JS Front End Interview Questions

```
/*
100 Front End Interview Questions Challenge

https://www.udemy.com/course/100-front-end-interview-questions-challenge/

DO DO:
- Watch & Understand
- Do (Write/Code the solution with you own words AND record and explain your own solut
ions)
- Rinse and Repeat
*/
```

Data Types

```
/*
JS Data Types

=> Primitives
    - Boolean (Boolean represents a logical entity and can have two values: true and f
alse.)

    - Null (Empty value user assigned. The Null type has exactly one value: null.)

    - Undefined (A variable that has not been assigned a value has the value undefine
d.)

    - Number (The Number type is a double-precision 64-bit binary format IEEE 754 valu
e (numbers between -(2^53 − 1) and 2^53 − 1). In addition to representing floating-poi
nt numbers, the number type has three symbolic values: +Infinity, -Infinity, and NaN
("Not a Number"). To check for the largest available value or smallest available value
within ±Infinity, you can use the constants Number.MAX_VALUE or Number.MIN_VALUE.)

    - BigInt (The BigInt type is a numeric primitive in JavaScript that can represent
integers with arbitrary precision. With BigInts, you can safely store and operate on l
arge integers even beyond the safe integer limit for Numbers.)

    - String (Used to represent textual data. It is a set of "elements" of 16-bit unsi
gned integer values. Each element in the String occupies a position in the String. The
first element is at index 0, the next at index 1, and so on. The length of a String is
the number of elements in it. Unlike some programming languages (such as C), JavaScrip
t strings are immutable. This means that once a string is created, it is not possible
to modify it.)

    - Symbol (A Symbol is a unique and immutable primitive value and may be used as th
e key of an Object property. In some programming languages, Symbols are called "atom
s".)

=> Objects (Complex data type that allows you to store collections of data. An object
contains properties, defined as a key-value pair. A property key (name) is always a st
ring, but the value can be any data type, like strings, numbers, booleans, or complex
data types like arrays, function and other objects.)
```

```
Not Data Type

    - Arrays (Type of object used for storing multiple values in single variable. Each value (also called an element) in an array has a numeric position, known as its index, and it may contain data of any data type-numbers, strings, booleans, functions, objects, and even other arrays.)

    - Function (Callable object that executes a block of code. Since functions are objects, so it is possible to assign them to variables)

    - Maps (The Map object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a key or a value.)

    - Sets (Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur once; it is unique in the Set's collection.)
*/
```

Const vs let vs var

```
/*
"const" and "let" were added on ES6 to solve the problem that "var" have with lexical scope and hoisting. Meaning that when declaring variable with "var", this variable will hoist even if its value is never run, creating an undesirable value for the outer scope code if this variable where to be used. This is a huge issue since the code will run/render differently that it would read, and that will cause inherent bug on our software.

"const" and "let" on the other hand implement what we call block scope, meaning thos variable will only exist on that specific block of code solving the issue there of "var"
*/
```

Pass by Value & pass by Reference

```
/*
Pass by Value and Reference like the name mention is when we assign or pass as parameter either by its value or by a reference to the value. Primitives are bound to pass by value and objects are passed by reference. When we passed by value like mentioned before when make a copy of the value, so if the copied value is modified the original value is still the same. On the other hand if we pass by reference, if we modified the copied value we will also modify the original since is only a copied reference of the value on memory, a pointer if you might, like on C++. This is why we need to be careful while upating objects on React if we want save/use the previous value, in that case the recommended strategy is to make a copy of the object eithe with the map() method for arrays or the JSON.stringify() and JSON.parse() for objects, or any other pre-existing methods on the current JS landscape (ES6 spread operator, lodash, underscore, etc).
*/
```

Array Methods

```
// => map() - The map() method creates a new array populated with the
// results of calling a provided function on every element in the calling array.
const array1 = [1, 4, 9, 16];
// pass a function to map
const map1 = array1.map((x) => x * 2);
```

```
console.log(map1); // Array [2, 8, 18, 32]

// => filter() - The filter() method creates a new array with all elements that
// pass the test implemented by the provided function.
const words = [
  "spray",
  "limit",
  "elite",
  "exuberant",
  "destruction",
  "present",
];
const result = words.filter((word) => word.length > 6);
console.log(result); // Array ["exuberant", "destruction", "present"]

// => reduce() - The reduce() method executes a user-supplied "reducer" callback
// function on each element of the array, in order, passing in the return value
// from the calculation on the preceding element. The final result of running the
// reducer across all elements of the array is a single value. The first time that
// the callback is run there is no "return value of the previous calculation".
// If supplied, an initial value may be used in its place. Otherwise the array
// element at index 0 is used as the initial value and iteration starts from the
// next element (index 1 instead of index 0).
const array2 = [1, 2, 3, 4];
const initialValue = 0;
const sumWithInitial = array2.reduce((previousValue, currentValue) => {
  return previousValue + currentValue;
}, initialValue);
console.log(sumWithInitial); // 10
```

