



Transitioning your Drupal 8 Site to ES6

Brian Reese & Lindsey Kopacz

Who we are

Lindsey Kopacz

- Front End Developer with Acquia Professional Services
- Passionate about Accessibility, JavaScript, pumping iron and good craft beers.

Brian Reese

- Senior Developer with Acquia Professional Services
- Front- and Back-end developer, with a passion for good javascript and visual regression testing

Overview

- Defining Variables
- Arrow Functions
- Template Literals
- Promises
- Modules and Classes

Defining Variables

ES5 Scope and declaring variables

- Historically var only had 2 scoping capabilities: functional scope and global scope

```
var pi = 3.14; // global scope
function calculateCircleArea(radius) {
  // functional scope that only exists in the function
  var area = pi * radius2;
}
```

- What happens when you want block scope?

Hoisting

- Hoisting is a JavaScript mechanism where variables are moved to the ***top of their scope*** before code execution.
- What happens when you want block scope?

```
function compareNumbers(x, y) {  
  if (x > y) {  
    var c = 2;  
  }  
}
```

- var c becomes ***hoisted*** into the functional scope so if y > x, c will become undefined

Solving the Hoisting problem

- ES6 introduced two new ways to declare variables: **let** and **const**
- **let** statement declares a **block** scope local variable, optionally initializing it to a value.
- **const** is much like **let** scope, except that it defines a constant that cannot change through reassignment, and it can't be redeclared.

Using const

```
// define pi as a constant  
const pi = 3.14;
```

```
// this will throw an error  
// Uncaught TypeError: Assignment to constant variable.  
pi = 3.1415;
```


Using `let`

```
let x = 1;
```

```
if (x === 1) {  
    let x = 2;
```

```
    console.log(x); // expected output: 2  
}
```

```
console.log(x); // expected output: 1
```

Arrow Functions

Functions (currently and pre-ES6)

- Functions are defined using a function keyword, optional parameter(s), and curly braces
- Every new function defined its own **this** value
- Can be used as constructors

Arrow Functions

- Arrow functions now use “fat arrow” syntax
- Arrow functions do not have their own **this** value.
- Arrow functions have convenient shorthands depending on how many lines of code are within the scope, how many parameters there are, etc

ES5 old school functions - using **this**

```
const pets = {  
  names: ['Baron', 'Chief', 'Axel'],  
  owner: 'Jason',  
  description: function(){  
    return this.names.map(function(pet){  
      return `${this.owner} knows an awesome dog named ${pet}.`  
    });  
  }  
};  
  
pets.description();  
  
// returns ['undefined knows an awesome dog named Baron', 'undefined knows  
an awesome dog named Chief', 'undefined knows an awesome dog named Axel']
```

Source: Medium Article [ES5 functions vs. ES6 'fat arrow' functions](#)

Why does it return **undefined**?

- **this** is referring to the function that called it which doesn't know what 'owner' is.
- Because ES5 functions has its own **this**, it's looking **within** the function
- What needs to happen is that 'this' needs to refer to the **pets object**, not the function.
- In ES5 there are a couple of ways around it (bind, creating a variable that is equal to this, pass the this value to the function). Let's focus on ES6 though.

Source: Medium Article [ES5 functions vs. ES6 'fat arrow' functions](#)

How to fix **this** with arrow functions

- Arrow functions don't have their own or don't change the context of **this**.
- When using arrow functions, the **this** value of the enclosing lexical context is used (IE **this** from its enclosing scope)

```
const pets = {  
  names: ['Baron', 'Chief', 'Axel'],  
  owner: 'Jason',  
  description: function(){  
    return this.names.map((pet) => {  
      return `${this.owner} knows an awesome dog named ${pet}.`  
    });  
  }  
};
```

Source: Medium Article [ES5 functions vs. ES6 'fat arrow' functions](#)

Arrow Functions syntax

- You define a function using a variable. You put the parameters in parentheses per usual, then use a fat arrow to continue to define your scope
- If you only have one parameter, but can omit the parentheses.
- If you only have one line of code that you're returning you can omit the curly braces and the **return** keyword.

Using arrow syntax

```
const double = (x) => {  
  return x * 2;  
}
```

// Below is the same exact function.

```
const double = x => {  
  return x * 2;  
}
```

// Below is ALSO the same exact function.

```
const double = x => x * 2;
```

Template Literals

Template Literals

- Template literals are string literals allowing embedded expressions.
- They are defined with backticks ``Some String``
- You can interject variables, math, object properties, etc with the following syntax: `${someVar}`
- The result: ``${name} is ${age} years old``

Using template literals

```
const person = {  
  name: 'Lindsey',  
  age: 28,  
  hometown: 'Frederick, MD',  
}
```

// creating a string in ES5 and below

```
var es5about = person.name + ' is ' + person.age + ' years old and is from'  
+ person.hometown + '.'
```

// creating a string in ES6

```
const aboutPerson = `${person.name} is ${person.age} years old and is from  
${person.hometown}.
```

Promises



What is a promise?

- An object which represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
- This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method ***returns a promise to supply the value at some point in the future.***
- You can chain your promises using **`.then()`** and **`.catch()`** which also return promises.

