**REDUX (Udacity Nanodegree)**

<https://youtu.be/zSHjvdQ7nZ8>

We've broken down your path to learning Redux into the following lessons:

* Lesson 1 - Managing State
* Lesson 2 - UI + Redux
* Lesson 3 - Redux Middleware
* Lesson 4 - Redux with React
* Lesson 5 - Asynchronous Redux
* Lesson 6 - react-redux
* Lesson 7 - Real World Redux

**Lesson 1 - Managing State**

You’ll learn techniques to make your state more **predictable** by moving your state to a **central** **location** and establishing strict rules for **getting, listening, and updating** that state.

**Lesson 2 - UI + Redux**

You’ll learn to move away from having state live in the DOM by creating a vanilla JavaScript application whose state is managed entirely by Redux.

**Lesson 3 - Redux Middleware**

You’ll learn to create custom middleware and add it to your Redux store. This middleware will allow you to **enhance your store by hooking into and intercepting actions before they reach any reducers**.

**Lesson 4 - Redux with React**

You’ll learn how to add React to a Redux application and have the state of that application be managed by Redux.

**Lesson 5 - Asynchronous Redux**

You’ll learn to better abstract asynchronous events by creating your own custom **Thunk** **middleware** and adding it to your store.

**Lesson 6 - react-redux**

You’ll learn to leverage the react-redux bindings in order to leverage the benefits of a UI library like React and a state management library like Redux.

