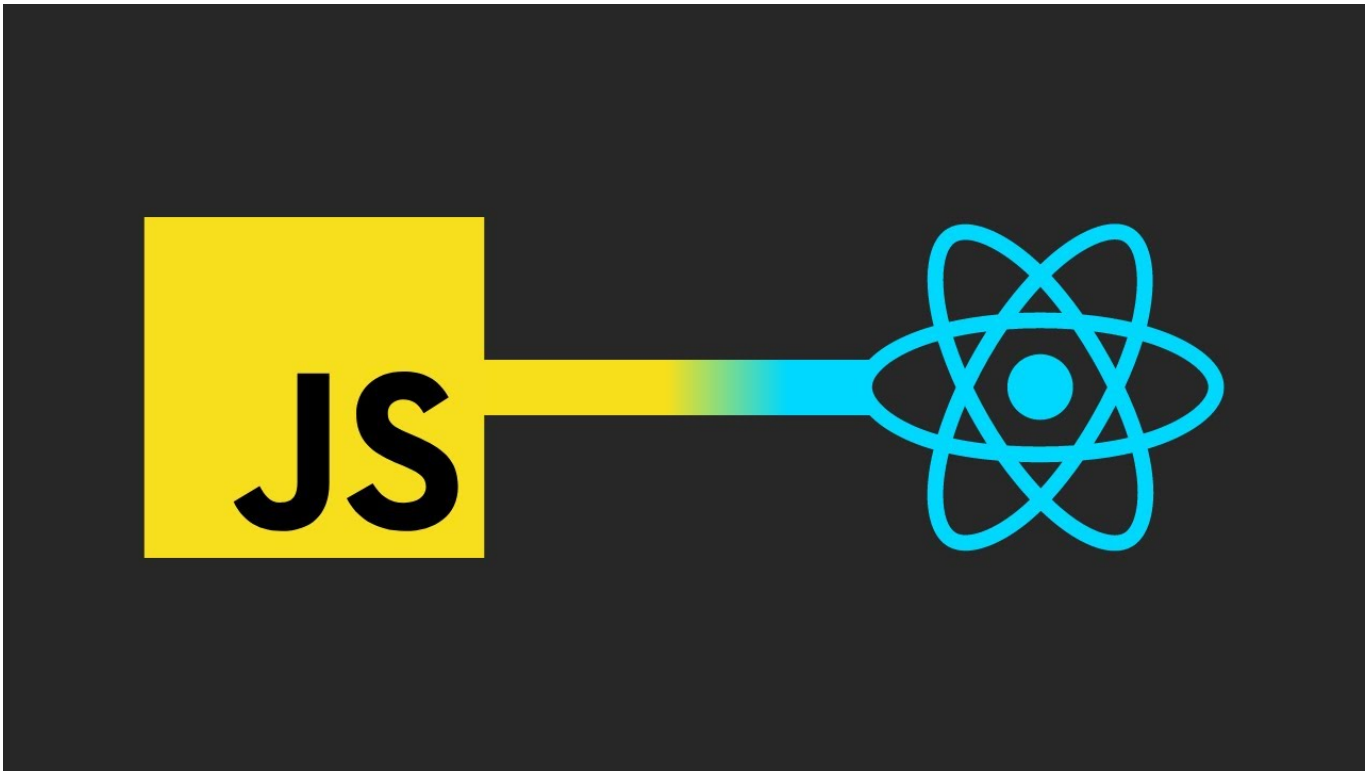
Khóa học Front-end ATT Lab - Những kiến thức cần nắm khi bước sang ReactJS

Tổng quan 🎯

1. Nội suy chuỗi trong JavaScript (String Interpolation)
2. Arrow function
3. Truthy và Falsy
4. Destructuring function parameters
5. Default parameter values
6. Property Value Shorthand
7. Rest và Spread
8. Object Destructuring
9. Array Destructuring
10. Javascript Modules



1. Nội suy chuỗi trong JavaScript (String Interpolation)

Nội suy chuỗi trong JavaScript (String Interpolation) Trong các phiên bản JavaScript cũ hơn, để tạo chuỗi động, chúng ta cần sử dụng phép toán nối (+):

```
const userName = 'Mai';
const dynamicString = 'hello ' + userName + '!';
```

Điều này hoạt động ổn, nhưng đôi khi nó cảm thấy khá rắc rối và có thể dẫn đến lỗi (ví dụ: quên khoảng trắng trong Hello).