Object Methods

```
// => for...in - The for...in statement iterates over all enumerable properties
// of an object that are keyed by strings (ignoring ones keyed by Symbols),
// including inherited enumerable properties.

const object = { a: 1, b: 2, c: 3 };

for (const property in object) {
  console.log(`${property}: ${object[property]}`);
}
```

Falsey Values

```
// => Falsey values

// undefined
// false
// null
// NaN
// '' or ""
// 0

console.log(Boolean(undefined));
console.log(Boolean(false));
console.log(Boolean(null));
```

```
console.log(Boolean(NaN));
console.log(Boolean(""));
console.log(Boolean(0));

// => Confusing values
console.log(Boolean([]));
console.log(Boolean({}));
```

this

```
// => this is used to access te current context
const exObj = {
  name: "Hello",
  last: "World",
  displayMessage: function () {
    return `${this.name} - ${this.last}`;
  },
};

console.log(exObj.displayMessage());
```

== vs ===

```
// == check for only value ( 5 == 5 => true ) ( "5" == 5 => true )

// === check for value and type ( 5 === 5 => true ) ( "5" === 5 => false )
```

Coercion

```
// Type coercion is the automatic or implicit conversion of values from one
// data type to another (such as strings to numbers). Type conversion is similar
// to type coercion because they both convert values from one data type to another
// with one key difference — type coercion is implicit whereas type conversion
// can be either implicit or explicit.

const value1 = "5";
const value2 = 9;
let sum = value1 + value2;

console.log(sum);
```

typeof

```
// The typeof operator returns a string indicating the type of the unevaluated
// operand.

console.log(typeof 42);
// expected output: "number"

console.log(typeof "blubber");
// expected output: "string"
```

```
console.log(typeof true);
// expected output: "boolean"

console.log(typeof undeclaredVariable);
// expected output: "undefined"

console.log(typeof function () {});
// expected output: "function"

// Gotchas: All not primitives will return "object", you will need to check what
// type of object is by using a prototype function instead
```

delete

```
// The JavaScript delete operator removes a property from an object; if no more
// references to the same property are held, it is eventually released
// automatically.

const Employee = {
  firstname: "John",
  lastname: "Doe",
};

console.log(Employee.firstname);
// expected output: "John"

delete Employee.firstname;

console.log(Employee.firstname);
// expected output: undefined

console.log(Employee);
// expected output: { lastname: 'Doe'}

// The use of delete is not recommended to use since we don't want to be changing
// the data model in our objects across our app, this could lead to crashed and
// bugs. Instead we could set the values of that property to some initial value
// like 0, null or empty string.
```

Object Notation

```
// Dot notation => We can access the object's properties and methods using dot notatio
n. The object name (person) acts as the namespace — it must be entered first to access
anything inside the object. Next you write a dot, then the item you want to access — t
his can be the name of a simple property, an item of an array property, or a call to o
ne of the object's methods, for example:

const person = {
  name: ["Bob", "Smith"],
  age: 32,
  bio() {
    return `${this.name[0]} ${this.name[1]} is ${this.age} years old.`;
  },
  introduceSelf() {
```

```
    return `Hi! I'm ${this.name[0]}.`;
  },
};

console.log(person.age);
console.log(person.bio());

// Bracket Notation => There is another way to access object properties — using bracke
t notation. This looks very similar to how you access the items in an array, and it is
basically the same thing — instead of using an index number to select an item, you are
using the name associated with each member's value. It is no wonder that objects are s
ometimes called associative arrays — they map strings to values in the same way that a
rrays map numbers to values.

console.log(person["age"]);
console.log(person["name"]);
```

