# Performance Best Practices

**Dan Wellman**
Senior Front-end Developer

# Why following Performance Best Practices Is Important

- The application will be performant and responsive—a far better user experience
- Performant code scales well and can handle growth without performance degradation
- Performant code can run on a wider range of devices
- SEO; faster pages usually rank higher

# Implement Lazy Loading

**Only load now what must be loaded now.**

**Anything that can be loaded later should be loaded later.**

# Lazy Loading

- Pattern where only the code needed for the initial app state is loaded at first
- Other code is loaded on demand as the user navigates the app
- Can speed up the initial load of the app considerably

Paul Irish

# “Performance is a part of design. It’s not just how it looks, it’s how it feels.”

# Lazy Loading

- Bundle savings from individual lazy-loaded files can be small
- But when many components can be lazy-loaded, the savings can add up
- Details here focused on Angular, but the idea of lazy loading is universal

**Never load more than you need to.**

**The rest can be loaded later.**

# Use Web-workers

# Web-workers

- JS is traditionally single-threaded, with one thread to handle all tasks—rendering, user interaction, data processing, etc.
- Too many tasks, or too intensive tasks, can block the thread, making the app unresponsive and slow

# Web-workers

- Web workers are a way to enable some tasks to be offloaded from the main thread to a background thread
- Can prevent the main thread from blocking and making the app unresponsive

```
import { Thing } from '../path';

self.onmessage = (e: MessageEvent) => {


  const dataToProcess = e.data;

  // heavy computation...

  self.postMessage(response);
};
```

- You can import things in a web-worker just like in a regular module
- Add an `onmessage` handler to receive messages from the main script. `self` represents the worker object
- Data received from the main script will be in `e.data`
- Safely do heavy computation without blocking main thread
- Computational result/data can be sent back to main script using `postMessage` again on the `self` object

```
const workerUrl = new URL(
  './data.worker', import.meta.url
);

const worker = new
Worker(workerUrl);
```

- To use a web-worker in TypeScript, you first need to get a URL to the worker file using the URL constructor

- To create a web-worker, you use the built-in Worker constructor and pass in the URL to the worker file

```
worker.addEventListener(
  'message',
  (event: MessageEvent) => {
    // handle event from web-worker
  }
);
```

- You can add an event listener to the worker to handle any message it sends back

```
worker.postMessage(dataToSend);
```

- You can send data into the worker using the worker's `postMessage` method

# Benefits of Using Web-workers

- Improved performance
- Increased stability
- Enhanced security
- Better resource utilization

# Break up Long-running Tasks

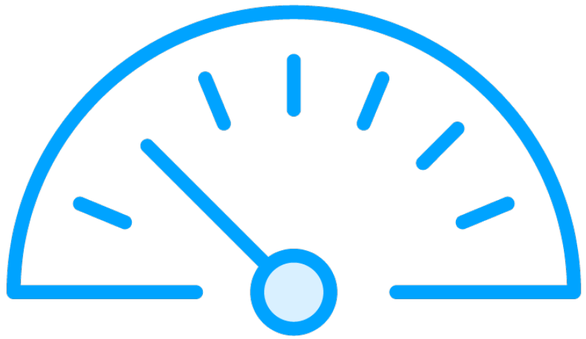
# Long-running Tasks

- Indicate a potential performance bottleneck
- Use `setTimeout` to break code up into smaller chunks that can be run as individual tasks
- This prevents the main thread from becoming unresponsive to user interactions

# Throttling and Debouncing

**Throttling and debouncing are both used to control the rate at which a function is executed.**

# Throttling



- Throttling ensures that a function is only invoked once in a specified period, regardless of how many times it’s called
- Often used with events like scrolling or resizing to limit how many times listeners are invoked

```
function throttle(fn, limit) {
  let timerId: number;
  let lastRan: number;
  return function(...args: any[]) {


    if (!lastRan) {
      fn.apply(this, args);
      lastRan = Date.now();
    } else {
      const now = Date.now();
      clearTimeout(timerId);
      timerId = setTimeout(() => {
        if ((now - lastRan) >= limit) {
          fn.apply(this, args);
          lastRan = now;
        }
      }, limit - (now - lastRan));
    }
  }
}
```

- **A basic throttle function takes another function and a time limit as parameters**
- **It returns a function which first checks whether the passed function has already run.**
- **If not, it calls the function and updates the `lastRan` variable with the current timestamp**
- **If it has already run, it clears any existing timer and then creates a new timer for the limit, minus the time between now and when it was last run**

```
const recalcLayout = () => {
  // recalculate layout or whatever...
};


window.addEventListener(
  'resize',
  throttle(recalcLayout, 200)
);
```

- You can pass an event listener to the throttle function, and it will ensure that the listener is only invoked once every 200 milliseconds, even if the event is fired continuously

  Think of throttling like slowing down a constant stream of events

  Without throttling, the resize event could fire many times, locking up the UI

# Debouncing

- Debouncing ensures that a function is only invoked after a specified delay has occurred since the function was last invoked
- If the function is called again before the timer has expired, the function is not invoked, and the timer is restarted

```
function debounce(fn, delay) {
```

```
let timerId: number;
  return function(...args: any[]) {
    clearTimeout(timerId);
    timerId = setTimeout(
      () => fn.apply(this, args),
      delay
    );
  };
}
```

- **A typical debounce function takes a function and a delay in milliseconds as parameters**

  **Often used with “lookahead” components**

- **It returns a function that first clears the timeout and then resets it, calling the passed function when the timeout expires**

```
const lookahead = () => {
  // get lookahead data...
};


input.addEventListener(
  ‘input’,
  debounce(lookahead, 300)
);
```

- You can pass an event listener to the debounce function, and it will ensure the function is only invoked if there has not been an input event for 300 milliseconds

  Think of debouncing as waiting for a stream of events to have stopped, and then invoking a function

  Without debouncing, a lookahead component might make too many HTTP requests than is necessary—a different problem to locking up the UI, but a problem nonetheless

# Throttling vs. Debouncing

## Throttling

Ensures the function executes at most once per specified period

Ideal for scenarios requiring regular updates at controlled intervals, such as scroll or resize events

Controls the rate of execution by limiting the number of times a function can be called

vs.

## Debouncing

Ensures the function executes once after a specified period of inactivity

Ideal for scenarios requiring a final action after a series of rapid events, such as typing or clicking

Controls the timing of execution by ensuring the function is called after a period of inactivity

**In scenarios where limiting how often a function is invoked is required...**

**Always consider throttling or debouncing.**

# Memoization and Caching

# Memoization

- Specific form of caching
- Store the result of function calls based on inputs
- Useful for functions with expensive computations that are often called with the same parameters

```
function memoize(fn: Function) {


  const cache = {};


  return function (...args: any[]) {
    const key = JSON.stringify(args);
    if (cache[key]) {
      return cache[key];
    }
    const result = fn(...args);
    cache[key] = result;
    return result;
  }
}
```

- A basic memoize function takes another function as a parameter
- The cache object stores the results of the function calls
- It returns a function which when called, checks if the result for the given arguments is already in the cache, and returns it if so
- If the arguments are not in the cache, it calls the function and stores the result in the cache before returning it

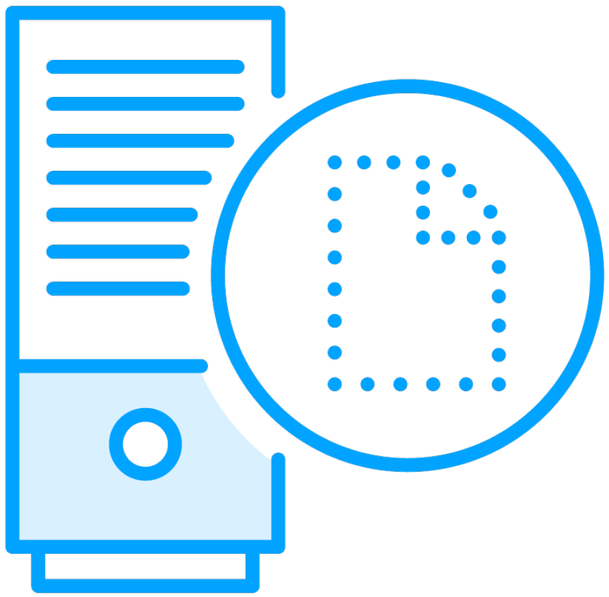
```
function fib(n: number): number {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

const memoizedFibonacci = memoize(fib);


memoizedFibonacci(10)
memoizedFibonacci(10)
```

- Calculating Fibonacci numbers can be computationally intensive

- You can memoize the fib function by passing it to the memoize function

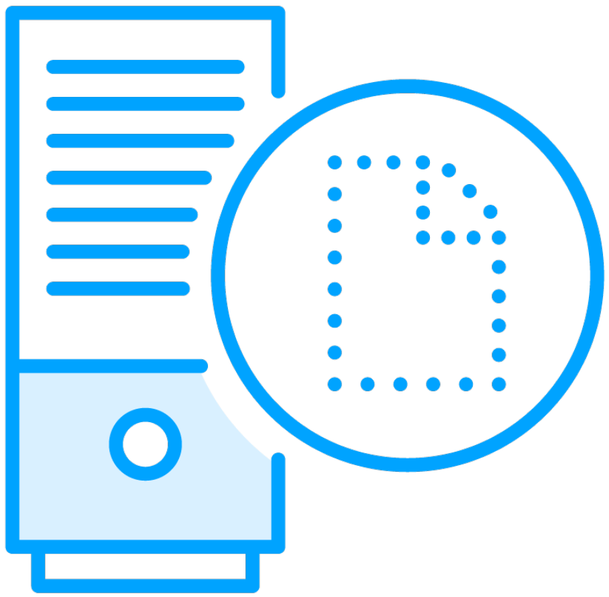
- Calling the memoized function with the same arguments will ensure that the result is only computed once—subsequent calls will return the already computed value from the cache

# Caching



- "Caching" is a more general term referring to storing data temporarily to avoid having to get or calculate that data again
- Not limited to function calls—any data or a resource that is expensive to get or compute can be cached

# Caching



- A great example of caching is static assets in the browser—images, etc., are stored in the browser's cache by domain
- When the domain was revisited, unchanged assets loaded from the cache instead of being fetched again

```
class DataCache {
  private cache = {};

  getData(key: string): string {
    if (this.cache[key]) {
      return this.cache[key];
    }
    const data = this.fetchData(key);
    this.cache[key] = data;
    return data;
  }

  private fetchData(key: string): string
  {
    // get data from API/DB/etc
  }
}
```

- A simple cache class could use an object as a private cache
- It has a method to get data—this method checks if the data is in the cache and returns it if so
- If the data is not in the cache already, it gets the data, stores it in the cache, and returns it

A more complex cache class might contain
- Expiry times for cached items
- A cache size limit
- Logic to determine which order to evict cache items in when the cache is full

**Both memoization and caching can improve performance by avoiding redundant data retrieval or computations.**