שאלות ידע:

מערך מול רשימה מקושרת:

מערך: נמצא בזכרון כבלוק כך שהאלמנטים ממוקמים באופן רציף

רשימה: האיברים מפוזרים בזכרון עם רפרנס לאיבר הבא

הכנסה למערך: אם זה בסוך אז מהיר, אם לא אז יכול להיות o(n)

רשימה:

In summary, arrays are better suited for scenarios where random access and memory locality are important, and the size of the collection is known in advance or doesn't change frequently. On the other hand, linked lists are more appropriate when frequent insertions and deletions are required, or when the size of the collection can change dynamically.

מערך: כאשר ידוע מראש הגודל וגודל המערך לא משתנה כל הזמן

רשימה: כאשר משתנה כל הזצן הגודל ויש הכנסות והוצאות כל הזמן

שפות תכנות:

C: תכנות פרוצדורלי: תבנית תכנות או גישה עצובית של כתיבת הקוד. למשל ב C, מחולק לאוסף של פונקציו תהקוראות אחת לשניה. ניתן ב C לתחזק את הזכרון עם מצביעים. משומש בעיקר בתוכנות שעוסקות בעיקר באינטראקציה עם החומרה ומערכת ההפעלה של המחשב

C++, JAVA: כאן הגישה היא תכנות מונחה עצמים שבין העצמים קיימת הירריכה ויחסים מסויימים.

פייתון: כאן יש מפענח, כלומר הקוד לא מקומפל לפני הריצה. כאן יש טיפוסים דינאמיים, מוגדרים בזמן ריצה. משתמשת לscientific computing, data analysis, machine learning web. יש לה ספריות רבות, סינטקס קל, קהילה גדולה ודוקומנטציה.

JavaScript:גם כאן שפה עם מפענח, לרוב משומשת בצד לקוח. even loop:

ה-stack:

כאן נשמרות הקריאות לפונקציות. אם יש קריאה לפונקציה פו, היא מתווספת לסטאק, ואם היא קראה לפונקציה באז, אז באז מתווספת לסטאק מעליה, ועד שבאז לא מסתיימת, ג'אווהסקריפט לא חוזרת לפו.

Event loop:

רכיב שתפקידו לבדוק אם הסטאק ריק, ואם הוא ריק לקחת משימה מה-

callback queue

ולהעביר אותה לסטאק.

Web API:

זה ממשק חיצוני, שקשור בדפדפן, ואחראי על הרצה של משימות אסינכרוניות (כמו קריאות לשרת).

כמובן שגם בנוד (וגם בסביבות אחרות), יש מקבילה לזה, שאחראית על הרצת משימות אסינכרוניות.

כשה-runtime מריץ פונקציה "רגילה" של קוד (כלומר סינכרונית) - היא מתווספת ל-call stack, ואז מתבצעת, ומוצאת משם.

כשהוא מריץ פקודה א-סינכרונית, הפקודה היא למעשה קריאה ל-web api (ולכן הת'רד יכול להמשיך בענייניו ולהריץ את פקודות מהסטאק).

ה-runtime קורא לפקודה מה-web api, וממשיך להריץ פקודות מהסטאק.

כשה-web api מסיים את הפקודה (נגיד הנתונים חזרו מהשרת), הוא מוסיף את הקוד שצריך הרצה ל- callback queue.

ה-event loop מחכה שהסטאק יהיה ריק, ואז לוקח את המשימה הבאה מה-callback queue, ומעביר אותה לסטאק, להרצה.

כך שהטריק הוא שקוד א-סינכורני לא באמת רץ בת'רד של ג'אווהסקריפט, וג'אווהסקריפט רק מתקשרת עם התוצאות של ההרצות האלה (כשאין לה משהו אחר לעשות).

https://www.youtube.com/watch?v=lqLSNG\_79lI&ab\_channel=JamesQQuick

event loop:

callStack: מה שנכנס ישר מבוצע אם לא א-סינכרוני.

webAPI: מגיע אליו דרך call stack, הוא שולח את כל א-סינכרוני

task queueץ כאן נשלח משימות אסינכרוניות בעדיםות גבוהה.

evebt queue: לכאן נכסו המשימות אחרי שסיימו את מה שביצעו ב webAPI.

event loop. יבדוק גאשר callstack ריק יבדוק את event queue (קודם יבדוק את טסק קיו) וזש ישלח משם ל call stack.

ASYNC/AWAIT

כאן נוצר המתנה בתוך הפונקציה המוגדרת א-סינכרונית. אבל מחוץ לפונקציה לא תהיה המתנה

יש לו עדיפות על טאסק קיו ולכן יתבצע קודם, כלומק משימותשהן פרומיס יתבצעו לפני סט טיימאוטMicrotask Queue.

Closure: היכולת של פונקציה פנימית לגעת בסקופ של פונקציית האם, כלומר ניתן לגעת בכל המשתנים כל פעם שהפונקציה הפנימית מופעלת

hoist: כאשר משתנים ופונקציות עוברות לראש הסקופ: 1. משתנים: הבדלים בין var, let const: const ו let עולים למעלה אבל לא מאותחלים עם undefined. ואילו var כן. לכן נקבל שגיאה כאשר נפנה לאיבר לפני שאותחלץ functions: גם הן מגיעות לראש הסקופ אבל לא פונקציות חץ.

This: הבדלים בין פונקציה רגילה פונקציית חץ:

פונקציה רגילה: אם הפונקציה היא מתודה בתוך אובייקט, אז ה this הוא האובייקט עצמו.

