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

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

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.

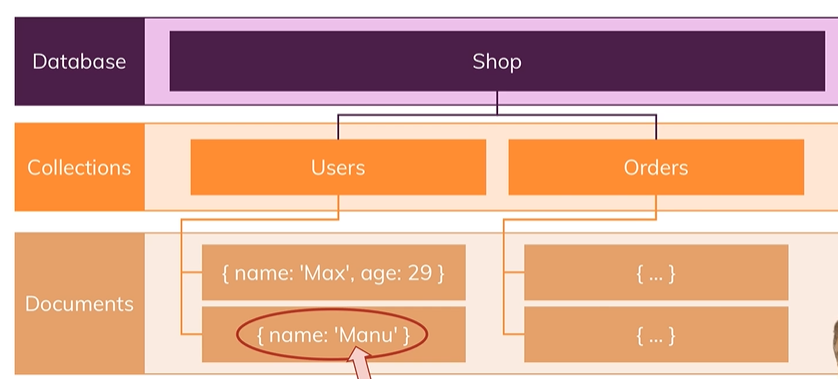
Props: אינפוט שיש לנו להעביר לקומפוננטה

MONGODB:

איך הוא בנוי:

בסיס נתונים=>אוספים=>documents

הסכימה כאן הרבה יותר גמישה מאשר sql. הפורמט הוא json

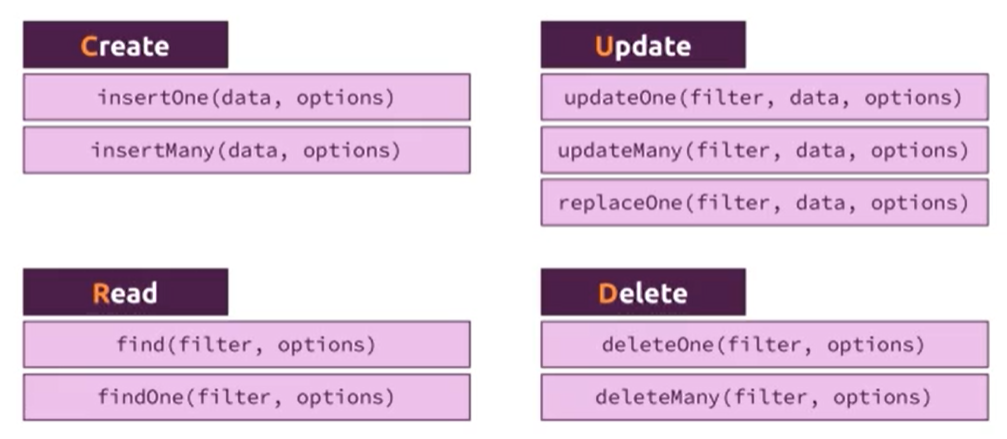


בניגוד לsql כאן ניתן לשמור מידע בתוך רשומה אחת ולא בכמה טבלאות, אפילו אם המידע הוא מערך, אובייקט וכו....

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.

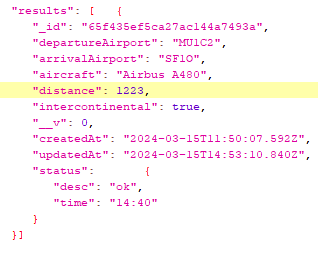
איך מעדכנים אובייקט פנימי:

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 }

    );