Redux:

is a predictable state container for JavaScript apps, most commonly used with libraries like React for building user interfaces. It helps in managing the application state in a more organized and predictable manner.

Here's a breakdown of Redux and how it works in a React application:

1. **State Management**: Redux helps manage the state of your entire application in a single store. This means that all of your application's data is stored in one centralized location.

2. **Store**: The store holds the state of the application. It's a plain JavaScript object that is managed by Redux. The store is created using the `createStore()` function provided by Redux.

3. **Actions**: Actions are plain JavaScript objects that represent an intention to change the state. They are dispatched to the Redux store. An action typically has a type field that describes the type of action being performed.

4. **Reducers**: Reducers are functions that specify how the application's state changes in response to actions sent to the store. They take the previous state and an action as arguments, and return the next state of the application.

5. **Dispatch**: Actions are dispatched to the store using the `dispatch()` method. When an action is dispatched, the store calls the corresponding reducer function, which updates the state according to the action.

6. **Connect**: In a React application, components need to access the state stored in the Redux store. The `connect()` function provided by the `react-redux` library is used to connect React components to the Redux store. This allows components to access the state as props and dispatch actions to update the state.

Here's a simple example of how Redux is used in a React application:

```javascript
// Define actions
const increment = () => {
  return { type: 'INCREMENT' };
};

const decrement = () => {
  return { type: 'DECREMENT' };
};

// Define reducer
```

```
const counterReducer = (state = 0, action) => {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};

// Create Redux store
import { createStore } from 'redux';
const store = createStore(counterReducer);

// Connect React component to Redux store
import { connect } from 'react-redux';

// React component
const Counter = ({ count, increment, decrement }) => {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

// Map state to props
const mapStateToProps = (state) => {
  return {
    count: state
  };
};

// Map dispatch to props
const mapDispatchToProps = {
  increment,
  decrement
};

// Connect component to Redux store
const ConnectedCounter = connect(mapStateToProps, mapDispatchToProps)(Counter);
```

```
export default ConnectedCounter;
```

In this example, we have a simple counter application. We define actions (`increment` and `decrement`), a reducer (`counterReducer`), and create a Redux store using `createStore()`. Then we connect a React component (`Counter`) to the Redux store using `connect()` from `react-redux`, mapping the state and action creators to props. Finally, we export the connected component (`ConnectedCounter`) which can be used in our application.