convention is to add a property to process.env with the key NODE\_ENV and a value of either production or development.

if (process.env.NODE\_ENV === 'development'){  
  console.log('Testing! Testing! Does everything work?');  
}

The process.memoryUsage() returns information on the CPU demands of the current process. It returns a property that looks similar to this:

{ rss: 26247168,  
  heapTotal: 5767168,  
  heapUsed: 3573032,  
  external: 8772 }

Heap can mean different things in different contexts: a heap can refer to [a specific data structure](https://en.wikipedia.org/wiki/Heap_(data_structure)), but it can also refer to the a block of [computer memory](https://en.wikipedia.org/wiki/Memory_management). process.memoryUsage().heapUsed will return a number representing how many bytes of memory the current process is using.

The process.argv property holds an array of command line values provided when the current process was initiated. The first element in the array is the absolute path to Node, which ran the process. The second element in the array is the path to the file that’s running. The following elements will be any command line arguments provided when the process was initiated. Command line arguments are separated from one another with spaces.

node myProgram.js testing several features

console.log(process.argv[3]); // Prints 'several'

We’ve only covered a few of the properties of the process object, so make sure to check out the [documentation on the process object](https://nodejs.org/api/process.html) to learn more about it and explore some of its other methods and properties.

Question:

**1.**

We want the program in **app.js** to store the starting amount of memory used (heapUsed), perform an operation, and then compare the final amount of memory used to the original amount. Right now, the initialMemory variable is assigned to null. Change this line, so that initialMemory is instead assigned the value of the heapUsed property on the object returned from invoking the process.memoryUsage() method.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

We want the user of the program to be able to fill in their own word when they run the program. Right now word is assigned to null. Change the program so that when a user initiates the program with an additional command line argument, word will be assigned that value. For example, running the program with the command: node app.js Codecademy would result in word being assigned the value 'Codecademy'

Checkpoint

Solution:

let initialMemory = process.memoryUsage().heapUsed;

let word = process.argv[2];

console.log(`Your word is ${word}`)

// Create a new array

let wordArray = [];

// Loop 1000 times, pushing into the array each time

for (let i = 0; i < 1000; i++){

  wordArray.push(`${word} count: ${i}`)

}

console.log(`Starting memory usage: ${initialMemory}. \nCurrent memory usage: ${process.memoryUsage().heapUsed}. \nAfter using the loop to add elements to the array, the process is using ${process.memoryUsage().heapUsed - initialMemory} more bytes of memory.`)

**Core Modules and Local Modules**

*Modularity* is a software design technique where one program has distinct parts each providing a single piece of the overall functionality. These separate *modules* come together to build a cohesive whole. Modularity is essential for creating scalable programs which incorporate libraries and frameworks and separate the program’s concerns into manageable chunks. Essentially, a module is a collection of code located in a file. Instead of having an entire program located in a single file, code is organized into separate files and combined through *requiring* them where needed using the require() function.4

Node has several modules included within the environment to efficiently perform common tasks. These are known as the core modules. The core modules are defined within Node.js’s source and are located in the lib/ folder. Core modules are required by passing a string with the name of the module into the require() function:

// Require in the 'events' core module:  
let events = require('events');

We can use the same require() function to require modules of our own creation. To handle these different tasks, the require() function includes some interesting logic “under the hood.” The require() function will first check to see if its argument is a core module, if not, it will move on to different attempts to locate it. Check out the [Node Modules documentation](https://nodejs.org/api/modules.html#modules_modules) to learn more about how require() works.

Let’s walk through the process of requiring a local module:

// dog.js  
module.exports = class Dog {  
   
  constructor(name) {  
    this.name = name;  
  }  
   
  praise() {  
    return `Good dog, ${this.name}!`;  
  }  
};

Above, in the **dog.js** file, we assign the Dog class as the value of module.exports. Each JavaScript file in the Node environment has a special JavaScript object called module.exports. It holds everything in that file, or module, that’s available to be required into a different file.

// app.js  
let Dog = require('./dog.js');  
const tadpole = new Dog('Tadpole');  
console.log(tadpole.praise());

In our **app.js** file we assign the variable Dog to the module.exports object of our **dog.js** file by invoking the require() function. Unlike when we require core modules which are required in with the name of the module as a string, local modules are required by passing in the path to the module. The require() function has some other quirks, like assuming file extensions if none are provided; this means we could have written let Dog = require('./dog'); in place of let Dog = require('./dog.js'); in the code above, and the require() function would have still correctly located and required in **dog.js**.

Question:

**1.**

We wrote a program where a Dog can fight a Cat, but it doesn’t work yet because we haven’t properly connected our code. We created two modules: **cat.js**, which contains our Cat class, and **dog.js**, which contains our Dog class. But we need to assign these modules to their module.exports. Let’s start with **cat.js**. Inside **cat.js** assign the Cat class as the value of module.exports.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

We’ll also need to export our Dog class. Navigate to **dog.js** and assign the Dog class as the value of module.exports.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Great! Now our Dog and Cat classes are being exported, but we still need to require them into **app.js** for our function to work properly. At the top of the **app.js** file, create a variable Dog and assign as its value invoking the require() function with the relative path of the **dog.js** file. Next, create a Cat variable and assign as its value invoking the require() function with the relative path of the **cat.js** file.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Let’s see this program in action! Use the node command to run **app.js** from the terminal.

Checkpoint

Solution:

Cat.js

module.exports = class Cat {

  constructor(name, clawStrength) {

    this.name = name;

    this.clawStrength = clawStrength;

  }

};

App.js

// Require modules in:

let Dog = require('./dog.js');

let Cat = require('./cat.js');

let fight = (dog, cat) => {

    if (dog.toothStrength > cat.clawStrength) {

        console.log(`${dog.name} wins!`);

    }

    else if (dog.toothStrength < cat.clawStrength) {

        console.log(`${cat.name} wins!`);

    }

    else {

        console.log(`${dog.name} and ${cat.name} are equally skilled fighters!`);

    }

}

const myDog = new Dog('Rex', Math.random());

const myCat = new Cat('Tabby', Math.random());

fight(myDog, myCat);

dog.js

module.exports = class Dog {

  constructor(name, toothStrength) {

    this.name = name;

    this.toothStrength = toothStrength;

  }

};

**Node Package Manager(npm):** NPM, which stands for **N**ode **P**ackage **M**anager, is an online collection, or registry, of software. Developers can share code they’ve written to the registry or download code provided by other developers.

**Event-Driven Architecture**

Node provides an EventEmitter class which we can access by requiring in the events core module:

// Require in the 'events' core module  
let events = require('events');  
   
// Create an instance of the EventEmitter class  
let myEmitter = new events.EventEmitter();

Each event emitter instance has an .on() method which assigns a listener callback function to a named event. The .on() method takes as its first argument the name of the event as a string and, as its second argument, the listener callback function.

Each event emitter instance also has an .emit() method which announces a named event has occurred. The .emit() method takes as its first argument the name of the event as a string and, as its second argument, the data that should be passed into the listener callback function.

let newUserListener = (data) => {  
  console.log(`We have a new user: ${data}.`);  
};  
   
// Assign the newUserListener function as the listener callback for 'new user' events  
myEmitter.on('new user', newUserListener)  
   
// Emit a 'new user' event  
myEmitter.emit('new user', 'Lily Pad') //newUserListener will be invoked with 'Lily Pad'

**1.**

In **app.js**, we’ve required in the events core module and written a function listenerCallback which expects to be passed data and will log a string to the console which incorporates that data. Now it’s time to create an event emitter. Create a new variable, myEmitter and assign as its value a new instance of the event emitter class.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Invoke myEmitter‘s .on() method passing in 'celebration' as the event name and listenerCallback as the listener callback function.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Let’s emit a 'celebration' event! Invoke myEmitter‘s .emit() method passing in 'celebration' as the event name and a string of your choice as the second argument.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Let’s see this program run! Use the node command to run **app.js** from the terminal.

Checkpoint

**Solution:**

**App.js**

// Here we require in the 'events' module and save a reference to it in an events variable

let events = require('events');

let listenerCallback = (data) => {

    console.log(`Celebrate ${data}`);

}

// Here we create an instance of the EventEmitter class

let myEmitter = new events.EventEmitter();

// Here we subscribe to 'celebration' events and provide a callback function which will be passed the event's data

myEmitter.on('celebration', listenerCallback);

// Here we emit an event, we pass the event type, 'celebration', as the first argument, and the event data as the second

myEmitter.emit('celebration', 'good times, come on!');

Asynchronous JavaScript with Node.js

Node provides a number of APIs for performing asynchronous tasks which expect callback functions to be passed in as arguments. Under the hood, these APIs trigger the subscription to and emitting of events to signal the completion of the operation. When the operation completes, the callback function is added to a queue, or line, of tasks waiting for their turn to be executed. When the current stack, or list, or synchronous tasks finish executing, the operations on the queue will be performed.

This means if synchronous tasks never end, operations waiting in the event-queue would never have the chance to run. Take a look at the following example code using the asynchronous Node setTimeout() API which asynchronously executes a provided callback function after a given delay:

let keepGoing = true;  
   
let callback = () => {  
  keepGoing = false;  
};  
   
setTimeout(callback, 1000); // Run callback after 1000ms  
   
while(keepGoing === true) {  
  console.log(`This is the song that never ends. Yes, it just goes on and on my friends. Some people started singing it, not knowing what it was, and they'll continue singing it forever just because...`)  
};

This while-loop will continue forever! Even though the callback changing the keepGoing variable to false is added to the event queue after 1 second, it will never have a chance to run— the synchronous code from the loop will always fill the stack! If we wanted to avoid the infinite loop, we could replace the while-loop with an asynchronous function— for example, the Node setInterval() API.

Note: The modern way of handling asynchronous tasks is through JavaScript Promises (developers also favor the newer async...await syntax). If you’re not familiar with these topics, check out [our lessons on them](https://www.codecademy.com/learn/asynchronous-javascript). Newer versions of Node (v8.0.0 and later) provide a collection of the traditional Node asynchronous APIs formatted for promises instead of callbacks. This can be found on util.promisify. Many contemporary 3rd party libraries also favour promise-based patterns over traditional callbacks.

**happen behind the scenes on the back-end! Understanding what User Input/Output:** In Node, we can also receive input from a user through the terminal using the stdin.on() method on the process object:

process.stdin.on('data', (userInput) => {  
  let input = userInput.toString()  
  console.log(input)  
});

Here, we were able to use .on() because under the hood process.stdin is an instance of EventEmitter. When a user enters text into the terminal and hits enter, a 'data' event will be fired and our anonymous listener callback will be invoked. The userInput we receive is an instance of [the Node Buffer class](https://nodejs.org/api/buffer.html#buffer_buffer), so we convert it to a string before printing.

**Errors**

The Node environment has all the standard JavaScript errors such as EvalError, SyntaxError, RangeError, ReferenceError, TypeError, and URIError as well as the JavaScript Error class for creating new error instances. Within our own code, we can generate errors and throw them, and, with synchronous code in Node, we can use [error handling](https://www.codecademy.com/learn/javascript-errors-debugging/modules/errors-and-error-handling) techniques such as try...catch statements.

Many asynchronous Node APIs use error-first callback functions: callback functions which have an error as the first expected argument and the data as the second argument. If the asynchronous task results in an error, it will be passed in as the first argument to the callback function. If no error was thrown, the first argument will be undefined.

const errorFirstCallback = (err, data)  => {  
  if (err) {  
    console.log(`There WAS an error: ${err}`);  
  } else {  
     // err was falsy  
      console.log(`There was NO error. Event data: ${data}`);  
  }  
}

**Filesystem**

All of the data on a computer is organized and accessed through a *filesystem*. In the back-end, however, less restricted interaction with the filesystem is essential. The Node fs core module is an API for interacting with the **f**ile **s**ystem. It was modeled after the [POSIX](https://en.wikipedia.org/wiki/POSIX) standard for interacting with the filesystem.

Each method available through the fs module has a synchronous version and an asynchronous version. One method available on the fs core module is the .readFile() method which **read**s data from a provided **file**:

const fs = require('fs');  
   
let readDataCallback = (err, data) => {  
  if (err) {  
    console.log(`Something went wrong: ${err}`);  
  } else {  
    console.log(`Provided file contained: ${data}`);  
  }  
};  
   
fs.readFile('./file.txt', 'utf-8', readDataCallback);

Let’s walk through the example above:

* We required in the fs core module.
* We define an error-first callback function which expects an error to be passed as the first argument and data as the second. If the error is present, the function will print Something went wrong: ${err}, otherwise, it will print Provided file contained: ${data}.
* We invoked the .readFile() method with three arguments:
  1. The first argument is a string that contains a path to the file **file.txt**.
  2. The second argument is a string specifying the file’s [character encoding](https://en.wikipedia.org/wiki/Character_encoding) (usually ‘utf-8’ for text files).
  3. The third argument is the callback function to be invoked when the asynchronous task of reading from the file system is complete. Node will pass the contents of **file.txt** into the provided callback as its second argument.