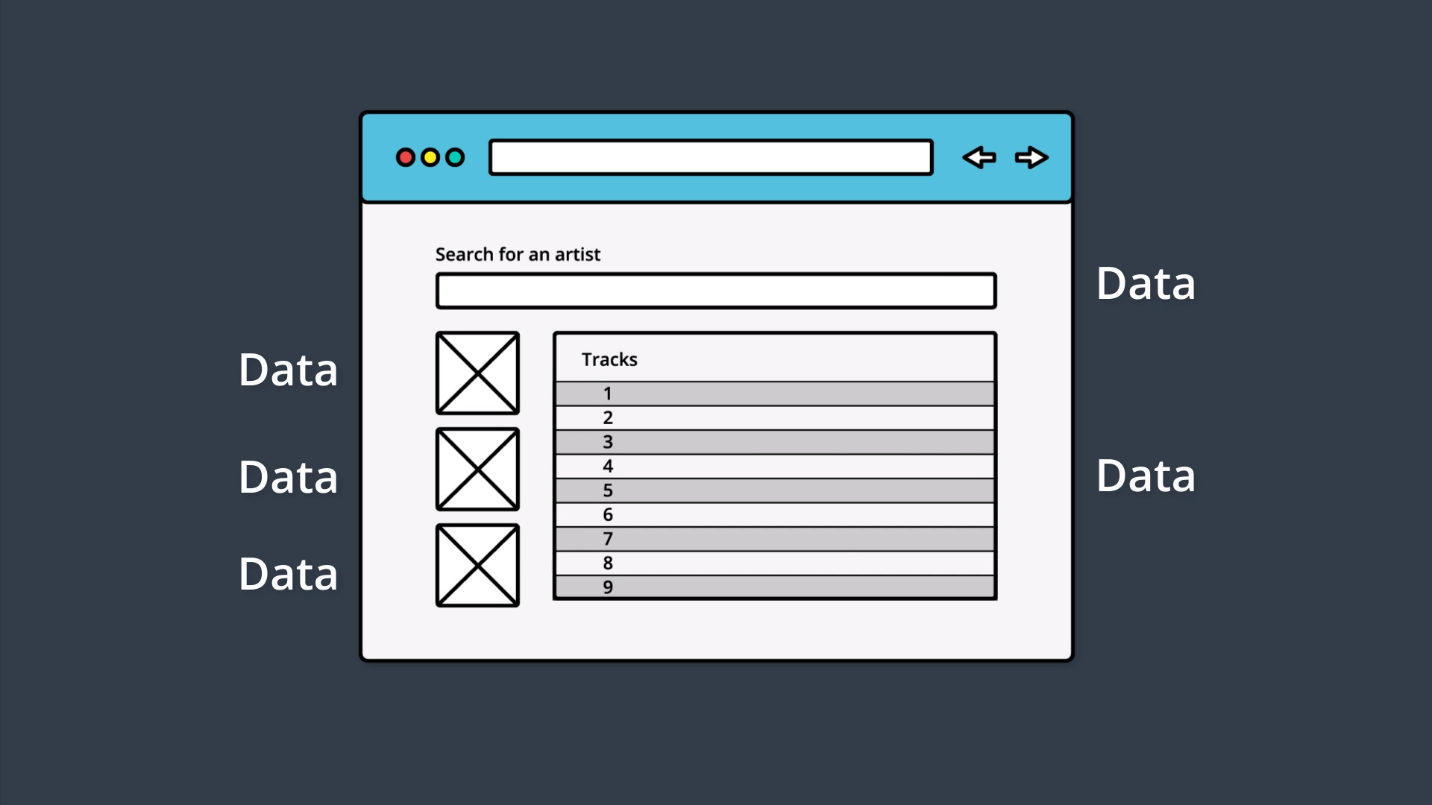
**Lesson 7 - Real World Redux**

You’ll take your knowledge of Redux and utilize it by building a real world Redux application. You’ll also learn advanced Redux topics like reducer composition and normalization.

**The Store**

<https://youtu.be/A1ZW3cRS65g>

A traditional app might look something like this:



**The application's data is sprinkled throughout the app.**

Notice in the image above, that this simple application has a lot of state:

* There are the images in the sidebar on the left.
* There are rows of tracks in the main area.
* Each Track will have its own information that it's maintaining.
* There's the search field at the top that introduces new state to the app (the searched for artist/track information).

And this is just one, simple page of this application. In most sites you use, there is information littered throughout every single page of the entire app.

Remember that the main goal of Redux is to make the state management of an application more predictable. Let's see what that might look like:



**Application data is stored outside of the app and is just referenced by the app.**

In this example, the app appears exactly the same to the end user, however, it's functioning quite differently under the hood. **All of the data is stored outside of the UI code and is just referenced from the UI code.**

With a change like this, if the data needs to be modified at all, **then all of the data is located in one place and needs to be only changed once. Then the areas of the app that are referencing pieces of data, will be updated since the source they're pulling from has changed.**

<https://youtu.be/IDdb6baBQyo>

**State Tree**

***One of the key points of Redux is that all of the data is stored in a single object called the state tree.***

But what does a state tree actually look like? Good question! Here's an example:

*{*

*recipes: [*

*{ … },*

*{ … },*

*{ … }*

*],*

*ingredients: [*

*{ … },*

*{ … },*

*{ … },*

*{ … },*

*{ … },*

*{ … }*

*],*

*products: [*

*{ … },*

*{ … },*

*{ … },*

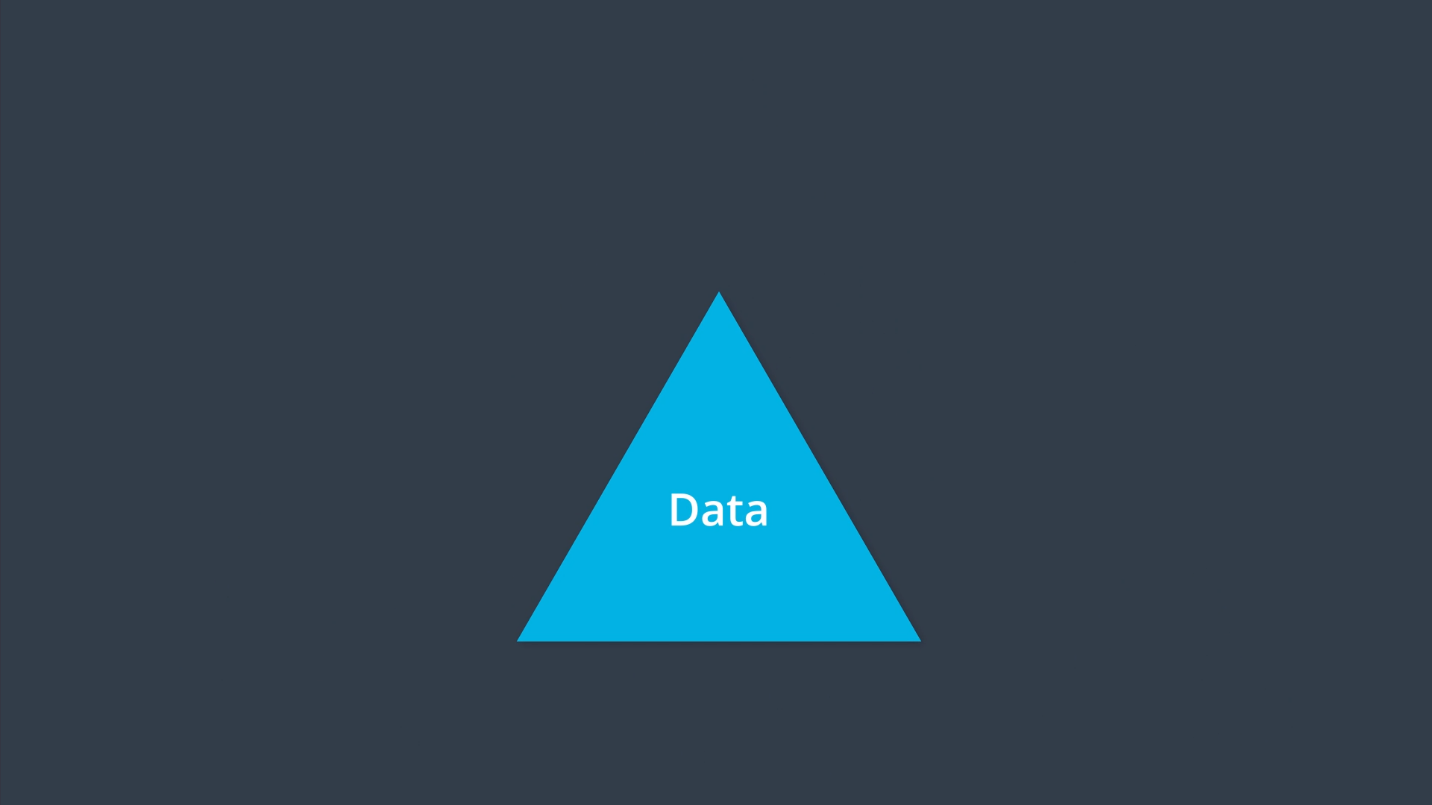
*{ … }*

*]*

*}*

See how **all of the data** for this imaginary cooking site is **stored in a single object?** So all of the **state (or "application data")** for this site is **stored** **in one, single location**. This is what we mean when we say **"state tree"…**. it's just all of the **data stored in a single object**.

Throughout this course, whenever we refer to an application's "state tree", we'll use a triangle to convey this concept.



<https://youtu.be/o8cEkLqR7VU>

**Summary**

In this lesson, we looked at the data in an application. We saw that **in traditional apps, the data is mixed in with the UI and markup**. This can lead to hard-to-find bugs where **updating the state in one location doesn't update it in every location.**

We learned that the **main goal that Redux is trying to offer is predictable state management.**

The way that Redux tries to accomplish this is through **having a single state tree.** **This state tree is an object that stores the entire state for an application**. Now that all state is stored in one location, we discovered three ways to interact with it:

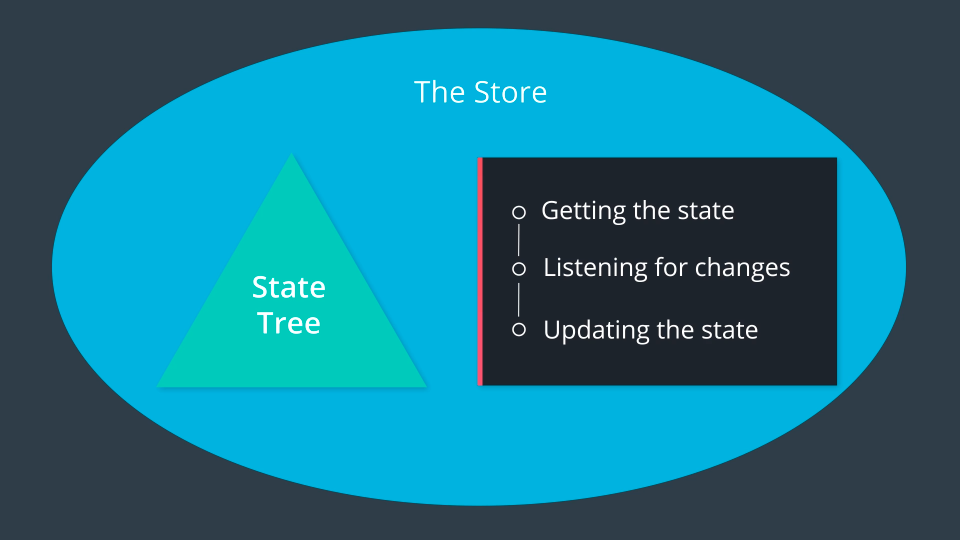
* getting the state
* listening for changes to the state
* updating the state

Then we **combine the three items above and the state tree object itself into one unit** which we called the **Store**. We'll look at creating this store in the next lesson.

**Create Store: Getting and Listening**

The store has the following information:

* the state tree
* a way to get the state tree
* a way to listen and respond to the state changing
* a way to update the state



**The Store contains the state tree and provides ways to interact with the state tree.**

So this is what we're going to do in this lesson - we're going to **actually create the store code ourselves, from scratch.**

In the following video(s), we'll start with a blank index.js file and create a factory function that creates store objects. Then we'll have the store keep track of the state, and we'll write the method to get the state from the store.

<https://youtu.be/YqmnAPNCxkQ>

<https://youtu.be/AWOuF_qoEh8>

<https://youtu.be/5jVn0L7nlBA>

<https://youtu.be/-MtD_RCqKK4>

We've got our first rule!

*“Only an event can change the state of the store.”*

Ok...well, without knowing what an "event" is, this rule is less than helpful :-\ Fear not, because we're going to look at what events are in this video:

<https://youtu.be/4SSkRoVunbI>

When an **event** takes place in a **Redux** application, we use a **plain JavaScript object** to keep track of what the specific event was. This object is called an **Action**.

Let's take another look at an Action:

*{*

*type: "ADD\_PRODUCT\_TO\_CART"*

*}*

As you can see, **an Action is clearly just a plain JavaScript object**. What makes this plain JavaScript object special in Redux, is that **every Action must have a type property**. **The purpose of the type property is to let our app (Redux) know exactly what event just took place.** This Action tells us that a product was added to the cart. That's incredibly descriptive and quite helpful, isn't it?

Now, since an Action is just a regular object, **we can include extra data about the event** that took place:

*{*

*type: "ADD\_PRODUCT\_TO\_CART",*

*productId: 17*

*}*

**In this Action, we're including the productId field**. Now we know exactly which product was added to the store!

One more note to keep in mind as you build your Action objects: it's better practice to **pass as little data as possible in each action.** That is, prefer passing the index or ID of a product rather than the entire product object itself.

Action Creators are functions that **create/return** action objects. For example:

*const addItem = item => ({*

*type: ADD\_ITEM,*

*item*

*});*

**or in ES5:**

*var addItem = function addItem(item) {*

*return {*

*type: ADD\_ITEM,*

*item: item*

*};*

*};*

**Updating State**

Let's step back one more time and think about what Redux is all about. **The whole goal of Redux is to increase predictability:**

**Redux is a predictable state container for JavaScript apps**.

With this in mind, let's see dig into how we can use **actions** and our state tree to predictably manage an application's state.