אם לא בתוך אובייקט אז ה this הוא global

Var: הוא מוגבל לסקופ של הפונקציה או לגלובלי, אבל לא לבלוק סקופ, כלומר for או if

Array.sort() הוא עושה מיון לפי ערך הסטרינג של האלמנטץ כלומר 2 יבוא אחרי 1 או 1000

**Shallow Copy**:: לעשות העתקה רדודה של אובייקט, כלומר אם באובייקט עצמו קיים אובייקט אז העתקה רדודה תגרום לאובייקט החדש להצביע על אותו אובייקט מכונן שנמצא אצלו ובכך לגרום לאובייקט הישן להיות חשוף לשינויים.

new.target: נותן לנו את המידע אם בנאי או פונקציה נקראו עםnew , אם כן שומש אז יחזיר את השם של הclass או את שם הפונקציה:

דוגמא:

class BaseClass {

  constructor() {

    console.log(new.target)//BaseClass

  }

}

let baseObj = new BaseClass();

function new and not new: כאשר משתמשים ב new אז מה שיחזור אלינו הוא אובייקט שמייצג את ה this שבתוך הפונקציה. למשל:

function f1() {

    this.y = 21;

    let x = 10;

    console.log("hi");

    return x;

  }

let noNewFunc2 = new f1();

console.log(noNewFunc2, noNewFunc2.y);//f1 { y: 21 }

בלי new פשוט נקבל את ה x שחוזר אלינו מהפונקציה. הthis בתוך הפונקציה יהיה הגלובלי

עוד דוגמא של new:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

const lydia = new Person('Lydia', 'Hallie');

const sarah = Person('Sarah', 'Smith');

console.log(lydia);

console.log(sarah);

כאשר משתמשים ב new יוצרים אובייקט חדש ומוסיפים לו את הנתונים מתוך הפונקציה ב this. כאשר לא משתמשים ב new אז ה this בפונקציה יהיה ה גלובל. ומה שיחזור לנו מהפונקציה זה undifiend כי אנחנו לא עושים return

Person {firstName: "Lydia", lastName: "Hallie"} and undefined

Event bubbling:

כאשר יש אירוע בטאג של html, למשל בכפתור, אז האבנט יפעיל שרשרת כלפי מעלה לכל האלמנטים העוטפים את הכפתור, למשל האבא שלו div, גם אצלו האירוע יגיע.

== ו === עם אובייקט לא עובד כמו עם משתנה פרמיטיבי. כאן בודקים את הreference שלו, כלומר אם מצביעים לאותו אובייקט:

function checkAge(data) {

if (data === { age: 18 }) {

console.log('You are an adult!');

} else if (data == { age: 18 }) {

console.log('You are still an adult.');

} else {

console.log(`Hmm.. You don't have an age I guess`);

}

}

checkAge({ age: 18 });

אז כאן נלך לelse

Browser storage: האפשרות לשמור מידע על גבי הדפדפן

יש כמה סוגים:

1. Cookies: שמירת מידע על המשתמש, ניתן להשתמש במידע של קוקיז גם בצד שרת וגם בצד לקוח. יש להם גבול של מידע עד 4 קילובייט
2. Local storage: נותן אפשרות לשמור מידע על ידי key value pair. ניתן לשמור עד 10 מגה בייט, ניתן לשמור מידע גם אם הדפדפן נסגר. ניתן לשימוש רק בצד לקוח.
3. Session storage. כמו local storage גם כאן זה אובייקט עם מפתח וערך. כאן המידע לא נשמר ברגע שסוגרים את הדפדפן לכן אין צורך לנקות

Object:

כל המפתחות של אובייקט הם string אפילו אם ציינת במפורש שהוא לא. לדוגמא:

const obj = { 1: 2 };

console.log(obj.hasOwnProperty("1"));//true

console.log(obj.hasOwnProperty(1));//true

אם אנחנו משתמשים ב Map אז המפתחות הם כפי שהגדרנו:

const obj = new Map();

obj.set(1, 2);

console.log(obj.has("1")); //false

console.log(obj.has(1)); //true

Generator Functions yield

הגנרטורים ב-ECMAScript 6 מאפשרים לנו ליצור פונקציות/לולאות או קטעי קוד אחרים ש'אפשר להפריע להם'. הפונקציה עוצרת, במקום שבו אנו בוחרים כמובן ומחזירה תוצאה. הקוד שמחוץ לפונקציה יכול לקבל את התוצאה ולהתניע מחדש את הפונקציה.

פונקציות כאלו, שיכולות לעצור את עצמן ואז להמשיך נקראות 'גנרטורים', הן נראות ממש כמו פונקציות רגילות, רק שיש להן כוכבית לפני השם. פקודת העצירה נקראת yield. ברגע שפונקציה קוראת ל-yield, קורים שני דברים – הראשון הוא שה-yield מחזיר ערך והדבר השני הוא שהפונקציה עוצרת עד שמשהו חיצוני ממשיך אותה.

ההתנעה הראשונית של הפונקציה וההתנעות הנוספות בעקבות כל yield נעשות באמצעות next.

דוגמא:

function\* generator(i) {

  yield i;

  yield i \* 2;

}

const gen = generator(10);

  console.log(gen.next().value);

  // Wait for 5 seconds

  await wait(5000);

  console.log(gen.next().value);

Class: כאשר עושים new אז נוצר מופע של המחלקה וה this יהיה קשור למחלקה שנוצרה.

כאשר אין new נקבל שגיאה

לסיכום: הבדלים עיקריים: כאשר משתמשים ב new זה פועל כמו בנאי ויוצר מופע של הפונקציה או המחלקה. בנוסף שימוש בthis הוא שונה כאשר משתמשים ב new.