**.**

We’ve created a devious treasure hunt for you! Your task is to use fs.readFile() to figure out the secret word and assign that value to the secretWord variable in **app.js**. Here’s your first clue, found scratched into walls of an abandoned castle: **fileOne.txt**.

There are many ways to complete this treasure hunt! You could write one program in **app.js** to solve the puzzle or you can run **app.js** to gain new insight and then change the program based on what you’ve figured out.

If you want some direction, but aren’t quite ready to check out the hint: we suggest you use the fs.readFile() method to print the contents of **fileOne.txt**.

Solution:

App.js

const fs = require('fs');

let secretWord = null;

let readDataCallback = (err, data) => {

  if (err) {

    console.log(`Something went wrong: ${err}`);

  } else {

    console.log(`Provided file contained: ${data}`);

  }

};

//fs.readFile('./fileOne.txt', 'utf-8', readDataCallback);

//fs.readFile('./anotherFile.txt', 'utf-8', readDataCallback);

fs.readFile('./finalFile.txt', 'utf-8', readDataCallback);

secretWord = "cheeseburgerpizzabagels"

**Readable Streams**

One of the simplest uses of streams is reading and writing to files line-by-line. To read files line-by-line, we can use the .createInterface() method from the readline core module. .createInterface() returns an EventEmitter set up to emit 'line' events:

const readline = require('readline');  
const fs = require('fs');  
   
const myInterface = readline.createInterface({  
  input: fs.createReadStream('text.txt')  
});  
   
myInterface.on('line', (fileLine) => {  
  console.log(`The line read: ${fileLine}`);  
});

Let’s walk through the above code:

* We require in the readline and fs core modules.
* We assign to myInterface the returned value from invoking readline.createInterface() with an object containing our designated input.
* We set our input to fs.createReadStream('text.txt') which will create a stream from the **text.txt** file.
* Next we assign a listener callback to execute when line events are emitted. A 'line' event will be emitted after each line from the file is read.
* Our listener callback will log to the console 'The line read: [fileLine]', where [fileLine] is the line just read.

**1.**

You’re going to create a program that reads each item off of a shopping list (located in **shoppingList.txt**) and prints it to the console. Let’s take it one step at a time.

Create a myInterface variable. Assign myInterface the value returned from invoking readline.createInterface().

You’ll want to invoke readline.createInterface() with an object with a key of input and a value of fs.createReadStream(). Remember that fs.createReadStream() expects the file (as a string) from which it should read.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Great work. Let’s create a listener callback function to use in the next step. Name this function printData. printData() should expect to receive some data (we named our parameter data) and it should log that data to the console in the format: Item: [data], where [data] is the argument passed into the function.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

We’re nearly there! Remember that a 'line' event will be emitted after each line from the file is read. Let’s assign our printData() function to execute whenever a 'line' event is emitted by using myInterface‘s .on() method.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Sweet! Let’s run the program in the terminal. Type node app.js in the terminal and press enter. If everything worked, each item from the shopping list should be printed to the terminal.

Solution:

App.js

const readline = require('readline');

const fs = require('fs');

const myInterface = readline.createInterface({

  input: fs.createReadStream('shoppingList.txt')

});

const printData = (data) => {

  console.log(`Item: ${data}`);

}

myInterface.on('line', printData);

**Writable Streams**

We can create a writeable stream to a file using the fs.createWriteStream() method:

const fs = require('fs')  
   
const fileStream = fs.createWriteStream('output.txt');  
   
fileStream.write('This is the first line!');   
fileStream.write('This is the second line!');  
fileStream.end();

In the code above, we set the output file as **output.txt**. Then we .write() lines to the file. Unlike a readable stream, which ends when it has no more data to read, a writable stream could remain open indefinitely. We can indicate the end of a writable stream with the .end() method.

Let’s combine our knowledge of readable and writable streams to create a program which reads from one text file and then writes to another.

**1.**

We’re going to create a writeable stream. We want to write to a file named **shoppingResults.txt**. Create a variable fileStream and assign as its value the writable stream.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Great work. Let’s create a listener callback function to use in the next step. Name this function transformData. transformData should expect to receive some data (we named our parameter line) and it should write() to the writable stream (fileStream) in the format They were out of: [line]\n, where [line] is the argument passed into the function.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

We’re nearly there! Let’s assign our transformData function to execute whenever a 'line' event is emitted on the myInterface stream.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

Sweet! Let’s run the program in the terminal. Type node app.js in the terminal and press enter. If everything worked, you should be able to open the **shoppingResults.txt** file and see that it has the correct contents.

Solution:

const readline = require('readline');

const fs = require('fs');

const myInterface = readline.createInterface({

  input: fs.createReadStream('shoppingList.txt')

});

const fileStream = fs.createWriteStream('shoppingResults.txt');

let transformData = (line) => {

 fileStream.write(`They were out of: ${line}\n`);

};

myInterface.on('line', transformData);

**Create an HTTP Server:** A Node core module designed to meet these needs is the http module. This module contains functions which simplify interacting with HTTP and streamline receiving and responding to requests.

The http.createServer() method returns an instance of an http.server. An http.server has a method .listen() which causes the server to “listen” for incoming connections. When we run http.createServer() we pass in a custom callback function (often referred to as the requestListener). This callback function will be triggered once the server is listening and receives a request.

Let’s break down how the requestListener callback function works:

* The function expects two arguments: a request object and a response object.
* Each time a request to the server is made, Node will invoke the provided requestListener callback function, passing in the request and response objects of the incoming request.
* Request and response objects come with a number of properties and methods of their own, and within the requestListener function, we can access information about the request via the request object passed in.
* The requestListener is responsible for setting the response header and body.
* The requestListener must signal that the interaction is complete by calling the response.end() method.

const http = require('http');  
   
let requestListener = (request, response) => {  
  response.writeHead(200, {'Content-Type': 'text/plain' });  
  response.write('Hello World!\n');  
  response.end();  
};  
   
const server = http.createServer(requestListener);  
   
server.listen(3000);

Let’s walk through the above code:

* We required in the http core module.
* We created a server variable assigned to the return value of the http.createServer() method.
* We invoked http.createServer() with our requestListener callback. This is similar to running the .on() of an EventEmitter: the requestListener will execute whenever an HTTP request is sent to the server on the correct port.
* Within the requestListener callback, we make changes to the response object, response, so that it can send the appropriate information to the client sending the request. The status code 200 means that no errors were encountered. The header communicates that the file type is text, rather than something like audio or compressed data.
* The last line starts the server with the port 3000. Every server on a given machine specifies a unique port so that traffic can be correctly routed.
* Node.js is a JavaScript runtime, an environment that allows us to execute our JavaScript code by converting it into something a computer can understand.
* REPLs are processes that **r**ead, **e**valuate, **p**rint, and repeat (**l**oop), and Node.js comes with its own REPL we can access in our terminal with the node command.
* We run JavaScript programs with Node in the terminal by typing node followed by the file name (if we’re in the same directory) or the absolute path of the file.
* Code can be organized into separate files, modules, and combined through requiring them where needed using the require() function.
* In addition to core modules, modules included within the environment to efficiently perform common tasks, we can also create our own modules using module.exports and the require() function.
* We can access NPM, a registry of hundreds of thousands of packages of re-usable code from other developers, directly through our terminal.
* Node has an event-driven architecture.
* We can make our own instances of the EventEmitter class and we can subscribe to listen for named events with the .on() method and emit events with the .emit() method.
* Node uses an event loop which enables asynchronous actions to be handled in a non-blocking way by adding callback functions to a queue of tasks to be executed when the callstack is empty.
* In order to handle errors during asynchronous operations, provided callback functions are expected to have an error as their first parameter.
* Node allows for both output, data/feedback to a user provided by a computer, and input data/feedback to the computer provided by the user.
* The Node fs core module is an API for interacting with the **f**ile **s**ystem.
* Streams allow us to read or write data piece by piece instead of all at once.
* The Node http core module allows for easy creation of web servers, computer processes that listen for requests from clients and return responses.

**LEARN EXPRESS ROUTES**

Introduction: Express is a powerful but flexible Javascript framework for creating web servers and [APIs](https://en.wikipedia.org/wiki/Web_API). It can be used for everything from simple static file servers to JSON APIs to full production servers.

**Starting A Server**

Express is a Node module, so in order to use it, we will need to import it into our program file. To create a server, the imported express function must be invoked.

const express = require('express');  
const app = express();

On the first line, we import the Express library with require. When invoked on the second line, it returns an instance of an Express application.

The purpose of a server is to listen for requests, perform whatever action is required to satisfy the request, and then return a response. In order for our server to start responding, we have to tell the server where to listen for new requests by providing a port number argument to a method called app.listen(). The server will then listen on the specified [port](https://en.wikipedia.org/wiki/Port_(computer_networking)) and respond to any requests that come into it.

The second argument is a callback function that will be called once the server is running and ready to receive responses.

const PORT = 4001;  
app.listen(PORT, () => {  
  console.log(`Server is listening on port ${PORT}`);  
});

In this example, our app.listen() call will start a server listening on port 4001, and once the server is started it will log 'Server is listening on port 4001'.

### Instructions

**1.**

Import express using require syntax and assign it to an express variable.

Create an instance of an Express server and save it to a variable named app.

Start the server listening on the port defined by the PORT variable. When the server has started, log a message to the console that the server is listening for requests.

To actually start your server listening, run the command node app.js to run your server in [Node](https://nodejs.org/en/). At this point, it won’t do much, but if you’ve completed the steps above, it will log your message to show that the server started successfully.

Solution: app.js

// Import the express library here

const express = require('express');

// Instantiate the app here

const app = express();

const PORT = process.env.PORT || 4001;

// Invoke the app's `.listen()` method below:

app.listen(PORT, () => {

  console.log(`Server is listining on port ${PORT}`);

})

**Writing Your First Route:**

Routes define the control flow for requests based on the request’s path and HTTP verb.

For example, if your server receives a GET request at /monsters, we will use a route to define the appropriate functionality for that HTTP verb (GET) and path (/monsters).

The path is the part of a request URL after the [hostname](https://en.wikipedia.org/wiki/Hostname) and port number, so in a request to localhost:4001/monsters, the path is /monsters (in this example, the hostname is localhost, the port number is 4001).

The HTTP verb is always included in the request, and it is one of a [finite number of options](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) used to specify expected functionality. GET requests are used for retrieving resources from a server, and we will discuss additional request types in later exercises.

Express uses app.get() to register routes to match GET requests. Express routes (including app.get()) usually take two arguments, a path (usually a string), and a callback function to handle the request and send a response.

const moods = [{ mood: 'excited about express!'}, { mood: 'route-tastic!' }];  
app.get('/moods', (req, res, next) => {  
  // Here we would send back the moods array in response  
});

The route above will match any GET request to '/moods' and call the callback function, passing in two objects as the first two arguments. These objects represent the request sent to the server and the response that the Express server should eventually send to the client.

If no routes are matched on a client request, the Express server will handle sending a 404 Not Found response to the client.

### Instructions

**1.**

Now that your server starting code should be working properly, you can start up the Express Yourself machine. Start your server from the terminal window with node app.js. Once it logs that it is running, you can refresh the browser window currently displaying Not Found.

Inside **app.js**, create a route handler to handle a GET request to '/expressions'. For now, give it a req, res, next callback. For now, log the req object inside the callback. Verify that the route works and logs the request by starting your server and clicking the Refresh Expressions button which will send a GET /expressions request.

We will complete this route in the next exercise and finish the first round of functionality to the Express Yourself machine.

You may notice that there’s a line with the command app.use(express.static('public'));. This is used to make sure that once the server is started, you can reload the browser and see the Express Yourself machine.

