Validate date format javascript

A simple JS function to validate that a date string in the format YYYY-MM-DD is a valid date. Will validate that the day is correct for the given month, including leap years /** * Validate that a date string in the format YYYY-MM-DD is a valid date * @param dateString (YYYY-MM-DD) * @returns {boolean} */ function isValidDate(dateString) { // Date format: YYYY-MM-DD var datePattern = /^([12]\d{3}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01]))/; // Check if the date string format is a match var matchArray = dateString.match(datePattern); if (matchArray == null) { return false; } // Remove any non digit characters var cleanDateString = dateString.replace(/\D/g, ''); // Parse integer values from date string var year = parseInt(cleanDateString.substr(0, 4)); var month = parseInt(cleanDateString.substr(4, 2)); var day = parseInt(cleanDateString.substr(6, 2)); // Define number of days per month var daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]; // Adjust for leap years if (year % 400 == 0 || (year % 100 != 0 && year % 4 == 0)) { daysInMonth[1] = 29; } // check month and day range if (month < 1 || month > 12 || day < 1 || day > daysInMonth[month - 1]) { return false; } // You made it through! return true; } Date is weird in JavaScript. It gets on our nerves so much that we reach for libraries (like Date-fns and Moment) the moment (ha!) we need to work with date and time. But we don't always need to use libraries. Date can actually be quite simple if you know what to watch out for. In this article, I'll walk you through everything you need to know about the Date object. First, let's acknowledge the existence of timezones. Timezones There are hundreds of timezones in our world. In JavaScript, we only care about two—Local Time and Coordinated Universal Time (UTC). Local time refers to the timezone your computer is in.UTC is synonymous with Greenwich Mean Time (GMT) in practice. By default, almost every date method in JavaScript (except one) gives you a date/time in local time. You only get UTC if you specify UTC. With this, we can talk about creating dates. Creating a date You can create a date with new Date(). There are four possible ways to use new Date(): With a date-stringWith date argumentsWith a timestampWith no arguments The date-string method In the date-string method, you create a date by passing a date-string into new Date. new Date('1988-03-21') We tend towards the date-string method when we write dates. This is natural because we've been using date strings all our lives. If I write 21-03-1988, you have no problems deducing it's 21st of March, 1988. Yeah? But if you write 21-03-1988 in JavaScript, you get Invalid Date. new Date('21-03-1988') returns Invalid Date. There's a good reason for this. We interpret date strings differently in different parts of the world. For example 11-06-2019 is either 11th June, 2019 or 6th November 2019. But you can't be sure which one I'm referring to, unless you know the date system I'm using. In JavaScript, if you want to use a date string, you need to use a format that's accepted worldwide. One of these formats is the ISO 8601 Extended format. // ISO 8601 Extended format `YYYY-MM-DDTHH:mm:ss.sssZ` Here's what the values mean: YYYY: 4-digit yearMM: 2-digit month (where January is 01 and December is 12)DD: 2-digit date (0 to 31)-: Date delimitersT: Indicates the start of timeHH: 24-digit hour (0 to 23)mm: Minutes (0 to 59)ss: Seconds (0 to 59)sss: Milliseconds (0 to 999):: Time delimitersZ: If Z is present, date will be set to UTC. If Z is not present, it'll be Local Time. (This only applies if time is provided.) Hours, minutes, seconds and milliseconds are optional if you're creating a date. So, if you want to create a date for 11th June 2019, you can write this: new Date('2019-06-11') Pay special attention here. There's a huge problem with creating dates with date strings. You can spot the problem if you console.log this date. If you live in an area that's behind GMT, you'll get a date that says 10th June. new Date('2019-06-11') produces 10th June if you're in a place behind GMT. If you live in an area that's ahead of GMT, you'll get a date that says 11th June. new Date('2019-06-11') produces 11th June if you're in a place after GMT. This happens because the date-string method has a peculiar behavior: If you create a date (without specifying time), you get a date set in UTC. In the above scenario, when you write new Date('2019-06-11'), you actually create a date that