
+
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:

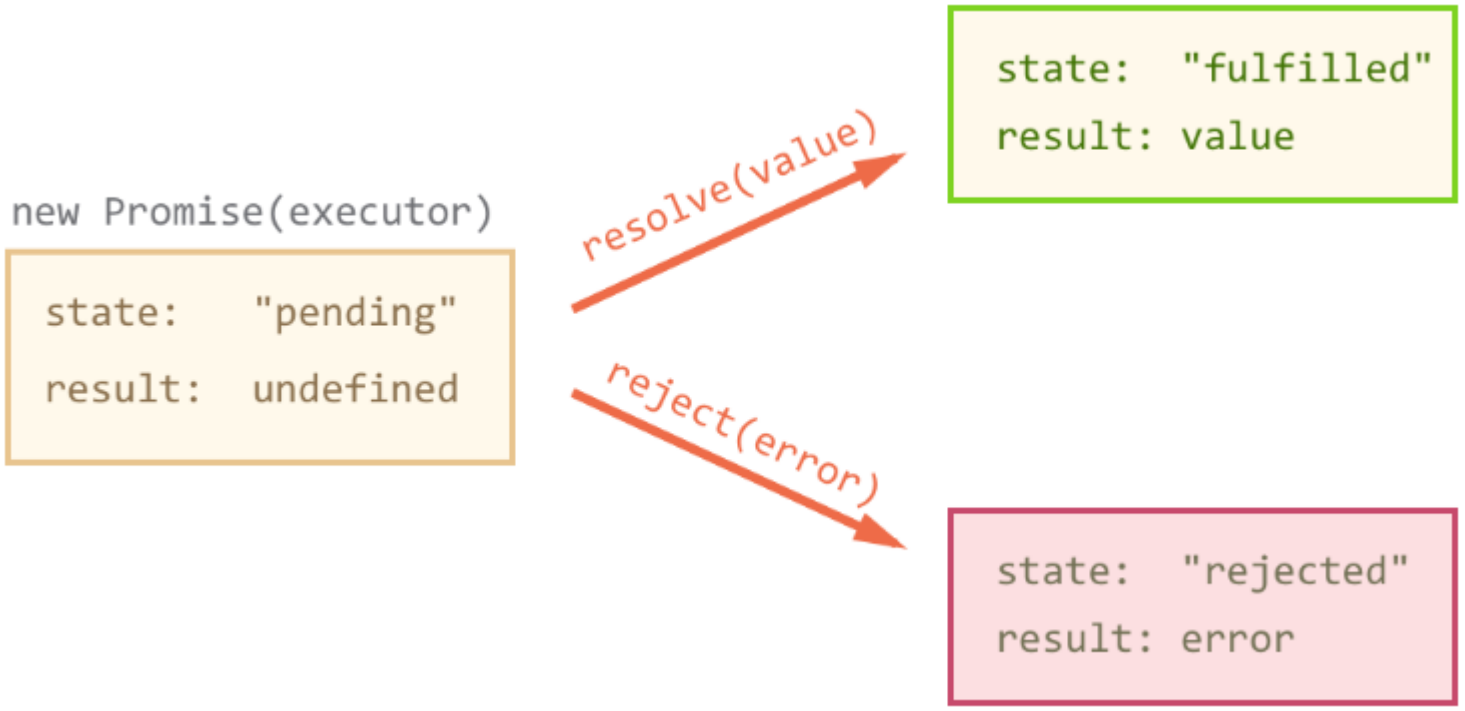
```
var ask(question, yes, no) => {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  () =>{ alert("You agreed."); },
  () =>{ alert("You canceled 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.

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 { StoreModule } from '@ngrx/store' ;

@NgModule({
        …
        imports :     [StoreModule.forRoot({prop : reducer})]
})
export class AppModule{}
```

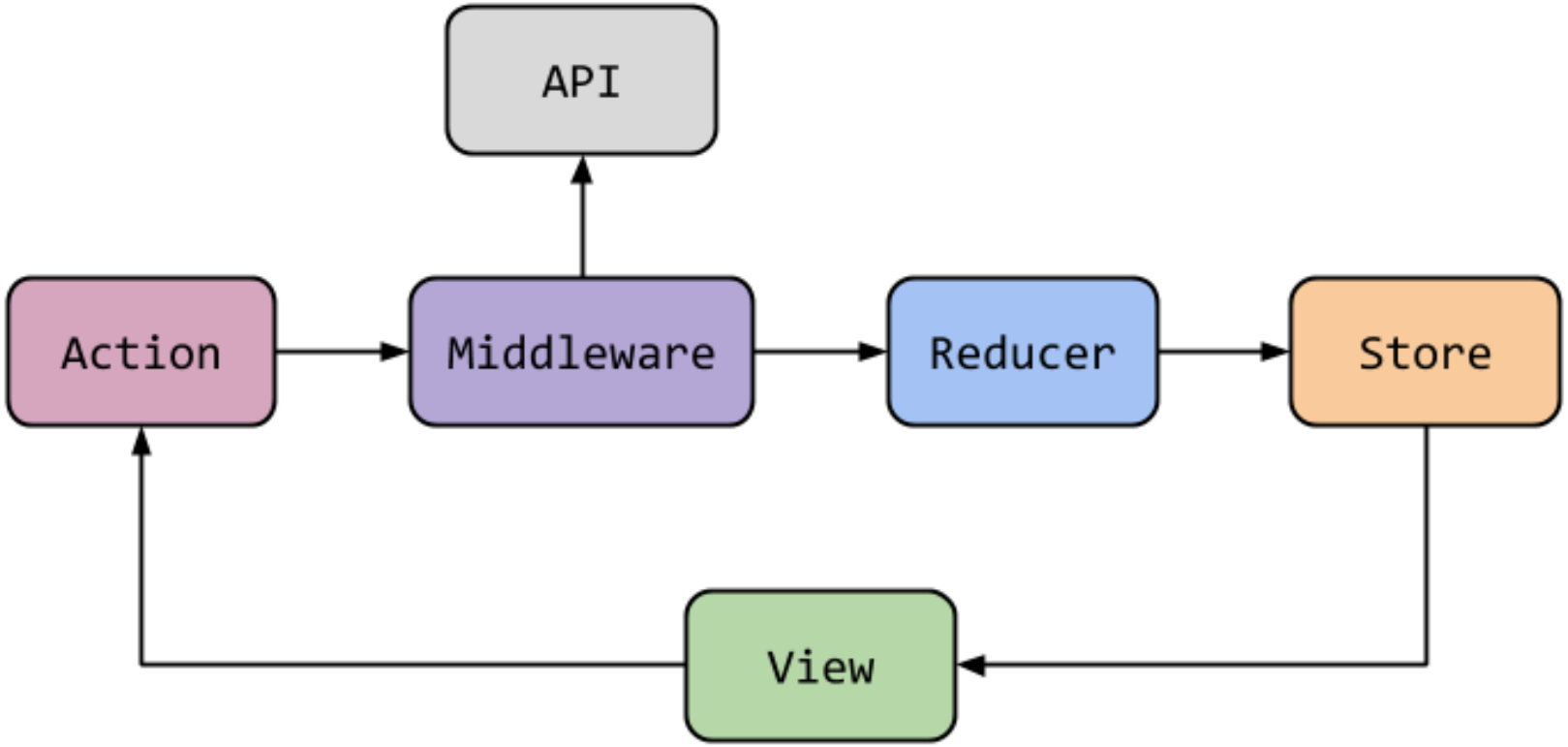
Data Flow

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



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