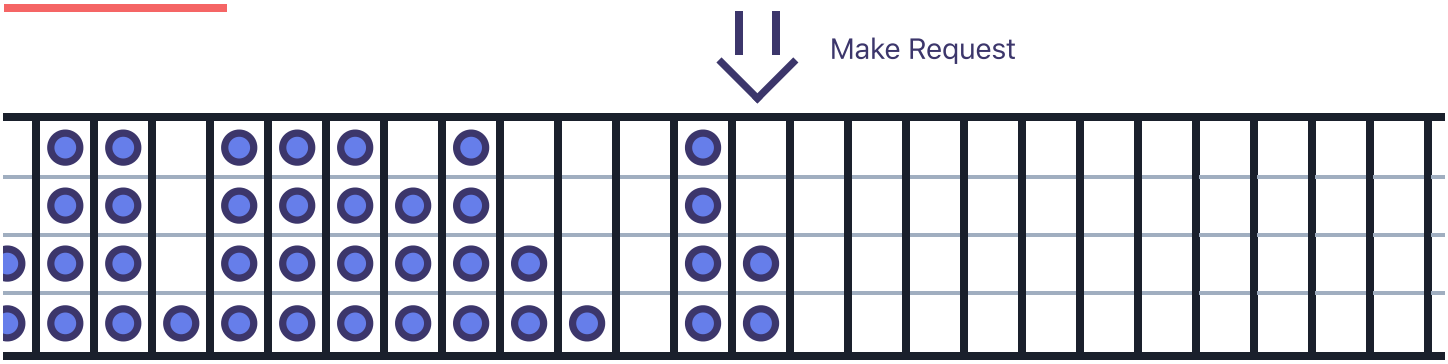
Fixed Window

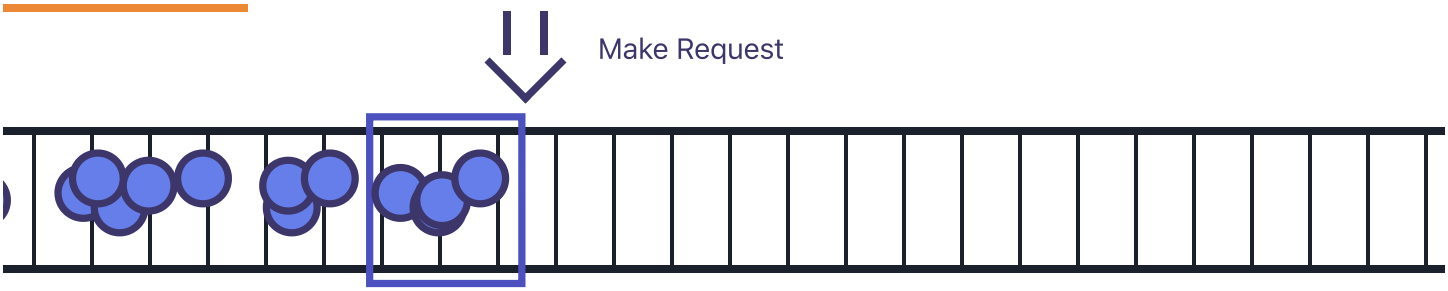


Allow X requests per Y second window.

```
function fixedWindow(maxRequestCount, intervalMs) {
  let requestCount = 0;
  let lastTimestamp = 0;

  return () => {
    const now = Date.now();

    if (now - lastTimestamp > intervalMs) {
      requestCount = 0;
      lastTimestamp = now;
    }

    if (requestCount < maxRequestCount) {
      requestCount++;
      return true;
    } else {
      return false;
    }

  };
}
```

- ✓ Simple to implement
- ✓ Only need to store 2 integers
- ✗ Painful for users when they go over
- ✗ Doesn't allow for bursty traffic
- ✗ Attackers can sustain max throughput

Sliding Window

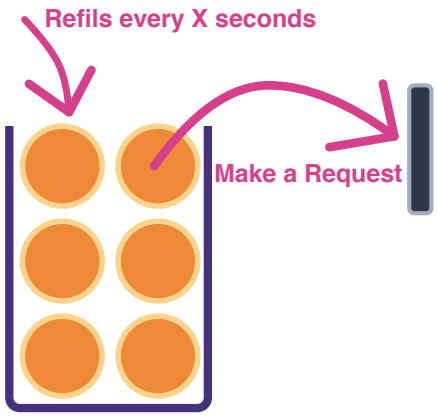


Allow no more than X requests in any Y second window.

```
function slidingWindow(maxRequestCount, windowMs) {
  let requests = [];

  return () => {
    const now = Date.now();

    requests = requests.filter(
      r => r > (now - windowMs)
    );

    if (requests.length < maxRequestCount) {
      requests.push(now);
      return true;
    } else {
      return false;
    }

  };
}
```

- ✓ Fairer than fixed window
- ✗ Uses lots of memory
- ✗ Bursts can exhaust limit for entire window
- ✗ Attackers can sustain max throughput

Bucket



Bucket overflows if no requests are made for a while

```
function bucketRateLimit(bucketCapacity, intervalMs) {
  let bucketContents = bucketCapacity;
  let lastTimestamp = Date.now();

  return () => {
    const now = Date.now();

    const extraTokens = Math.min(
      bucketCapacity - bucketContents,
      Math.floor((now - lastTimestamp) / intervalMs),
    );

    bucketContents += extraTokens;
    if (bucketContents === bucketCapacity) {
      lastTimestamp = now;
    } else {
      lastTimestamp += extraTokens * intervalMs;
    }

    if (bucketContents > 0) {
      bucketContents--;
      return true;
    } else {
      return false;
    }

  };
}
```

- ✓ Only need to store 2 integers
- ✓ Allows for bursts
- ✓ Attackers limited by the interval
- ✗ More complex to explain to API consumers

Exponential Delay

> Start with an inial delay, then multiply that by a number greater than 1 for each failed passcode attempt.

```
function exponential(freeAttemptsCount, initialDelayMs, factor) {
  let attemptsCount = 0;
  let lastTimestamp = Date.now();

  return {
    reset() {
      attemptsCount = 0;
    },

    attempt() {
      const now = Date.now();

      if (attemptsCount < freeAttempts + 1) {
        attemptsCount++;
        lastTimestamp = now;
        return true;
      }

      const delay = initialDelayMs * Math.pow(factor, attemptsCount - (freeAttempts + 1));
      if (now - lastTimestamp > delay) {
        attemptsCount++;
        lastTimestamp = now;
        return true;
      } else {
        return false;
      }
    },
  };
}
```

- ✓ Minimal impact on legitimate users
- ✓ Massive impact on malicious users
- ✓ Only need to store 2 integers
- ✗ Needs some way to reset attempts counter

Recommendations

- ✓ Use Bucket Rate Limiting - for APIs
- ✓ Use Exponential Delay - for password attempts
- ✓ Use the `@authentication/rate-limit` npm package.
- ✓ Combine both where possible
- ✗ Do not use Fixed Window Rate Limiting
- ✗ Do not use Sliding Window Rate Limiting