Checkpoint

**Solution:app.js**

const express = require('express');

const app = express();

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

// Open a call to `app.get()` below:

const expressions = [{expression: 'excited about express!'}, { expression: 'route-tastic!' }];

app.get('/expressions', (req, res, next) => {

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Sending A Response:** Express servers send responses using the .send() method on the response object. .send() will take any input and include it in the response body.

const monsters = [  
  { type: 'werewolf' },   
  { type: 'hydra' },   
  { type: 'chupacabra' }  
];  
app.get('/monsters', (req, res, next) => {  
  res.send(monsters);  
});

In this example, a GET /monsters request will match the route, Express will call the callback function, and the res.send() method will send back an array of spooky monsters.

In addition to .send(), .json() can be used to explicitly send JSON-formatted responses. .json() sends any JavaScript object passed into it.

### Instructions

**1.**

Send the expressions array from your app.get handler. Now that you have a complete route, you can test it out by reloading the browser window and clicking the ‘Refresh Expressions’ button on the machine.

If you make changes to **app.js**, you will need to restart your server to see the changes in effect. You can do this by pressing Ctrl + C in the terminal window to stop the old server, and you can start it again with node app.js.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, seedElements } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Matching Route Paths:** In the example to the right, you can see two .get() routes registered at /another-route and /expressions. When a GET /expressions request arrives to the Express server, it first checks /another-route‘s path because it is registered before the /expressions route. Because /another-route does not match the path, Express moves on to the next registered middleware. Since the route matches the path, the callback is invoked, and it sends a response.

**Request**

**GET/Expressions : diagram on notebook**

**Getting A Single Expression:** Express servers provide this functionality with named route parameters. Parameters are route path segments that begin with : in their Express route definitions. They act as [wildcards](https://expressjs.com/en/guide/routing.html#route-parameters), matching any text at that path segment. For example /monsters/:id will match both/monsters/1 and /monsters/45.

Express parses any parameters, extracts their actual values, and attaches them as an object to the request object: req.params. This object’s keys are any parameter names in the route, and each key’s value is the actual value of that field per request.

const monsters = {   
  hydra: { height: 3, age: 4 },   
  dragon: { height: 200, age: 350 }   
};  
// GET /monsters/hydra  
app.get('/monsters/:name', (req, res, next) => {  
  console.log(req.params)' // { name: 'hydra' }  
  res.send(monsters[req.params.name]);  
});

In this code snippet, a .get() route is defined to match /monsters/:name path. When a GET request arrives for /monsters/hydra, the callback is called. Inside the callback, req.params is an object with the key name and the value hydra, which was present in the actual request path.

The appropriate monster is retrieved by name (the object key) from the monsters object and sent back to the client with res.send().

### Instructions

**1.**

Create a GET /expressions/:id get route that you will use to send back a single expression. You can use req.params object and the pre-written helper function getElementById(id, array) to find the correct expression before sending it back.

For instance, to find ID 560 from expressions, you would call getElementById(560, expressions);. This function returns the element object if it exists and undefined if it does not.

Don’t forget to restart your server when you make changes to **app.js**. To test the Express Yourself machine, use the box in the upper-left corner to send a GET request for a specified ID.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, seedElements } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

// Add a new call to app.get('/expressions/:id') here

app.get('/expressions/:id', (req, res, next) => {

  console.log(req.params.id)

};

res.send(expressions[req.params.name]);

})

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Setting Status Codes:** Response codes provide information to clients about how their requests were handled. Until now, we have been allowing the Express server to set status codes for us. For example, any res.send() has by default sent a 200 OK status code.

The res object has a .status() method to allow us to set the status code, and other methods like .send() can be chained from it.

const monsterStoreInventory = { fenrirs: 4, banshees: 1, jerseyDevils: 4, krakens: 3 };  
app.get('/monsters-inventory/:name', (req, res, next) => {  
  const monsterInventory = monsterStoreInventory[req.params.name];  
  if (monsterInventory) {  
    res.send(monsterInventory);  
  } else {  
    res.status(404).send('Monster not found');  
  }  
});

In this example, we’ve implemented a route to retrieve inventory levels from a Monster Store. Inventory levels are kept in the monsterStoreInventory variable. When a request arrives for /monsters-inventory/mothMen, the route matches and so the callback is invoked. req.params.name will be equal to 'mothMen' and so our program accesses monsterStoreInventory['mothMen']. Since there are no mothMen in our inventory,res.status() sets a 404 status code on the response, and .send() sends the response.

### Instructions

**1.**

Let’s make sure that our GET /expressions/:id route handles invalid requests properly, for instance if we request an expression ID that does not exist.

Complete your route so that it sends back the correct expression object if it exists and sends back a 404 response if it does not.

Solution: ap.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, seedElements } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Matching Longer Paths:** Route parameters will match anything in their specific part of the path, so a route matching /monsters/:name would match all the following request paths:

/monsters/hydra  
/monsters/jörmungandr  
/monsters/manticore  
/monsters/123

In order for a request to match a route path, it must match the entire path, as shown in the diagram to the right. The request arrives for /expressions/1. It first tries to match the /expressions route, but because it has additional path segments after /expressions, it does not match this route and moves on to the next. It matches /expressions/:id because :id will match any value at that level of the path segment. The route matches, so the Express server calls the callback function, which in turn handles the request and sends a response.

**Diagram on notebook.**

**Other HTTP Methods:** This course will cover three other important HTTP methods: PUT, POST, and DELETE. Express provides methods for each one: app.put(), app.post(), and app.delete().

PUT requests are used for updating existing resources. In our Express Yourself machine, a PUT request will be used to update the name or emoji of an expression already saved in our database. For this reason, we will need to include a unique identifier as a route parameter to determine which specific resource to update.

### Instructions

**1.**

For now, open a PUT /expressions/:id route with an empty (req, res, next) callback function. We will fully implement its functionality in the next exercise.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, seedElements } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Add your PUT route handler below:

app.put('/expressions/:id', (req, res, next) => {

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Using Queries:** Query strings do not count as part of the route path. Instead, the Express server parses them into a JavaScript object and attaches it to the request body as the value of req.query. The key: value relationship is indicated by the = character in a query string, and key-value pairs are separated by &. In the above example route, the req.query object would be { name: 'chimera', age: '1' }.

const monsters = { '1': { name: 'cerberus', age: '4'  } };  
// PUT /monsters/1?name=chimera&age=1  
app.put('/monsters/:id', (req, res, next) => {  
  const monsterUpdates = req.query;  
  monsters[req.params.id] = monsterUpdates;  
  res.send(monsters[req.params.id]);  
});

Here, we have a route for updating monsters by ID. When a PUT /monsters/1?name=chimera&age=1 request arrives, our callback function is called and, we create a monsterUpdates variable to store req.query. Since req.params.id is '1', we replace monsters['1']‘s value with monsterUpdates . Finally, Express sends back the new monsters['1'].

When updating, many servers will send back the updated resource after the updates are applied so that the client has the exact same version of the resource as the server and database.

### Instructions

**1.**

Use req.query to update the proper element in the expressions array.

We’ve imported a helper function from **/utils.js** to help with this task.

You can use the updateElement() helper function in your PUT /expressions/:id route.

It takes three arguments:

* id (the ID number of the element)
* queryArguments (the new, updated expression object from req.query)
* elementList (the array which contains the element to update)

updateElement() updates that specific element in the elementList array (you’ll pass in the expressions array), and then returns the updated element.

Be sure to check that an expression with the id you provide exists in the expressions array (getIndexById() can help)!

To test your functionality with the Express Yourself machine, make sure your server is running, get all expressions, and then use the UPDATE tab to select an individual expression, select updates, and send the PUT request.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Matching By HTTP Verb**

Express matches routes using both path and HTTP method verb. In the diagram to the right, we see a request with a PUT verb and /expressions (remember that the query is not part of the route path). The path for the first route matches, but the method verb is wrong, so the Express server will continue to the next registered route. This route matches both method and path, and so its callback is called, the necessary updating logic is executed, and the response is sent.

**Diagram in notebook:**

Creating An Expression

POST is the HTTP method verb used for creating new resources. Because POST routes create new data, their paths do not end with a route parameter, but instead end with the type of resource to be created.

For example, to create a new monster, a client would make a POST request to /monsters. The client does not know the id of the monster until it is created and sent back by the server, therefore POST /monsters/:id doesn’t make sense because a client couldn’t know the unique id of a monster before it exists.

Express uses .post() as its method for POST requests. POST requests can use many ways of sending data to create new resources, including query strings.

The HTTP status code for a newly-created resource is 201 Created.

### Instructions

**1.**

Create a POST /expressions route. It should send create and add a new expression to the expressions array if it is a valid new expression (meaning it has an emoji and name key). It should send back the new element with a 201 status code if it is valid, and it should send a 400 status code if the object is not valid.

You can use the createElement(elementType, objectToCreate) helper function to create a valid expression. The first argument is the type of element, so it should be 'expressions' in this case. The second argument should be the query object with an emoji and a name property. This function will return false if the objectToCreate does not contain all necessary key-value pairs, and it will return the newly-created element if object to create is valid. It does not add the created element to any arrays, you will need to do so yourself.

Don’t forget to restart your server and test as you implement the functionality. To test your route, use the POST tab in the upper left corner. Select a name and emoji and send the request to see if your route works as intended.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, getIndexById, updateElement,

        seedElements, createElement } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Deleting Old Expressions**

DELETE is the HTTP method verb used to delete resources. Because DELETE routes delete currently existing data, their paths should usually end with a route parameter to indicate which resource to delete.

Express uses .delete() as its method for DELETE requests.

Servers often send a 204 No Content status code if deletion occurs without error.

### Instructions

**1.**

Create a DELETE /expressions/:id route. It should send back a 404 response for a request with an invalid id, and it should delete the proper element from the expressions array and send a 204 status with a valid id.

To test your functionality, use the DELETE tab in the upper left. Select the ID to delete and send the request.

Solution: app.js

const express = require('express');

const app = express();

// Serves Express Yourself website

app.use(express.static('public'));

const { getElementById, getIndexById, updateElement,

        seedElements, createElement } = require('./utils');

const expressions = [];

seedElements(expressions, 'expressions');

const animals = [];

seedElements(animals, 'animals');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Listening on port ${PORT}`);

});

**Adding Animals Routes:** You are going to add an additional set of functionality to our machine: Animal Mode! This will involve creating similar GET, POST, PUT, and DELETE routes.

### Instructions

**1.**

In your **app.js** file, Create a GET /animals route to return an array of all animals.

Checkpoint 2 Passed

**2.**

Create a GET /animals/:id route to respond with a single animal.

Checkpoint 3 Passed

**3.**

Create a PUT /animals/:id route to update an animal in animals and send back the updated animal.

Checkpoint 4 Passed

**4.**

Create a POST /animals route to add new animals to the animals and respond with the new animal.

Checkpoint 5 Passed

**5.**

Create a DELETE /animals/:id route to delete animals by ID.

Checkpoint

**Solution:app.js**

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

const expressions = [];

seedElements(expressions, 'expressions');

const animals = [];

seedElements(animals, 'animals');

// Get all expressions

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**Wrap Up:** In this exercise, you were able to create a full server allowing users to implement all CRUD operations for two kinds of resources: Expressions and Animals! With these skills and knowledge of the HTTP request-response cycle, you could implement an API for any project needing CRUD functionality. You could build a trip planner, an address book, a grocery list, an image-sharing application, an anonymous message board, the sky’s the limit!

Continue on to the next lesson to learn more about how to keep your code clean and modular with Express Routers!

Solution: app.js

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

const expressions = [];

seedElements(expressions, 'expressions');

const animals = [];

seedElements(animals, 'animals');

// Get all expressions

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**This File Is Too Big!:** Express provides functionality to alleviate this problem: Routers. Routers are mini versions of Express applications — they provide functionality for handling route matching, requests, and sending responses, but they do not start a separate server or listen on their own ports. Routers use all the .get(), .put(), .post(), and .delete()

Solution:app.js

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

const expressions = [];

seedElements(expressions, 'expressions');

const animals = [];

seedElements(animals, 'animals');

// Get all expressions

app.get('/expressions', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**Express.Router**

An Express router provides a subset of Express methods. To create an instance of one, we invoke the .Router() method on the top-level Express import.

To use a router, we mount it at a certain path using app.use() and pass in the router as the second argument. This router will now be used for all paths that begin with that path segment. To create a router to handle all requests beginning with /monsters, the code would look like this:

const express = require('express');  
const app = express();  
   
const monsters = {  
  '1': {  
    name: 'godzilla',  
    age: 250000000  
  },  
  '2': {  
    name: 'manticore',  
    age: 21  
  }  
}  
   
const monstersRouter = express.Router();  
   
app.use('/monsters', monstersRouter);  
   
monstersRouter.get('/:id', (req, res, next) => {  
  const monster = monsters[req.params.id];  
  if (monster) {  
    res.send(monster);  
  } else {  
    res.status(404).send();  
  }  
});

Inside the monstersRouter, all matching routes are assumed to have /monsters prepended, as it is mounted at that path. monstersRouter.get('/:id') matches the full path /monsters/:id.

When a GET /monsters/1 request arrives, Express matches /monsters in app.use() because the beginning of the path ('/monsters') matches. Express’ route-matching algorithm enters the monstersRouter‘s routes to search for full path matches. Since monstersRouter.get('/:id) is mounted at /monsters, the two paths together match the entire request path (/monsters/1), so the route matches and the callback is invoked. The 'godzilla' monster is fetched from the monsters array and sent back.

### Instructions

**1.**

Create an expressionsRouter instance of Express.Router. Mount it at /expressions at your base app level with app.use.

After doing so, create a route for your expressionsRouter that will send all expressions for a GET request.

Solution:app.js

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

const expressions = [];

seedElements(expressions, 'expressions');

const animals = [];

seedElements(animals, 'animals');

//answer

const expressionsRouter = express.Router();

app.use('/expressions', expressionsRouter);

// Get all expressions

expressionsRouter.get('/', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**Exercise: Using Multiple Router Files**

Generally, we will keep each router in its own file, and require them in the main application. This allows us to keep our code clean and our files short.

To do this with monstersRouter, we would create a new file **monsters.js** and move all code related to /monsters requests into it.

// monsters.js  
const express = require('express');  
const monstersRouter = express.Router();  
   
const monsters = {  
  '1': {  
    name: 'godzilla',  
    age: 250000000  
  },  
  '2': {  
    Name: 'manticore',  
    age: 21  
  }  
}  
   
monstersRouter.get('/:id', (req, res, next) => {  
  const monster = monsters[req.params.id];  
  if (monster) {  
    res.send(monster);  
  } else {  
    res.status(404).send();  
  }  
});  
   
module.exports = monstersRouter;

This code contains all the monsters specific code. In a more full-fledged API, this file would contain multiple routes. To use this router in another file, we use module.exports so that other files can access monstersRouter. The only other new line of code required is that Express must be required in each file, since we’ll need to create a router with express.Router().

Our **main.js** file could then be refactored to import the monstersRouter:

// main.js  
const express = require('express');  
const app = express();  
const monstersRouter = require('./monsters.js');  
   
app.use('/monsters', monstersRouter);

In this example, the monstersRouter is required in **main.js** from **monsters.js** and used exactly as it was before.

### Instructions

**1.**

Let’s start to refactor our /expressions routes to **expressions.js**.

Open the **expressions.js** file. Create an expressionsRouter instance of Express.Router.

Export expressionsRouter from that file with module.exports. Remove your old expressionsRouter from **app.js**. require your expressionsRouter from **expressions.js** into **app.js** and make sure it is mounted at /expressions.

Move your GET /expressions handler from **app.js** to **expressions.js**. You’ll have to also move the expressions array to that file. Finally, make sure that you move the seedElements(expressions) line into **expressions.js**.

Solution: app.js

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

let animals = [];

seedElements(animals, 'animals');

const expressionsRouter = require('./expressions.js');

app.use('/expressions', expressionsRouter);

// Get a single expression

app.get('/expressions/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

app.put('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

app.post('/expressions', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

app.delete('/expressions/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**expression.js**

const express = require('express');

const { seedElements, getElementById, createElement, updateElement, getIndexById } = require('./utils');

let expressions = [];

seedElements(expressions, 'expressions');

const expressionsRouter = express.Router();

module.exports = expressionsRouter;

// Get all expressions

expressionsRouter.get('/', (req, res, next) => {

  res.send(expressions);

});

**Matching In Nested Routers**

As you saw in the previous exercise, when using routers, it’s important to remember that the full path of a request can be segmented.

In the diagram to the left, you can create an Express application using two routers. A GET request arrives for /expressions/1. Because the beginning of the path does not match /animals in the first app.use(), the Express server moves on to the next app.use(), which matches /expressions.

Express’ route matching algorithm then enters the expressionsRouter instance which is required from **expressions.js**. Inside this router, the path matching changes. Even though the whole request path is /expressions/1, inside the expressionsRouter, all paths are matched from the parts of the path after /expressions, meaning that in this context, the router is trying to match the path /1.

Because the path is /1, the path does not match the first .get() method at /. The Express server moves on to the next route, which has a route parameter of /:id, so it matches! This route handles the necessary logic and sends the response.

Routers can be nested as many times as necessary for an application, so understanding nested route matching is important for created complicated APIs.

**Diagram in notebook**

**Refactoring Expressions Routes**

Now that you’ve learned about nested route matching, let’s refactor the rest of the /expressions routes into **expressions.js**.

### Instructions

**1.**

Move all your /expressions routes to your router into **expressions.js**. Make sure that they still match the same request paths, and remove the duplicate code from **app.js**.

Move the following routes to **expresions.js**:

* GET /expressions/:id
* PUT /expressions/:id
* POST /expressions
* DELETE /expressions/:id

Remember to change the paths for each route handler as you move them, as they should already be mounted at /expressions inside **expressions.js**.

Make sure that you still require the same helper functions from **utils.js** in **expressions.js**

**Expression.js**

const express = require('express');

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

let expressions = [];

seedElements(expressions, 'expressions');

expressionsRouter = express.Router();

// Get all expressions

expressionsRouter.get('/', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

expressionsRouter.get('/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

expressionsRouter.put('/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

expressionsRouter.post('/', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

expressionsRouter.delete('/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

module.exports = expressionsRouter;

**app.js**

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

let animals = [];

seedElements(animals, 'animals');

// Import and mount the expressionsRouter

const expressionsRouter = require('./expressions.js');

app.use('/expressions', expressionsRouter);

// Get all animals

app.get('/animals', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

app.get('/animals/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

app.post('/animals', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

app.put('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

app.delete('/animals/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

**Refactoring Animals Routes**

Okay, training wheels off! Now refactor all your /animals routes to **animals.js**.

### Instructions

**1.**

Now, do the same refactoring for /animals routes into an animalsRouter Router that you create in **animals.js**.

App.js

const express = require('express');

const app = express();

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

const PORT = process.env.PORT || 4001;

// Use static server to serve the Express Yourself Website

app.use(express.static('public'));

// Import and mount the expressionsRouter

const expressionsRouter = require('./expressions.js');

app.use('/expressions', expressionsRouter);

// Import and mount the expressionsRouter

const animalsRouter = require('./animals.js');

app.use('/animals', animalsRouter);

app.listen(PORT, () => {

  console.log(`Server is listening on ${PORT}`);

});

Expression.js

const express = require('express');

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

let expressions = [];

seedElements(expressions, 'expressions');

expressionsRouter = express.Router();

// Get all expressions

expressionsRouter.get('/', (req, res, next) => {

  res.send(expressions);

});

// Get a single expression

expressionsRouter.get('/:id', (req, res, next) => {

  const foundExpression = getElementById(req.params.id, expressions);

  if (foundExpression) {

    res.send(foundExpression);

  } else {

    res.status(404).send();

  }

});

// Update an expression

expressionsRouter.put('/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    updateElement(req.params.id, req.query, expressions);

    res.send(expressions[expressionIndex]);

  } else {

    res.status(404).send();

  }

});

// Create an expression

expressionsRouter.post('/', (req, res, next) => {

  const receivedExpression = createElement('expressions', req.query);

  if (receivedExpression) {

    expressions.push(receivedExpression);

    res.status(201).send(receivedExpression);

  } else {

    res.status(400).send();

  }

});

// Delete an expression

expressionsRouter.delete('/:id', (req, res, next) => {

  const expressionIndex = getIndexById(req.params.id, expressions);

  if (expressionIndex !== -1) {

    expressions.splice(expressionIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

module.exports = expressionsRouter;

animals.js

const express = require('express');

const { getElementById, getIndexById, updateElement,

  seedElements, createElement } = require('./utils');

let animals = [];

seedElements(animals, 'animals');

animalsRouter = express.Router();

// Get all animals

animalsRouter.get('/', (req, res, next) => {

  res.send(animals);

});

// Get a single animal

animalsRouter.get('/:id', (req, res, next) => {

  const animal = getElementById(req.params.id, animals);

  if (animal) {

    res.send(animal);

  } else {

    res.status(404).send();

  }

});

// Create an animal

animalsRouter.post('/', (req, res, next) => {

  const receivedAnimal = createElement('animals', req.query);

  if (receivedAnimal) {

    animals.push(receivedAnimal);

    res.status(201).send(receivedAnimal);

  } else {

    res.status(400).send();

  }

});

// Update an animal

animalsRouter.put('/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    updateElement(req.params.id, req.query, animals);

    res.send(animals[animalIndex]);

  } else {

    res.status(404).send();

  }

});

// Delete a single animal

animalsRouter.delete('/:id', (req, res, next) => {

  const animalIndex = getIndexById(req.params.id, animals);

  if (animalIndex !== -1) {

    animals.splice(animalIndex, 1);

    res.status(204).send();

  } else {

    res.status(404).send();

  }

});

module.exports = animalsRouter;

## The Postman App

Postman is a GUI that aids in the development of APIs by making it easy to test requests and their responses in an organized way. Everything you can do in Postman you can also do through the command line, but Postman makes the process quicker and easier by providing a clean interface and powerful set of tools.

# What is Middleware? A simple explanation.

Middleware is helpful if you want to run code on each request made to a server and take action on request or response data.

**MIDDLEWARE: DRYing Code With Functions**

Take a look at the following code:

const addFive = number => {  
  const fiveAdded = number + 5;  
  console.log(`Your number plus 5 is ${fiveAdded}`);  
}  
   
const addTen = number => {  
  const tenAdded = number + 10;  
  console.log(`Your number plus 10 is ${tenAdded}`);  
}  
   
const addTwenty = number => {  
  const twentyAdded = number + 20;  
  console.log(`Your number plus 20 is ${twentyAdded}`);  
}

While these three function definitions are not exact duplicates of each other, a well-designed application will be flexible enough to join similar functionality in a single element.

const addNumber = (number, addend) => {  
  const numAdded = number + addend;  
  console.log(`Your number plus ${addend} is ${numAdded}`);  
}

As you can see, by adding an argument to the earlier functions we can simplify our application code which will ultimately save time should we realize that we also want an addFifty() function and an addHundred() function.

Code that performs the same task in multiple places is repetitive, and the quality coder’s credo is “Don’t Repeat Yourself” (DRY). If a program performs similar tasks without refactoring into a function, it is said to “violate DRY”. “Violating DRY” is a programmer’s way of complaining: “This script is saying the same thing over and over! We can do the same thing with less code!” Let’s try to not repeat ourselves in this codebase by repurposing some of the more glaringly repeated code into functions we can call instead.

### Instructions

**1.**

We have provided a front-end for testing out your routes throughout this lesson. To get it to display in each exercise, start your server (node app.js) and then refresh the browser to the right. A tool should appear that allows you to set request verbs, paths, and body information, and then make requests using that information. Use this tool to ensure your server is working as expected throughout this lesson, checking your server logs and examining the returned responses as you make changes.

Checkpoint 2 Passed

**2.**

Currently, each route logs a message with the HTTP method and a message that the request was received (i.e. 'GET Request Received'). Write a function logRequest that takes a single string parameter verb and logs a message formatted in the same fashion.

Checkpoint 3 Passed

**3.**

Replace the console.log calls that open each route and replace each with a call to logRequest. Pass in the method name for each route.

Checkpoint

**SOLUTION: APP.JS**

const express = require('express');

const app = express();

const PORT = process.env.PORT || 4001;

app.use(express.static('public'));

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Add your logging function here:

const logRequest = verb => {

  console.log(`${verb} Request Received`);

}

app.get('/beans/', (req, res, next) => {

  logRequest('GET');

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  logRequest('POST');

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(queryData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(400).send('Bag with that name already exists!');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  logRequest('GET');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  logRequest('POST');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    jellybeanBag[beanName].number += numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  logRequest('POST');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    if (jellybeanBag[beanName].number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    jellybeanBag[beanName].number -= numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  logRequest('DELETE');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  logRequest('PUT');

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const newName = JSON.parse(queryData).name;

    jellybeanBag[newName] = jellybeanBag[beanName];

    jellybeanBag[beanName] = null;

    res.send(jellybeanBag[newName]);

    console.log('Response Sent');

  });

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**DRYing Routes With app.use():** Middleware is code that executes between a server receiving a request and sending a response. It operates on the boundary, so to speak, between those two HTTP actions.