This: יש הבדלים בין סביבות עבודה בין node.js לבין עבודה בדפדפן:

דפדפן: ה this הגלובלי יהיה window. גם בחוץ וגם בתוך פונקציה

Node.js: כאן יש התנהגות שונה: בתוך פונקציה: this יהיה ה גלובלי. מחוץ לפונקציה זה יהיה module.export

התנהגות דוגמאות:

function f() {

  console.log(this); //global

}

console.log(this); //module.export

const obj = {

  title: "party",

  names: ["ran", "ran2"],

  checkName() {

    this.names.map(function (item, index) {

      console.log(this.title); //undefined inside regular function this is global

      return item;

    });

  },

  checkName2() {

    this.names.map((item, index) => {

      console.log(this.title); //party, in arrow function this wiil be the this from the outside object

      return item;

    });

  },

  checkName3() {

    let that = this;

    this.names.map(function (item, index) {

      console.log(that.title); //party, we are saving the this before the function and then we will use it

      return item;

    });

  },

};

a prototype is a mechanism that allows objects to inherit properties and methods from other objects. Every JavaScript object has a prototype property, which points to another object. This prototype object itself may have a prototype, and so on, forming a prototype chain

Prototyping היא דרך להוספת מתודות ותכונות לאובייקטים מסוג Native (מקומיים) של JS. ניתן להוסיף תכונות ומתודות לאובייקטים בצורה פשוטה מאוד,  
כך למשל, ניתן להוסיף ל-String מתודה שמוחקת את הרווחים המיותרים משני צידיו של הטקסט (ב-VBS המתודה הזאת נקראת Trim)

לדוגמא:

class Person {

    constructor(name, age) {

      this.name = name;

      this.age = age;

    }

  }

  let p1 = new Person("ran", 34);

  let p2 = new Person("ran2", 234);

  Person.prototype.checkName = function () {

    console.log("Hello, my name is " + this.name);

  };

  p1.checkName();

  p2.checkName();

call, apply, bind:

1. Call: זאת פונקציה שמקבלת אליה אובייקט שהוא ישמש כ this שלה בתוך הפונקציה

דוגמא:

  const obj = { name: "ran" };

  let f1 = function (age, home) {

    console.log(this.name, age, home);

  };

  //call

  f1(); //undifiend

  f1.call(obj, 15, "Kramim"); //ran 15 Kramim

1. Apply: מקבלת גם את האובייקט כthis אבל כאן ארגומנטים נוספים יהיו בתוך מערך

f1.apply(obj,[15, "Kramim"])

1. Bind: מחזירה פונקציה חדשה שאליה אפשר להוסיף ארגומנטים

let f = f1.bind(obj);

  f(15, "Kramim");

**What is the difference between slice and splice**

Some of the major differences in a tabular form:

| **Slice** | **Splice** |
| --- | --- |
| Doesn't modify the original array(immutable) | Modifies the original array(mutable) |
| Returns the subset of original array | Returns the deleted elements as array |
| Used to pick the elements from array | Used to insert/delete elements to/from array |

אובייקט מול map. צורתם דומים אבל יש שוני בינהם:

Map: ניתן לשים בו כל משתנה כמפתח: אובייקט, מערך וכו..

כדי לגעת בערך לא מספיק לשים את מערך עם אותו ערכים אלא את המערך המקורי עצמו

המפתחות שנכנסים יש להם סדר מסויים, לפי סדר ההכנסה שלהם.

const obj = {};

  const map = new Map();

  let obj2 = { test: "test1" };

  let arr = [1, 2, 3];

  map.set(obj2, "obj2");

  map.set(arr, "arr");

  console.log(map); //Map(2) { { test: 'test1' } => 'obj2', [ 1, 2, 3 ] => 'arr' }

  obj[obj2] = "obj2";

  obj[arr] = "arr";

  console.log(obj); //{ '[object Object]': 'obj2', '1,2,3': 'arr' }

פונקציה חץ:

An arrow function is a shorter/concise syntax for a function expression and does not have its own **this, arguments, super, or new.target**. These functions are best suited for non-method functions, and they cannot be used as constructors

Singleton:

יצירה של אובייקט שאפשרי לעשות לו אתחול רק פעם אחת. דוגמא:

class MySingletonClass {

    static instance = null; //this will check if the class was created

    constructor() {

      if (MySingletonClass.instance) {

        //instance is no null so object was created so throw exception

        throw new Error("Object was already init");

      }

      this.name = "single";

      this.age = 20;

      MySingletonClass.instance = this;

    }

    static getInstance() {

      if (MySingletonClass.instance) {

        return MySingletonClass.instance;

      }

      return new MySingletonClass();

    }

  }

  try {

    const obj = MySingletonClass.getInstance();

    const obj2 = MySingletonClass.getInstance();

    console.log(obj, obj2);

    const obj3 = new MySingletonClass(); //error

  } catch (error) {

    console.log(error);

  }

Static:

static methods and properties belong to the class itself rather than to instances of the class

ההבדלים בין קריאות API:

GET: עושים בה שימוש בכי לקבל מידע, בדך השימוש בה הוא רק לקריאה ולא לשינוי

POST: שולחת מידע לSERVER ויוצרת מידע חדש. בניגוד לGET כאן נצטרך להעביר גם HEADERS ו BODY. בGET בדך הם דיפולטיביים ואין צורך לשנות משהו או לציין אותם

PUT: עושים בה שימוש בכדי לעדכן מידע קיים. בגוף שלה נוכל לשלוח את כל מידע שאנחנו רוצים לשנות כך שיחליף לחלוטין את הרשומה הקיימת

PATCH: דומה לPUT ההבדל הוא שעאן מעדכנים רק חלק מהרשומה ולא מחליפים אותה.

DELETE: מחיקה של רשומה

| **var** | **let** |
| --- | --- |
| It has been available from the beginning of JavaScript | Introduced as part of ES6 |
| It has function scope | It has block scope |
| Variable declaration will be hoisted | Hoisted but not initialized |
| It is possible to re-declare the variable in the same scope | It is not possible to re-declare the variable |

CORS ERROR:

כאשר יש ישרת ואתרי אינטרנט שמנסים לתקשר.

