Best Practices and Production Concerns

Maaike van Putten
Lead Trainer & Software Developer

@brightboost | www.brightboost.nl

Overview

Debug generator functions

Evaluate the performance of generators

Performance optimization

Security and generators

Debugging generators

Debugging Generators

Similar to debugging other JS code

Common bugs

Challenging due to paused and resumed execution

Logging and debugging tools

Internal State

Can lead to bugs when not managed or fully understood correctly

Careful with external state that is crucial for the inner state of the generator

Demo

Debugging a problem with the state of the generator

Errors in the Generator

This can lead to unexpected outcomes that are challenging to debug.

Demo

Debugging an error thrown in the generator function

Errors occurring in a generator can be tricky. Let’s implement the best practice of adding a try catch.

Demo

Handling the error inside the generator

Performance profiling

Profiling the Performance of a Generator

Measuring:

- Execution time
- Memory usage
- Identify bottlenecks

Demo

Every situation of profiling is different

Walk through the different steps:

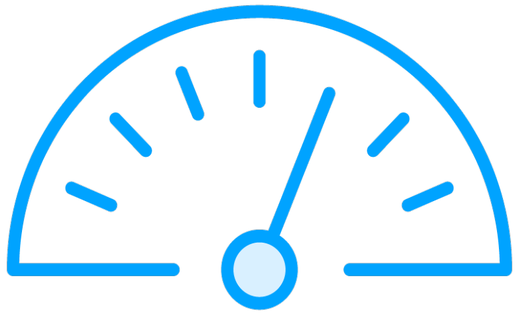
- Measure time with console.time()
- Using DevTools for a detailed analysis

Check the code where the memory usage spikes

These are the bottlenecks

For generators this could be the generator itself, but also the code processing the yielded values

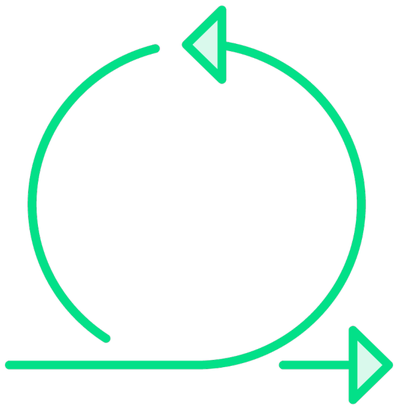
Optimize Code



- Identified bottlenecks
- Rewrite certain parts of the generator
- More efficient algorithms
- Optimize processing of yielded data
- Try without generator if possible
- Re-run the profiler to see the difference

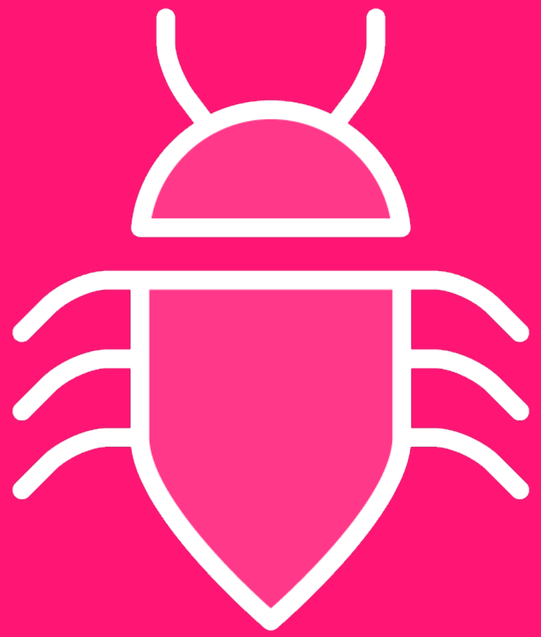
Iterative Process

Profile



Optimize

Retest



Speed Is Just One Part

We're also looking at memory efficiency.

Performance optimization tips

Optimization techniques

- Memory management
- Lazy evaluation
- Asynchronous operations

Pitfalls

- Additional complexity
- Overusing generators
- Infinite loops

General best practices

Memory Management

Generators are typically memory-efficient

Yield smaller chunks

Find balance: making API calls and database calls is heavy on performance too

Because the dataset can be processed in chunks.

Memory Management

Generators are typically memory-efficient

Yield smaller chunks

Find balance: making API calls and database calls is heavy on performance too

Lazy Evaluation

Strong point of generators

Reduce initial load time

Reduce memory footprint

Avoid unnecessary computations

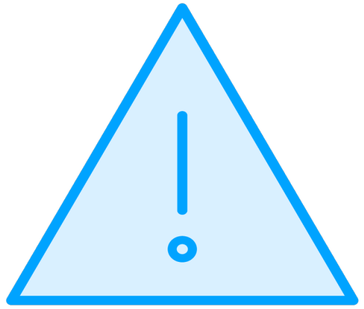
Complex Asynchronous Situation

Handle enormous amounts of data async

More readable that complex nested callbacks and promise chains

More control than just async/await

Generator Pitfalls



Don’t overuse generators

Use the Best Tool for the Job

Whenever a simpler solution is available, such as:

- Loop
- Array
- Async/await

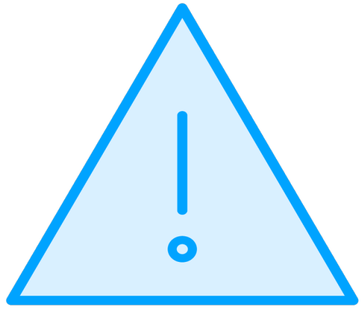
Generators are great for their use cases but add complexity

Whenever generators yield the result of an API or database call they generate overhead

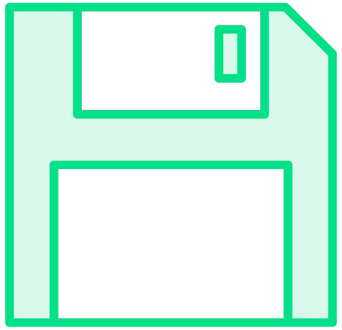
Using Generators or Alternatives

Assess the requirement, consider the possible solutions and choose the cleanest and highest performing solution.