In Express, middleware is a function. Middleware can perform logic on the request and response objects, such as: inspecting a request, performing some logic based on the request, attaching information to the response, attaching a status to the response, sending the response back to the user, or simply passing the request and response to another middleware. Middleware can do any combination of those things or anything else a Javascript function can do.

app.use((req, res, next) => {  
  console.log('Request received');  
});

The previous code snippet is an example of middleware in action. app.use() takes a callback function that it will call for every received request. In this example, every time the server receives a request, it will find the first registered middleware function and call it. In this case, the server will find the callback function specified above, call it, and print out 'Request received'.

You might be wondering what else our application is responsible for that isn’t related to middleware. The answer is not much. To quote the [Express documentation](http://expressjs.com/en/guide/using-middleware.html):

An Express application is essentially a series of middleware function calls.

It is precisely this service that we leverage Express for. In addition to performing the routing that allows us to communicate appropriate data for each separate endpoint, we can perform application logic we need by implementing the necessary middleware.

### Instructions

**1.**

After your logRequest function, there is an unfinished call to app.use(). Its callback will be called before every route. We’ll be moving the logging out of logRequest, so we no longer have access to the verb string. Since we can access the req object, however, we can use the req.method property which will always be equal to the verb of the request! Finish the app.use() callback by replicating the logging behavior of logRequest.

Don’t be afraid if your server no longer returns responses. We will fix this in the next exercise.

Checkpoint 2 Passed

**2.**

All the calls to logRequest should now be redundant. Remove them from every route, and remove the logRequest function itself.

Checkpoint 3 Passed

**3.**

Now we’ve removed a significant amount of code, but our routes aren’t returning responses, because something is still missing from our first app.use() call. Move on to the next exercise when you’re ready.

Checkpoint

**Solution:app.js**

const express = require('express');

const app = express();

const PORT = process.env.PORT || 4001;

app.use(express.static('public'));

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

//1 answer

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(queryData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(404).send('Bean with that name does not exist');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    jellybeanBag[beanName].number += numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(queryData).number) || 0;

    if (jellybeanBag[beanName].number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    jellybeanBag[beanName].number -= numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    const newName = JSON.parse(queryData).name;

    jellybeanBag[newName] = jellybeanBag[beanName];

    jellybeanBag[beanName] = null;

    res.send(jellybeanBag[newName]);

    console.log('Response Sent');

  });

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**next():** an app.use() that occurs after an app.get() will get called after the app.get(). Observe the following code:

app.use((req, res, next) => {  
  console.log("A sorcerer approaches!");  
  next();  
});  
   
app.get('/magic/:spellname', (req, res, next) => {  
  console.log("The sorcerer is casting a spell!");  
  next();  
});  
   
app.get('/magic/:spellname', (req, res, next) => {  
  console.log(`The sorcerer has cast ${req.params.spellname}`);  
  res.status(200).send();  
});  
   
app.get('/magic/:spellname', (req, res, next) => {  
  console.log("The sorcerer is leaving!");  
});  
   
// Accessing http://localhost:4001/magic/fireball   
// Console Output:  
// "A sorcerer approaches!"  
// "The sorcerer is casting a spell!"  
// "The sorcerer has cast fireball"

In the above code, the routes are called in the order that they appear in the file, provided the previous route called next() and thus passed control to the next middleware. We can see that the final matching call was not printed. This is because the previous middleware did not invoke the next() function to run the following middleware.

An Express middleware is a function with three parameters: (req, res, next). The sequence is expressed by a set of callback functions invoked progressively after each middleware performs its purpose. The third argument to a middleware function, next, should get explicitly called as the last part of the middleware’s body. This will hand off the processing of the request and the construction of the response to the next middleware in the stack.

### Instructions

**1.**

Add a call to next after your logging statement so that the middleware will be executed before moving on to your routes.

Checkpoint 2 Passed

**Solution: app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(bodyData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(400).send('Bean with that name already exists!');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    jellybeanBag[beanName].number += numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    if (jellybeanBag[beanName].number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    jellybeanBag[beanName].number -= numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Request And Response Parameters**

Recall the function signature of an Express middleware, i.e., (req, res, next). Express routes are middleware. Every route created in Express is also a middleware function handling the request and response objects at that part of the stack. Express routes also have the option of sending a response body and status code and closing the connection. These two features are a by product of Express routes being middleware, because all Express middleware functions have access to the request, the response, and the next middleware in the stack.

App.js

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(bodyData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(400).send('Bean with that name already exists!');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    jellybeanBag[beanName].number += numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    if (jellybeanBag[beanName].number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    jellybeanBag[beanName].number -= numberOfBeans;

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Route-Level app.use() - Single Path**

Let’s see what the [Express documentation](https://expressjs.com/en/4x/api.html) for app.use() has to say about this use case. This is the app.use() function signature:

app.use([path,] callback [, callback...])

In documentation for many programming languages, optional arguments for functions are placed in square brackets ([]). This means that app.use() takes an optional path parameter as its first argument. We can now write middleware that will run for every request at a specific path.

app.use('/sorcerer', (req, res, next) => {  
  console.log('User has hit endpoint /sorcerer');  
  next();  
});

In the example above the console will print 'User has hit endpoint /sorcerer', if someone visits our web page’s ‘/sorcerer’ endpoint. Since the method app.use() was used, it won’t matter if the user is performing a GET,a POST, or any other kind of HTTP request. Since the path was given as an argument to app.use(), this middleware function will not execute if the user hits a different path (for instance: '/spells' or '/sorcerer/:sorcerer\_id').

### Instructions

**1.**

We’re going to refactor all the logic that checks the existence of a jelly bean into a new middleware function. Currently, this logic is used in every route that begins with beans/:beanName and looks like this:

const beanName = req.params.beanName;  
  if (!jellybeanBag[beanName]) {  
    console.log('Response Sent');  
    return res.status(404).send('Bean with that name does not exist');  
  }

We check to see if the bean with the supplied name exists in jellybeanBag, and we send a 404 response if it does not. The return statement ensures that we break out of the middleware and don’t try any operations on a nonexistent jelly bean.

Create a new app.use call after your logging middleware. It should be called for all /beans/:beanName routes. You can leave the callback empty at this point.

Checkpoint 2 Passed

**2.**

Copy all the checking logic (from const beanName through the if statement) from a route into your middleware callback. Remove those lines from every route that uses them.

Checkpoint 3 Passed

**3.**

After the checking logic, we’re going to attach the correct bean object to the request by setting a bean property on the request (req.bean). Set it equal to the correct bean from the bean object. For good measure, also attach the bean name to the request as req.beanName.

After these properties are set, be sure to call next.

Checkpoint 4 Passed

Stuck? Get a hint

**4.**

You can now remove the duplicate checking logic from all /beans/:beanName routes. To make sure that all your routes still work if we remove const beanName = req.params.beanName; from them, make sure that you use req.beanName any place where you need to access the bean by name. For instance, inside app.delete, you’ll have to change

jellybeanBag[beanName] = null;

to

jellybeanBag[req.beanName] = null;

Check your routes to make sure that they use req.beanName.

Checkpoint

**Solution:app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

// Add your code below:

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(bodyData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(400).send('Bean with that name already exists!');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    req.bean.number += numberOfBeans;

    res.send(req.bean);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    if (req.bean.number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    req.bean.number -= numberOfBeans;

    res.send(req.bean);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  req.bean = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const newName = JSON.parse(bodyData).name;

    jellybeanBag[newName] = jellybeanBag[req.beanName];

    jellybeanBag[req.beanName] = null;

    res.send(jellybeanBag[newName]);

    console.log('Response Sent');

  });

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Control Flow With next():** Notice how our middleware correctly calls next. If your if block is entered (meaning the bean does not exist), the function returns to break from the middleware. We could also achieve the same result by putting all the code after the if block in an else.

next is called at the end of the middleware callback function. This placement ensures that if a bean does not exist, the proper error status is sent, but if it does exist, we attach it to the request object and proceed to the next matching route/middleware to complete the request-response cycle.

**Solution: app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next('Bean with that name does not exist');

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    res.status(404).send('Bean with that name does not exist');

    return console.log('Response Sent');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const body = JSON.parse(bodyData);

    const beanName = body.name;

    if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

      return res.status(400).send('Bean with that name already exists!');

    }

    const numberOfBeans = Number(body.number) || 0;

    jellybeanBag[beanName] = {

      number: numberOfBeans

    };

    res.send(jellybeanBag[beanName]);

    console.log('Response Sent');

  });

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    req.bean.number += numberOfBeans;

    res.send(req.bean);

    console.log('Response Sent');

  });

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    const numberOfBeans = Number(JSON.parse(bodyData).number) || 0;

    if (req.bean.number < numberOfBeans) {

      return res.status(400).send('Not enough beans in the jar to remove!');

    }

    req.bean.number -= numberOfBeans;

    res.send(req.bean);

    console.log('Response Sent');

  });

});

app.delete('/beans/:beanName', (req, res, next) => {

  jellybeanBag[req.beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

Route-Level app.use() - Multiple Paths

We learned that app.use() takes a path parameter, but we never fully investigated what that path parameter could be. Let’s take another look at the Express documentation for app.use():

“argument: path

description: The path for which the middleware function is invoked; can be any of:

* A string representing a path.
* A path pattern.
* A regular expression pattern to match paths.
* An array of combinations of any of the above. “

So app.use() can take an array of paths! That seems like a handy way to rewrite the code from our last exercise so that we don’t have to put the same code in two different routes with different paths.

### Instructions

**1.**

Now we’ll add some more advanced middleware. You might have noticed that in each PUT and POST route, there is code that looks like this:

let bodyData = '';  
req.on('data', (data) => {  
  bodyData += data;  
});  
req.on('end', () => {  
  // ...  
});

You don’t need to worry too much about how this code works right now since we’ll eventually be replacing it with a better solution, but it is used for combining the HTTP request body into a single string. The req.on('end' .. callback will be called once the whole request has been received. We are going to move this logic to middleware so that it attaches the body to the request object once it’s fully received and then calls next.

Open a new call to app.use below the previous middleware. Make sure that it matches all routes for '/beans/' and '/beans/:beanName' using the array of routes syntax. You can leave your callback function body empty for now.

Checkpoint 2 Passed

**2.**

Now, copy the lines from the bodyData variable declaration to the end of the first req.on call into your middleware callback.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The next step will be a bit different from the routes that are already present. Add req.on('end', () => {}). Complete the callback by adding the following lines inside the body of the callback function:

if (bodyData) {  
  req.body = JSON.parse(bodyData);  
}

This will parse the request body into a JavaScript object and attach it to the request object. Finish the middleware by calling next at the end of the req.on('end') callback function outside of the if statement.

Checkpoint 4 Passed

**4.**

Now to refactor! You can remove the lines

let bodyData = '';  
req.on('data', (data) => {  
  bodyData += data;  
});

from all your routes. Then you can remove the req.on('end' ...) method calls, but you’ll need to preserve the callback functions’ internal logic. You can simply remove the lines with req.on(... and the }); line at the end of the method call. Do this for all routes that have this duplicate code.

Checkpoint 5 Passed

**5.**

To finish refactoring, you can replace all instances of JSON.parse(bodyData) in the same routes and replace them with req.body since the body has already been parsed!

Checkpoint 6 Passed

**6.**

Great job, you removed duplicate code from four routes!

SOLUTION; APP.JS

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

// Add your code below:

app.use(['/beans/', '/beans/:beanName'], (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  req.bean = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Middleware Stacks**

Recall that middleware is just a function with a specific signature, namely (req, res, next). We have, for the most part, been using anonymous function definitions for this because our middleware has only been relevant to the route invoking it. There is nothing stopping us from defining functions and using them as middleware though. That is to say:

const logging = (req, res, next) => {  
  console.log(req);  
  next();  
};  
   
app.use(logging);

is a valid and reasonable way to introduce logging throughout all paths. It is also modifiable so that you can remove the app.use() line and replace it with a specific route method, or sprinkle it throughout the application without it being universal.