אתר מנסה לשלוח בקשות לשרת כאשר אין לו הרשאה לעשות כך.

הSERVER יכול לשנות את ההגדרות שלו וכך יוכל לקבל פניות של אתרים.

const corsOptions = {

  origin: "http://localhost:3000", // Replace with your React app's domain

  credentials: true,

};

app.use(cors(corsOptions));

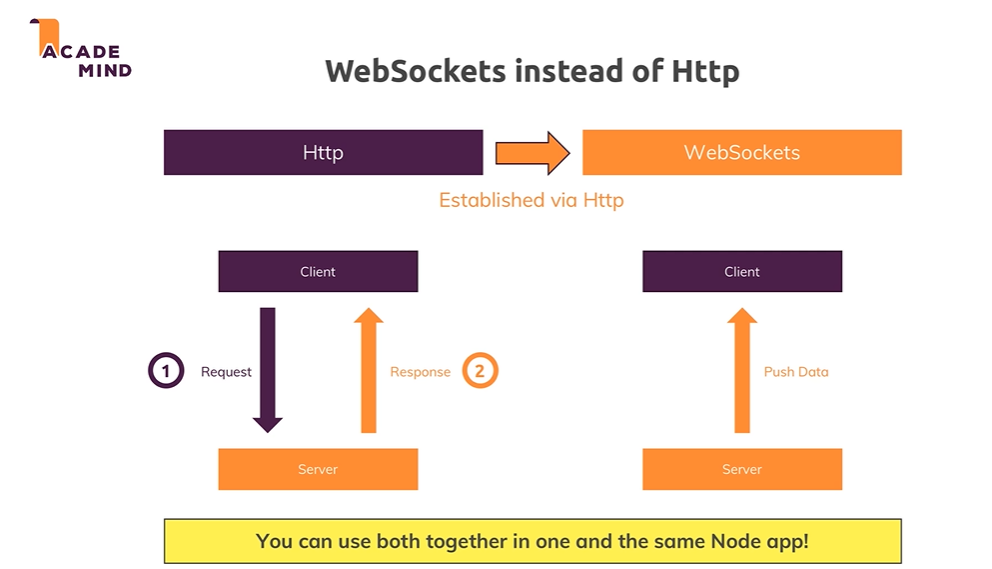
origin: מי יוכל לגשת לי לשרת

Bubling event listner

Socket: תקשורת בין שרת ללקוח הוא חד כיווני מפבחינת הפנייה. הלקוח תמיד יפנה לשרת והשרת יחזיר תגובה.

מה קורה כאשר השרת צריך לשלוח מידע ללקוח בלי בקשה שלו?

דוגמא: אפליקציית צ'אטים, שלקוח1 שולח הודעה היא מגיעה ל DB ואז לקוח שתיים יקבל אותה מהשרת. איך זה יקרה? WEB SOCKET



The **Array.from()** static method creates a new, shallow-copied Array instance from an [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol) or [array-like](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#working_with_array-like_objects) object.

הוצאת אלמנט ממערך:

let arr = [1, 2, 3, 4, 5];

let indexToRemove = 2; // index of the element to remove

arr.splice(indexToRemove, 1); // remove 1 element starting from indexToRemove

console.log(arr); // Output: [1, 2, 4, 5]

PROMISE:

נתחיל בתיאור קצר של הבעיה אותה Promise בא לפתור, למעשה JavaScript היא שפה המושתתת על Singe-thread מה שבעצם אומר שכל פעולה שמתבצעת תתקע את שאר הקוד עד לכשתסתיים ולשם כך אנו משתמשים ב- callbacks ו- Promises שעוזרים לנו ליצור אסינכורניות בקוד ולקרוא לחלקים ממנו ברגע שבו פעולה כלשהי תסתיים ובינתיים להמשיך עם שאר הקוד

Promise מאפשר לנו להמתין לפעולות שאנחנו לא בהכרח יכולים לדעת מתי יסתיימו והאם יצליחו או יכשלו.

const promise = new Promise((resolve, reject) => {

// Condition to resolve or reject the promise

});

דוגמא מלאה בelta

Promise.race()

לוקחת כמה פרומיס ביחד ומריצה, הראשון שמסתיים אותו נקבל בלבד וכל השאר לא נקבל תשובה

const promise1 = new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve("yes");

    }, 500);

  });

  const promise2 = new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve("yes fast");

    }, 100);

  });

  Promise.race([promise1, promise2]).then((value) => {

    console.log(value);

    // Both resolve, but promise2 is faster

  });

Promise.all()

פונקציה שלוקחת מערך של פרומיס ומריצה את כולם. במידה וכולם resolve אז תחזיר במערך את כל התוצאות.

אם אחד מהפרומיס נכשל אז תציג רק את הכשלון שלו

 const promise1 = Promise.resolve(3);

  const promise2 = 42;

  const promise3 = new Promise((resolve, reject) => {

    setTimeout(reject, 100, "foo");

  });

  Promise.all([promise1, promise2, promise3])

    .then((values) => {

      console.log(values);// [ 3, 42, 'foo' ]

    })

    .catch(() => {

      console.log("reject");

    });

Promise.allSettled()

כאן נשלח שוב כמה פרומיס ביחד אבל בניגוד לקודם נקבל מערך שכולל גם אם באחד מהפרומיס יש שגיאה

const promise1 = Promise.resolve(3);

  const promise2 = new Promise((resolve, reject) =>

    setTimeout(reject, 100, "foo")

  );

  const promises = [promise1, promise2];

  Promise.allSettled(promises).then((results) =>

    results.forEach((result) => console.log(result.status))

  );

Promise.any

כאן נשלח שוב כמה פרומיס ורק המהירה ביותר שאושרה ניקח, אם אחת מהן יהיה reject אז פשוט נמשיך הלאה ולא נתחשב בשגיאה, נחפש רק את הראשונה שעושה resolve

const promise1 = Promise.reject(0);

const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));

const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));

const promises = [promise1, promise2, promise3];

Promise.any(promises).then((value) => console.log(value));//quick

Array.map

עוברת על כל אחד מהאלמנט של המערך ומחזירה אותו עם שינוי או בלי.