JavaScript hiện đại cho phép chúng ta nhúng các biến và các biểu thức khác ngay bên trong chuỗi

```
const dynamicString = `hello ${userName}!`;
```

Các chuỗi được tạo bằng backticks được gọi là “template strings”. Về phần lớn, chúng hoạt động giống như bất kỳ chuỗi nào khác, nhưng chúng có siêu năng lực này: chúng có thể nhúng các phân đoạn động.

Chúng tôi tạo một phân đoạn động trong chuỗi của mình bằng cách viết ${}. Mọi thứ được đặt giữa các dấu ngoặc nhọn sẽ được đánh giá là biểu thức JavaScript.

Điều này có nghĩa là chúng ta có thể làm những điều thú vị như thế này:

```
const age = 7;
console.log(`Next year, you'll be ${age + 1} years old.`);

<div style="page-break-after: always;"></div>```

## 2. Arrow function
Trong lịch sử, các hàm trong JavaScript đã được viết bằng từ khóa hàm:

```js
function exclaim(string) {
  return string + '!';
}
```

Vào năm 2015, ngôn ngữ này đã nhận được một cú pháp thay thế để tạo hàm: arrow function. Chúng trông như thế này:

```
const exclaim = string => string + '!';
```
```

Các `arrow function` được lấy cảm hứng từ các hàm lambda từ các ngôn ngữ lập trình hàm khác. Lợi ích chính của chúng là ngắn hơn và sạch hơn nhiều. Giảm "sự lộn xộn của hàm" có vẻ như là một lợi ích không đáng kể, nhưng nó thực sự có thể giúp cải thiện khả năng đọc khi làm việc với các hàm ẩn danh?. Ví dụ:

```
const arr = ['hey', 'ho', 'let\'s go'];

// This:
arr
  .map(function(string) {
    return string + '!'
  })
  .join(' ');

// …Becomes this:
arr
  .map(string => string + '!')
  .join(' ');
```

3. Truthy và Falsy

Trong JavaScript, mọi giá trị đều được phân loại thành "truthy" hoặc "falsy". Giá trị truthy được coi là "true" khi sử dụng trong các điều kiện.

Hãy xem xét đoạn mã JavaScript sau:

```
const user = {
  name: 'J. Script',
};

if (user.name) {
  console.log('This user has a name!');
}
```

Trong đoạn code này, chúng ta kiểm tra xem tên người dùng có tồn tại hay không bằng cách sử dụng biểu thức `user.name`. Điều thú vị là `user.name` không phải là một giá trị kiểu Boolean (true/false). Tên người dùng có thể là một chuỗi ký tự bất kỳ.

Trong nhiều ngôn ngữ lập trình khác, đây có thể là một phép toán không hợp lệ. Vậy làm thế nào để JavaScript biết một chuỗi ký tự bất kỳ có đủ điều kiện hay không?

Giá trị truthy

JavaScript giải quyết vấn đề này bằng cách sử dụng các giá trị truthy và falsy. Giá trị truthy là giá trị được coi là "true" khi sử dụng trong các điều kiện. Hầu hết các giá trị trong JavaScript đều là truthy.

Danh sách các giá trị falsy:

```
false (sai)
null (rỗng)
undefined (chưa được định nghĩa)
'' (chuỗi rỗng)
0 (số 0) và các giá trị liên quan như 0.0 và -0
NaN (Not a Number - không phải là số)
```

Trong ví dụ trên, `user.name` là một giá trị truthy vì nó là một chuỗi ký tự có ít nhất 1 ký tự. Bất kỳ chuỗi nào khác ngoài chuỗi rỗng '' đều được coi là truthy.