Up until this point we’ve only been giving each middleware-accepting method a single callback. With modular pieces like this, it is useful to know that methods such as app.use(), app.get(), app.post(), and so on all can take multiple callbacks as additional parameters. This results in code that looks like the following:

const authenticate = (req, res, next) => {  
  ...  
};  
   
const validateData = (req, res, next) => {  
  ...  
};  
   
const getSpell = (req, res, next) => {  
  res.status(200).send(getSpellById(req.params.id));  
};  
   
const createSpell = (req, res, next) => {  
  createSpellFromRequest(req);  
  res.status(201).send();  
};  
   
const updateSpell = (req, res, next) => {  
  updateSpellFromRequest(req);  
  res.status(204).send();  
}  
   
app.get('/spells/:id', authenticate, getSpell);  
   
app.post('/spells', authenticate, validateData, createSpell);  
   
app.put('/spells/:id', authenticate, validateData, updateSpell);

In the above code sample, we created reusable middleware for authentication and data validation. We use the authenticate() middleware to verify a user is logged in before proceeding with the request and we use the validateData() middleware before performing the appropriate create or update function. Additional middleware can be placed at any point in this chain.

### Instructions

**1.**

Since we don’t need any request body for GET or DELETE routes, let’s refactor the behavior of our body-parsing middleware to use the in-route middleware stack. Start by saving the body-parsing middleware to a const variable bodyParser and removing the app.use call handling body parsing for ['/beans/', '/beans/:beanName'].

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now, insert the bodyParser as the first callback for all routes handling POST requests.

**Solution: app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

const bodyParser = (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

};

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

Route-Level app.use() - Multiple Paths

We learned that app.use() takes a path parameter, but we never fully investigated what that path parameter could be. Let’s take another look at the Express documentation for app.use():

“argument: path

description: The path for which the middleware function is invoked; can be any of:

* A string representing a path.
* A path pattern.
* A regular expression pattern to match paths.
* An array of combinations of any of the above. “

So app.use() can take an array of paths! That seems like a handy way to rewrite the code from our last exercise so that we don’t have to put the same code in two different routes with different paths.

### Instructions

**1.**

Now we’ll add some more advanced middleware. You might have noticed that in each PUT and POST route, there is code that looks like this:

let bodyData = '';  
req.on('data', (data) => {  
  bodyData += data;  
});  
req.on('end', () => {  
  // ...  
});

You don’t need to worry too much about how this code works right now since we’ll eventually be replacing it with a better solution, but it is used for combining the HTTP request body into a single string. The req.on('end' .. callback will be called once the whole request has been received. We are going to move this logic to middleware so that it attaches the body to the request object once it’s fully received and then calls next.

Open a new call to app.use below the previous middleware. Make sure that it matches all routes for '/beans/' and '/beans/:beanName' using the array of routes syntax. You can leave your callback function body empty for now.

Checkpoint 2 Passed

**2.**

Now, copy the lines from the bodyData variable declaration to the end of the first req.on call into your middleware callback.

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

The next step will be a bit different from the routes that are already present. Add req.on('end', () => {}). Complete the callback by adding the following lines inside the body of the callback function:

if (bodyData) {  
  req.body = JSON.parse(bodyData);  
}

This will parse the request body into a JavaScript object and attach it to the request object. Finish the middleware by calling next at the end of the req.on('end') callback function outside of the if statement.

Checkpoint 4 Passed

**4.**

Now to refactor! You can remove the lines

let bodyData = '';  
req.on('data', (data) => {  
  bodyData += data;  
});

from all your routes. Then you can remove the req.on('end' ...) method calls, but you’ll need to preserve the callback functions’ internal logic. You can simply remove the lines with req.on(... and the }); line at the end of the method call. Do this for all routes that have this duplicate code.

Checkpoint 5 Passed

**5.**

To finish refactoring, you can replace all instances of JSON.parse(bodyData) in the same routes and replace them with req.body since the body has already been parsed!

Checkpoint 6 Passed

**6.**

Great job, you removed duplicate code from four routes!

Checkpoint 7 Passed

**Solution: app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

// Add your code below:

app.use(['/beans/', '/beans/:beanName'], (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  req.bean = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!req.bean) {

    return res.status(404).send('Bag with that name does not exist');

  }

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  req.bean = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Middleware Stacks**

Recall that middleware is just a function with a specific signature, namely (req, res, next). We have, for the most part, been using anonymous function definitions for this because our middleware has only been relevant to the route invoking it. There is nothing stopping us from defining functions and using them as middleware though. That is to say:

const logging = (req, res, next) => {  
  console.log(req);  
  next();  
};  
   
app.use(logging);

to know that methods such as app.use(), app.get(), app.post(), and so on all can take multiple callbacks as additional parameters. This results in code that looks like the following:

const authenticate = (req, res, next) => {  
  ...  
};  
   
const validateData = (req, res, next) => {  
  ...  
};  
   
const getSpell = (req, res, next) => {  
  res.status(200).send(getSpellById(req.params.id));  
};  
   
const createSpell = (req, res, next) => {  
  createSpellFromRequest(req);  
  res.status(201).send();  
};  
   
const updateSpell = (req, res, next) => {  
  updateSpellFromRequest(req);  
  res.status(204).send();  
}  
   
app.get('/spells/:id', authenticate, getSpell);  
   
app.post('/spells', authenticate, validateData, createSpell);  
   
app.put('/spells/:id', authenticate, validateData, updateSpell);

In the above code sample, we created reusable middleware for authentication and data validation. We use the authenticate() middleware to verify a user is logged in before proceeding with the request and we use the validateData() middleware before performing the appropriate create or update function

**1.**

Since we don’t need any request body for GET or DELETE routes, let’s refactor the behavior of our body-parsing middleware to use the in-route middleware stack. Start by saving the body-parsing middleware to a const variable bodyParser and removing the app.use call handling body parsing for ['/beans/', '/beans/:beanName'].

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Now, insert the bodyParser as the first callback for all routes handling POST requests.

Checkpoint

**Solution: app.js**

const express = require('express');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use((req, res, next) => {

  console.log(`${req.method} Request Received`);

  next();

});

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    console.log('Response Sent');

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

const bodyParser = (req, res, next) => {

  let bodyData = '';

  req.on('data', (data) => {

    bodyData += data;

  });

  req.on('end', () => {

    if (bodyData) {

      req.body = JSON.parse(bodyData);

    }

    next();

  });

};

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

  console.log('Response Sent');

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

  console.log('Response Sent');

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

  console.log('Response Sent');

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

  console.log('Response Sent');

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

  console.log('Response Sent');

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Open-Source Middleware: Logging:** Express already exists as an open-source package that we can install and use to build upon. There is a huge ecosystem of Javascript packages that will solve so many of the problems that developers frequently run into.

In the workspace you’ll see what code looks like using unnecessary custom solutions and lots of lines calling console.log().

We will replace the logging code in the workspace with [morgan](https://github.com/expressjs/morgan), an open-source library for logging information about the HTTP request-response cycle in a server application. morgan() is a function that will return a middleware function, to reiterate: the return value of morgan() will be a function, that function will have the function signature (req, res, next) that can be inserted into an app.use(), and that function will be called before all following middleware functions. Morgan takes an argument to describe the formatting of the logging output. For example, morgan('tiny') will return a middleware function that does a “tiny” amount of logging. With morgan in place, we’ll be able to remove the existing logging code. Once we see how fast it is to add logging with morgan, we won’t have to spend time in the future trying to figure out how to replicate that functionality.

Require morgan at the top of the app where you import Express, and save it to a const morgan.

Checkpoint 2 Passed

**2.**

Replace your logging middleware with morgan('tiny').

Checkpoint 3 Passed

Stuck? Get a hint

**3.**

Morgan will log response codes after the response is sent, so you can get rid of all the remaining console.log statements that log 'Response Sent'. At the end of this refactor, you should only have one console.log left in your code (inside app.listen).

Checkpoint 4 Passed

**Solution: app.js**

const express = require('express');

const morgan = require('morgan');

const app = express();

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

const bodyParser = (req, res, next) => {

  let queryData = '';

  req.on('data', (data) => {

    data = data.toString();

    queryData += data;

  });

  req.on('end', () => {

    if (queryData) {

      req.body = JSON.parse(queryData);

    }

    next();

  });

};

// Logging Middleware

app.use(morgan('tiny'));

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', bodyParser, (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Documentation:** Documentation is a resource, presented by the package’s author(s), that includes information about what software is, what it does, and how to use it. We’ve seen the Express documentation in this course, and now we’re going to look at the [morgan documentation](https://github.com/expressjs/morgan#api).

### Instructions

**1.**

Morgan provides a number of pre-defined formats. Let’s change from using 'tiny' to one with a bit more information. We want one with the HTTP method, URL, status code, response time, content length, and one that changes colors of the status code output based on the code. Try to find the logging format that will create output as we expect and replace 'tiny' with that format name.

Solution: app.js

const express = require('express');

const app = express();

const morgan = require('morgan');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

const bodyParser = (req, res, next) => {

  let queryData = '';

  req.on('data', (data) => {

    queryData += data;

  });

  req.on('end', () => {

    if (queryData) {

      req.body = JSON.parse(queryData);

    }

    next();

  });

};

// Logging Middleware

app.use(morgan('dev'));

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', bodyParser, (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bean with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', bodyParser, (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Open-Source Middleware: Body Parsing**

**1.**

Our bodyParser function is okay for now, but there are bound to be edge cases and all sorts of request bodies it can’t handle well. Let’s replace it with a well-maintained open-source package, body-parser. Require 'body-parser' at the top of the app in the same way, and save it to a const bodyParser.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Remove the bodyParser middleware that you previously wrote. You can also now remove it from the middleware stacks for all PUT and POST routes. bodyParser will automatically attach the parsed body object to req.body.

Open a new app.use call directly after your morgan logging middleware. bodyParser has multiple [methods](https://github.com/expressjs/body-parser#api) for returning middleware functions. For now, let’s use bodyParser.json() to parse all request bodies in JSON format.

Solution: app.js

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Logging Middleware

app.use(morgan('dev'));

// Body parsing middleware

app.use(bodyParser.json());

app.use('/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bag with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Error-Handling Middleware:** Error handling middleware needs to be the last app.use() in your file. If an error happens in any of our routes, we want to make sure it gets passed to our error handler. The middleware stack progresses through routes as they are presented in a file, therefore the error handler should sit at the bottom of the file. How do we write it?

app.use((err, req, res, next) => {  
  console.error(err.stack);  
  res.status(500).send('Something broke!');  
});

Based on the code above, we can see that error-handling middleware is written much like other kinds of middleware. The biggest difference is that there is an additional parameter in our callback function, err. This represents the error object, and we can use it to investigate the error and perform different tasks depending on what kind of error was thrown. For now, we only want to send an HTTP 500 status response to the user.

Express has its own error-handler, which catches errors that we haven’t handled. But if we anticipate an operation might fail, we can invoke our error-handling middleware. We do this by passing an error object as an argument to next(). Usually, next() is called without arguments and will proceed through the middleware stack as expected. When called with an error as the first argument, however, it will call any applicable error-handling middleware.

app.use((req, res, next) => {  
  const newValue = possiblyProblematicOperation();  
  if (newValue === undefined) {  
    let undefinedError = new Error('newValue was not defined!');  
    return next(undefinedError);  
  }  
  next();  
});  
   
app.use((err, req, res, next) => {  
  const status = err.status || 500;  
  res.status(status).send(err.message);  
});

In this segment we assign the return value of the function possiblyProblematicOperation() to newValue.

**1.**

Add a very simple error handler as the last handler in your file, immediately before app.listen. The callback function should have four arguments. It should set the status of the response equal to the status property of the error object if it exists or set it to 500 by default. Finally, your error handler should send back the error object’s message property.

If you want to see your errors in the terminal console as you test, log out the error or it’s message property inside your error handler.

Checkpoint 2 Passed

**2.**

Now, refactor the routes that send error responses (any that are greater than or equal to 400) to use this error handler. This means instead of a line like this

return res.status(404).send('<error message>');

You should instead create a new Error with the correct error message, set its .status property, and then call next and pass in the error. Be sure to still return the next call so that the route/middleware callback breaks out and the error handler takes over.

Solution: app.js

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Body-parsing Middleware

app.use(bodyParser.json());

// Logging Middleware

if (!process.env.IS\_TEST\_ENV) {

  app.use(morgan('dev'));

}

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    const error = new Error('Bean with that name does not exist')

    error.status = 404;

    return next(error);

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    const error = new Error('Bean with that name already exists!')

    error.status = 400;

    return next(error);

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    const error = new Error('Not enough beans in the jar to remove!')

    error.status = 400;

    return next(error);

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

// Add your error handler here:

app.use((err, req, res, next) => {

  if (!err.status) {

    err.status = 500;

  }

  res.status(err.status).send(err.message);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Discovering Open-Source Middleware:**

### Instructions

**1.**

Require the error handling package that you found. Replace your catch-all error handler with the middleware created by the error handler package. You can use the default settings.