Strict Mode

```
//  JavaScript's strict mode, introduced in ECMAScript 5, is a way to opt in to a rest
ricted variant of JavaScript, thereby implicitly opting-out of "sloppy mode". Strict m
ode isn't just a subset: it intentionally has different semantics from normal code. Br
owsers not supporting strict mode will run strict mode code with different behavior fr
om browsers that do, so don't rely on strict mode without feature-testing for support
for the relevant aspects of strict mode. Strict mode code and non-strict mode code can
coexist, so scripts can opt into strict mode incrementally.

// Strict mode makes several changes to normal JavaScript semantics:

// Eliminates some JavaScript silent errors by changing them to throw errors.
// Fixes mistakes that make it difficult for JavaScript engines to perform optimizatio
ns: strict mode code can sometimes be made to run faster than identical code that's no
t strict mode.
// Prohibits some syntax likely to be defined in future versions of ECMAScript.
```

Anonymous Functions

```
//  An anonymous function is a function without a name. An anonymous function is not a
ccessible after its initial creation. Therefore, you often need to assign it to a vari
able.

const func = function () {
  console.log("AnonFunc");
};

func();
```

Callbacks

```
// A callback function is a function passed into another function as an argument, whic
h is then invoked inside the outer function to complete some kind of routine or actio
n.
```

```
function greeting(name) {
  alert("Hello " + name);
}

function processUserInput(callback) {
  var name = prompt("Please enter your name.");
  callback(name);
}

processUserInput(greeting);

// The above example is a synchronous callback, as it is executed immediately.

// Note, however, that callbacks are often used to continue code execution after an as
ynchronous operation has completed — these are called asynchronous callbacks. A good e
xample is the callback functions executed inside a .then() block chained onto the end
of a promise after that promise fulfills or rejects. This structure is used in many mo
dern web APIs, such as fetch().
```

Closures

```
// A closure is the combination of a function bundled together (enclosed) with referen
ces to its surrounding state (the lexical environment). In other words, a closure give
s you access to an outer function's scope from an inner function. In JavaScript, closu
res are created every time a function is created, at function creation time.

const add = (function () {
  let counter = 0;

  return function () {
    counter++;
    return counter;
  };
})();

console.log(add()); // 1
console.log(add()); // 2
console.log(add()); // 3
```

Naming and Standards

```
// For JS, camel case, when dealing with booleans try to use the "isHas" prefix or som
ething similar, when dealing with function call that return something prefixed with a
verb "getValue", "setValue" etc. and have some consistency across the board with the a
rrange convention, ex. lets name "update..." to the functions that use PUT ajax calls
and so forth.

// CSS dashed naming

// Ultimately the ideal set up will be where you insert yourself in the company's work
flow and improve and add code following all the conventions of that the company is alr
eady in place, so this is a very open ended question that really depend on the company
preexistign workflow and architectures.
```

Empty Array

```
// How to empty an array

// array = [];

// array.length = 0;

// while(array.length > 0) {
//     array.pop();
// }

// while(array.length > 0) {
//     array.shift();
// }
```

MUL Function

```
// In this function, we call the function that required an argument as a first number,
and that function calls another function that required another argument and this step
goes on.

function mul(x) {
  return function (y) {
    return function (z) {
      return x * y * z;
    };
  };
}

console.log(mul(5)(2)(3));
```

Create Array

```
// const arr = [1];

// const arr2 = arr1.map((val) => {
//   return val + 1;
// });

// const badArr = new Array();
```

Undefined vs Undeclared vs null

```
//  undeclared => variable does not exist

//  undefined => The value undefined means value is not assigned & you don’t know its
value. It is an unintentional absence of value. It means that a variable has been decl
ared but has not yet been assigned a value. In case of non primitives like objects or
arrays it can mean lack of existence, but the object does exist in that case.

//  null => The value null indicates that you know that the field does not have a valu
```

```
e. It is an intentional absence of value. The system does not return null, is an inten
tional value.
```

0.2 + 0.1

```
//  Problem, math in JS is a little weird using decimals

// Problem
console.log(0.2 + 0.1);

// Solution
console.log((0.2 + 0.1).toFixed(1));
```

instanceof

```
// Is use when check for prototypes in JS. Not very useful to check against primitive
s.

console.log(4 instanceof Number);
console.log("Hello world" instanceof String);

console.log({} instanceof Object);
console.log([] instanceof Object);

class Circle {}
const circle = new Circle();
console.log(circle instanceof Circle);
console.log(circle instanceof Object);
```

IIFE

```
//  Inmediately Invoqued Function Expression

function noIIFE() {
  console.log("IIFE");
}

(function IIFE() {
  console.log("IIFE");
})();
```