<https://youtu.be/15sTwJsyWbU>

And we've got our **second** rule!

The function that returns the new state needs to be a **pure function.**

So far, our rules are:

1. Only an event can change the state of the store.
2. The function that returns the new state needs to be a pure function.

A pure function can be a bit theoretical, so we'll take it step by step and explain why a pure function is so powerful and how it helps improve predictability.

<https://youtu.be/o9cWPrOMuyU>

**Remember: pure functions are predictable.**

**What are Pure Functions?**

Pure functions are integral to how state in Redux applications is updated. By definition, pure functions:

* Return the same result if the same arguments are passed in
* Depend solely on the arguments passed into them
* Do not produce side effects, such as API requests and I/O operations

Let’s check out an example of a pure function, square():

*const square = x => x \* x;*

square() is a pure function because it outputs the same value every single time, given that the same argument is passed into it. There is no dependence on any other values to produce that result, and we can safely expect just that result to be returned -- no side effects (more on this in a bit!).

On the other hand, let’s check out an example of an impure function, calculateTip():

**// `calculateTip()` is an impure function**

*const tipPercentage = 0.15;*

*const calculateTip = cost => cost \* tipPercentage;*

calculateTip() calculates and returns a number value. However, **it relies on a variable (tipPercentage) that lives outside the function to produce that value.** Since it fails one of the requirements of pure functions, calculateTip() is an impure function.

However, we could convert this function to a pure function **by passing in the outside variable, tipPercentage, as a second argument to this function!**

const calculateTip = (cost, tipPercentage = 0.15) => cost \* tipPercentage;

**Why Pure Functions Are Great**

For our purposes, the most important feature of a pure function is that it's **predictable**.

**If we have a function that takes in our state and an action that occurred, the function should (if it's pure!) return the exact same result every single time.**

<https://youtu.be/QU_WvPaC6cM>

<https://youtu.be/z5yJhTOxmMU>

* **Action:** the action represents the different events that will change the state of our store.
* **Reducer:** the reducer is a function which takes in current state and an action which occurred and returns a new state.
* **createStore:** creates the actual store.

1. state tree
2. getting the state
3. listening for changes(subscribe function)
4. update the state

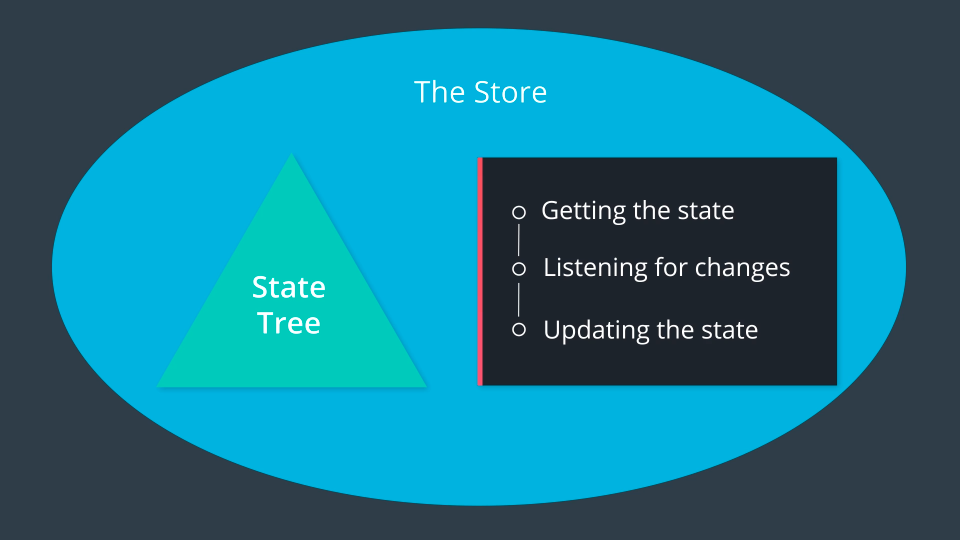
<https://youtu.be/wIyRfRSpvDo>

<https://youtu.be/P09BK4IXzmk>

* dispatch() is called with an Action
* the reducer that was passed to createStore() is called with the current state tree and the action...this updates the state tree
* because the state has (potentially) changed, all listener functions that have been registered with the subscribe() method are called

