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:

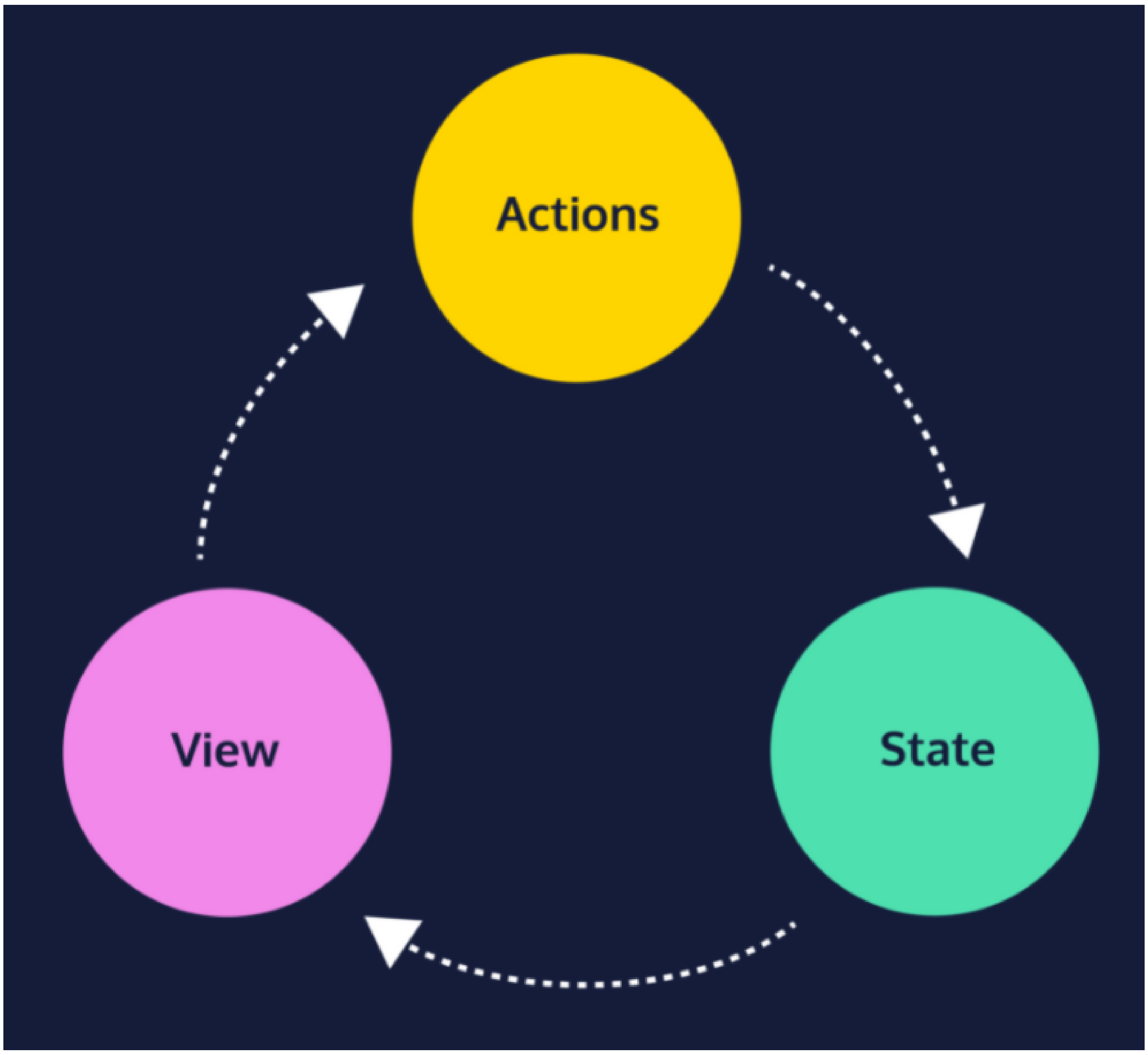
- The state holds the current data used by the app’s components.
- The view components display that state data.
- When a user interacts with the view, like clicking a button, the state will be updated in some way.
- The view is updated to display the new state.

With plain React, these three parts overlap quite a bit. Components not only render the user interface, but they also may manage their own state. When actions that may change the state occur, components need to directly communicate these changes to each other.

Redux helps separate the state, the view, and actions by requiring that the state be managed by a single source. Requests to change the state are sent to this single source by view components in the form of an action. Any components of the view that would be affected by these changes are informed by this single source. By imposing this structure, Redux makes our code more readable, reliable, and maintainable.

