2758. Next Day

Easy

Problem Description

In this coding task, the goal is to extend the functionality of the JavaScript `Date` object by introducing a new method called `nextDay()`. When called on a `Date` object, this method should calculate the date for the following day and return it as a string formatted as "YYYY-MM-DD". For example, if the method is invoked on a `Date` object representing June 20, 2014, it should return the string "2014-06-21" which represents June 21, 2014.

Intuition

To achieve the result of getting the next day's date, we employ a straightforward approach. By using the `Date` object's built-in methods, we can carry out this task with a few steps:

1. Get the current date from the `this` reference inside our new `nextDay` method. The `this` keyword refers to the `Date` object that `nextDay` is called upon.
2. Create a copy of the current date to avoid mutating the original `Date` object.
3. Increment the date of this copied object by one, using the `getDate` and `setDate` methods. This action effectively moves the date forward by one day.
4. Convert the updated `Date` object into a string formatted as ISO 8601 using `toISOString`. This will yield a string in the format "YYYY-MM-DDTHH:mm:ss.sssZ".
5. Extract and return only the date portion from this ISO string (i.e., the first 10 characters, "YYYY-MM-DD").

Solution Approach

The solution leverages the native JavaScript `Date` object's methods to achieve the desired functionality. Here's a step-by-step breakdown of the algorithm, patterns, and data structure used in the implementation:

1. **Monkey Patching**: The solution uses a pattern known as "monkey patching," where a method is added to an existing built-in object at runtime. By extending the `Date` object's prototype, we introduce a new method `nextDay` that becomes available to all instances of `Date`.
2. **Creating a New Date Instance**: A new `Date` object is instantiated using `this.valueOf()`. The `valueOf` method returns the primitive value of the specified object. In the case of a `Date` object, it returns the number of milliseconds since January 1, 1970, 00:00:00 UTC. This number is used to create a new `Date` object which is a copy of the original, ensuring that the original `Date` object remains unchanged.
3. **Incrementing the Date**: By calling `getDate` on the newly created `Date` object, we get the day of the month for the specified date according to local time. We then use `setDate` to advance this value by one, which moves the date to the next day.
4. **Formatting the Result**: The `toISOString` method is used, which converts a `Date` object into a string in ISO format (YYYY-MM-DDTHH:mm:ss.sssZ). Since the problem only requires the date component in "YYYY-MM-DD" format, `slice(0, 10)` is applied to this string to extract the first ten characters, discarding the time portion.
5. **Returning the Result**: The string representation of the next day in "YYYY-MM-DD" format is then returned.

The solution doesn't directly employ complex data structures or algorithms as it mainly revolves around date manipulation using built-in `Date` object methods. The pattern used here is simple and efficient since JavaScript's `Date` object internally handles all the complexities related to leap years, time zones, daylight saving time changes, and other date-related quirks.

Here's the implementation based on the approach described:

```
declare global {
    interface Date {
        nextDay(): string;
    }
}

Date.prototype.nextDay = function () {
    const date = new Date(this.valueOf());
    date.setDate(date.getDate() + 1);
    return date.toISOString().slice(0, 10);
};

/**
 * Example usage:
 * const date = new Date("2014-06-20");
 * console.log(date.nextDay()); // "2014-06-21"
 */
```

By following these steps, the `nextDay` method reliably calculates the next day, formats it appropriately, and integrates seamlessly into the `Date` object for convenient usage.

Example Walkthrough

To provide a better understanding of the `nextDay` method implementation, let's walk through a small example where we apply the solution approach to a `Date` object representing April 30, 2021. This date is of particular interest because the following day is May 1, 2021, which is the start of a new month. Demonstrating the function works across month boundaries ensures it handles more complex cases of date manipulation.

1. **Creating the Date Object**: We first construct a `Date` object representing April 30, 2021.

   ```
   const date = new Date("2021-04-30");
   ```

2. **Adding the `nextDay` Method**: We could imagine the `nextDay` method has already been added to the `Date` prototype as described in the solution approach.

3. **Calling `nextDay`**: When we call the `nextDay` method on our date object:

   ```
   const nextDayStr = date.nextDay();
   ```

4. **Inside `nextDay`**: Here's what happens within the `nextDay` method:

   - It creates a new `Date` object that represents the same moment in time as `date` by using `this.valueOf()`.
   - It then calls `getDate()` on the new `Date` object to retrieve the day of the month, which is 30. It then uses `setDate()` to increment this value by one. The JavaScript `Date` object automatically handles the transition to the next month, so the internal date representation becomes May 1, 2021.
   - Finally, it uses `toISOString()` to convert the `Date` object to a string in ISO format, followed by `slice(0, 10)` to extract the year, month, and day parts.

5. **Result**: The `nextDay` method returns the string "2021-05-01" which is the formatted string representing the next day, conforming to the "YYYY-MM-DD" format. This result is stored in `nextDayStr`.

In summary, on invoking `date.nextDay()`, where `date` is April 30, 2021, our output will be:

```
console.log(nextDayStr); // "2021-05-01"
```

This confirmed our function correctly moves from the last day of April to the first day of May, thereby validating the method's ability to handle month changes seamlessly.

