
+
ES6

# Training Agenda

- ES6+ Features
- Redux

# Prerequisites:

- Sound knowledge of JavaScript
- Interest to learn

# ES6 : New features

- Arrow Functions
- Promises
- Block Scoping
- Rest & Spread Operators
- Default Values
- Destructuring
- Template Strings
- Symbols, Iterators, and Generators

# Arrow Functions =>

- Arrow functions are handy for one-liners.
- They come in two flavors:
  - Without curly braces: `(...args) => expression` – the right side is an expression: the function evaluates it and returns the result.
  - With curly braces: `(...args) => { body }` – brackets allow us to write multiple statements inside the function, but we need an explicit *'return'* to return something.

```
let func = (arg1, arg2, ...argN) => expression
```

# Arrow functions : Limitations

- Do not have **this**.
- Do not have **arguments**.
- Can’t be called with **new**.

# Arrow Function : Task

- Replace Function Expressions with arrow functions in the code:

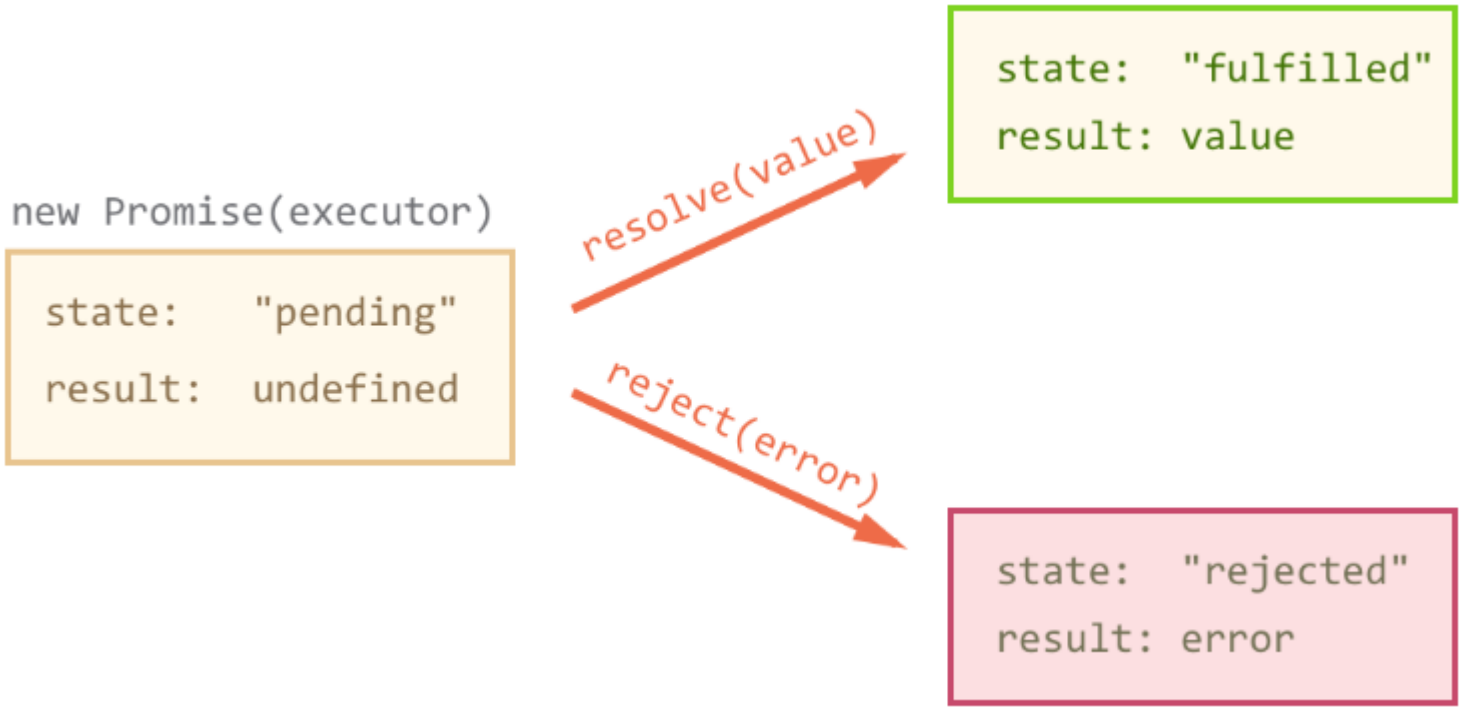
```
function ask(question, yes, no){
      if(confirm(question)) yes();
      else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function(){ alert("You cancelled the execution."); }
);
```