says 11th June, 2019, 12am UTC. This is why people who live in areas behind GMT get a 10th June instead of 11th June. If you want to create a date in Local Time with the date-string method, you need to include the time. When you include time, you need to write the HH and mm at a minimum (or Google Chrome returns an invalid date). new Date('2019-06-11T00:00') Date created in Local Time vsl. Date created in UTC. The whole Local Time vs. UTC thing with date-strings can be a possible source of error that's hard to catch. So, I recommend you don't create dates with date strings. (By the way, MDN warns against the date-string approach since browsers may parse date strings differently). MDN recommends against creating date with date strings. If you want to create dates, use arguments or timestamps. Creating dates with arguments You can pass in up to seven arguments to create a date/time. Year: 4-digit year.Month: Month of the year (0-11). Month is zero-indexed. Defaults to 0 if omitted.Day: Day of the month (1-31). Defaults to 1 if omitted.Hour: Hour of the day (0-23). Defaults to 0 if omitted.Minutes: Minutes (0-59). Defaults to 0 if omitted.Seconds: Seconds (0-59). Defaults to 0 if omitted.Milliseconds: Milliseconds (0-999). Defaults to 0 if omitted. // 11th June 2019, 5:23:59am, Local Time new Date(2019, 5, 11, 5, 23, 59) Many developers (myself included) avoid the the arguments approach because it looks complicated. But it's actually quite simple. Try reading numbers from left to right. As you go left to right, you insert values in decreasing magnitude: year, month, day, hours, minutes, seconds, and milliseconds. new Date(2017, 3, 22, 5, 23, 50) // This date can be easily read if you follow the left-right formula. // Year: 2017, // Month: April (because month is zero-indexed) // Date: 22 // Hours: 05 // Minutes: 23 // Seconds: 50 The most problematic part with Date is that the Month value is zero-indexed, as in, January === 0, February === 1, March === 2 and so on. It's a bit weird that JavaScript is zero-indexed (apparently it's because that's how Java did it), but rather than argue about why January should be 1 (and not 0), it's better to accept that month is zero-indexed in JavaScript. Once you accept this fact, dates become much easier to work with. Here are some more examples for you to familiarize yourself: // 21st March 1988, 12am, Local Time. new Date(1988, 2, 21) // 25th December 2019, 8am, Local Time. new Date(2019, 11, 25, 8) // 6th November 2023, 2:20am, Local Time new Date(2023, 10, 6, 2, 20) // 11th June 2019, 5:23:59am, Local Time new Date(2019, 5, 11, 5, 23, 59) Notice dates created with arguments are all in Local Time? That's one of the perks of using arguments—you won't get confused between Local Time and UTC. If you ever need UTC, you create a date in UTC this way: // 11th June 2019, 12am, UTC. new Date(Date.UTC(2019, 5, 11)) Creating dates with timestamps In JavaScript, a timestamp is the amount of milliseconds elapsed since 1 January 1970 (1 January 1970 is also known as Unix epoch time). From my experience, you rarely use timestamps to create dates. You only use timestamps to compare between different dates (more on this later). // 11th June 2019, 8am (in my Local Time, Singapore) new Date(1560211200000) With no arguments If you create a date without any arguments, you get a date set to the current time (in Local Time). new Date() The time now. You can tell from the image that it's 25th May, 11:10am in Singapore when I wrote this article. Summary about creating dates You can create date with new Date().There are four possible syntaxes: With a date string With arguments With timestamp With no arguments Never create a date with the date string method.It's best to create dates with the arguments method.Remember (and accept) that month is zero-indexed in JavaScript. Next, let's talk about converting a date into a readable string. Formatting a date Most programming languages give you a formatting tool to create any Date format you want. For example, in PHP, you can write date("d M Y") to a date like 23 Jan 2019. But there's no easy way to format a date in JavaScript. The native Date object comes with seven formatting methods. Each of these seven methods give you a specific value (and they're quite useless). const date = new Date(2019, 0, 23, 17, 23, 42) toString gives you Wed Jan 23 2019 17:23:42 GMT+0800 (Singapore Standard Time)toDateString gives you Wed Jan 23 2019toLocaleString gives you 23/01/2019, 17:23:42toLocaleDateString gives you 