Một điều bất ngờ: Mảng rỗng [] và object rỗng {} cũng là các giá trị truthy, không phải falsy. Điều này có nghĩa là mọi object và mảng đều được coi là truthy.

Chuyển đổi sang kiểu Boolean

Đôi khi, chúng ta cần chuyển đổi một giá trị truthy thành true (đúng) hoặc falsy thành false (sai).

Có hai cách để thực hiện điều này:

Sử dụng hàm Boolean(): Đây là cách rõ ràng nhất để chuyển đổi. Ví dụ:

```
Boolean(4); // true
Boolean(0); // false
Boolean([]); // true
Boolean(''); // false
Hãy thận trọng khi sử dụng các đoạn mã.
content_copy
Sử dụng toán tử NOT (!): Đây là cách phổ biến hơn nhưng có thể trông hơi
lạ đối với người mới bắt đầu. Toán tử NOT được sử dụng lặp lại hai lần
(!!). Ví dụ:
JavaScript
!!4; // true
!!0; // false
!![]; // true
!!''; // false
```

Lưu ý rằng `!!` không phải là một toán tử thực sự trong JavaScript. Chúng ta đang sử dụng toán tử NOT (!) hai lần.

Cách toán tử NOT hoạt động:

Toán tử NOT đảo ngược giá trị Boolean. Ví dụ:

```
!true; // false
!false; // true
```

Nếu chúng ta sử dụng toán tử NOT với một giá trị không phải kiểu Boolean, nó sẽ đảo ngược giá trị truthy thành false (sai) và falsy thành true (đúng). Ví dụ:

```
!4; // false, vì 4 là truthy
!0; // true, vì 0 là falsy
```

Chúng ta có thể sử dụng nhiều toán tử NOT (!) cùng nhau để đảo ngược giá trị nhiều lần.

Ví dụ:

```
!4; // false
!!4; // true
!!!4; // false
!!!!4; // true

<div style="page-break-after: always;"></div>```

## 4. Destructuring function parameters
```js
```
```

```
function validateUser(user) {
  if (typeof user.name !== 'string') {
    return false;
  }

  if (user.password.length < 12) {
    return false;
  }

  return true;
}
```

Cách viết khác

```
function validateUser(user) {
  const { name, password } = user;

  if (typeof name !== 'string') {
    return false;
  }

  if (password.length < 12) {
    return false;
  }

  return true;
}
```

Bằng cách sử dụng tính năng `parameter destructuring`, chúng ta có thể thực hiện việc `destructuring` này ngay trong các tham số của hàm:

```
function validateUser({ name, password }) {
  if (typeof name !== 'string') {
    return false;
  }

  if (password.length < 12) {
    return false;
  }

  return true;
}
```

> Cả 3 đoạn mã này đều tương đương nhau, nhưng nhiều nhà phát triển thích sử dụng tính năng `parameter destructuring`. Nó đặc biệt phổ biến trong React để phá hủy các props trong các

thành phần của chúng ta.

5. Default parameter values

Giống như việc phá hủy đối tượng thông thường, chúng ta có thể cung cấp các giá trị mặc định để sử dụng cho các tham số.

Đây là một ví dụ nhanh:

```
function sendApiRequest({ method = 'GET', numOfRetries }) {
  // Stuff
}
```

Khi tôi gọi hàm này, tôi có thể cung cấp giá trị của riêng mình cho `method`:

```
sendApiRequest({ method: 'PUT', numOfRetries: 4 });
```

...Hoặc, nếu tôi muốn nó bằng `GET`, tôi có thể bỏ qua hoàn toàn:

```
sendApiRequest({numOfRetries: 4 });

<div style="page-break-after: always;"></div>```

## 6. Property Value Shorthand
Các đối tượng trong JavaScript hiện đại có một thủ thuật nhỏ tiện lợi. Đó là một việc nhỏ nhưng nếu bạn không để ý thì có thể gây ra rất nhiều nhầm lẫn.

Giả sử chúng ta có đoạn mã sau:

```js
const name = 'Ahmed';
const age = 26;

const user = {
  name: name,
  age: age,
};

console.log(user);
// { name: 'Ahmed', age: 26 }
```

Chúng tôi đang tạo một object `user` với hai thuộc tính, tên và tuổi. Chúng tôi đang gán các thuộc tính đó cho các biến có cùng tên.

Tuy nhiên, nó có vẻ hơi dư thừa phải không? Chúng ta đang gán tên cho tên và gán tuổi cho tuổi.
```

JavaScript hiện đại cung cấp cho chúng ta một cách viết tắt thuận tiện mà chúng ta có thể sử dụng trong tình huống này:

```
const name = 'Ahmed';
const age = 26;

const user = {
  name,
  age,
};

console.log(user);
// { name: 'Ahmed', age: 26 }
```

Điều này được gọi là `Property Value Shorthand`. Khi tạo một đối tượng, chúng ta có thể bỏ qua các giá trị nếu chúng có cùng tên với thuộc tính.

Kết quả cuối cùng là như nhau, nhưng ngắn gọn hơn một chút.

7. Rest và Spread

7.1. Rest

Giả sử chúng ta đang viết một hàm có số lượng tham số thay đổi.

Ví dụ: có thể chúng ta muốn tạo một hàm sẽ cộng tất cả các số được cung cấp, bất kể số lượng có bao nhiêu:

```
addNums(1, 1); // 2
addNums(1, 2, 3, 4); // 10
addNums(1, 10, 100, 1000, 10000); // 11111
```

Chúng ta có thể làm điều này bằng cách sử dụng các `rest parameters`:

```
function addNums(...nums) {
  let sum = 0;

  nums.forEach(num => {
    sum += num;
  });

  return sum;
}
```

`nums` là một tham số duy nhất sẽ tập hợp tất cả các tham số khác vào một mảng:

```
function logArgs(...args) {
  console.log(args);
}

logArgs(1, 2, 'hi!');
// logs: [1, 2, 'hi']
```

Giống như bất kỳ tham số hàm nào khác, chúng ta có thể đặt tên cho nó theo bất cứ tên nào chúng ta muốn. Việc sử dụng `...rest` được sử dụng phổ biến, nhưng nó cũng có thể đặt là `...allTheThings`.

Đôi khi, chúng ta sẽ có một hoặc nhiều tham số đã biết/cố định và một số tham số có thể thay đổi. Chúng ta có thể sử dụng tham số còn lại song song với các tham số khác, như sau:

```
function removeFirstArg(first, ...rest) {
  return rest;
}

removeFirstArg(1, 2, 3, 4); // Produces [2, 3, 4]
```

Cả hai chức năng này đều không hợp lệ:

```
function nope(...firstRest, ...rest) {}

function notAllowed(...rest, second, third) {}
```

7.2. Spread

Cú pháp `spread` ngược lại với các tham số `rest`. Thay vì thu thập một loạt tham số vào một mảng, nó sẽ trải một mảng dữ liệu thành các đối số riêng lẻ.

Ví dụ:

```
function createDate(year, month, day) {
  return new Date(year, month, day);
}

const myDateInfo = [2020, 01, 01];

createDate(...myDateInfo);
```

Chúng tôi "giải nén" myDateInfo để nó cung cấp 3 đối số riêng lẻ, thay vì một đối số mảng duy nhất. Nó tương đương với việc làm này:

```
const myDateInfo = [2020, 01, 01];

createDate(
  myDateInfo[0],
  myDateInfo[1],
  myDateInfo[2]
);
```

Có rất nhiều cách sử dụng hữu ích cho việc này.

Ví dụ: chúng ta có thể sử dụng nó để sao chép các phần tử con từ mảng này sang mảng khác:

```
const myArray = [1, 2, 3, 4];

const arrayCopy = [...myArray];

console.log(arrayCopy); // [1, 2, 3, 4];
console.log(myArray === arrayCopy); // false. Different arrays.
```

Tương tự, chúng ta có thể sử dụng nó để hợp nhất hai mảng:

```
const myNumbers = [1, 2, 3];
const yourNumbers = [4, 5, 6];

const ourNumbers = [...myNumbers, ...yourNumbers];

console.log(ourNumbers); // [1, 2, 3, 4, 5, 6]
```

7.3. Sử dụng với Object

Phiên bản đầu tiên của cú pháp này chỉ hoạt động với mảng, nhưng ngày nay chúng ta cũng có thể sử dụng nó với các đối tượng!

Ví dụ: chúng ta có thể tạo một bản sao mới của một đối tượng hiện có:

```
const originalObject = {
  latitude: 1.234,
  longitude: 4.321,
};

const clonedObj = { ...originalObject };
```

Chúng ta có thể mở rộng các đối tượng hiện có thành các đối tượng mới với các thuộc tính bổ sung:

```
const sharedCharacteristics = {
  species: 'human',
  location: 'earth',
};

const human1 = {
  ...sharedCharacteristics,
  name: 'Tina',
  eyeColor: 'green',
};

const human2 = {
  ...sharedCharacteristics,
  name: 'James',
  eyeColor: 'brown',
};
```

Và chúng ta có thể hợp nhất hai hoặc nhiều đối tượng như thế này:

```
const myObj = { hi: 5 };
const yourObj = { bye: 10 };
```

```
const ourObj = { ...myObj, ...yourObj };
console.log(ourObj); // { hi: 5, bye: 10 }

<div style="page-break-after: always;"></div>```

## 9. Array Destructuring
Giả sử chúng ta có một số dữ liệu nằm trong một mảng và chúng ta muốn lấy nó ra và gán nó cho một biến.

Theo truyền thống, điều này được thực hiện bằng cách truy cập một mục theo chỉ mục và gán cho nó một giá trị bằng một câu lệnh gán điển hình:

```js
const fruits = ['apple', 'banana', 'cantaloupe'];
const firstFruit = fruits[0];
const secondFruit = fruits[1];
```

Chúng ta có thể dùng cách `Array Destructuring`

```
const fruits = ['apple', 'banana', 'cantaloupe'];
const [firstFruit, secondFruit] = fruits;
```

Lần đầu tiên bạn nhìn thấy nó, các dấu ngoặc mảng trước dấu = trông khá là lạ.

☐ được sử dụng sau toán tử gán (=), chúng được dùng để đóng gói các mục thành một mảng. Khi được sử dụng trước đó, chúng làm ngược lại và giải nén các mục từ một mảng:

```
// Packing 3 values into an array:
const array = [1, 2, 3];

// Unpacking 3 values from an array:
const [first, second, third] = array;
```

Cuối cùng, đây không thực sự là một công cụ thay đổi cuộc chơi, nhưng nó là một thủ thuật nhỏ gọn gàng trong bữa tiệc có thể giúp mọi thứ gọn gàng hơn một chút.
```

10. Object Destructuring

Phá hủy đối tượng cung cấp một cách dễ dàng để trích xuất một số biến từ một đối tượng.

Đây là một ví dụ nhanh:

```
const user = {
  name: 'Trung tâm ATT Lab',
  city: 'Bình Thanh, HCM',
  country: 'VietNam',
};

const { name, country } = user;

console.log(name); // ‘Trung tâm ATT Lab’
console.log(country); // VietNam’
```

Đây thực sự là điều tương tự như làm điều này:

```
const name = user.name;
const country = user.country;
```

Chúng ta có thể lấy ra nhiều hoặc ít giá trị tùy thích.

10.1. Đổi tên các giá trị được trích xuất

Hãy xem xét tình huống này:

```
const name = 'Hello!';

const user = { name: 'Trung tâm ATT Lab' };

const { name } = user;
// Uncaught SyntaxError:
// Identifier 'name' has already been declared
```

Ta đã cố gắng phân tách thuộc tính `name` thành biến riêng của nó nhưng gặp phải một vấn đề: đã có một biến có tên là `name`!