צריך להיזהר כי אפילו אם עושים if ואז return אם זה הולך ל else הוא יחזיר undifine

Filter:

מסנן איברים לא רצויים אבל יכול להחזיר אך ורק את האיבר השלם עצמו, לא ניתן לבחור מה להחזיר ממנו

Null VS Undefined:

null הוא ערך **ריק** או ערך **לא קיים**.  
null הוא ערך שהושם במתכוון, כלומר, אם משתנה מחזיר null זה מכיוון ש-null הושם לתוכו, כך לדוגמה:

נקבל undefined כאשר משתנה **הוצהר** אך לא **הוגדר**.  
JavaScript תשים למשתנה את הערך Undefined באופן אוטומטי.

\*כאשר נבדוק type אז null יחזיר לנו אובייקט ואילו undefined יחזיר לנו undefined

Null==undefined true

When you use the loose equality operator (==) to compare var2 and var1, JavaScript treats null and undefined as equal because both represent the absence of a value. As a result, the expression var2 == var1 evaluates to true.

Null===undefined false

However, when you use the strict equality operator (===) to compare var2 and var1, JavaScript does not perform any type coercion and compares both the type and value. Since var2 is of type object (null is considered an object in JavaScript) and var1 is of type undefined, they are not the same type, so the expression var2 === var1 evaluates to false

Array.prototype.some()

עובר על המערך ובודק אם משתנה אחד מתאים לתנאי שהצבנו.

Class: אפילו אם לא מציינים ברגע שאתה יורש ממחלקה מסויימת אתה אוטומטי מפעיל את הבנאי שלו

class Employee {

  constructor(name, id) {

    this.name = name;

    this.id = id;

  }

}

class Manager extends Employee {

  setDepartment(dep) {

    this.dep = dep;

  }

}

const manager = new Manager("asd", "asd");

console.log(manager);//Manager { name: 'asd', id: 'asd' }

Debouncing is a technique used to limit the rate at which a function is called. It ensures that the function is only called once after a specified time period has elapsed since the last invocation

//function to activate after timer is done

  const saveInput = (name) => {

    console.log("save name = ", name);

  };

  const debounce = (func, timeOut) => {

    let timer;

    return (...args) => {

      //clearTimeout(timer) cancels the execution of the setTimeout function if it hasn't already been executed.

      //if 2 seconds didnt pass so we will cancel setTimeOut.

      //if 2 seconds pass so we will trigger setTimeOut

      clearTimeout(timer);

      timer = setTimeout(() => {

        func.apply(this, args);

      }, timeOut);

    };

  };

  const change = debounce(saveInput, 2000);

  change("1");

  change("2");

  change("3");

Throttling: Throttling is a technique used to limit the rate at which a function is called

SHALLOW COMPARISION:

כאשר המשתנים פרימיטיבים אז js בודק את הערך של המשתנה, אם הם לא מאותו סוג אז הוא עושה המרה. למשל:

console.log(2=="2")//true

כאשר המשתנים הם מורכבים, כמו מערך או אובייקט אז ההשואה תהיה למקום בו הם נמצאים בזכרון. לכן אובייקט עם אותו ערך לא יהיה שווה לאובייקט עם בדיוק אותו ערך אם הם לא מצביעים לאותו מקום בזכרון

console.log({a:"1"}=={a:"1"})//false

פונקציה:

const shallowComparision = (obj1, obj2) => {

  //checking all tyes

  if (typeof obj1 !== typeof obj2) {

    return false;

  }

  if (Array.isArray(obj1) && obj1.length === obj2.length) {

    //array

    return obj1.every((element, index) => {

      return element === obj2[index];

    });

  }

  if (typeof obj1 === "object" && obj1 !== null) {

    if (Object.keys(obj1).length !== Object.keys(obj2).length) {

      return false;

    }

    for (const property in obj1) {

      if (obj1[property] !== obj2[property]) {

        return false;

      }

    }

    return true;

  }

  //checking primetive types

  if (obj1 === obj2) {

    return true;

  }

  return false;

};

DEEP COMPARE

כאן אנחנו נצטרך לעבור על כל מערך ועל כל אובייקט בצורה ריקורסיבית על מנת לוודא שגם בתוכם אין אובייקט שהוא לא מתאים.

למשל מהסוג הזה:

[{a:1}]

const deepComparison = (obj1, obj2) => {

  //checking all tyes

  if (typeof obj1 !== typeof obj2) {

    return false;

  }

  if (Array.isArray(obj1) && obj1.length === obj2.length) {

    //array

    return obj1.every((element, index) => {

      //recursive call

      return deepComparison(element, obj2[index]);

    });

  }

  if (typeof obj1 === "object" && obj1 !== null) {

    if (Object.keys(obj1).length !== Object.keys(obj2).length) {

      return false;

    }

    return Object.keys(obj1).every((key) => {

      return deepComparison(obj1[key], obj2[key]);

    });

  }

  //checking primetive types

  if (obj1 === obj2) {

    return true;

  }

  return false;

};

React:

ספריית פרונט-אנד של ג'אווהסקריפט.