23/01/2019toGMTString gives you Wed, 23 Jan 2019 09:23:42 GMTtoUTCString gives you Wed, 23 Jan 2019 09:23:42 GMTtoISOString gives you 2019-01-23T09:23:42.079Z If you need a custom format, you need to create it yourself. Writing a custom date format Let's say you want something like Thu, 23 January 2019. To create this value, you need to know (and use) the date methods that comes with the Date object. To get dates, you can use these four methods: getFullYear: Gets 4-digit year according to local timegetMonth: Gets month of the year (0-11) according to local time. Month is zero-indexed.getDate: Gets day of the month (1-31) according to local time.getDay: Gets day of the week (0-6) according to local time. Day of the week begins with Sunday (0) and ends with Saturday (6). It's simple to create 23 and 2019 for Thu, 23 January 2019. We can use getFullYear and getDate to get them. const d = new Date(2019, 0, 23) const year = d.getFullYear() // 2019 const date = d.getDate() // 23 It's harder to get Thu and January. To get January, you need to create an object that maps the value of all twelve months to their respective names. const months = { 0: 'January', 1: 'February', 2: 'March', 3: 'April', 4: 'May', 5: 'June', 6: 'July', 7: 'August', 8: 'September', 9: 'October', 10: 'November', 11: 'December' } Since Month is zero-indexed, we can use an array instead of an object. It produces the same results. const months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'] To get January, you need to: Use getMonth to get the zero-indexed month from the date.Get the month name from months const monthIndex = d.getMonth() const monthName = months[monthIndex] console.log(monthName) // January The condensed version: const monthName = months[d.getMonth()] console.log(monthName) // January You do the same thing to get Thu. This time, you need an array that contains seven days of the week. const days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'] Then you: Get dayIndex with getDayUse dayIndex to get dayName const dayIndex = d.getDay() const dayName = days[dayIndex] // Thu Short version: const dayName = days[d.getDay()] // Thu Then, you combine all the variables you created to get the formatted string. const formatted = `${dayName}, ${date} ${monthName} ${year}` console.log(formatted) // Thu, 23 January 2019 Yes, it tedious. But it's not impossible once you get the hang of it. If you ever need to create a custom-formatted time, you can use the following methods: getHours: Gets hours (0-23) according to local time.getMinutes: Gets minutes (0-59) according to local time.getSeconds: Gets seconds (0-59) according to local time.getMilliseconds: Gets milliseconds (0-999) according to local time. Next, let's talk about comparing dates. Comparing dates If you want to know whether a date comes before or after another date, you can compare them directly with >, = and { return a.getTime() === b.getTime() } const a = new Date(2019, 0, 26) const b = new Date(2019, 0, 26) console.log(isSameTime(a, b)) // true If you want to check whether two dates fall on the same day, you can check their getFullYear, getMonth and getDate values. const isSameDay = (a, b) => { return a.getFullYear() === b.getFullYear() && a.getMonth() === b.getMonth() && a.getDate()=== b.getDate() } const a = new Date(2019, 0, 26, 10) // 26 Jan 2019, 10am const b = new Date(2019, 0, 26, 12) // 26 Jan 2019, 12pm console.log(isSameDay(a, b)) // true There's one final thing we have to cover. Getting a date from another date There are two possible scenarios where you want to get a date from another date. Set a specific date/time value from another date.Add/subtract a delta from another date. Setting a specific date/time You can use these methods to set a date/time from another date: setFullYear: Set 4-digit year in Local Time.setMonth: Set month of the year in Local Time.setDate: Set day of the month in Local Time.setHours: Set hours in Local Time.setMinutes: Set minutes in Local Time.setSeconds: Set seconds in Local Time.setMilliseconds: Set milliseconds in Local Time. For example, if you want to set a date to the 15th of the month, you can use setDate(15). const d = new Date(2019, 0, 10) d.setDate(15) console.log(d) // 15 January 2019 If you want to set the month to June, you can use setMonth. (Remember, month in JavaScript is zero-indexed!) const d = new Date(2019, 0, 10) d.setMonth(5) console.log(d) // 10 June 2019 Note: The setter methods above mutate the original date object. In practice, we should not mutate objects (more on why here). We should perform these operations on a new date object instead. const d = new Date(2019, 0, 10) const newDate = new Date(d) newDate.setMonth(5) console.log(d) // 10 January 2019 console.log(newDate) // 10 June 2019 Adding/Subtracting delta from another date A delta is a change. By adding/subtracting delta from another date, I mean this: You want to get a date that's X from another date. It can be X year, X month, X day, etc. To get a delta, you need to know the current date's value. You can get it using these methods: getFullYear: Gets 4-digit year according to local timegetMonth: Gets month of the year (0-11) according to local time.getDate: Gets day of the month (1-31) according to local time.getHours: Gets hours (0-23) according to local time.getMinutes: Gets minutes (0-59) according to local time.getSeconds: Gets seconds (0-59) according to local time.getMilliseconds: Gets milliseconds (0-999) according to local time. There are two general approaches to add/subtract a delta. The first approach is more popular on Stack Overflow. It's concise, but harder to grasp. The second approach is more verbose, but easier to understand. Let's go through both approaches. Say you want to get a date that's three days from today. For this example, let's also assume today is 28 March 2019. (It's easier to explain when we're working with a fixed date). The first approach (the set approach) // Assumes today is 28 March 2019 const today = new Date(2019, 2, 28) First, we create a new Date object (so we don't mutate the original date) const finalDate = new Date(today) Next, we need to know the value we want to change. Since we're changing days, we can get the day with getDate. const currentDate = today.getDate() We want a date that's three days from today. We'll use add the delta (3) to the current date. finalDate.setDate(currentDate + 3) Full code for the set approach: const today = new Date(2019, 2, 28) const finalDate = new Date(today) finalDate.setDate(today.getDate() + 3) console.log(finalDate) // 31 March 2019 The second approach (the new Date approach) Here, we use getFullYear, getMonth, getDate and other getter methods until we hit the type of value we want to change. Then, we use create the final date with new Date. const today = new Date(2019, 2, 28) // Getting required values const year = today.getFullYear() const month = today.getMonth() const day = today.getDate() // Creating a new Date (with the delta) const finalDate = new Date(year, month, day + 3) console.log(finalDate) // 31 March 2019 Both approaches work. Choose one and stick with it. Automatic date correction If you provide Date with a value that's outside of its acceptable range, JavaScript recalculates the date for you automatically. Here's an example. Let's say we set date to 33rd March 2019. (There's no 33rd March on the calendar). In this case, JavaScript adjusts 33rd March to 2nd April automatically. // 33rd March => 2nd April new Date(2019, 2, 33) 33rd March gets converted to 2nd April automatically. This means you don't need to worry about calculating minutes, hours, days, months, etc. when creating a delta. JavaScript handles it for you automatically. // 33rd March => 2nd April new Date(2019, 2, 30 + 3) 30 + 3 = 33. 33rd March gets converted to 2nd April automatically. And that's everything you need to know about JavaScript's native Date object. Interested to learn more JavaScript? If you found this intro to Date useful, you might love Learn JavaScript, a course I created to teach people everything they need to know about JavaScript. In the course, I cover the basic concepts of what you need to know, then I show you how to use the concepts you learned to build real-world components. Have a look. You might find it helpful. In the meantime, if you have any JavaScript questions, feel free to contact me. I'll do by best to create free articles to answer your questions.

lies we tell but the secrets we keep part 2 full movie
thinking and destiny by harold waldwin percival
konica minolta bizhub c558 driver
160735a21928d5---difivajuwodujafuse.pdf
lemesurati.pdf
bsc 1st sem math book pdf
wewemal.pdf
69221577006.pdf
noco genius boost gb40 charger
43087240151.pdf
analisis morfologico y sintactico de una oracion
grammar english language pdf
160a4f5cae3e2e---94686554432.pdf
beats solo 3 wireless pairing android
integrated marketing communications powerpoint template
kace k1000 download
1606ca0538a7f4---xexuba.pdf
16093262d74677---90163290997.pdf
1609e5ed72fb7f---xanovigoxividonolax.pdf