Solution Implementation

Python

```
from datetime import datetime, timedelta

# Extend the datetime class by creating a subclass that includes the next_day method.
class ExtendedDate(datetime):
    def next_day(self):
        """
        Calculate and return the ISO format string of the next day.

        Returns:
            str: The date string in the format YYYY-MM-DD for the next day.
        """
        # Add one day to the current date
        next_date = self + timedelta(days=1)
        # Format the date as an ISO date string and return
        return next_date.strftime('%Y-%m-%d')

# Example usage:
# Create an instance of ExtendedDate with a specific date
today = ExtendedDate(2014, 6, 20)
# Print the string representation of the next day
print(today.next_day())  # Outputs: "2014-06-21"
```

Java

```
import java.util.Calendar;
import java.util.Date;
import java.text.SimpleDateFormat;

public class ExtendedDate {
    private Date date;

    // Constructor that initializes the ExtendedDate object with a specific date
    public ExtendedDate(Date date) {
        this.date = date;
    }

    // This method returns the ISO string representation of the next day
    public String nextDay() {
        // Getting a Calendar instance based on the current date
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);

        // Adding one day to the current date in the Calendar
        calendar.add(Calendar.DAY_OF_MONTH, 1);

        // Converting the next day's date to the ISO format (YYYY-MM-DD)
        SimpleDateFormat isoFormat = new SimpleDateFormat("yyyy-MM-dd");
        return isoFormat.format(calendar.getTime());
    }

    // Example usage of the ExtendedDate class
    public static void main(String[] args) {
        // Initialize a Calendar object with a specific date
        Calendar calendar = Calendar.getInstance();
        calendar.set(2014, Calendar.JUNE, 20);

        // Create an ExtendedDate object with the specific date
        ExtendedDate today = new ExtendedDate(calendar.getTime());

        // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD)
        System.out.println(today.nextDay());
    }
}
```

C++

```
#include <iostream>
#include <ctime>
#include <sstream>
#include <iomanip>

// Extending the std::tm structure to include the next_day method, by creating a wrapper class around std::tm.
class Date {
public:
    std::tm timeStruct;

    // Constructor that initializes Date object from a string in the format "YYYY-MM-DD".
    Date(const std::string &isoDate) {
        std::istringstream ssIsoDate(isoDate);
        ss >> std::get_time(&timeStruct, "%Y-%m-%d");
        // Manual check if get_time failed, if so, populate with current time.
        if (ss.fail()) {
            std::time_t t = std::time(nullptr);
            localtime_s(&timeStruct, &t); // For safety, using localtime_s as it's thread-safe.
        }
    }

    // Method to calculate the next day and return it as an ISO string (YYYY-MM-DD).
    std::string nextDay() const {
        // Making a copy of timeStruct to modify
        std::tm nextDayTm = timeStruct;
        // Increment the day by one; mktime will normalize the tm struct if the day goes beyond the last day of the month
        nextDayTm.tm_mday++;
        std::time_t nextDayTime = std::mktime(&nextDayTm);

        // Create a string stream to hold the ISO string representation of the next day
        std::ostringstream oss;
        oss << std::put_time(localtime(&nextDayTime), "%Y-%m-%d");

        // Return the next day as an ISO string
        return oss.str();
    }
};

/**
 * Example usage:
 * Date today("2014-06-20"); // Initialize a Date object with a specific date.
 * std::cout << today.nextDay() << std::endl; // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD).
 */

int main() {
    Date today("2014-06-20"); // Initialize a Date object with a specific date.
    std::cout << today.nextDay() << std::endl; // Outputs: "2014-06-21"
    return 0;
}
```

TypeScript

```
// Extending the global Date interface to include the nextDay method.
interface Date {
    nextDay(): string;
}

// Implementing the nextDay method for the Date prototype to get the ISO string of the next day.
Date.prototype.nextDay = function (): string {
    const currentDate = new Date(this.valueOf()); // Creating a new date object that represents the current date.
    currentDate.setDate(currentDate.getDate() + 1); // Adding one day to the current date.
    return currentDate.toISOString().slice(0, 10); // Converting the next day's date to an ISO string and slicing to get the date par
};

/**
 * Example usage:
 * const today = new Date("2014-06-20"); // Initialize a Date object with a specific date.
 * console.log(today.nextDay()); // Outputs: "2014-06-21", the next day in the ISO date format (YYYY-MM-DD).
 */
```

```
from datetime import datetime, timedelta

# Extend the datetime class by creating a subclass that includes the next_day method.
class ExtendedDate(datetime):
    def next_day(self):
        """
        Calculate and return the ISO format string of the next day.

        Returns:
            str: The date string in the format YYYY-MM-DD for the next day.
        """
        # Add one day to the current date
        next_date = self + timedelta(days=1)
        # Format the date as an ISO date string and return
        return next_date.strftime('%Y-%m-%d')

# Example usage:
# Create an instance of ExtendedDate with a specific date
today = ExtendedDate(2014, 6, 20)
# Print the string representation of the next day
print(today.next_day())  # Outputs: "2014-06-21"
```

Time and Space Complexity

The time complexity of the `nextDay` method is $O(1)$, indicating that it takes a constant amount of time to compute the next day regardless of the size of the input. This is because the internal operations, such as `getDate`, `setDate`, and `toISOString`, are all constant-time operations provided by the Date object's API.