אלמנטים מרכזיים:

1. JSX, סינטקס שנותן אפשרות לשלב בין קוד ל HTML.
2. Virtual DOM: נדבר קודם על DOM זה המסמל את הדף שאנחנו רואים בפרונטאנד. סמסמך נראה את כל האלמנטים ואת ההיררכייה בינהם.
3. VIRTUAL DOM: הוא העתק של הDOM, הוא משפר את הביצועיים של שינויים שקורים בדף WEB. למה לעדכן את הDOOM זה איטי? כי ביחד איתו נרנדר גם את כל הCSS וכל שאר הקבצים.

בכל פעם שה-state של האפליקציה שלנו משתנה ריאקט יוצרת את כל ה- DOM הווירטואלי כולו מאפס. יצירת עץ אובייקטים שלם היא מהירה מאוד ולכן זה לא משפיע על הביצועים שלנו. בעצם בכל זמן נתון ריאקט מחזיקה שני Virtual DOM, אחד עם המצב המעודכן והשני עם המצב הקודם.

לאחר שהיא מעדכנת את הקומפוננטות הדרושות ב- Virtual DOM. ריאקט משווה בין שני ה-virtual DOM שהיא מחזיקה באמצעות אלגוריתם (diff algorithm) כדי למצוא את מספר הצעדים המינימלי לעדכון ה-Real DOM ולאחר מכן מעדכנת ב- Real DOM רק את השינויים הנדרשים ולא את כל ה DOM

1. שימוש חוזר בקומפוננטות

קומפוננטה פונקציה ומחלקה:

ממומלץ להשתמש בפונקציות ולא קלאס. יותר מהיר וקצר.

לפני שהכניסו hooks. הקלאס יהיה עדיף כי רק שם אפשר היה להשתמש בstate משלה.

הוקס מאפשרים לנו גם בפונקציות לנההל state

State: מידע שקיים בקומפוננטה שיכול להשתנות בזמן שהקומפוננטה עדיין בחיים. ברגע שהמידע משתנה הקומפוננטה תרנדר את עצמה. כדי לעדכן state, לא נעשה ישירות כי כך הקומפוננטה לא תרנדר את עצמה.

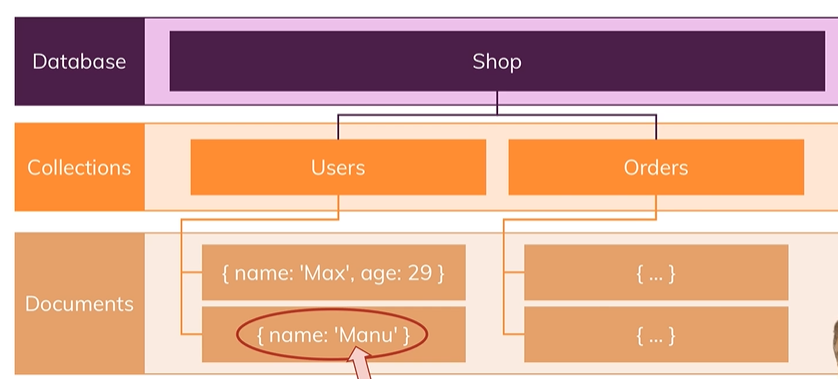
Props: אינפוט שיש לנו להעביר לקומפוננטה

MONGODB:

איך הוא בנוי:

בסיס נתונים=>אוספים=>documents

הסכימה כאן הרבה יותר גמישה מאשר sql. הפורמט הוא json

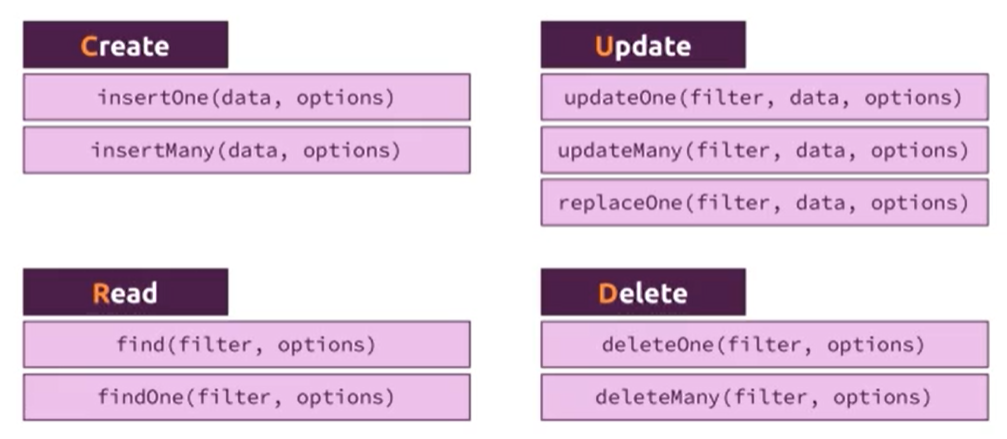


בניגוד לsql כאן ניתן לשמור מידע בתוך רשומה אחת ולא בכמה טבלאות, אפילו אם המידע הוא מערך, אובייקט וכו....

אז מה השוני?

1. אין סכימה
2. פחות יחסים בין אוספים כי אין בזה צורך כל כך
3. מתאים לאפליקציות שיש בו המון מידע

CRUD: create, read, update, delete



UPDATE VS UPDAEMANY

כאשר עושים set אז הפעולה תהיה אותו דבר, אם לא נשעשה set אז update ייקח את המשתנה ופשוט יחליף את האובייקט בdb עם מה שהעברנו. updateMany יזרוק שגיאה

FIND: לא מחזיר לנו את כל הרשומות באוסף. הוא מחזיר לנו אובייקט, האובייקט מחזיק הרבה meta data. וניתן לשלוף בעזרתו עוד תוצאותץ אם נעשה toArray אחרי find כן נקבל את כל התוצאות.