Checkpoint

**Solution: app.js**

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

const errorHandler = require('errorhandler')

app.use(errorHandler());

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const jellybeanBag = {

  mystery: {

    number: 4

  },

  lemon: {

    number: 5

  },

  rootBeer: {

    number: 25

  },

  cherry: {

    number: 3

  },

  licorice: {

    number: 1

  }

};

// Body-parsing Middleware

app.use(bodyParser.json());

// Logging Middleware

app.use(morgan('dev'));

app.use('/beans/:beanName', (req, res, next) => {

  const beanName = req.params.beanName;

  if (!jellybeanBag[beanName]) {

    return res.status(404).send('Bean with that name does not exist');

  }

  req.bean = jellybeanBag[beanName];

  req.beanName = beanName;

  next();

});

app.get('/beans/', (req, res, next) => {

  res.send(jellybeanBag);

});

app.post('/beans/', (req, res, next) => {

  const body = req.body;

  const beanName = body.name;

  if (jellybeanBag[beanName] || jellybeanBag[beanName] === 0) {

    return res.status(400).send('Bean with that name already exists!');

  }

  const numberOfBeans = Number(body.number) || 0;

  jellybeanBag[beanName] = {

    number: numberOfBeans

  };

  res.send(jellybeanBag[beanName]);

});

app.get('/beans/:beanName', (req, res, next) => {

  res.send(req.bean);

});

app.post('/beans/:beanName/add', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  req.bean.number += numberOfBeans;

  res.send(req.bean);

});

app.post('/beans/:beanName/remove', (req, res, next) => {

  const numberOfBeans = Number(req.body.number) || 0;

  if (req.bean.number < numberOfBeans) {

    return res.status(400).send('Not enough beans in the jar to remove!');

  }

  req.bean.number -= numberOfBeans;

  res.send(req.bean);

});

app.delete('/beans/:beanName', (req, res, next) => {

  const beanName = req.beanName;

  jellybeanBag[beanName] = null;

  res.status(204).send();

});

app.put('/beans/:beanName/name', (req, res, next) => {

  const beanName = req.beanName;

  const newName = req.body.name;

  jellybeanBag[newName] = req.bean;

  jellybeanBag[beanName] = null;

  res.send(jellybeanBag[newName]);

});

app.use((err, req, res, next) => {

  res.status(500).send(err);

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

### Instructions

**1.**

There’s something missing from the custom body-parsing middleware function, fix it to move on.

Checkpoint 2 Passed

**2.**

Now our app should work, but let’s replace the custom body-parsing with the body-parser package. Use bodyParser.json() for all routes.

Checkpoint 3 Passed

**3.**

The following existence checking logic appears at all /cards/:cardId routes. Refactor it to a middleware function that matches all /cards/:cardId routes. If the card exists, add it to the req object as req.cardIndex and refactor routes to use req.cardIndex where necessary.

const cardId = Number(req.params.cardId);  
const cardIndex = cards.findIndex(card => card.id === cardId);  
if (cardIndex === -1) {  
  return res.status(404).send('Card not found');  
}

Checkpoint 4 Passed

**4.**

You probably noticed that these lines get replicated for POST and PUT requests

const newCard = req.body;  
const validSuits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'];  
const validRanks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace'];  
if (validSuits.indexOf(newCard.suit) === -1 || validRanks.indexOf(newCard.rank) === -1) {  
  return res.status(400).send('Invalid card!');  
}

Create a custom middleware function named validateCard that replicated this logic. Use it in the middleware stack for the POST and PUT routes.

Checkpoint 5 Passed

**5.**

Congratulations! You’ve fixed the broken functionality and greatly increased the code quality using your middleware skills!

Checkpoint

**Solution: app.js**

const express = require('express');

const app = express();

const morgan = require('morgan');

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const cards = [

  {

    id: 1,

    suit: 'Clubs',

    rank: '2'

  },

  {

    id: 2,

    suit: 'Diamonds',

    rank: 'Jack'

  },

  {

    id: 3,

    suit: 'Hearts',

    rank: '10'

  }

];

let nextId = 4;

// Logging

if (!process.env.IS\_TEST\_ENV) {

  app.use(morgan('short'));

}

// Parsing

app.use(bodyParser.json());

// Find card

app.use('/cards/:cardId', (req, res, next) => {

  const cardId = Number(req.params.cardId);

  const cardIndex = cards.findIndex(card => card.id === cardId);

  if (cardIndex === -1) {

    return res.status(404).send('Card not found');

  }

  req.cardIndex = cardIndex;

  next();

});

const validateCard = (req, res, next) => {

  const newCard = req.body;

  const validSuits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'];

  const validRanks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace'];

  if (validSuits.indexOf(newCard.suit) === -1 || validRanks.indexOf(newCard.rank) === -1) {

    return res.status(400).send('Invalid card!');

  }

  next();

};

// Get all Cards

app.get('/cards/', (req, res, next) => {

  res.send(cards);

});

// Create a new Card

app.post('/cards/', validateCard, (req, res, next) => {

  const newCard = req.body;

  newCard.id = nextId++;

  cards.push(newCard);

  res.status(201).send(newCard);

});

// Get a single Card

app.get('/cards/:cardId', (req, res, next) => {

  res.send(cards[req.cardIndex]);

});

// Update a Card

app.put('/cards/:cardId', validateCard, (req, res, next) => {

  const newCard = req.body;

  const cardId = Number(req.params.cardId);

  if (!newCard.id || newCard.id !== cardId) {

    newCard.id = cardId;

  }

  cards[req.cardIndex] = newCard;

  res.send(newCard);

});

// Delete a Card

app.delete('/cards/:cardId', (req, res, next) => {

  cards.splice(req.cardIndex, 1);

  res.status(204).send();

});

// Start the server

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**ROUTER PARAMETERS: Introduction**

When building interfaces with Express, we will run into the pattern of expecting certain information in a requested URL and using that information to identify the data that is being requested. To give an example:

app.get('/sorcerers/:sorcererName', (req, res, next) => {  
  const sorcerer = Sorcerers[req.params.sorcererName];  
  res.send(sorcerer.info);  
});  
   
app.get('/sorcerers/:sorcererName/spellhistory', (req, res, next) => {  
  const sorcerer = Sorcerers[req.params.sorcererName];  
  const spellHistory = Spells[sorcerer.id].history;  
  res.send(spellHistory);  
});

In the above code we need to extract the request parameter :sorcererName from the url in both instances, and end up duplicating the necessary code so that it appears in both routes. When working with routes that require parameters, we might find ourselves in a position where multiple different routes require the same parameter and use it to identify the same piece of data. While writing this duplicate code will get the job done, copy-and-pasting functionality does leave a bitter taste in the mouth of the principled developer. We should investigate if there is a better way to accomplish this.

**router.param():** When a specific parameter is present in a route, we can write a function that will perform the necessary lookup and attach it to the req object in subsequent middleware that is run.

app.param('spellId', (req, res, next, id) => {  
  let spellId = Number(id);  
    try {  
      const found = SpellBook.find((spell) => {  
        return spellId === spell.id;  
      })  
      if (found) {  
        req.spell = found;  
        next();  
      } else {  
        next(new Error('Your magic spell was not found in any of our tomes'));  
      };  
    } catch (err) {  
      next(err)  
    }  
});

In the code above we intercept any request to a route handler with the :spellId parameter. Note that in the app.param function signature, 'spellId' does not have the leading :. The actual ID will be passed in as the fourth argument, id in this case, to the app.param callback function when a request arrives.

We look up the spell in our SpellBook array using the .find() method. If SpellBook does not exist, or some other error is thrown in this process, we pass the error to the following middleware (hopefully we’ve written some error-handling middleware, or included some externally-sourced error-handling middleware). If the spell exists in SpellBook, the .find() method will store the spell in found, and we attach it as a property of the request object (so future routes can reference it via req.spell). If the requested spell does not exist, .find() will store undefined in found, and we let future middlewares know there was an error with the request by creating a new Error object and passing it to next().

Note that inside an app.param callback, you should use the fourth argument as the parameter’s value, not a key from the req.params object.

### Instructions

**1.**

Let’s refactor this code to use app.param for all /spices/:spiceId routes.

First, start your code with a call to app.param. Write functionality that will look for the spiceIndex and attach it to the req object as req.spiceIndex if it exists. Call next after that. If it does not exist, send a 404 error response and do not call next.

Checkpoint 2 Passed

**2.**

Now, refactor your current code to get rid of any index-checking logic in /spices/:spiceId routes. Use req.spiceIndex to do any necessary operations on the spiceRack object. You should also get rid of anything that would send an error response, as non-existent ids will already have been handled by router.param.

Solution: app.js

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const spiceRack = [

  {

    id: 1,

    name: 'cardamom',

    grams: 45,

  },

  {

    id: 2,

    name: 'pimentn',

    grams: 20,

  },

  {

    id: 3,

    name: 'cumin',

    grams: 450,

  },

  {

    id: 4,

    name: 'sichuan peppercorns',

    grams: 107,

  }

];

let nextSpiceId = 5;

app.use(bodyParser.json());

// Add your code here:

app.param('spiceId', (req, res, next, id) => {

  const spiceId = Number(id);

  const spiceIndex = spiceRack.findIndex(spice => spice.id === spiceId);

  if (spiceIndex !== -1){

    req.spiceIndex = spiceIndex;

    next();

  } else {

    res.sendStatus(404);

  }

})

app.get('/spices/', (req, res, next) => {

  res.send(spiceRack);

});

app.post('/spices/', (req, res, next) => {

  const newSpice = req.body.spice;

  if (newSpice.name  && newSpice.grams) {

    newSpice.id = nextSpiceId++;

    spiceRack.push(newSpice);

    res.send(newSpice);

  } else {

    res.status(400).send();

  }

});

app.get('/spices/:spiceId', (req, res, next) => {

  res.send(spiceRack[req.spiceIndex]);

});

app.put('/spices/:spiceId', (req, res, next) => {

  spiceRack[req.spiceIndex] = req.body.spice;

  res.send(spiceRack[req.spiceIndex]);

});

app.delete('/spices/:spiceId', (req, res, next) => {

  spiceRack.splice(req.spiceIndex, 1);

  res.status(204).send();

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

**Merge Parameters:** This would be a router that is invoked within another router. In order to pass parameters the parent router has access to, we pass a special configuration object when defining the router.

const sorcererRouter = express.Router();  
const familiarRouter = express.Router({mergeParams: true});  
   
sorcererRouter.use('/:sorcererId/familiars', familiarRouter);  
   
sorcererRouter.get('/', (req, res, next) => {  
  res.status(200).send(Sorcerers);  
  next();  
});  
   
sorcererRouter.param('sorcererId', (req, res, next, id) => {  
  const sorcerer = getSorcererById(id);     
  req.sorcerer = sorcerer;  
  next();  
});  
   
familiarRouter.get('/', (req, res, next) => {  
  res.status(200).send(`Sorcerer ${req.sorcerer} has familiars ${getFamiliars(sorcerer)}`);  
});  
   
app.use('/sorcerer', sorcererRouter);

In the code above we define two endpoints: /sorcerer and /sorcerer/:sorcererId/familiars. The familiars are nested into the sorcerer endpoint — indicating the relationship that a sorcerer has multiple familiars. Take careful note of the {mergeParameters: true} argument that gets passed when creating the familiarRouter. This argument tells Express that the familiarRouter should have access to parents passed into its parent router, that is, the sorcererRouter. We then tell express that the path for the familiarRouter is the same as the path for the sorcererRouter with the additional path /:sorcererId/familiars. We then can create a family of routes (a router) built by appending routes to familiarRouter‘s base: /sorcerer/:sorcererId/familiars.

### Instructions

**1.**

Let’s make our spices API more flexible and allow spices to be associated with different spice racks. The goal for this exercise will be to ensure that when new spices are created or updated, they will be associated with the correct spice rack.

In the workspace, you have a new root **app.js** file and a **spicesRouter.js** with code from the last exercise. **app.js** will handle interactions retrieving, creating, updating, and deleting spice racks, and **spicesRouter.js** will be nested to handle individual spices with the spice racks. Each file has a param method call (app.param in **app.js**, router.param in **spicesRouter.js**).