**Putting it all together**

<https://youtu.be/HEQR3KYjG24>



We've finally finished creating the **createStore** function! Using the image above as a guide, let's break down what we've accomplished:

we created a function called **createStore()** that returns a store object

* createStore() must be passed a "reducer" function when invoked
* the store object has three methods on it:
  + .getState() - used to get the current state from the store
  + .subscribe() - used to provide a listener function the store will call when the state changes
  + .dispatch() - used to make changes to the store's state
* **the store object's methods have access to the state of the store via closure**

**Managing more State**

As of right now, our code is handling the **ADD\_TODO** action. There are still a couple more actions that our app is supposed to be able to handle:

1. the REMOVE\_TODO action
2. the TOGGLE\_TODO action

<https://youtu.be/Yqeks3OSY6M>

<https://youtu.be/kPYmzsY2RAo>

we now have two reducer functions:

* todos
* goals

However, the createStore() function we built can only handle a single reducer function:

// createStore takes one reducer function as an argument

const store = createStore(todos);

We can't call createStore() passing it two reducer functions:

// this will not work

const store = createStore(todos, goals);

So we've got a problem...

We need a Root Reducer that combines existing reducers

<https://youtu.be/QTNV7BP7dWs>

<https://youtu.be/qL0HB_kmiQ0>

Whenever **dispatch** is called, we invoke our app function. The app function will then invoke the todos reducer as well as the goals reducer. Those will return their specific portions of the state. And then, the app function will return a state object with a todos property (the value of which is what the todos reducer returned) and a goals property (the value of which is what the goals reducer returned).

*function todos (state = [], action) {*

*switch(action.type) {*

*case 'ADD\_TODO' :*

*return state.concat([action.todo])*

*case 'REMOVE\_TODO' :*

*return state.filter((todo) => todo.id !== action.id)*

*case 'TOGGLE\_TODO' :*

*return state.map((todo) => todo.id !== action.id ? todo :*

*Object.assign({}, todo, { complete: !todo.complete }))*

*default :*

*return state*

*}*

*}*

*function goals (state = [], action) {*

*switch(action.type) {*

*case 'ADD\_GOAL' :*

*return state.concat([action.goal])*

*case 'REMOVE\_GOAL' :*

*return state.filter((goal) => goal.id !== action.id)*

*default :*

*return state*

*}*

*}*

*function app (state = {}, action) {*

*return {*

*todos: todos(state.todos, action),*

*goals: goals(state.goals, action),*

*}*

*}*

*/\**

*Passing the root reducer to our store since our createStore function can only take one reducer.*

*\*/*

*const store = createStore(app);*

**Best Practices**

<https://youtu.be/BnX0BPQPuY4>

Prefer constants rather than strings as the values of type properties. Both work -- but when using constants, the console will throw an error rather than fail silently should there be any misspellings (e.g. LOAD\_PROFILE vs. LOAD\_PROFILE)

We can ensure an error will be thrown for misspelled action types.

<https://youtu.be/oPC21DNJwyo>

Use Action creators to avoid redundant code.

In this section, we converted our actions to use JavaScript constants instead of strings. We also refactored our .dispatch() calls from passing in unique objects directly to them, to calling special functions that create the action objects - these special functions that create action objects are called **Action Creators**.

**UI + Redux**

<https://youtu.be/i2pSr-MlCPU>

<https://youtu.be/8IkNVrCqtvo>

<https://youtu.be/0M2gm4-IbGs>

<https://youtu.be/b9HpVHhDvL4>

<https://youtu.be/p3PtYdpqSO0>