הדיפולטיבי הוא 100 רשומות.

Sort: ממיין את הרשומות שהגיעו

const dbResponse = await Flight.find()

      .sort({ createdAt: 1})

כאן מוסיפים 1 או 1-, אם רוצים סדר עולה או יורד:

1 for ascending and -1 for descending

סדר עולה: 900=>1000=>2000

ניתן גם להשתמש בשני פרמטרים ב sort דוגמא:

const dbResponse = await Flight.find()

      .sort({ createdAt: 1, distance: 1 })

הראשון מסדר לפי תאריך, השני יסדר לפי רוחק, כך שאם שני רשומות עם אותו תאריך התחלה אז יסודרו לפי המרחק

Projection: לעשות שאילתא שתחזיר לי רשומה אבל רק שדות מסויימים ממנה

const dbResponse = await Flight.find(

      {},

      { arrivalAirport: 1, departureAirport: 1 }

    ).limit(5); //projection

אנחנו מציינים באובביקט השני איזה שדות נרצה שיחזרו

ניתן גם הפוך, לציין מה לא יחזור:

const dbResponse = await Flight.find({}, { arrivalAirport: 0 }).limit(5)

embedded docs and array:

embedded: רשומה שבתוכה יש רשומה נוספת (אובייקט בוך אובייקט), ניתן להגיע עד לרמה של 100.

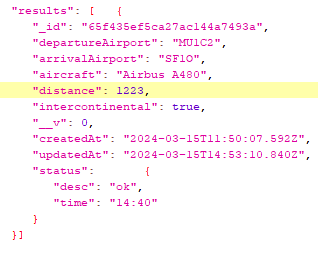
איך מעדכנים אובייקט פנימי:

dbResponse = await Flight.updateOne(

      { \_id: "65f435ef5ca27ac144a7493a" },

      { $set: { "status.time": "15:50" } }

    );



ARRAYS:

הוספה למערך:

dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $push: { passengers: "bar" } }

    );

הוספה רק אם הערך לא קיים במערך:

dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $$addToSet: { passengers: "bar" } }

    );

הוצאת ערך מסויים מהמערך

 dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { pull: { passengers: "ran" } }

    );

הוצאת כמה ערכים

dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $pullAll: { arrayField: ["ran", "bar"] } }

    );

הוצאת משתנה ראשון(1-) ואחרון (1)

 dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $pop: { passengers: 1 } }

    );

    dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $pop: { passengers: -1 } }

    );

שינוי משתנה במערך לפי אינדקס

const index = 4; // Dynamically determine the index

    const updateQuery = {};

    updateQuery[`arrayField.${index}`] = "newValue";

    dbResponse = await Flight.updateOne(

      { arrivalAirport: "SF1O" },

      { $set: updateQuery }

    );

עוד דוגמאות בפרויקט...

סכימות:

בעיקרון אין סכימות במונגו, ניתן להכניס לאותו אוסף סכימות שונות עם types שונים ושדות שונים.

כאשר מגדירים סכימה ניתן לציין שדות שהם חובה ושדות שהם רשות.

Data types: צריך לציין בסכימה את הטייפ של כל שדה שאתה רוצה להכניס

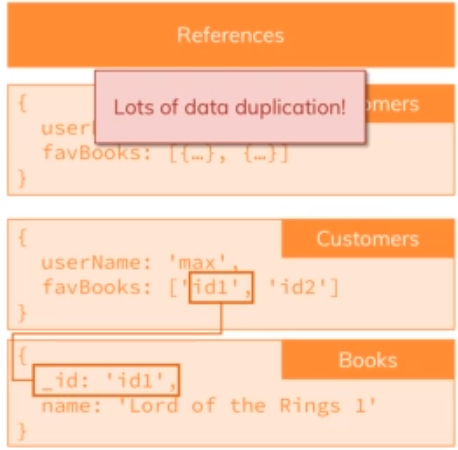
Relations: יש אפשרות שיש כמה אוספים שקשורים אחד לשני. למשל: user ו address.

2 אפשרויות:

1. Embedded: שבתוך האוסף של הuser נעשה אובייקט של כתובת:



1. להשתמש בreference:



כאן יש לנו אוסף של לקוח ואוסף של ספרים.

כל לקוח יכול להיות מחובר למערך של ספרים שמוצגים על ידי מספר הזהות שלהם.

עשיתי דוגמא בפרוייקט: ספרים ולקוחות.

יחסים ומתי כדי להשתמש בשיטות רפרנס ואמבדד:

כאשר יש יחס של אחד לאחד, עדיף לעשות אמבדד דוקץ כלומר לא לעשות שני אוספים

יחס אחד לרבים, למשל לכל אדם יכול להיות הרבה רכבים אבל לכל רכב יש בעלים אחד

אנחנו צריכים לקבוע איך לעצב את ה בסיס נתונים לפי ההגיון.

למשל: יחס אחד לרבים, עיר לתושבים. השאלה אם אתה רוצה לקבל את העיר ניויורק האם תרצה שיגיע גם רשימה לש כל התושבים? למרות שרק רצית את המידע של העיר?

אז אולי במקרה הזה כדי להפריד לשני אוספים. אוסף של ערים ואוסף של תושבים שבתושבים יהיה הפנייה לעיר שהם משתייכים אליה.

יחס רבים לרבים: דוגמא, לקוח ומוצר. לקוח יכול לקנות כמה מוצרים. ומוצר יכול להיות קנוי על ידי כמה לקוחות.

כנראה שעדיף רפרנס כדי להימנע משכפול של מידע. למשל אם עודים אמבדד אז לקוח אחד יהיה עם מוצר 1 עם כל המידע עליו וגם לקוח 2 שיקנה את אותו מוצר ישכפל את המידע של המוצר.

במקום זה שני הלקוחות יכולים להצביע על אותו רפרנס.

ישנן שתי פונקציות שעוזרות לנו לקבל מידע משני אוספים בשאילתא אחת:

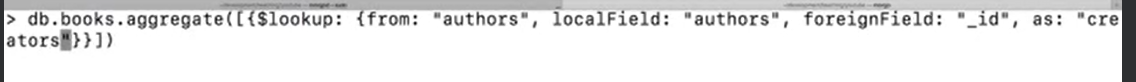
Populate

למשל בדוגמא של לקוח וספרים. אם יש ללקוח מערך של ספרים שהוא קנה והם רפרנס אז ניתן לקבל את המידע על הספרים עם הפונקציה בתוספת שם האוסף

let dbResponse = await Customer.find({}).populate("books");

lookup:

עושה פעולה דומה אבל סינטקס שונה:



אם אנחנו עושים את הקריאה מספרים אז:

From

לציין את שמו של האוסף

localField

לציין את שמו של השדה הרלוונטי באוסף הספרים שמציין את הרפרנס לאוסף השני.

Foreign field:

הקשר אליו השדה מה"נל מתקשר איתו, כלומר המספר זהות באוסף הזר.

As:

בתוך האוסף ספרים נקבל שדה חדש עם המידע מהוסף של הסופרים

משימה:

לעשות בסיס נתונים של משתמש, פוסט ותגובות.

התכנון שלי:

User: שם, מספר זהות, ומערך של פוסטים.

Post: כותרת, תוכן, מספר זהות של משתמש, שם משתמש, מערך של תגובות

Comment: שם זהות של הפוסט, מערך של תגובות שהוא אובייקט עם כותרת תגובה, תוכן, שם משתמש, זהות ומערך של תגובות לתגובה.

Inserts many:

ההתנהגות הדיפולטיבית היא שאם מכניסים 3 אובייקטים והשני נכשל אז הוא יכשיל את כל מי שהגיע אחריו אבל הראשון יישמר.

נעשה transaction

async function insertManyWithTransaction(docs) {

  const session = await mongoose.startSession();

  session.startTransaction();

  try {

      await Model.insertMany(docs, { session: session });

      await session.commitTransaction();

      session.endSession();

      console.log("All documents inserted successfully.");

  } catch (error) {

      await session.abortTransaction();

      session.endSession();

      console.error("Error inserting documents:", error);

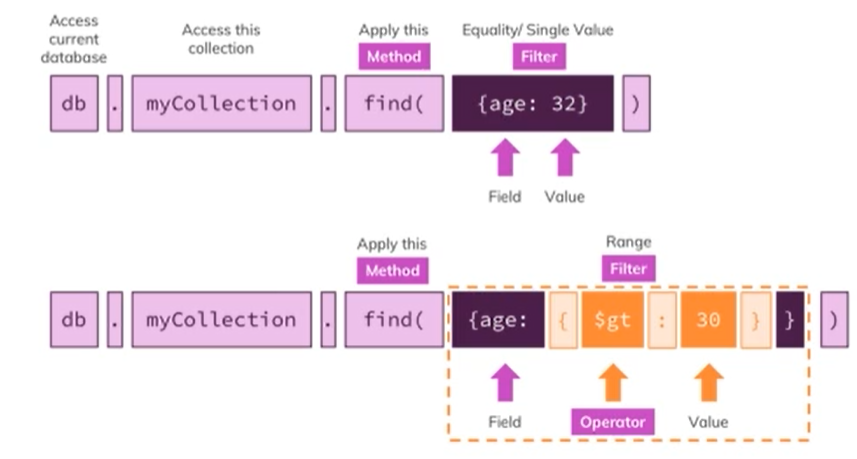
      throw error; // Re-throw the error to handle it in the caller function

  }

}

Read:

בעיקרון לקבל מידע מהבסיס נתונים. לרוב נעשה תנאים, לדוגמא:



Query Selectors: משתמשים בהם בכדי לפלטר את התוצאות של השאילתא

Projection Operators: משתמשים כדי לעצב את תוכן המידע שנקבל מהשאילתא

Querying Embedded Fields & Arrays:



אז כאן צריך לציין את הדרך לשם של השדה שהוא אובייקט: rating, ואז בתוכו יש average, ואז שהוא יהיה גדול מ7

מערך:

שתי אפשרויות עם תוצאות שונות.

אם המערך כזה: genres = [“Drama”,”Action”]

אז אם נעשה שאילתא כזו:



כאן הוא ימצא את כל הרשומות שבהן יש גם את drama. כלומר כמו הדוגמא, המערך מכיל את drama.

אבל אם נשאל כך:



נחפש מערך שהוא מכיל רק את drama ושום דבר אחר.

IN, NIN

כאן נוכל לחפש משתנה עם כמה ערכים ששווים לו, למשל:



אז runtime יכול להיות 30 או 42

אותו דבר עם NIN, כלומר לא שווים ל 30 ו 42

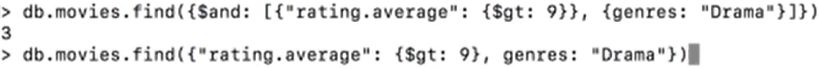
OR, AND:



נציין את האופרטור בהתחלה ואז בתוך המערך נשים את התנאים

And:

כאן השאילתא הדיפולטיבית היא and. כלומר לא חייב לציין זאת. אז ניתן לעשות זאת בשתי אפשרויות:



Not:



Element operators:

Element operators return data based on field existence or data types

לדוגמא:



אם יש שדה age אז יחזיר את כל הרשומות שזה קיים בהן

ניתן לבדוק גם סוג השדה:



אפשר גם לשלוח כמה אפשרויות:

