

1. VPS:

- First,

```
2. init(pageNum = 1) {
  window.scrollTo(0, 0)
  const { sortPrice } = this.state
  this.setState({
    loading: true
  }, () => {
    queryTicketList({
      pageNum: pageNum,
      reorder: sortPrice || '',
      searchInput: '上海',
      cityName: '上海'
    }).then(data => {
      const { pageCount, viewInfoList } = data
      this.setState({
        loading: false,
        dataSource: viewInfoList,
        total: pageCount,
        current: pageNum
      })
    })
  })
}

paiXu = (sortPrice) => {
  this.setState({ sortPrice }, () => this.init()) //this.init()需要
  ↳ 在this.setState()的回调函数里面执行
}
```

3. often used DOM properties and methods (scrollHeight, offsetHeight, clientHeight, scrollTop, offsetTop)

```
document.body.innerHTML = 'This is the text of the body element';

document.body.firstChild.parentNode.innerHTML = 'I am the parent and my
↳ inner HTML has been replaced!';
```

```

document.querySelector('h1').innerHTML = 'Most popular TV show searches
→ in 2016'; // returns the first selected element

document.querySelector('#fourth').innerHTML = 'Fourth element';

document.body.style.backgroundColor = '#201F2E';

let liRef = document.createElement('li');
liRef.id = 'oaxaca';
liRef.innerHTML = "Oaxaca, Mexico";

document.getElementById('more-destinations').appendChild(liRef);

parent.removeChild(child);

```

4. webpack.config.js

```

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

// common.js syntax
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.join(__dirname, '/dist'),
    filename: 'index_bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }, Or
        use: [
          // apply multiple loaders and options
          "htmlint-loader",
          {
            loader: "html-loader",

```

```

        options: {
            /* ... */
        }
    }
]
}
]
},
plugins: [
    new HtmlWebpackPlugin({
        template: './src/index.html',
        filename: 'index.html',
        inject: 'body'
    })
]
}

```

5. Computed property names

Starting with ECMAScript 2015, the object initializer syntax also supports computed property names. That allows you to put an expression in brackets [], that will be computed and used as the property name.

```

// Computed property names (ES2015)
var i = 0;
var a = {
    ['foo' + ++i]: i,
    ['foo' + ++i]: i,
    ['foo' + ++i]: i
};

console.log(a.foo1); // 1
console.log(a.foo2); // 2
console.log(a.foo3); // 3

var param = 'size';
var config = {
    [param]: 12,
    ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};

```

```
console.log(config); // {size: 12, mobileSize: 4}
```

6. A presentational component can often be written as a *stateless functional component*

```
// A component class written in the usual way:
export class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}

// The same component class, written as a stateless functional
↪ component:
export const MyComponentClass = () => {
  return <h1>Hello world</h1>;
}

// Works the same either way:
ReactDOM.render(
  <MyComponentClass />,
  document.getElementById('app')
);
```

7. Child Components Update Their Parents' state in React

Parent.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Child } from './Child';

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.changeName = this.changeName.bind(this);
    this.state = { name: 'Frarthur' };
  }

  changeName(newName) {
    this.setState({ name: newName });
  }
}
```

```

render() {
  return <Child name={this.state.name} onChange={this.changeName} />
}
}

ReactDOM.render(
  <Parent />,
  document.getElementById('app')
);

```

You cannot declare method `changeName(newName)` as `changeName: function(newName)`, otherwise won't work. The same goes for `render()`

Child.js

```

import React from 'react';

export class Child extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    const name = e.target.value;
    this.props.onChange(name);
  }

  render() {
    return (
      <div>
        <h1>
          Hey my name is {this.props.name}!
        </h1>
        <select id="great-names" onChange={this.handleChange}>
          <option value="Frarthur">
            Frarthur
          </option>

```

```

        <option value="Gromulus">
            Gromulus
        </option>

        <option value="Thinkpiece">
            Thinkpiece
        </option>
    </select>
</div>
);
}
}

```

8. class inheritance

```

class HospitalEmployee {
    constructor(name) {
        this._name = name;
        this._remainingVacationDays = 20;
    }

    get name() {
        return this._name;
    }

    get remainingVacationDays() {
        return this._remainingVacationDays;
    }

    takeVacationDays(daysOff) {
        this._remainingVacationDays -= daysOff;
    }
}

class Nurse extends HospitalEmployee {
    constructor(name, certifications) {
        super(name);
        this._certifications = certifications;
    }
}

```

```

get certifications() {
    return this._certifications;
}

addCertification(newCertification) {
    this.certifications.push(newCertification);
}
}

const nurseOlynyk = new Nurse('Olynyk', ['Trauma', 'Pediatrics']);
nurseOlynyk.takeVacationDays(5);
console.log(nurseOlynyk.remainingVacationDays);
nurseOlynyk.addCertification('Genetics');
console.log(nurseOlynyk.certifications);

```

9. The major difference between a GET request and POST request is that a POST request requires additional information to be sent through the request. This additional information is sent in the body of the post request.

10. async await POST

```

// async await POST

async function getData(){
    try {
        const response = await fetch('http://api-to-call.com/endpoint', { //
            ↪ sends request
            method: 'POST',
            body: JSON.stringify({id: '200'})
        });
        if (response.ok){ // handles response if successful
            const jsonResponse = await response.json();
            // Code to execute with jsonResponse
        }
        throw new Error('Request Failed!');
    } catch (error){ // handles response if unsuccessful
        console.log(error);
    }
}

```

11. async await GET

```
// async await GET

async function getData(){
  try {
    const response = await fetch('http://api-to-call.com/endpoint');
    if (response.ok){ // handles response if successful
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) { // handles response if unsuccessful
    console.log(error);
  }
}
```

12.
 - used `fetch()` to make GET and POST requests
 - check the status of the responses coming back
 - catch errors that might possibly arise
 - taking successful responses and rendering it on the webpage

13. `fetch()` POST Requests

```
// fetch POST

fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'}) // sends request
}).then(response => {
  if (response.ok){
    return response.json(); // converts response object to JSON
  }
  throw new Error('Request failed!');
}, networkError => console.log(networkError.message) // handles errors
).then(jsonResponse => {
  // Code to execute with jsonResponse // handles success
});
```

14. `fetch()` GET Requests


```
// fetch GET

fetch('http://api-to-call.com/endpoint').then(response => { // sends
  ↪ request
    if (response.ok){
      return response.json(); // converts response object to JSON
    }
    throw new Error('Request failed!');
  }, networkError => console.log(networkError.message) // handles errors
).then(jsonResponse => {
  // Code to execute with jsonResponse // handles success
});
```

15. Boilerplate code for making an XHR POST request

From codecademy:

```
// XMLHttpRequest POST

const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';
const data = JSON.stringify({id: '200'}); // Converts data to a JSON
↪ string

// handles response
xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE){
    // Code to execute with response
  }
};

xhr.open('POST', url);
xhr.send(data);
```

16. Boilerplate code for making an XHR GET request

From codecademy:

```
// XMLHttpRequest GET

const xhr = new XMLHttpRequest(); // creates new object
```

```

const url = 'http://api-to-call.com/endpoint';

// handle responses
xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if(xhr.readyState === XMLHttpRequest.DONE){
    // Code to execute with response
  }
};

//opens request and sends object
xhr.open('GET', url);
xhr.send();

```

From w3schools

```

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    // Typical action to be performed when the document is ready:
    document.getElementById("demo").innerHTML = xhttp.responseText;
  }
};
xhttp.open("GET", "filename", true);
xhttp.send();

```

17. With a GET request, we're retrieving, or *getting*, information from some source (usually a website). For a POST request, we're *posting* information to a source that will process the information and send it back
18. json example. **Data types:** Number, String, Boolean, Array, Object, Null

```

{
  "name": "Brad Traversy",
  "age": 35,
  "address": {
    "street": "5 main st",
    "city": "Boston"
  },
  "children": ["Brianna", "Nicholas"]
}

```

19. javascript object example

```
var person = {  
  name: "Brad",  
  age: 35,  
  email: function(){  
    return 'brad@gmail.com';  
  }  
};  
console.log(person.name);  
console.log(person.email());
```

20. Section DOM Template Parsing Caveats covered is is special attribute offers a workaround

21. Wrap form widgets inside a p tag

```
<p>  
  <label for="name">  
    <span>Name: </span>  
    <strong><abbr title="required">*</abbr></strong>  
  </label>  
  <input type="text" id="name" name="username">  
</p>
```

22. Three ways to change elements' visibility

```
display: none; /* completely gone, never existed */  
visibility: hidden; /* still occupies space */  
opacity: 0; /* still occupies space */
```

23. CSS Margins: You can set the margin property to auto to horizontally center the element within its container. To horizontally center a block element (like <div>), use margin: auto; **Note:** Center aligning has no effect if the width property is not set (or set to 100%).

To horizontally center an image use

```
<img style="display: block; margin: 8px auto 15px auto;"  
display: block; 一定要加
```

24. The default font-size is 16px, so 1em equals 16px

25. width and height can only be applied to elements that are not inline elements. Some examples of inline and block elements, also see [w3schools](#) (块级元素内联元素)

```
<section id="inline">
  <span>inline</span>
  <a>inline</a>
  <b>inline</b>
  <em>inline</em>
</section>
<section id="block">
  <div>block</div>
  <nav>nav</nav>
  <aside>main</aside>
  <main>main</main>
</section>
```

26. <h1> is the most important part of a html doc

27. self-closing tags in HTML5:
 <embed> <hr> <iframe> <input> <link> <meta>

28. Layout elements

```
<body>
  <header>
    <nav>

    </nav>
  </header>
  <section>
    <main>
      <article>

      </article>
    </main>
    <aside>

    </aside>
  </section>
  <footer>

  </footer>
</body>
```

- 29. In addition to data properties, Vue instances expose a number of useful instance properties and methods. These are prefixed with `$` to differentiate them from user-defined properties
- 30. Out of the box, webpack won't require you to use a configuration file. However, it will assume the entry point of your project is `src/index.js` and will output the result in `dist/main.js` minified and optimized for production
- 31. When installing a package that will be bundled into your production bundle, you should use `npm install --save`. If you're installing a package for development purposes (e.g. a linter, testing libraries, etc.) then you should use `npm install --save-dev`
- 32. Popular CSS pre-processors including LESS, SASS, Stylus, and PostCSS
- 33. Follow this guide if the built-in configuration of Vue CLI does not suit your needs, or you'd rather create your own webpack config from scratch
- 34. `vue-loader` is a loader for **webpack** that allows you to author Vue components in a format called **Single-File Components (SFCs)**
- 35. `http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.
- 36. With `http.request()` one must always call `req.end()` to signify the end of the request – even if there is no data being written to the request body
- 37. `querystring.parse(str[, sep[, eq[, options]]])` parses a URL query string (`str`) into a collection of key and value pairs. For example, the query string `'foo=bar&abc=xyz&abc=123'` is parsed into:

```
{
  foo: 'bar',
  abc: ['xyz', '123']
}
```

- 38. `querystring.stringify(obj[, sep[, eq[, options]]])` produces a URL query string from a given `obj` by iterating through the object's "own properties"

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' });
// returns 'foo=bar&baz=qux&baz=quux&corge='
```

- 39. `JSON.stringify(value[, replacer[, space]])` converts a JavaScript value to a JSON string

```

let person = {
  name: "Brad",
  age: 35
};

person = JSON.stringify(person);
// person = JSON.parse(person); // back to an object

console.log(person);

```

40. `JSON.parse(text[, reviver])` parses a JSON string, constructing the JavaScript value or object described by the string. trailing commas are not allowed, `JSON.parse('[1, 2, 3, 4,]')`; will throw an error
41. If you access a method **without** `()`, it will return the **function definition**
42. JS index position starts at zero!

Anatomy of an HTTP transaction

server.js:

```

const http = require('http');

http.createServer((request, response) => {
  console.log(request.method);
  console.log(request.url);

  request.on('error', (err) => {
    console.error(err);
    response.statusCode = 400;
    response.end();
  });

  response.on('error', (err) => {
    console.error(err);
  });

  if(request.method === 'POST' && request.url === '/echo'){
    // let body = [];
    // request.on('data', (chunk) => {
    //   body.push(chunk);
    // }).on('end', () => {
    //   body = Buffer.concat(body).toString();

```

```
//      response.writeHead(200, {'Content-Type': 'text/plain'});
//      response.end(body);
// });
request.pipe(response);
}else {
    response.statusCode = 404;
    response.end();
}
}).listen(8080);

console.log('Server listening on port 8080');
```

client.js

```
var http = require('http');
var querystring = require('querystring');

var postData = querystring.stringify({
    'msg': 'hello world!'
});

var options = {
    hostname: 'localhost',
    port: 8080,
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Content-Length': postData.length
    },
    agent: false,
    path: '/echo'
};

var req = http.request(options, function (res) {// function emitted when
    ↪ a response is received to this request
    //res is of type <http.IncomingMessage> and can be used to access
    ↪ response status, headers and data.
    //<http.IncomingMessage> implements Readable Stream interface
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
```

```

res.setEncoding('utf8');

// get data as chunks (stream or buffer)
res.on('data', function (chunk) {
  console.log('BODY: ' + chunk);
});

// end response
res.on('end', function () {
  console.log('No more data in response.')
});
});

req.on('error', function (e) {
  // console.log('problem with request: ' + e.message);
  console.error(e.stack);
});

// write data to request body
req.write(postData, 'utf8');

req.end(); //With http.request() one must always call req.end() to
→ signify the end of the request

```

[2-5] to \cite{2,3,4,5}

```

const input = "[2-5]";

var numRangeArr = input.match(/\d/gm);

var len = numRangeArr[1] - numRangeArr[0];

var resArr = [];

for(var i = 0; i <= len; i++){
  var res = Number(numRangeArr[0])+ i;
  resArr[i] = res;
  console.log(resArr);
}

```



```
var midRes = resArr.toString();

var result = "\\cite{" + midRes + "}";
```

Anonymous function:

```
var myArray = ["Sam", "Mark", "Tim", "Sam"];

/*anonymous function*/
var result = myArray.filter(function (value, index, array){return
    ↪ array.indexOf(value) == index;});

document.write(result);
```

```
/*let add = function(a,b){
    return a + b;
}

let multiply = function(a,b){
    return a * b;
}*/

let calc = function(num1, num2, callback){
    return callback(num1, num2);
}

console.log(calc(1, 2, function(a, b){
    return a-b;
})));
```

Factory pattern:

```
var peopleFactory = function(name, age, state){

    var temp = {};
    //var temp = new Object();

    temp.age = age;
    temp.name = name;
    temp.state = state;
```

```
temp.printPerson = function(){
    console.log(this.name + ", " + this.age + ", " + this.state);
}

return temp;
}

var person1 = peopleFactory("john", 23, "CA");
var person2 = peopleFactory("kim", 27, "SC");

person1.printPerson();
person2.printPerson();
```

Constructor pattern

```
var peopleConstructor = function(name, age, state){

    this.name = name;
    this.age = age;
    this.state = state;

    this.printPerson = function(){
        console.log(this.name + ", " + this.age + ", " + this.state);
    }
}

var person1 = new peopleConstructor("john", 23, "CA");
var person2 = new peopleConstructor("kim", 27, "SC");

person1.printPerson();
person2.printPerson();
```

Prototype pattern

```
var peopleProto = function(){

}

//prototype properties
peopleProto.prototype.age = 0;
peopleProto.prototype.name = "no name";
```

```
peopleProto.prototype.city = "no city";

peopleProto.prototype.printPerson = function(){
  console.log(this.name + ", " + this.age + ", " + this.city);
}

var person1 = new peopleProto();
person1.name = "John";
person1.age = 23;
person1.city = "CA";

console.log("name" in person1);
console.log(person1.hasOwnProperty("name"));

person1.printPerson();
```

Dynamic prototype pattern

```
//dynamic prototype pattern
var peopleDynamicProto = function(name, age, state){
  this.age = age;
  this.name = name;
  this.state = state;

  // create function only once
  if(typeof this.printPerson !== "function"){
    peopleDynamicProto.prototype.printPerson = function(){
      console.log(this.name + ", " + this.age + ", " + this.state);
    }
  }
}

var person1 = new peopleDynamicProto("John", 24, "CA");
var person2 = new peopleDynamicProto("Yu", 23, "ZJ");

console.log("name" in person1);
console.log(person1.hasOwnProperty("name"));

person1.printPerson();
person2.printPerson();
```

Closure:

```
var addTo = function(passed){  
  
    var add = function(inner){  
        return passed + inner;  
    }  
  
    return add;  
}  
  
var addTwo = addTo(2);  
var addThree = addTo(3);  
  
//console.dir(addTwo);  
//console.dir(addThree);  
  
console.log(addTwo(1));  
console.log(addThree(1));
```

callback function: A callback is a function that is passed as an argument to another function and is executed after its parent function has completed

```
let x = function(){  
    console.log("i am called from inside a function");  
}  
  
let y = function(callback){  
    console.log("do something");  
    callback();  
}  
  
y(x);
```

```
/*let calc = function(num1, num2, calcType){  
  
    if(calcType === "add"){  
        return num1 + num2;  
    }else if(calcType === "multiply"){  
        return num1 * num2;  
    }  
}
```

```
}

console.log(calc(1, 2, "multiply"));*/

let add = function(a,b){
  return a + b;
}

let multiply = function(a,b){
  return a * b;
}

let doWhatever = function(a,b){
  console.log("Here are the two numbers: ", a + ", " + b);
}

let calc = function(num1, num2, callback){
  if(typeof callback === "function"){
    return callback(num1, num2);
  }
}

console.log(calc(1, 10, add));
```

```
var myArr = [{
  num: 5,
  str: "apple"
},{
  num: 7,
  str: "cabbage"
},{
  num: 1,
  str: "ban"
}];

//anonymous function
myArr.sort(function(val1, val2){
  if(val1.str < val2.str){
    return -1;
  }
});
```

```
    }else{
        return 1;
    }
})

console.log(myArr);
```

promises

```
let promiseToCleanTheRoom = new Promise(function(resolve, reject){
    //cleaning the room
    let isClean = false;

    if(isClean){
        resolve("Cleaned up");
    }else{
        reject("not clean");
    }
})

promiseToCleanTheRoom.then(function(fromResolve){
    console.log("The room is " + fromResolve);
}).catch(function(fromReject){
    console.log("The room is " + fromReject);
})
```

```
let cleanRoom =function(){
    return new Promise(function(resolve, reject){
        resolve("Cleaned the room ");
    })
}

let removeGarbage = function(message){
    return new Promise(function(resolve, reject){
        resolve(message + "Remove garbage ");
    })
}

let winIcecream = function(message){
    return new Promise(function(resolve, reject){
```

```

    resolve(message + "Won icecream");
  })
}

cleanRoom().then(function(result){
  return removeGarbage(result);
}).then(function(result){
  return winIcecream(result);
}).then(function(result){
  console.log("Finished " + result);
})

//do everything in parallel
/*Promise.all([cleanRoom(), removeGarbage(),
  ↪ winIcecream()]).then(function(){
  console.log("All finished");
})*//

//any one of them
/*Promise.race([cleanRoom(), removeGarbage(),
  ↪ winIcecream()]).then(function(){
  console.log("One of them is finished");
})*//

```

call, apply and bind

```

var obj = {num:3};
var addToThis = function(a, b, c){
  return this.num + a + b + c;
}

// call
console.log(addToThis.call(obj, 1, 2, 3));

// apply
var arr = [1,2,3]; // only difference from `call`
console.log(addToThis.apply(obj, arr));

// bind
var bound = addToThis.bind(obj);

```

```
console.log(bound(1, 2, 3));
```

prototype inheritance

```
var x = function(j){
  this.i = 0;
  this.j = j;

  this.getJ = function(){
    return this.j;
  }
}

x.prototype.getJ = function(){
  return this.j;
}

var x1 = new x(1);
var x2 = new x(4);

console.log(x1.getJ()); // use the method from the parent class, instead
↳ of creating one of own
console.log(x2.getJ()); // use the method from the parent class, instead
↳ of creating one of own
```

```
// baseclass
var Job = function(){
  this.pays = true;
}

// prototype method
Job.prototype.print = function(){
  console.log(this.pays ? 'Please hire me' : 'no thank you');
}

// subclass
var TechJob = function(title, pays){
  Job.call(this); // inherits properties and methods from Job function

  this.title = title;
```



```

    this.pays = pays;
}

TechJob.prototype = Object.create(Job.prototype); // inherits from the
↳ prototype of Job
TechJob.prototype.constructor = TechJob; // set a constructor for
↳ TechJob

TechJob.prototype.print = function(){
    console.log(this.pays ? this.title + ' job is great, please hire me' :
↳ 'I would rather learn Javascript');
}

var softwarePosition = new TechJob('Javascript Programmer', true);
var softwarePosition2 = new TechJob('vb Programmer', false);

console.log(softwarePosition.print());
console.log(softwarePosition2.print());

```

HTML codes

```

<textarea name="text" row="5000" cols="100" id="inputtext"
↳ style="width:1200px; height:300px; background-color: rgb(204,232,204);
↳ border: 2px solid Tomato; font-size: 15px"></textarea>

```