Instructions

This diagram represents the one-way data flow in Redux: from state to view to action to state and so on. With Redux, data ALWAYS flows in this direction.



State
State is the current information behind a web application.

For a calendar application it includes the events (name, date, label, etc.), the current timezone, and the display filters. For a todo app it includes the todo items (description, completed/not completed), the current order of the items, and display filters. For a word editor, it includes the contents of the document, the print settings, and comments.

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.

Instructions

1.
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

Checkpoint 2 Passed

Hint
Here's a similar example:

```
const differentState = [ 'eenie', 'meenie', 'miney' ];
```

Actions
Most well-designed applications will have separate components that need to communicate and share data with each other.

A todo list might have an input field where the user can type in a new todo item. The application might transfer this data from the input field, add it to an array of all todos, and then render them as text on the screen. 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'
}
```

Instructions

1.
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

Checkpoint 2 Passed

Hint
As a reminder, here’s what our initial state looked like:

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

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

Checkpoint 3 Passed

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'`

Checkpoint 4 Passed
Hint
Not every action needs a `payload` property.

index.js

```
const state = [ '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'
}
```

Reducers

So far, we’ve defined the state of our application and the actions representing requests to change that state, but we haven’t seen how these changes are carried out in JavaScript. The answer is a *reducer*.

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 for the `state` parameter.
2. We use the spread operator (`...`) 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.

Instructions

1.

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

Checkpoint 2 Passed

Hint

Here’s the outline for a reducer function:

```
const appReducer = (state = initialState, action) => {
  switch (action.type) {
    default:
      return state;
  }
}
```

2.

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

Hint

Make sure to return a NEW copy of the state object. Follow the todo example in the narrative.

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](#).

You can expect an action like this:

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

Checkpoint 4 Passed

Hint

Use the `filter()` method to make a copy of `state` without the specified song.

The function passed to `filter()` should return true if the current song does not equal the action payload.

For example, `smallNums` will include numbers that are less than 6:

```
const nums = [0, 3, 2, 4, 10, 6];

const smallNums = nums.filter(num => num < 6);
```

index.js

```
// 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', 'Respe
ct' ];

const addNewSong = {
  type: 'songs/addSong',
  payload: 'Halo'
};

const removeSong = {
  type: 'songs/removeSong',
  payload: 'Take Five'
};

const removeAll = {
  type: 'songs/removeAll'
}
```

Rules of Reducers

In the previous exercise, we wrote reducers that returned a new copy of the state rather than editing it directly. We did this to adhere to the rules of reducers provided by the Redux documentation:

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".

By asynchronous logic or "side effects", we mean anything that the function does aside from returning a value, e.g. logging to the console, saving a file, setting a timer, making an HTTP request, generating random numbers.

These rules make Redux code predictable and easy to debug: tests run reliably and other developers know what to expect from your code.

Instructions

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.

Checkpoint 2 Passed

Hint

If a reducer is using global variables to calculate its next state, then it is using something other than the passed arguments.

After making your changes, the reducer should work given an action and function call like this:

```
addGlobalAction = {
  type: 'songs/addGlobalSong',
  payload: 'We are the World'
};
const newPlaylist = playlistReducer(undefined,
addGlobalAction);
```

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`.

Checkpoint 3 Passed

Hint

Using `state.push()` modifies the existing `state`. Use the spread operator (`...state`) instead.

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`.

Checkpoint 4 Passed

Hint

Calling `Math.random()` is a side effect, making the reducer unpredictable and hard to test. If we test this reducer today, it will behave differently tomorrow!

After making your changes, the reducer should work given an action and function call like this:

```
const addRandomAction = { type: 'numbers/addRandom',
payload: Math.random() };
const newState = reducer(undefined, addRandomAction);
```

app1.js

```
// Reducer violates rule 1:
// They should only calculate the new state value based on t
he 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: 'W
e are the World' };
const newState = playlistReducer(state, addAction);
```

app2.js

```
// Reducer violates rule 2:
// They are not allowed to modify the existing state.
// Instead, they must copy the existing state and make chang
es 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 th
e mountain' ];
const addTodoAction = { type: 'todos/addTodo', payload: 'Des
cend' };
const newState = todoReducer(state, addTodoAction);
```

app3.js

```
 // Reducer violates rule 3:
 // They must not do any asynchronous logic or have other “s
ide 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: M
ath.random() };
const newState = reducer(undefined, randomAction);
```

Immutable Updates and Pure Functions

In programming, there is a more general way to describe the three rules of reducers in Redux: 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];
};
```

Instructions

1.

The function in **immutable.js** mutates its arguments because it uses the array `splice()` function. Rewrite it using the `slice()` method and the spread operator.

If done correctly, the output should still be:

```
[ 'a', 'c', 'd' ]
Checkpoint 2 Passed
```

Hint

`splice()` changes the array it is called upon. `slice()` and the spread operator can copy the contents of an array into a new array.

Here's an example that cuts out the third item in a list:

```
const removeItemAtThree = (list) => {
  return [
    ...list.slice(0,3),
    ...list.slice(4)
  ];
};
```

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.

immutable.js

```
const removeItemAtIndex = (list, index) => {
 return [
   ...list.slice(0,index),
   ...list.slice(index+1, list.length)
 ]
};

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

pure.js

```
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
So far we have covered state, actions, reducers, and how they participate in the one-way data flow. Where, in JavaScript, does all of this take place?

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.

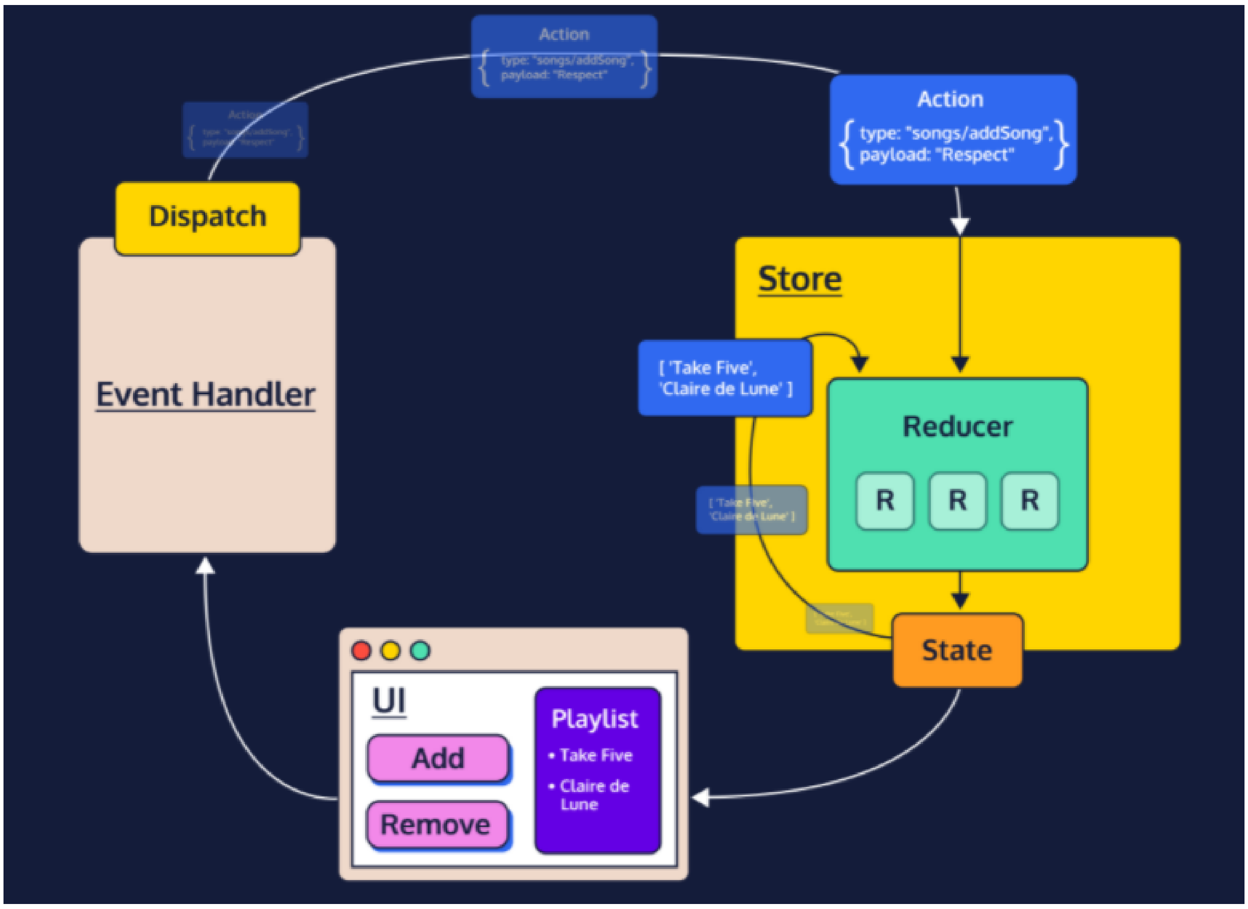
We can rephrase our data flow using the new term:

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.