To begin, let’s hook the router up to the main application. At the end of **app.js**, use the spicesRouter for all /spice-racks/:spiceRackId/spices routes.

Checkpoint 2 Passed

**2.**

Now, let’s make sure that the spicesRouter is merging parameters from parent **app.js** Express instance. Add the proper options to the .Router() method at the top of your **spicesRouter.js** file.

Checkpoint 3 Passed

**3.**

Okay, now let’s make sure that newly created spices inside **spicesRouter.js** are associated with the correct spice rack. Inside your .post() route, make sure to set the newSpice.spiceRackId equal to the req.params.spiceRackId that the parent router attached if mergeParams has performed as expected. Don’t forget to convert the spiceRackId to a number before attaching it. Make sure to set this before it is pushed onto the spices array.

Solution: app.js

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

app.use(bodyParser.json());

const spiceRacks = [

  {

    id: 1,

    location: 'Kitchen Countertop',

  },

  {

    id: 2,

    location: 'Pantry',

  },

  {

    id: 3,

    location: 'Outdoor Barbecue',

  }

];

let nextSpiceRackId = 4;

app.param('spiceRackId', (req, res, next, id) => {

  const idToFind = Number(id);

  const rackIndex = spiceRacks.findIndex(spiceRack => spiceRack.id === idToFind);

  if (rackIndex !== -1) {

    req.spiceRack = spiceRacks[rackIndex];

    req.spiceRackIndex = rackIndex;

    next();

  } else {

    res.status(404).send('Spice Rack Not Found.');

  }

});

app.get('/spice-racks/', (req, res, next) => {

 res.send(spiceRacks);

});

app.post('/spice-racks/', (req, res, next) => {

  const newRack = req.body.spiceRacks;

  newRack.id = nextSpiceRackId++;

  spiceRacks.push(newRack);

  res.send(newRack);

});

app.get('/spice-racks/:spiceRackId', (req, res, next) => {

  res.send(req.spiceRack);

});

app.put('/spice-racks/:spiceRackId', (req, res, next) => {

  const updatedRack = req.body.spiceRack;

  if (req.spiceRack.id !== updatedRack.id) {

    res.status(400).send('Cannot update Spice Rack Id');

  } else {

    spiceRacks[req.spiceRackIndex] = updatedRack;

    res.send(spiceRacks[req.spiceRackIndex]);

  }

});

app.delete('/spice-racks/:spiceRackId', (req, res, next) => {

  spiceRacks.splice(req.spiceRackIndex, 1);

  res.status(204).send();

});

const spicesRouter = require('./spicesRouter');

// Write your code below:

app.use('/spice-racks/:spiceRackId/spices', spicesRouter);

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

Review

router.param is a powerful tool that we can use to keep our code from repeating core functionality through routes. This is a pattern we want to frequently follow: identify multiple pieces of code that accomplish the same goal, put it into a single component, let that component do that thing (and update it when we want the thing it does to change — in a single place).

Let’s try applying that knowledge again, to another codebase. If you look at the workspace you’ll find the same problem of data-lookup happening, based on a URL parameter, multiple times in different places. Try combining that logic in a single place using router.param.

### Instructions

**1.**

Refactor the current application to use an app.param to handle all routes with snackId. It should set the req.snackIndex if it exists and send the proper 404 response if not. Make sure to fix all routes to use the req.snackIndex and remove duplicate code.

Checkpoint 2 Passed

**solution: app.js**

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

app.use(express.static('public'));

const PORT = process.env.PORT || 4001;

const vendingMachine = [

  {

    id: 1,

    name: 'Gum',

    price: 1.25,

  },

  {

    id: 7,

    name: 'Bag of chips',

    price: 3.50,

  },

  {

    id: 23,

    name: 'cumin',

    price: .75,

  }

];

let nextSnackId = 24;

app.use(bodyParser.json());

// Add your code here:

app.param('snackId', (req, res, next, id) => {

  const snackId = Number(id);

  const snackIndex = vendingMachine.findIndex(snack => snack.id === snackId);

  if (snackIndex === -1) {

    res.status(404).send('Snack not found!');

  } else {

    req.snackIndex = snackIndex;

    next();

  }

});

app.get('/snacks/', (req, res, next) => {

  res.send(vendingMachine);

});

app.post('/snacks/', (req, res, next) => {

  const newSnack = req.body;

  if (!newSnack.name || !newSnack.price) {

    res.status(400).send('Snack not found!');

  } else {

    newSnack.id = nextSnackId++;

    vendingMachine.push(newSnack);

    res.send(newSnack);

  }

});

app.get('/snacks/:snackId', (req, res, next) => {

  res.send(vendingMachine[req.snackIndex]);

});

app.put('/snacks/:snackId', (req, res, next) => {

  vendingMachine[req.snackIndex] = req.body;

  res.send(vendingMachine[req.snackIndex]);

});

app.delete('/snacks/:snackId', (req, res, next) => {

  vendingMachine.splice(req.snackIndex, 1);

  res.status(204).send();

});

app.listen(PORT, () => {

  console.log(`Server is listening on port ${PORT}`);

});

# What is CORS?

A request for a resource (like an image or a font) outside of the origin is known as a *cross-origin* request. CORS (cross-origin resource sharing) manages cross-origin requests.

Once again, consider the following URL:

http://www.example.com/foo-bar.html

Let’s call it *URL1* (for short).

Unlike same-origin, navigating to **https://www.ejemplo.com/hola.html** from **URL1** could be allowed with CORS. Allowing cross-origin requests is helpful, as many websites today load resources from different places on the Internet (stylesheets, scripts, images, and more).

Cross-origin requests, however, mean that servers must implement ways to handle requests from origins outside of their own. CORS allows servers to specify who (i.e., which origins) can access the assets on the server, among many other things.

You can think of these interactions as a building with a security entrance. For example, if you need to borrow a ladder, you could ask a neighbor in the building who has one. The building’s security would likely not have a problem with this request (i.e., same-origin). If you needed a particular tool, however, and you ordered it from an outside source like an online marketplace (i.e., cross-origin), the security at the entrance may request that the delivery person provide identification when your tool arrives.

### Why is CORS necessary?

The CORS standard is needed because it allows servers to specify not only who can access the assets, but also how they can be accessed.

Cross-origin requests are made using the standard HTTP request methods. Most servers will allow **GET** requests, meaning they will allow resources from external origins (say, a web page) to read their assets. HTTP requests methods like **PATCH**, **PUT**, or **DELETE**, however, may be denied to prevent malicious behavior. For many servers, this is intentional. For example, it is likely that server A does not want servers B, C, or D to edit or delete its assets.

With CORS, a server can specify who can access its assets and which HTTP request methods are allowed from external resources.

### How does CORS manage requests from external resources?

An HTTP header is a piece of information associated with a request or a response. Headers are passed back and forth between your web browser (also referred to as a client) and a server when the web page you are on wants to use resources hosted on a different server. Headers are used to describe requests and responses. The CORS standard manages cross-origin requests by adding new HTTP headers to the standard list of headers. The following are the new HTTP headers added by the CORS standard:

* **Access-Control-Allow-Origin**
* **Access-Control-Allow-Credentials**
* **Access-Control-Allow-Headers**
* **Access-Control-Allow-Methods**
* **Access-Control-Expose-Headers**
* **Access-Control-Max-Age**
* **Access-Control-Request-Headers**
* **Access-Control-Request-Method**
* **Origin**

These are all important, but let’s focus on the following header:

* **Access-Control-Allow-Origin**

The **Access-Control-Allow-Origin** header allows servers to specify how their resources are shared with external domains. When a **GET** request is made to access a resource on Server A, Server A will respond with a value for the **Access-Control-Allow-Origin** header. Many times, this value will be **\***, meaning that Server A will share the requested resources with any domain on the Internet. Other times, the value of this header may be set to a particular domain (or list of domains), meaning that Server A will share its resources with that specific domain (or list of domains). The **Access-Control-Allow-Origin** header is critical to resource security.

You can find a description of each CORS header at the following: [CORS Headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers#CORS).

### Pre-flight Requests

As mentioned before, most servers will allow **GET** requests but may block requests to modify resources on the server. Servers don’t just blindly block such requests though; they have a process in place that first checks and then communicates to the client (your web browser) which requests are allowed.

When a request is made using any of the following HTTP request methods, a standard preflight request will be made before the original request.

* **PUT**
* **DELETE**
* **CONNECT**
* **OPTIONS**
* **TRACE**
* **PATCH**

Preflight requests use the **OPTIONS** header. The preflight request is sent before the original request, hence the term “preflight.” The purpose of the preflight request is to determine whether or not the original request is safe (for example, a **DELETE** request). The server will respond to the preflight request and indicate whether or not the original request is safe. If the server specifies that the original request is safe, it will allow the original request. Otherwise, it will block the original request.

The request methods above aren’t the only thing that will trigger a preflight request. If any of the headers that are automatically set by your browser (i.e., user agent) are modified, that will also trigger a preflight request.

### How do I implement CORS?

Implementing the request headers to set up CORS correctly depends on the language and framework of the backend.

For example, if you are using Node, you can use **setHeader()**, as shown below:

response.setHeader('Content-Type', 'text/html');

If you are using Express, you can use CORS middleware:

$ npm install cors

var express = require('express');  
var cors = require('cors');  
var app = express();  
   
app.use(cors());  
   
app.get('/hello/:id', function (req, res, next) {  
  res.json({msg: 'Hello world, we are CORS-enabled!'});  
});  
   
app.listen(80, function () {  
  console.log('CORS-enabled web server is listening on port 80');  
});

### Conclusion

There are many resource sharing solutions for all web technologies. The overall concepts, however, will always be the same. By understanding security policies like CORS, you can understand how risky behavior — like downloading assets from external origins — are mitigated.

### What is a security policy?

Servers are used to host web pages, applications, images, fonts, and much more. When you use a web browser, you are likely attempting to access a distinct website (hosted on a server). Websites often request these hosted resources from different locations (servers) on the Internet. Security policies on servers mitigate the risks associated with requesting assets hosted on different server. Let’s take a look at an example of a security policy: same-origin.

The same-origin policy is very restrictive. Under this policy, a document (i.e., like a web page) hosted on server A can only interact with other documents that are also on server A. In short, the same-origin policy enforces that documents that interact with each other have the same origin.

An origin is made up of the following three parts: the protocol, host, and port number. The details of these individual parts aren’t necessary at the moment, but it is important to illustrate how the same-origin policy uses these parts.

Consider the following URL:

http://www.example.com/foo-bar.html

Let’s call it **URL1** (for short).

If you used a web browser to navigate from **URL1** to **http://www.example.com/hello-world.html**, you would be allowed to do so because the protocol (HTTP), host (example.com), and port (80) of each URL match one another. (Port 80 is the default port.) The same-origin policy requires that all parts of the origin match.

Navigating to **https://www.en.example.com/hello.html** from URL1, however, would not be allowed because of the different protocol (HTTPS) and host (en.example.com).

As you can see, not having a security policy can be risky, but a security policy like same-origin is a bit too restrictive. Thankfully, there are security policies that strike a mix of both, like cross-origin, which has evolved into the cross-origin resource sharing standard, often abbreviated as CORS.

How does CORS manage requests from external resources?

Correct! This is how the CORS standard manages cross-origin requests.by adding new HTTP header to the standard lists of header.(eg: Access- Control-Allow-Origin).

What is one disadvantage of using the module **swagger-node-express**?

Correct! Unfortunately using this module requires a manual setup and you need to add quite a few paths on your server on top of copying the Swagger UI manually into your source code. The answer is Setup.

can be overwhelming because the back-end has a lot of different parts, and different websites or web applications can have dramatically different back-ends. We covered a lot in this lesson, so let’s review what we learned:

* The front-end of a website or application consists of the HTML, CSS, JavaScript, and static assets sent to a client, like a web browser.
* A web server is a process running on a computer somewhere that listens for incoming requests for information over the internet and sends back responses.
* Storing, accessing, and manipulating data is a large part of a web application’s back-end
* Data is stored in databases which can be relational databases or NoSQL databases.
* The server-side of a web application, sometimes called the application server, handles important tasks such as authorization and authentication.
* The back-end of web application often has a web API which is a way of interacting with an application’s data through HTTP requests and responses.
* Together the technologies used to build the front-end and back-end of a web application are known as the stack, and many different languages and frameworks can be used to build a robust back-end.

Now that you have a sense for server-side web development and what the back-end is, you’re ready to dive in and learn about the different parts in more depth!