# Promises

- A *promise* is a special JavaScript object that links the “producing code” and the “consuming code” together.
- A “producing code” that does something and takes time. For instance, the code loads a remote script.
- A “consuming code” that wants the result of the “producing code” once it’s ready.
- The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

# The Promise Object

- The resulting promise object has internal properties:
  - state — initially "*pending*", then changes to either "*fulfilled*" or "*rejected*",
  - result — an arbitrary value of your choice, initially "*undefined*".



# Promise : Task

- The function `delay(ms)` should return a promise. That promise should resolve after **`ms`** milliseconds, so that we can add **`.then`** to it :

```
function delay(ms) {
  // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

# Block Scoping

Restricts the scope of variables to the nearest curly braces :

- ***let*** : for all type of variables
- ***const*** : converts the variable to a constant

**const != immutable**

# Rest/Spread Operator (...)

- Rest Parameters :
  - A function can be called with any number of arguments, no matter how it is defined.
  - The rest parameters must be at the end.
  - Usage : create functions that accept any number of arguments.

- Spread Operator :
  - Spread operator looks similar to rest parameters, also using (...), but does quite the opposite.
  - It  is used in the function call, it “expands” an iterable object into the list of arguments.
  - Usage : pass an array to functions that normally require a list of many arguments.

# Destructuring

*Destructuring assignment* is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables.

*Array destructuring :* the array is destructured into variables, but the array itself is not modified.

If there are fewer values in the array than variables in the assignment, there will be no error. Absent values are considered ***undefined***.

*Object destructuring :* We have an existing object at the right side, that we want to split into variables

*Nested destructuring :* If an object or an array contain other objects and arrays, we can use more complex left-side patterns to extract deeper portions.

# Destructuring : Task

- Write the destructuring assignment that reads:

```
let user = {
  name: "John",
  years: 30
};
```

- **name** property into the variable name.
- **years** property into the variable age.
- **isAdmin** property into the variable isAdmin (false if absent)

# Template Strings

Template literals are string literals allowing embedded expressions.

We can use multi-line strings and string interpolation features with them.

Template literals are enclosed by the back-tick (` `) character instead of double or single quotes.

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}).

The tag expression (usually a function) gets called with the processed template literal, which you can then manipulate before outputting.

# Symbols

- Symbol is a primitive type for unique identifiers.
- Symbols are created with `Symbol()` call with an optional description.
- Symbols are always different values, even if they have the same name.
- If we want same-named symbols to be equal, then we should use the global registry: `Symbol.for(key)` returns (creates if needed) a global Symbol with key as the name.
- Symbols have two main use cases:
  - “Hidden” object properties.
  - A symbolic property does not appear in for..in.
- *Well-Known Symbols :* There are many system symbols used by JavaScript which are accessible as Symbol.*. We can use them to alter some built-in behaviours.

# Iterables & Generators

Objects that can be used in *for..of* are called *iterable*.

*Iterables* must implement the method named **Symbol.iterator.**

The result of obj[Symbol.iterator] is called an *iterator*. It handles the further iteration process.

An *iterator* must have the method named next() that returns an object *{done: Boolean, value: any},* here **done:true** denotes the iteration end, otherwise the value is the next value.

The *Symbol.iterator* method is called automatically by *for..of,* but we also can do it directly.

Built-in *iterables* like strings or arrays, also implement *Symbol.iterator.*

# Iterables & Generators

Creation of Iterators requires careful programming due to the need to explicitly maintain their internal state.

Generators allow you to define an iterative algorithm by writing a single function whose execution is not continuous.

Generator functions are written using the `function*` syntax.

When called initially, generator functions do not execute any of their code, instead returning a type of iterator called a Generator.

When a value is consumed by calling the generator's **next** method, the Generator function executes until it encounters the **yield** keyword.

The function can be called as many times as desired and returns a new Generator each time, however each Generator may only be iterated once.

# Code Challenge

- Suppose that you're working in a small town administration and you're in-charge of two town elements :
  - Parks
  - Streets
- It's very small town, so there are only 3 Parks and 4 Streets. All parks and streets have name and build year.

- At an end-of-year meeting, your boss wants a final report with the following :
  1. Tree density of each park in the town *(formula : number of trees / park area)*
  2. Average age of each town's park *(formula : sum of all ages / number of parks)*
  3. The name of park that has more than 1000 trees
  4. Total and average length of town's streets
  5. Size classification of all streets – tiny / small / normal / big / huge *(if the size if unknown, default is normal)*
- All the report data should be printed to the console.

***Hint :*** *Use some of ES6 features : classes, subclasses, default parameter, template string, arrow functions, maps, destructuring, rest and spread, block scopes etc*


A predictable state container for JavaScript apps.

# Should I Use Redux?

| | |
|---|---|
| Some suggestions: | You have reasonable amounts of data changing over time |
| | You need a single source of truth for your state |
| | You find that keeping all your state in a top-level component is no longer sufficient |

# Redux

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

It provides a great developer experience, such as live code editing combined with a time traveling debugger.