Using promises (a real-world example)

- Axios is a Promise-based HTTP client for JavaScript which can be used in your front-end application and in your Node.js backend.

```
axios.get('https://api.github.com/users/darkcody')  
  .then(response => console.log(response.data.name))  
  .catch(error => console.log(error));
```


Creating your own promises

```
const x = 10;

const promise = new Promise((resolve, reject) => {
  if (x < 15) {
    axios.get('https://api.github.com/users/lkopacz').then((content) => {
      Content.title = 'My custom title';
      resolve(content);
    });
  }
  else {
    reject('not valid');
  }
});

promise.then(result => console.log('It worked'));
promise.catch(error => console.log('It didn't work'));
```

Async/Await

- A syntax for working with promises that reads more like synchronous code
- Async functions are functions that return a promise
- Within an async function, code execution can be paused using the **await** keyword. Code execution continues with the resolved promise.
- Async/await is **not an official part of ES6** but its coming!
It can be used today using the Babel transpiler

Async/Await examples

Using Promises:

```
function doThingsInSequence() {  
  return doThing1()  
    .then((result1) => {  
      return doThing2(result1);  
    })  
    .then((result2) => {  
      return doThing3(result2);  
    })  
    .then((result3) => {  
      return doThing4(result3);  
    });  
}
```

Using Async/Await:

```
async function doThingsInSequence() {  
  let result1 = await doThing1();  
  let result2 = await doThing2();  
  let result3 = await doThing3();  
  return await doThing4();  
}
```

Async/Await examples

- The “await” keyword only works inside of a function defined with “async”

```
async function getVisitorLocation() { ... }
```

```
// Will throw an exception
```

```
await getVisitorLocation()
```

```
// Will also throw an exception
```

```
function myRegularFunction() {
```

```
    let location = await getVisitorLocation();
```

```
}
```

Async/Await in 7 seconds

<https://twitter.com/manekinekk/status/855824609299636230>

ES6 Modules and Classes

What is an ES6 Class

- At its most **basic** level, the class keyword in ES6 is equivalent to a constructor function definition that conforms to prototype-based inheritance.
- The **class** keyword is just a special function and exhibits expected function behavior.
- Moving towards the object oriented model, but note that it is primarily syntactical sugar for JavaScript's existing prototype-based inheritance and not yet object oriented.

ES5 vs. ES6 constructor syntax

// ES5 Constructor Function

```
function Animal(type, name, sound) {  
  this.type = type;  
  this.name = name;  
  this.sound = sound;  
}
```

// ES6 Class

```
class Animal {  
  constructor(type, name, sound) {  
    this.type = type;  
    this.name = name;  
    this.sound = sound;  
  }  
}
```

// Works for both ES5 and ES6!

```
const frog = new Animal('frog', 'George', 'ribbit');
```

Source: Medium Article [ES6 Class vs Object.prototype](#)

What is Prototypal Inheritance?

```
function Person(first, last, age) {  
  this.name = {  
    first,  
    last  
  };  
  this.age = age;  
};  
  
Person.prototype.greeting = function() {  
  console.log(`Hi! I'm ${this.name.first}.`);  
};  
  
const Lindsey = new Person('Lindsey', 'Kopacz', 28);  
Lindsey.greeting(); // logs 'Hi! I'm Lindsey.' to the console
```

What is Prototypal Inheritance?

- All functions get initialized with a prototype object.
- By placing greeting on Person.prototype, we made it available to all instances of Person
- In actuality, there's no Lindsey.greeting property. Instead, Lindsey has access to the greeting() method on Person.prototype because it's an instance of Person.
- If I logged "Lindsey" to the console, I would not see the greeting in my object.
- This is commonly referred to as the prototype chain.

ES6 prototypal inheritance

- ES6 gives us the ability to create a class that inherits properties from a parent **super** class.
- It uses the **extends** keyword.
- Let's take our frog example, and this time we'll create it as a class.

Source: Medium Article [ES6 Class vs Object.prototype](#)

ES6 prototypal inheritance

```
class Frog extends Animal {  
  constructor(name) {  
    super('frog', name, 'ribbit');  
  }  
}
```

```
const george = new Frog('George');
```

```
Animal.prototype.makeSound = function() {  
  console.log(`The ${this.type} goes ${this.sound}!`);  
}
```

```
george.makeSound(); // Logs 'The frog goes ribbit!' to the console
```

What is a JavaScript Module?

- At its most **basic** level, modules refer to small units of independent, reusable code.
- Modules can be imported and exported.
- Use **import** and **export** keyword to take full advantage of modular JavaScript
- There is a problem though: importing files into applications wasn't built into browsers - so we need a way to bundle the code so that we can render it.
- One Solution: Use Webpack!

What is Webpack?

- Webpack is a module bundler.
- The bundling process begins from user-defined entries.
- Entries themselves are modules and can point to other modules through imports.
- Plays nicely with babel - which compiles ES6 code into ES5 friendly code for browser compatibility.

Other Things to Learn

- Rest Parameters
- Spread operators
- Destructuring Arrays and Objects

Drupal Example

Resources

- [Medium Blog Post: ES6 Classes vs Object Inheritance](#)
- [Medium Blog Post: ES5 functions vs. ES6 'fat arrow' functions](#)
- [In-depth dive on Async/Await: Truly understanding Async/Await](#)
- [Understanding JavaScript Modules](#)
- [A Detailed Introduction to Webpack](#)
- [Using Babel and Webpack](#)
- [Prototypal Inheritance in JavaScript](#)
- [MDN: Using Promises](#)
- [MDN: Using Fetch](#)
- [MDN: Template Literals](#)
- [MDN: let declaration](#)
- [MDN: const declaration](#)
- [MDN: Destructuring Assignment](#)
- [MDN: Rest parameters](#)
- [MDN: Spread Syntax](#)

ES6 Drupal Example

<https://github.com/darkcody/weather-module-example>