Trong những trường hợp này, chúng ta có thể đổi tên giá trị khi giải nén nó:

```
const name = 'Hello!';

const user = { name: 'Trung tâm ATT Lab' };
```

```
const { name: userName } = user;

console.log(name); // ‘Hello!’
console.log(userName); // Trung tâm ATT Lab’
```

10.2. Default value

Đây là câu hỏi: điều gì sẽ xảy ra nếu chúng ta cố gắng hủy cấu trúc khóa khỏi một đối tượng không được xác định?

```
const user = { name: 'Trung tâm ATT Labr' };

const { status } = user;

console.log(status); // ???
```

Chà, `user.status` không được xác định và do đó `status` sẽ được đặt thành `undefined`.

Nếu muốn, chúng ta có thể đặt giá trị mặc định bằng toán tử gán:

```
const { status = 'idle' } = user;
```

Nếu `user` có thuộc tính `status`, giá trị đó sẽ được lấy ra và gán cho hằng số trạng thái mới. Nếu không, `status` sẽ được gán cho chuỗi `'idle'`.

Nói cách khác, đây là phiên bản hiện đại của điều này:

```
const status = typeof user.status === 'undefined'
  ? 'idle'
  : user.status;
```

10. JavaScript Modules

Named exports

Mỗi `file` có thể xác định một hoặc nhiều `named exports`:

```
export const significantNum = 5;

export function doubleNum(num) {
 return num * 2;
}
```

Trong tệp chính của chúng tôi, `index.js`, chúng tôi đang nhập cả hai bản xuất này:

```
import { significantNum, doubleNum } from './data';
```

Export statements

```
// C1
export const significantNum = 5;
```

hoặc có thể viết bằng cách này

```
// C2
const significantNum = 5;

function sum(a, b){
    return a + b;
};

export { significantNum, sum };
```

Renaming imports

Đôi khi, chúng ta sẽ gặp phải xung đột về đặt tên với các mục nhập được đặt tên:

```
import { Wrapper } from './Header';
import { Wrapper } from './Footer';
// 🚫 Identifier 'Wrapper' has already been declared.
```

Điều này xảy ra vì các tên xuất khẩu không nhất thiết phải là duy nhất trên global. Việc cả Đầu trang và Chân trang sử dụng cùng một tên là hoàn toàn hợp lệ:

```
// Header.js
export function Wrapper() {
  return <header>Hello</header>;
}
```

```
// Footer.js
export function Wrapper() {
  return <footer>World</footer>;
}
```

Chúng ta có thể đổi tên các `import` nhập bằng từ khóa `as`:

```
import { Wrapper as HeaderWrapper } from './Header';
import { Wrapper as FooterWrapper } from './Footer';
// ✅ No problems
```

Default exports

Có một loại xuất riêng biệt trong mô-đun JS: xuất mặc định.

Hãy xem một ví dụ:

```
const magicNumber = 100;

export default magicNumber;
```

Khi nói đến `default export`, chúng tôi luôn xuất một biểu thức:

```
// ✅ Correct:
const hi = 5;
export default hi;

// 🚫 Incorrect
export default const hi = 10;
```

Mỗi mô-đun JS được giới hạn ở một lần `single default export`. Ví dụ: điều này không hợp lệ:

```
const hi = 5;
const bye = 10;

export default hi;
export default bye;
// 🚫 SyntaxError: Too many default exports!
```

Khi nhập dữ liệu xuất mặc định, chúng tôi không sử dụng dấu ngoặc nhọn:

```
// ✅ Correct:
import magicNumber from './data';

// 🚫 Incorrect:
import { magicNumber }  from './data';
```

Khi chúng tôi `import` một `default export`, chúng ta có thể đặt tên cho nó bất cứ thứ gì chúng ta muốn; nó không nhất thiết phải khớp:

```
// ✅ This works
import magicNumber from './data';

// ✅ This also works!
import helloWorld from './data';
```