```
npm install --save redux
```

# Redux : Three Principles

## Single source of truth

- The *state* of your whole application is stored in an object tree within a single *store*.

## State is read-only

- The only way to change the state is to emit an *action*, an object describing what happened.

## Changes are made with pure functions

- To specify how the state tree is transformed by actions, you write pure *reducers*.

# Actions

- *Actions* are payloads of information that send data from your application to your *store*.
- They are the only source of information for the store.
- You send them to the store using `store.dispatch()`.

```
{
    type: ADD_TODO,
    text: 'Build my first Redux app’
}
```

# Reducers

- The *reducer* is a pure function that takes the previous *state* and an *action*, and returns the next state.
- *Actions* only describe *what happened*, but don't describe how the application's state changes.
- **Reducers** specify how the application's state changes in response to *actions* sent to the *store*.

```
(previousState, action) => newState
```

# Reducers : Don’ts

Things you should **never** do inside a reducer:

- Mutate its arguments.
- Perform side effects like API calls and routing transitions.
- Call non-pure functions, e.g. `Date.now()` or `Math.random()`.

# Store

The Store is the single *object* that has the following responsibilities:

- Holds application state.
- Allows access to state via `getState()`.
- Allows state to be updated via `dispatch(action)`.
- Registers listeners via `subscribe(listener)`.
- Handles unregistering of listeners via the function returned by `subscribe(listener)`.

# Creating Store with Root Reducer

```
import { createStore } from 'redux’ ;
import rootReducer from './reducers’ ;