We won't be writing any code for the store during this lesson, but it's important that you understand this term for future Redux lessons.

Instructions

Follow the diagram's one-way data flow. Notice how the store contains the reducer and state. It receives actions and calls the reducer with the action and current state.



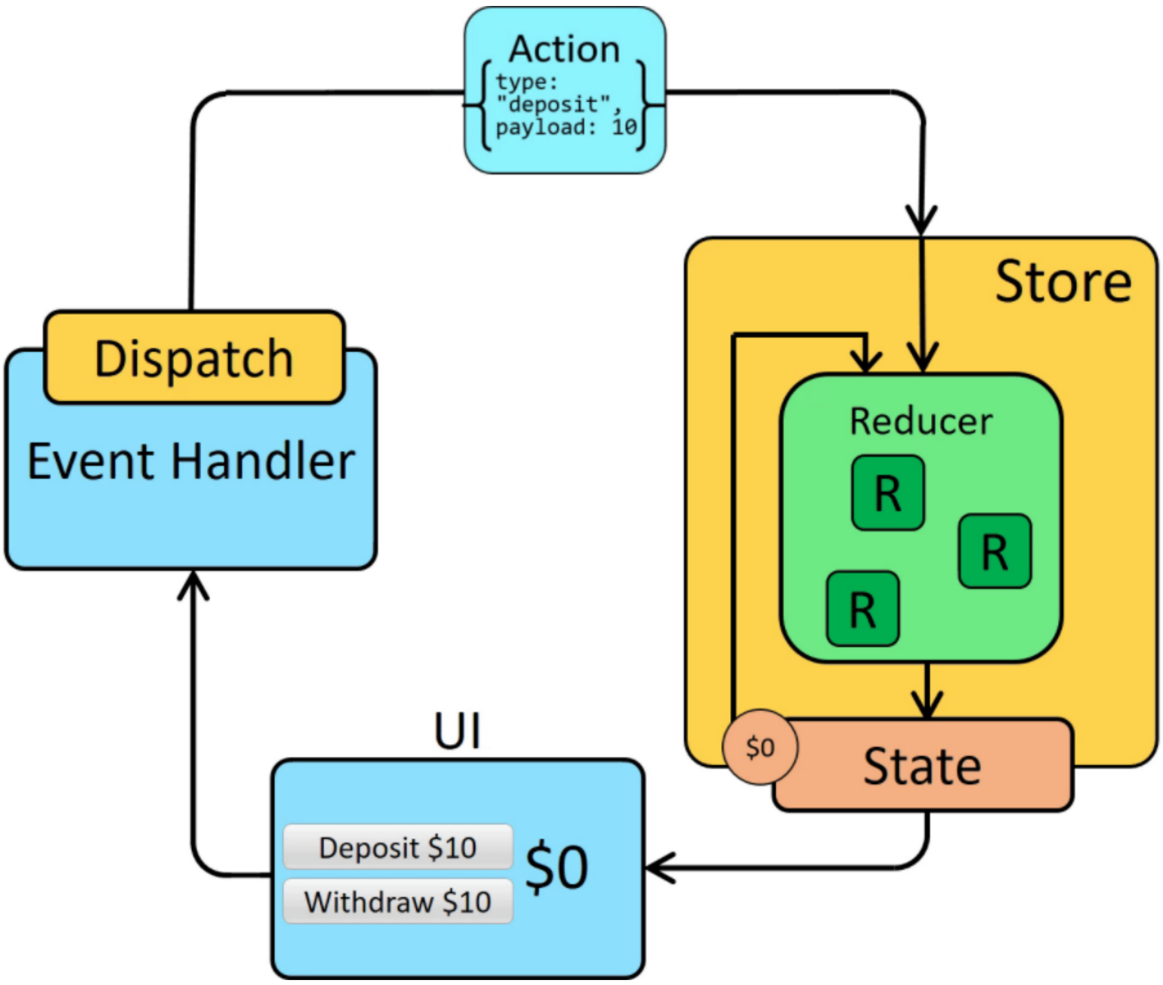
Review

Congratulations! In this lesson you've built a strong conceptual foundation of Redux and built a state object, some actions, and a reducer along the way. Here's what else you learned:

- 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.

Instructions

Take another look at the diagram depicting data flow in a Redux application. Make sure you can explain every part of this diagram before moving on.


Action
{type: "deposit", payload: 10}
Store
Dispatch
Event Handler
Reducer
R
R
R
UI
$0
State
Deposit $10
Withdraw $10
$0