Maintaining State and Local Storage

```
// Using a 3rd Party Store like Redux
// Using Local Storage || Session Storage
// Using cookies
// Global variables (bad idea)

localStorage.setItem("firstName", "Marx");
console.log(localStorage.getItem("firstName")); // Marx
```

```
const user = JSON.stringify({
  name: "John",
  last: "Doe",
});

localStorage.setItem("user", user);
console.log(JSON.parse(localStorage.getItem("user"))); // { name: "John", last: "Doe"
}

localStorage.clear();
localStorage.removeItem();

// etc
```

HTML5 API

```
// Geolocation API

if (navigator.geolocation) {
  // navigator object gives you access to location, check first if exist, some browser
s block it
  navigator.geolocation.getCurrentPosition(
    (pos) => {
      console.log("Location", pos);
    },
    () => {
      console.log("No location");
    }
  );
}

// Drang and Drop API
// Wep Workers API
// and more...
```

Destructuring Objects and Arrays

```
// A way to selectively work with objects and array.

const [month, day, year] = [08, 09, 1990];
console.log(`${month}/${day}/${year}`);

const { name: firstName, nickname } = {
  last: "Doe",
  name: "John",
  middle: "Arthur",
  nickname: "Johnny",
};

console.log(firstName);
console.log(nickname);
```

Rest Parameter and Spread Operator

```
// The rest parameter syntax allows a function to accept an indefinite number of argum
ents as an array, providing a way to represent variadic functions in JavaScript.

// Old code
function add1(num) {
  console.log(num); // 5
  console.log(Array.prototype.slice.call(arguments)); // [5, 6, 7, 8] Do the job but n
ot very clean
  console.log([].slice.call(arguments)); // [5, 6, 7, 8] Do the job but not very clean
  console.log(Array.from(arguments)); // [5, 6, 7, 8] Do the job but not very clean
}
add1(1, 2, 3, 4, 5);

// New code
function add2(...num) {
  console.log(num); // [6, 7, 8, 9, 10]
}
add2(6, 7, 8, 9, 10);
```

```
// Spread syntax (...) allows an iterable such as an array expression or string to be
expanded in places where zero or more arguments (for function calls) or elements (for
array literals) are expected, or an object expression to be expanded in places where z
ero or more key-value pairs (for object literals) are expected.
function add3(num) {
  const vals = [1, 2, 3, 4, 5, ...num];
  console.log(vals); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
add3([6, 7, 8, 9, 10]);
```

Arrow Functions

```
// An arrow function expression is a compact alternative to a traditional function exp
ression, but is limited and can't be used in all situations.

// Differences & Limitations:

// Does not have its own bindings to this or super, and should not be used as methods.
// Does not have new.target keyword.
// Not suitable for call, apply and bind methods, which generally rely on establishing
a scope.
// Can not be used as constructors.
// Can not use yield, within its body.

// Old code
function example1(param1) {
  return param1 + 1;
}
console.log(example1(9));

// New code
const example2 = (param2) => param2 + 2;
console.log(example2(8));

// New code variant
const example3 = (param3) => {
  return param3 + 3;
```

```
};
console.log(example3(7));
```

NaN

```
// typeof Nan = Number
// We will get NaN when we try to do something not supported by the language

console.log("" / 0);
console.log(0 / 0);
console.log(parseInt("hello"));

console.log(isNaN(NaN));
```

DOM Selectors

```
// There are 4 main way of do this

document.getElementById("id-example");

document.getElementsByTagName("h1");

document.getElementsByClassName("class-example");

// Slower solutions
document.querySelector("body");
document.querySelectorAll("h1.some-class");
```

Copy Objects

```
const obj1 = {
  first: "John",
  last: "Doe",
  contry: "USA",
};
console.log(obj1);

const obj2 = { ...obj1 }; // Best way
obj2.first = "Jane";
console.log(obj1);
console.log(obj2);

const obj4 = JSON.parse(JSON.stringify(obj1));
obj4.first = "Isaak";
console.log(obj1);
console.log(obj4);

const obj3 = obj1; // Bad idea. Pass by reference.
obj3.first = "Joe";
console.log(obj1);
console.log(obj3);

// How to check if 2 objects are the same. Not have the sane value, are the same objec
```

```
t itself, aka are pointing to the same space in memory.
console.log(Object.is(obj1, obj2));
console.log(Object.is(obj1, obj3));
console.log(Object.is(obj1, obj4));
```

Comparing 2 Objects

```
const user1 = {
  first: "John",
  last: "Doe",
  country: "USA",
};

const user2 = {
  first: "John",
  last: "Doe",
  country: "USA",
};

console.log(user1 === user2); // No good. These comparisons don't work with non primit
ives

// If keys are in the same order
const stringUser1 = JSON.stringify(user1);
const stringUser2 = JSON.stringify(user2);
console.log(stringUser1 === stringUser2);

function isSameObject(obj1, obj2) {
  const prop1 = Object.getOwnPropertyNames(obj1);
  const prop2 = Object.getOwnPropertyNames(obj2);

  console.log(prop1);
  console.log(prop2);

  if (prop1.length != prop2.length) {
    return false;
  }

  for (let i = 0; i < prop1.length; i++) {
    // Create an array with the properties so we can use after. Doesnt matter is if pr
op1 or prop2, they both have the same properties as filtered on the previous if statem
ent.
    const properties = prop1[i];

    if (obj1[properties] !== obj2[properties]) {
      return false;
    }
  }

  return true;
}

console.log(isSameObject(user1, user2));
```

Service Workers

```
// Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server. They will also allow access to push notifications and background sync APIs.

// The Worker interface of the Web Workers API represents a background task that can be created via script, which can send messages back to its creator.

// A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web-page/site that it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available).

// A service worker is run in a worker context: it therefore has no DOM access, and runs on a different thread to the main JavaScript that powers your app, so it is non-blocking. It is designed to be fully async; as a consequence, APIs such as synchronous XHR and Web Storage can't be used inside a service worker.

// Service workers only run over HTTPS, for security reasons. Having modified network requests, wide open to man in the middle attacks would be really bad. In Firefox, Service Worker APIs are also hidden and cannot be used when the user is in private browsing mode.
```

Removing Duplicates

```
// Removing duplicate from arra can be done by 2 methos. One is convert the array in a set, which doesnt allow duplicates, and we are done. And the second methos involves iterate our array and call the method includes against the original aray while we build our final array with no duplicated.

// Method 1

const nums = [1, 1, 2, 2, 2, 3, 3, 3];
const numSet = new Set();
nums.forEach((value) => numSet.add(value));
const newNums = Array.from(numSet);
console.log(newNums);

// Method 2

const singleValArr = [];
nums.forEach((value) => {
    if(!singleValArr.includes(value)) {
        singleValArr.push(value);
    }
});

console.log(singleValArr);
```

Async & Await

```
<script>
    // async function getUsers() {
    //     console.log(1);
    //     try {
    //         const res = await fetch("https://api.github.com/users/carlostrujillo9
0");
    //         const response = await res.json();
    //         console.log(response);
    //     } catch (e) {
    //         console.log(e);
    //     }
    //     console.log(2);
    // }
    // getUsers();


    function getUsers() {
        console.log(1);

        fetch("https://api.github.com/users/carlostrujillo90")
            .then((res) => console.log(res.json()));

        console.log(2);
    }
    getUsers();
</script>
```

Promises

```
// The Promise object represents the eventual completion (or failure) of an asynchrono
us operation and its resulting value.

// const myPromise = new Promise((resolve, reject) => {
//   setTimeout(() => {
//     resolve("foo");
//   }, 300);
// });

// myPromise
//   .then(handleResolvedA, handleRejectedA)
//   .then(handleResolvedB, handleRejectedB)
//   .then(handleResolvedC, handleRejectedC);

let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was successful, and r
eject(...) when it failed.
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5 API.
  setTimeout(function () {
    resolve("Success!"); // Yay! Everything went well!
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function above.
  // It doesn't have to be a string, but if it is only a succeed message, it probably
will be.
```

```
  console.log("Yay! " + successMessage);
});
```

HTTP Methods

```
// HTTP defines a set of request methods to indicate the desired action to be performe
d for a given resource. Although they can also be nouns, these request methods are som
etimes referred to as HTTP verbs. Each of them implements a different semantic, but so
me common features are shared by a group of them: e.g. a request method can be safe, i
dempotent, or cacheable.

// GET
// The GET method requests a representation of the specified resource. Requests using
GET should only retrieve data.

// HEAD
// The HEAD method asks for a response identical to a GET request, but without the res
ponse body.

// POST
// The POST method submits an entity to the specified resource, often causing a change
in state or side effects on the server.

// PUT
// The PUT method replaces all current representations of the target resource with the
request payload.

// DELETE
// The DELETE method deletes the specified resource.

// CONNECT
// The CONNECT method establishes a tunnel to the server identified by the target reso
urce.

// OPTIONS
// The OPTIONS method describes the communication options for the target resource.

// TRACE
// The TRACE method performs a message loop-back test along the path to the target res
ource.

// PATCH
// The PATCH method applies partial modifications to a resource.
```

HTTP Response Status

```
// HTTP response status codes indicate whether a specific HTTP request has been succes
sfully completed. Responses are grouped in five classes:

// Informational responses (100–199)
// Successful responses (200–299)
// Redirection messages (300–399)
// Client error responses (400–499)
// Server error responses (500–599)
```

REST

```
// REST (Representational State Transfer) refers to a group of software architecture d
esign constraints that bring about efficient, reliable and scalable distributed system
s.

// The basic idea of REST is that a resource, e.g. a document, is transferred via well
-recognized, language-agnostic, and reliably standardized client/server interactions.
Services are deemed RESTful when they adhere to these constraints.

// HTTP APIs in general are sometimes colloquially referred to as RESTful APIs, RESTfu
l services, or REST services, although they don't necessarily adhere to all REST const
raints. Beginners can assume a REST API means an HTTP service that can be called using
standard web libraries and tools.


// The six constraints are:

// Uniform Interface: The uniform interface constraint defines the interface between c
lients and servers. It simplifies and decouples the architecture, which enables each p
art to evolve independently.

// Stateless: As REST is an acronym for REpresentational State Transfer, statelessness
is key. Essentially, what this means is that the necessary state to handle the request
is contained within the request itself, whether as part of the URI, query-string param
eters, body, or headers. The URI uniquely identifies the resource and the body contain
s the state (or state change) of that resource. Then after the server does it's proces
sing, the appropriate state, or the piece(s) of state that matter, are communicated ba
ck to the client via headers, status and response body.

// Cacheable: As on the World Wide Web, clients can cache responses. Responses must th
erefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent
clients reusing stale or inappropriate data in response to further requests. Well-mana
ged caching partially or completely eliminates some client–server interactions, furthe
r improving scalability and performance.

// Client-Server: The uniform interface separates clients from servers. This separatio
n of concerns means that, for example, clients are not concerned with data storage, wh
ich remains internal to each server, so that the portability of client code is improve
d. Servers are not concerned with the user interface or user state, so that servers ca
n be simpler and more scalable. Servers and clients may also be replaced and developed
independently, as long as the interface is not altered.

// Layered System: A client cannot ordinarily tell whether it is connected directly to
the end server, or to an intermediary along the way. Intermediary servers may improve
system scalability by enabling load-balancing and by providing shared caches. Layers m
ay also enforce security policies.

// Code on Demand (optional): Servers are able to temporarily extend or customize the
functionality of a client by transferring logic to it that it can execute. Examples of
this may include compiled components such as Java applets and client-side scripts such
as JavaScript. Complying with these constraints, and thus conforming to the REST archi
tectural style, will enable any kind of distributed hypermedia system to have desirabl
e emergent properties, such as performance, scalability, simplicity, modifiability, vi
sibility, portability and reliability.
```

Synchronous vs Asynchronous

```
// In synchronous operations tasks are performed one at a time and only when one is co
mpleted, the following is unblocked. In other words, you need to wait for a task to fi
nish to move to the next one.

// In asynchronous operations, on the other hand, you can move to another task before
the previous one finishes. This way, with asynchronous programming you’re able to deal
with multiple requests simultaneously, thus completing more tasks in a much shorter pe
riod of time.
```

Timers

```
<script>
    // The setTimeout() method calls a function after a number of milliseconds.
    let timer = setTimeout(() => { console.log("Hello World") }, 3000);

    // The setInterval() method, offered on the Window and Worker interfaces, repeated
ly calls a function or executes a code snippet, with a fixed time delay between each c
all.
    // This method returns an interval ID which uniquely identifies the interval, so y
ou can remove it later by calling clearInterval() or clearTimeout().
    let interval = setInterval(myCallback, 3000, 'Parameter 1', 'Parameter 2');
    function myCallback(a, b) {
        // Your code here
        // Parameters are purely optional.
        console.log(a);
        console.log(b);
    }

    // The global clearTimeout() or clearInterval() method cancels a timer previously
established.
    clearTimeout(interval);
</script>
```