Generator Pitfalls



Don’t overuse generators



Don’t use generators for small datasets

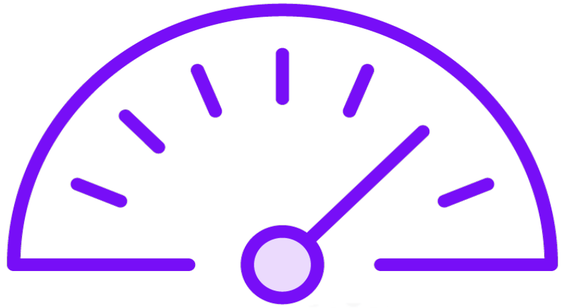
Generators and Overhead

Generators don’t add overhead

Often they yield the result of a DB or API call

Those calls are heavy on the performance

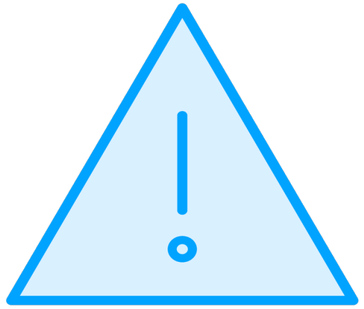
It’s only considered overhead when it’s a small dataset



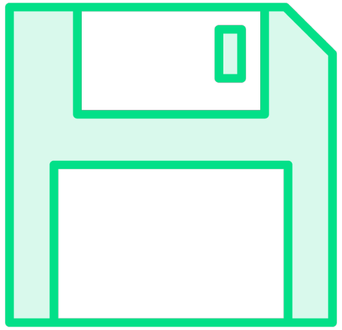
Choosing the Best Solution

Profile your code and try it with and without a generator to see what it does to performance.

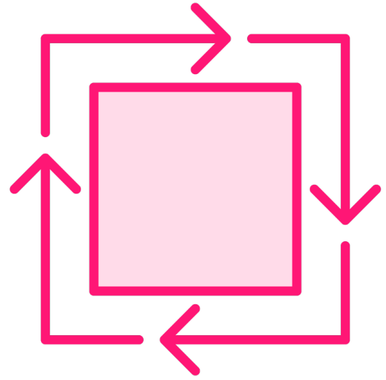
Generator Pitfalls



Don’t overuse generators



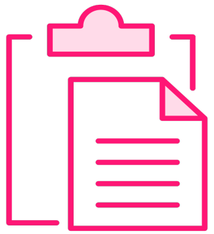
Don’t use generators for small datasets



Be cautious of infinite loops

Best Practices

Testing



Documentation

Keep It Simple (Stupid)

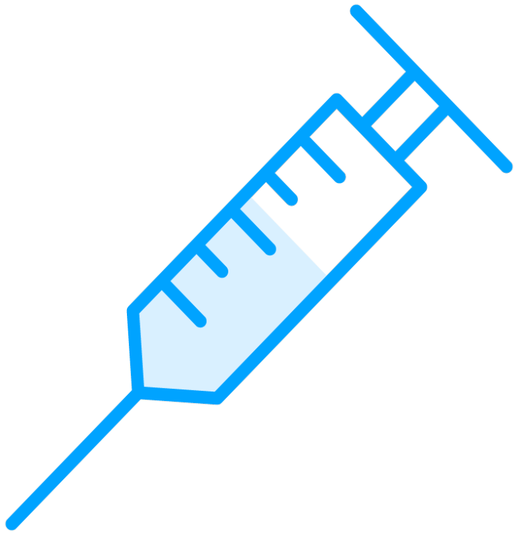
Security and generators

Security and Generators

Generators are not especially dangerous

Considerations and potential issues can arise for any code construct

Injection Attacks in Generator Functions



Not generator specific

Injection attacks possible when generators interact with the database or external APIs

Validate and sanitize user inputs and use prepared statements

Resource Exhaustion

Risk in the context of large datasets, intensive computations or long running processes

Could be exploited in DoS attacks

Implement limits and handling resource limits

Monitor and manage resource usage

Side Effects and State Management

Generators maintain state

State could include sensitive data

Carefully manage state to avoid unintended side effects

Generators Are Not Inherently Insecure

Like any code construct, security best practices must be applied to reduce risks.

Course wrap up

Summary

- Debug generator functions
- Evaluate the performance of generators
- Performance optimization
- Security and generators

Reflect on what you've learned

Be Like the Generators in Learning

Don’t process everything all at once

Learning should be done in manageable chunks

Iterate through each concept

Gradually build your skillset

Good luck!