const store = createStore(rootReducer);
```

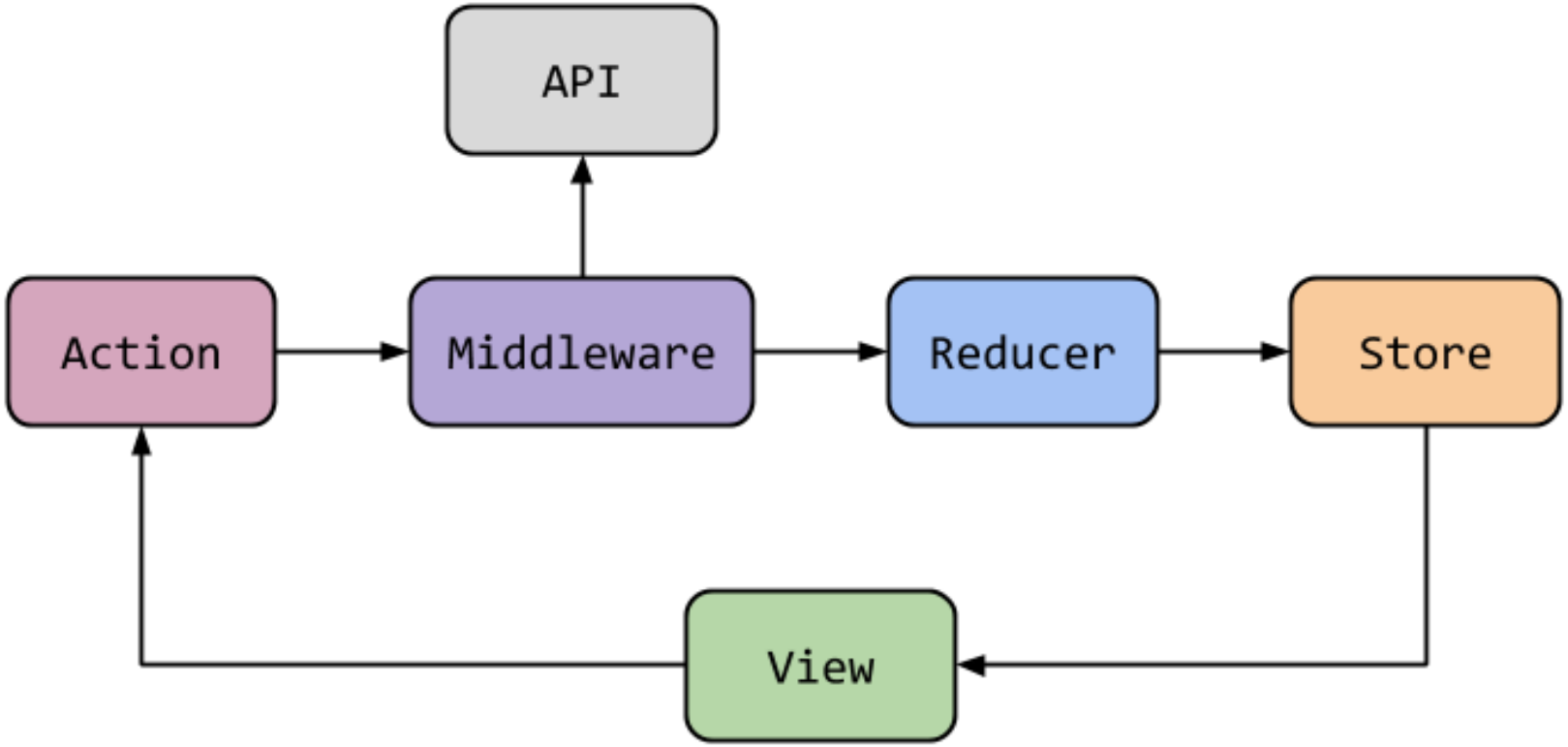
# Data Flow

Redux architecture revolves around a **strict unidirectional data flow**.

The data lifecycle in any Redux app follows these 4 steps:

- You call `store.dispatch(action)`.
- The Redux store calls the reducer function you gave it.
- The root reducer may combine the output of multiple reducers into a single state tree.
- The Redux store saves the complete state tree returned by the root reducer.

# Redux Architecture & Data Flow



# Redux Middleware : Thunk

Thunks are middlewares for basic Redux side effects logic including complex synchronous operation that needs access to the store and/or async operation like AJAX requests.

Redux Thunk middleware allows you to write action creators that return a function instead of an action.

The thunk can be used to delay the dispatch of an action or to dispatch only if a certain condition is met. The inner function receives the store methods **dispatch** and **getState** as parameters.

**> npm install redux-thunk**

# Redux Middleware : Freeze

Redux middleware that prevents state from being mutated anywhere in the app. When mutation occurs, an error will be thrown by the runtime.

Useful during development mode to ensure that no part of the app accidentally mutates the state.

# References


http://javascript.info

https://redux.js.org

https://stackoverflow.com

https://www.npmjs.com

https://nodejs.org

https://gist.github.com/danharper/3ca2273125f500429945

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol