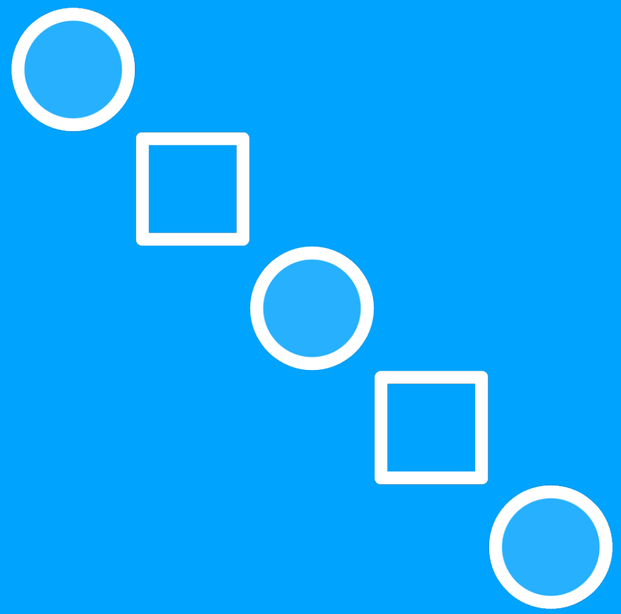
Working with Generator Patterns

Maaike van Putten

Lead Trainer & Software Developer

@brightboost | www.brightboost.nl



No Typical Design Patterns

We'll be dealing with combinations of common coding constructs with generators.

Overview

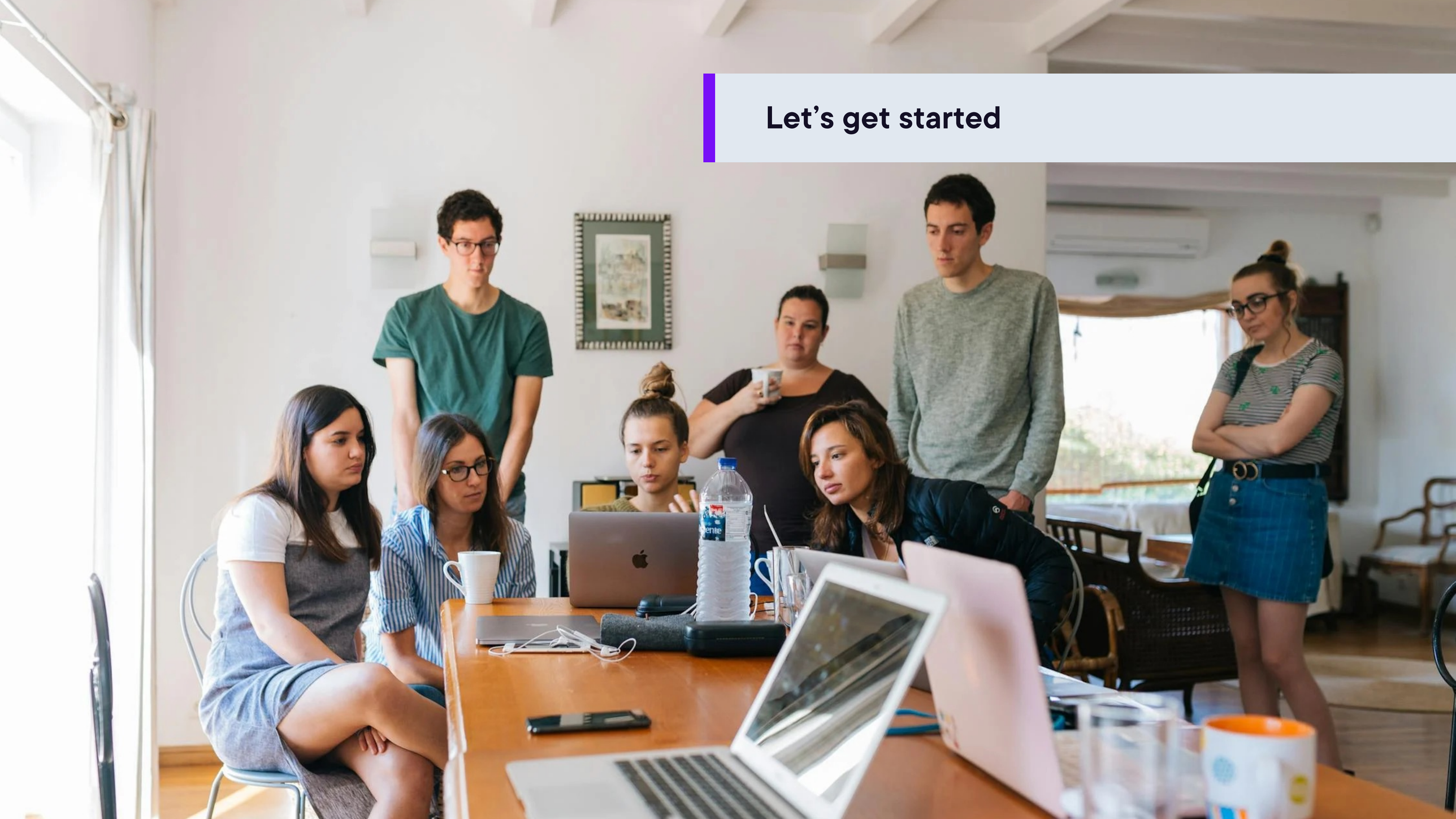
Generators and promises

Async/await with generators

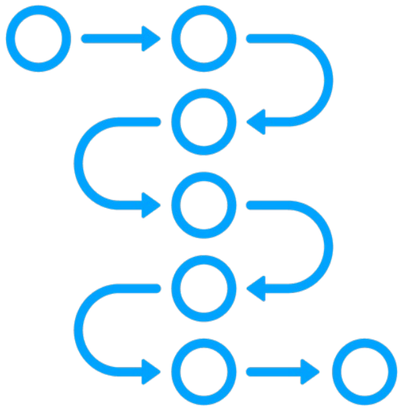
Error handling for generators

Generators in enterprise frameworks

Let’s get started

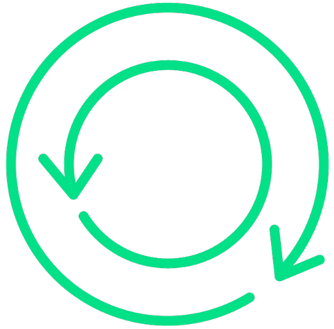


Generators and Async/Await



Generators

Functions that can be paused and resumed



Async/Await

Async functions that can be waited for with the await keyword

They can be seen both as alternatives and as complementary, depending on the scenario.

Async/Await

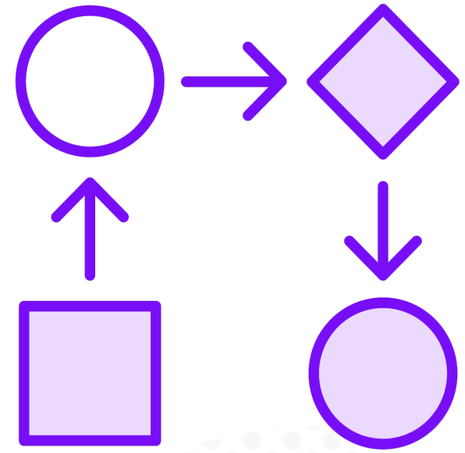
Simplifies working with promises

Makes async code look like sync code

Async functions return a promise

Await is used to pause the execution until the promise is resolved

Great for sequential async operations

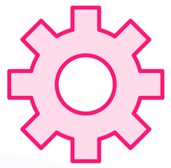


Generators

Functions that can be paused and resumed

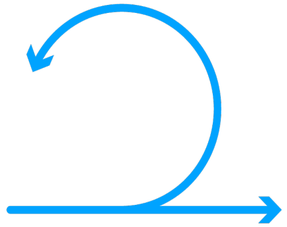
Yield multiple values over time

Not necessarily asynchronous but can be used to handle async code

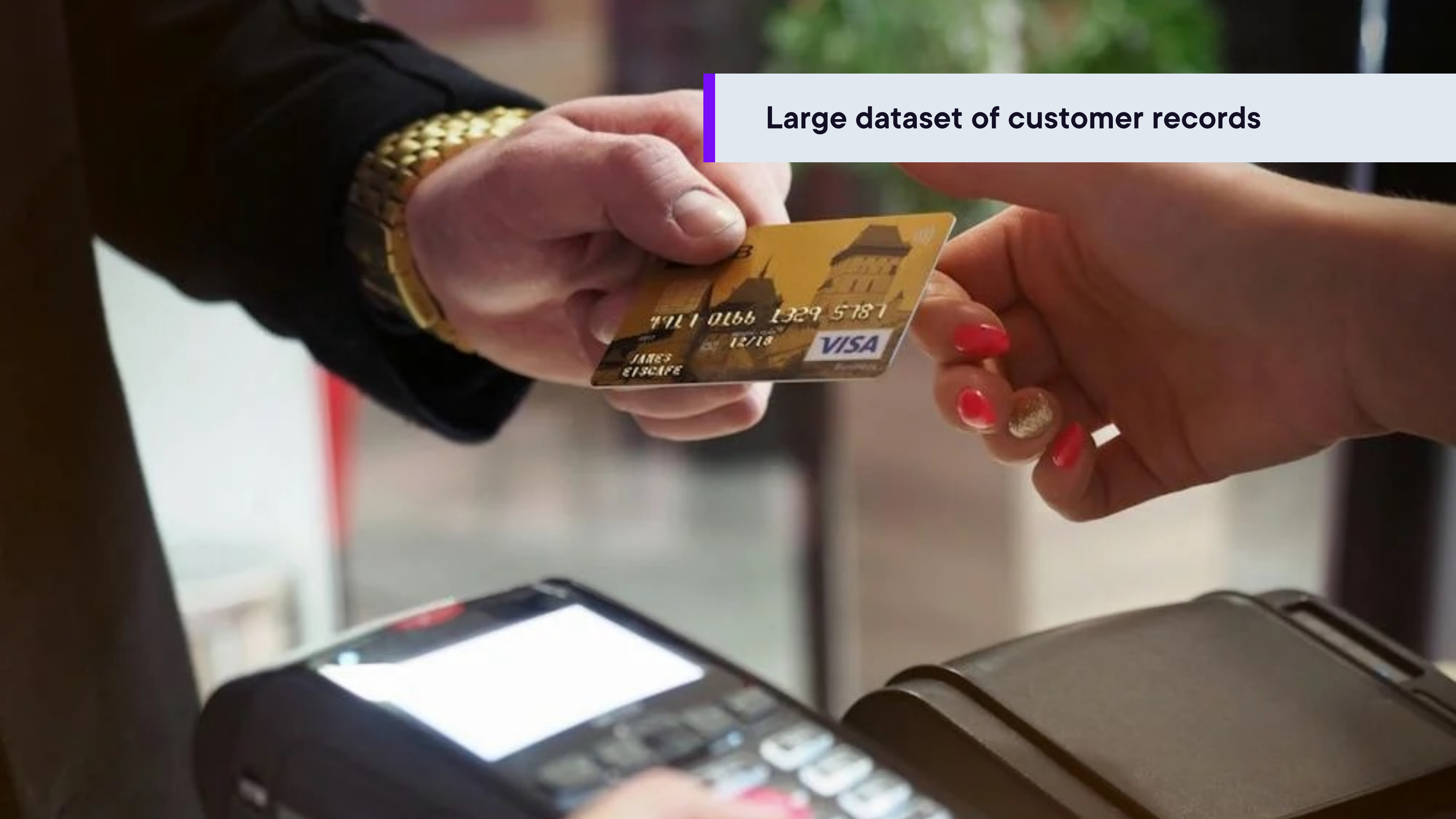


Yielding promises and using external mechanisms to resume execution

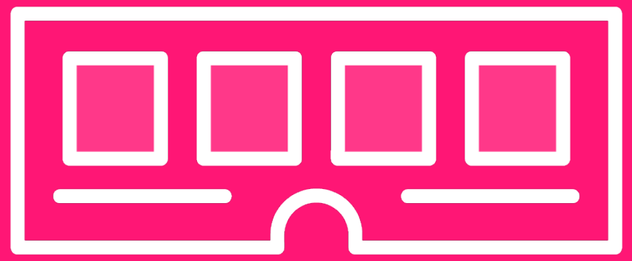
Use Cases for Generators



Complex iteration scenarios

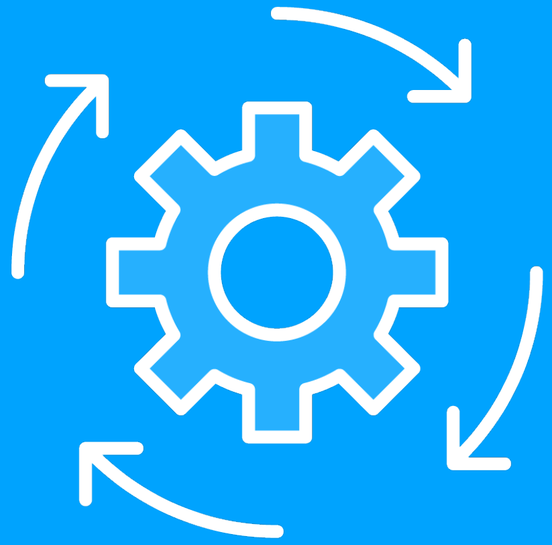


Large dataset of customer records



Memory Constraints

This dataset is too large to load into memory all at once. The data also needs to be processed first.



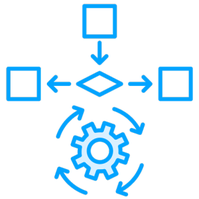
Generator

A JavaScript generator is perfect for this scenario.

Generators

Read from the dataset in chunks

With each yield, the generator can pause after processing each chunk



Ability to perform complex operations

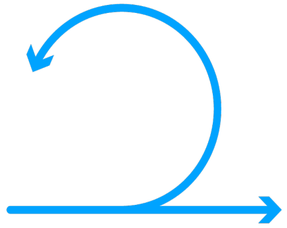
Memory friendly

Process Data On the Fly

Process data before the entire dataset is available

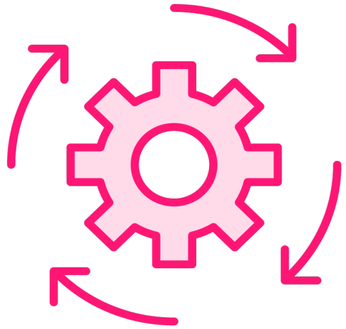
Real-time and time-sensitive data analysis

Use Cases for Generators



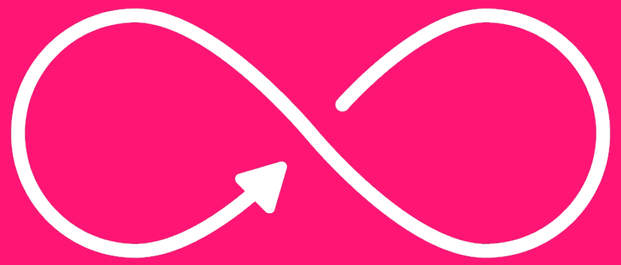
Complex iteration scenarios

Lazy evaluation



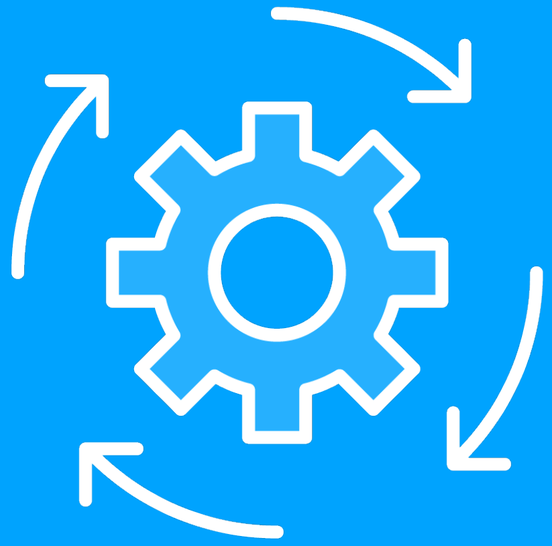
Streams of data


Real-time financial data
12:23
BTC / USDT
37 296,08
+4,45 %
38 708,27
90 305,20
Low
35 657,69
Vol(USDT)
3 365 304 410,56
Depth
More
1d
4h
1h
15m
Line
MA(7):37 595,02
MA(25):37 144,45
MA(99):34 954,87
37 296,08
39 000,00
38 708,27
37 000,00
35 000,00
35 000,00
02-04
02-03
02-02
31 153,26
VOL:3 440,43
RSI
KDJ
MACD
VOL
Info
0.01
0,174239
0,004500
Market History
Ask
37 296,08
37 296,07
37 296,89
37 294,88
37 297,71
37 292,57
EMA
BOLL
MA
Order Book
Bid
0,149695
0,021357
0,041040
Buy
Alert



Infinite Stream

This stream is continuous and potentially infinite, so you can't store all of it in memory.

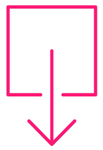


Generators

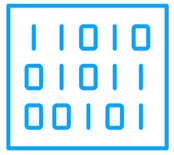
Yield each incoming data point, which allows the rest of your application to process the data in a just-in-time manner.

Generators

Evaluate and display information

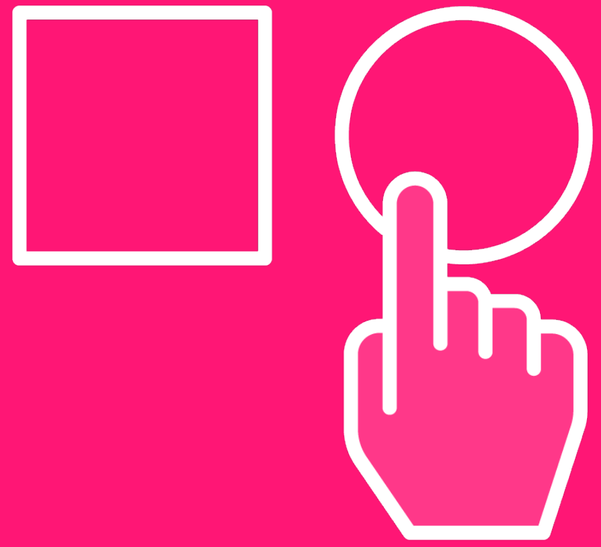


On demand



No pre-loading unnecessary data

Efficiency and responsiveness



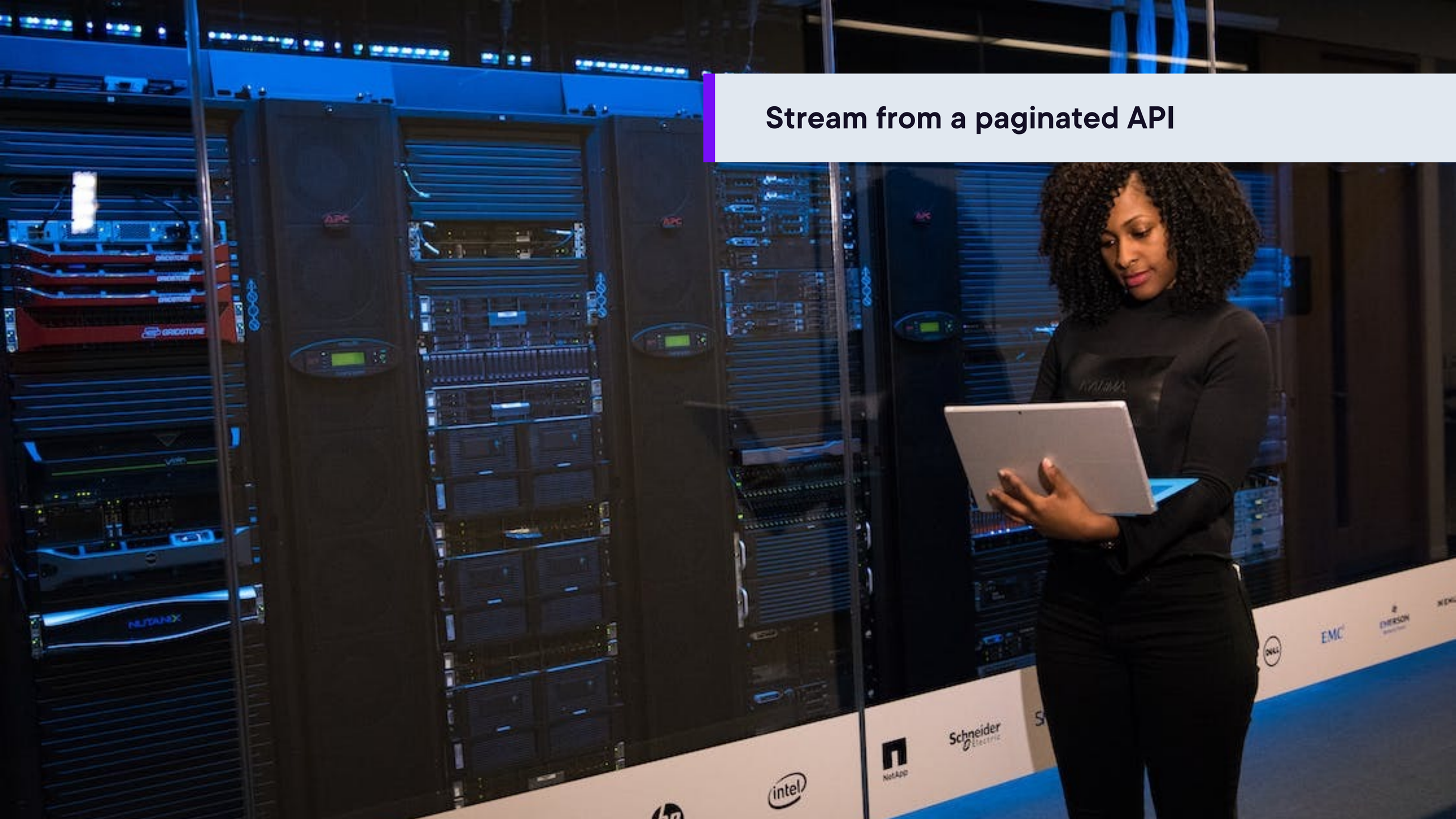
Alternatives?

If we look at them as alternatives, very often async/await is enough for the requirements.

Complementary

There are scenarios in which combining async/await with generators is ideal.

Stream from a paginated API



Page By Page

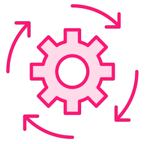
Loading all products at once would be inefficient and resource-intensive, so instead we are going to load and process the data page by page.

Async/await within a generator

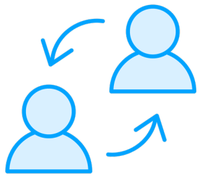
Allows for asynchronous operations to be handled with a synchronous-like control flow.

Async/await within a Generator

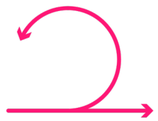
Yielding each product as it becomes available



Enabling the consuming code to start processing right away



Better responsiveness



Display or process incrementally

Demo

Create async generators

Combine the features of async/await and generators

Also know as IIFE: Immediately Invoked Function Expression

Why Not Just Async/Await?

Higher level of control and incremental processing

Process data in chunks without loading the entire set into memory at once

Handling errors in generators

We need to be prepared for unexpected situations

Generators can handle errors gracefully

Avoiding Unhandled Errors

We can surround the generator with a try/catch to avoid unhandled errors.

Better to Handle Inside the Generator

It’s typically better to surround the code inside the generator with a try/catch.

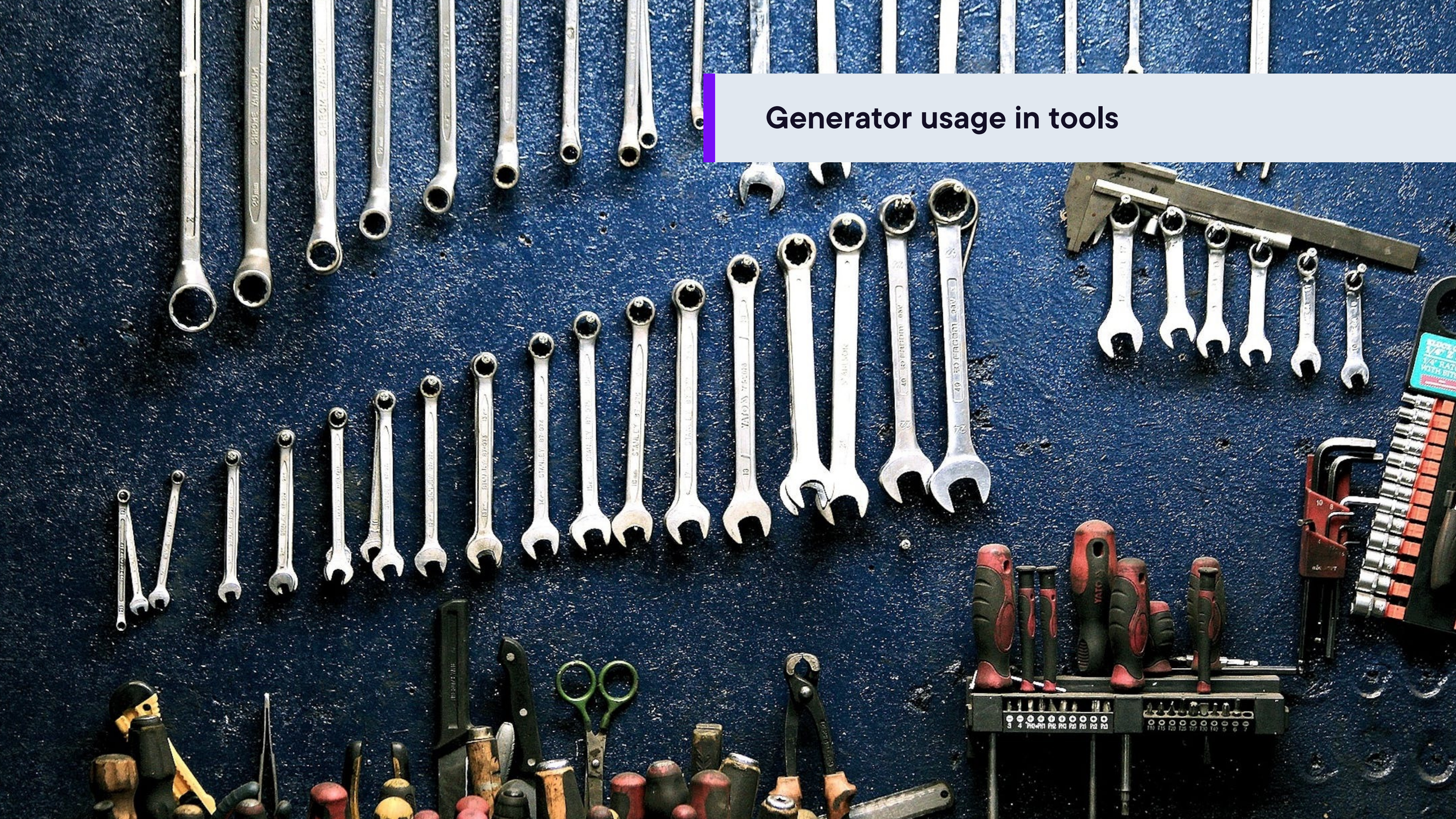
Problem Solved

With try/catch the generator won't crash

Yields an error message

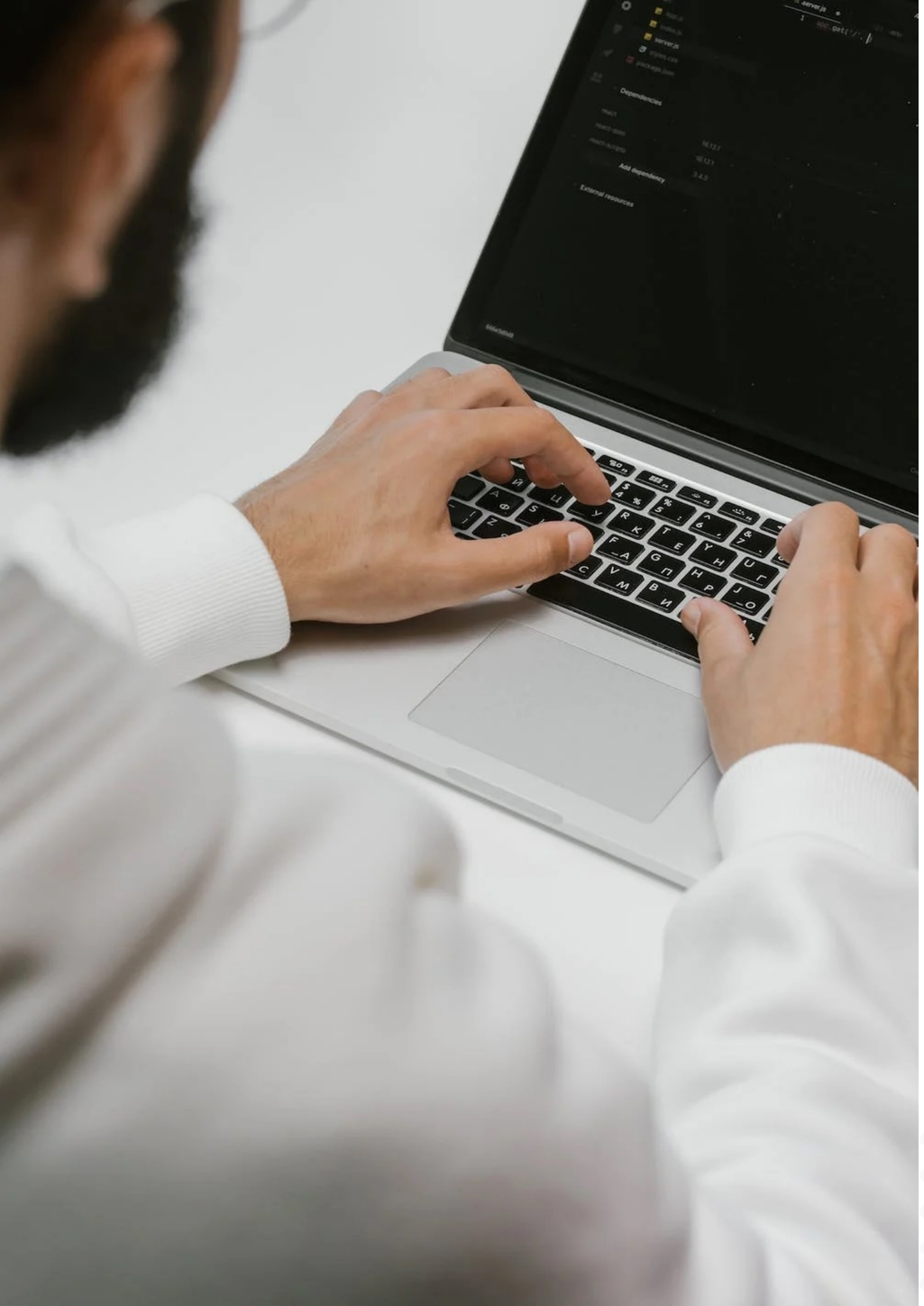
This message can be logged or handled

Generator usage in tools



Generators are often replaced with the async/await syntax

Low-level construct that is crucial for certain types of handling, such as streams of data



Some packages require generator functions to use the package

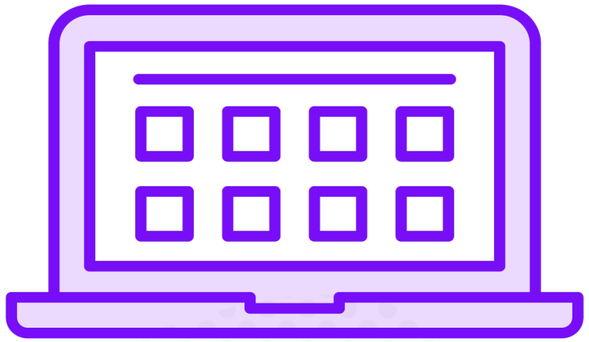
An example is redux saga

Redux Saga

Modern JS library

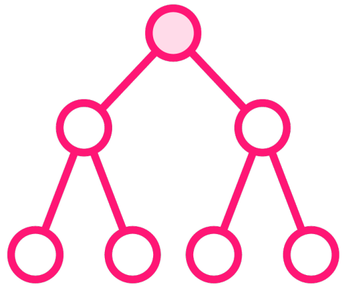
Utilizes generator functions

It is made for managing side effects in redux applications

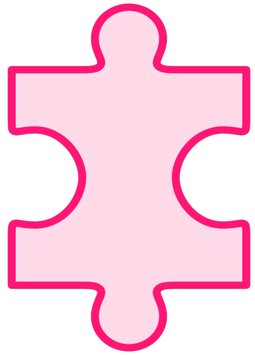


Redux Saga

Special because of:



Complex async flow



Straightforward logic

Easy to follow

Handling Complex Side Effects

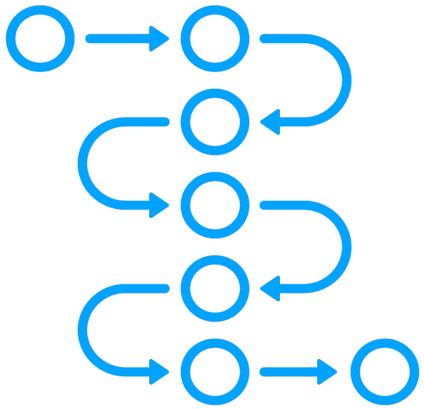
Cancellation

Parallel execution

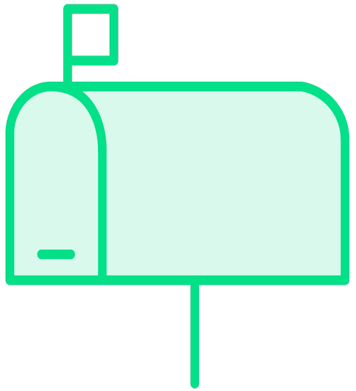
Easy testing

Promises and generators

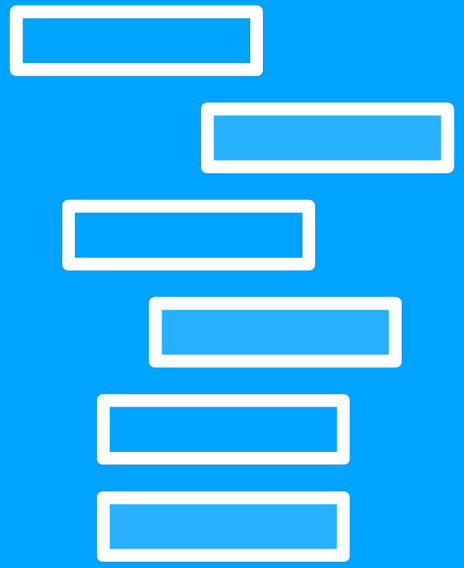
Generators and Promises



Generators
Functions that can be paused and resumed



Promises
Objects that produce resolved values



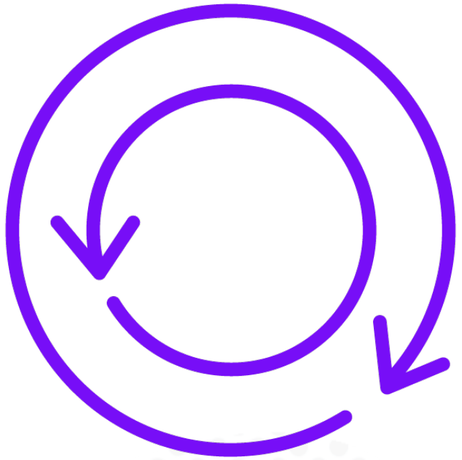
Combining Promises and Generators

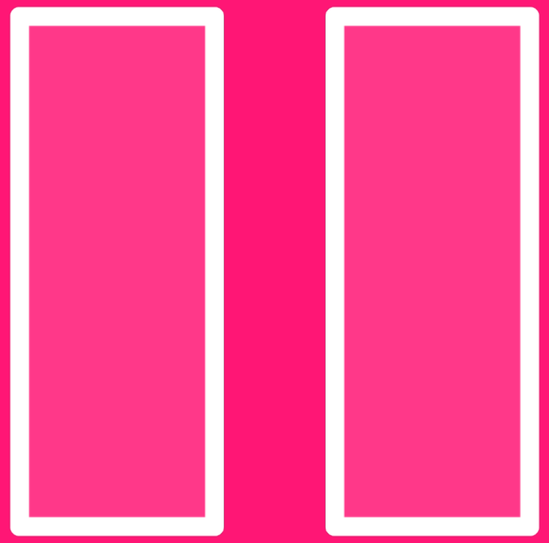
The pattern we're about to discuss was one of the preferred ways to manage asynchronous code in a synchronous manner.

Managing Asynchronous Flow

Generators can yield promises

This allows a function to pause until the promise is resolved





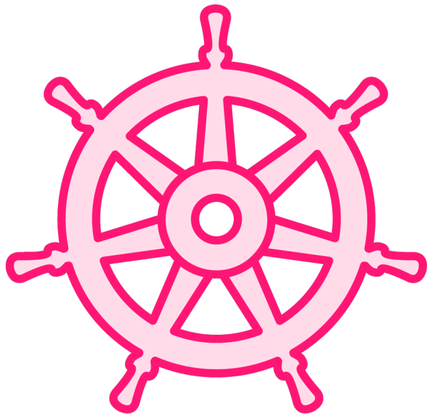
Make Async Look Sync

Pausing a function until a promise is resolved allows for asynchronous operations to appear synchronous within the flow of your code.

Benefits of Making Async Code Look Sync

Higher code readability

Error handling simplified



Fine-grained control over execution flow

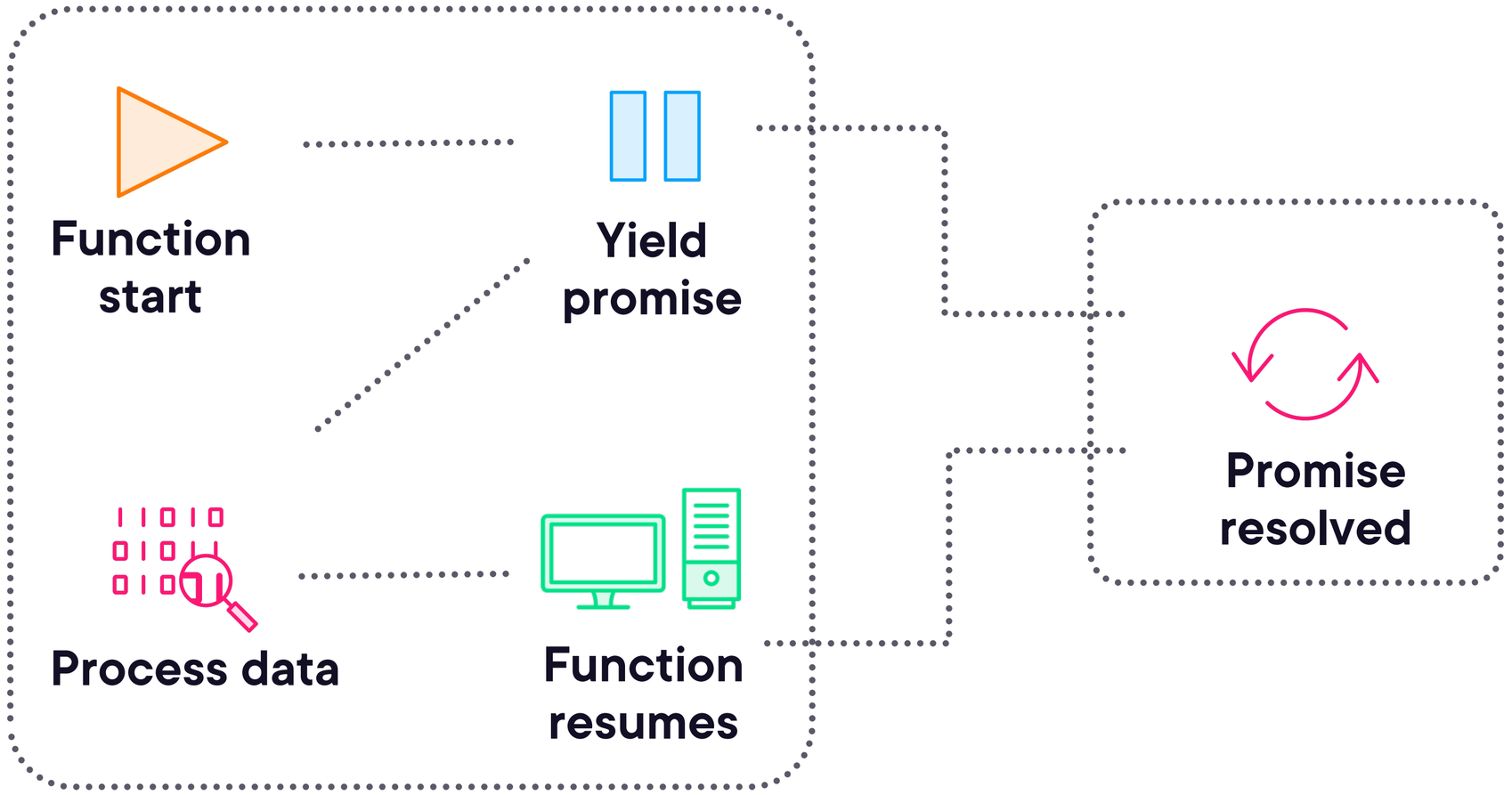
Data Fetching in a Web Application

Using generators and promises

Application can fetch initial user data and then pause until the data is received and processed

Then proceed to the next data fetch

Generator and Promise Flow Diagram

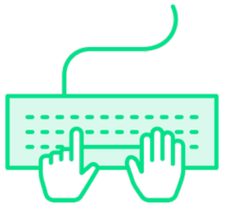


Demo

Yielding promises in a generator

Would I do it like this if I had to implement this specific use case?

Generator Benefits and Promise Chain Benefits



Control

Simplicity

Flexibility

Less boilerplate

Error handling

Wide support

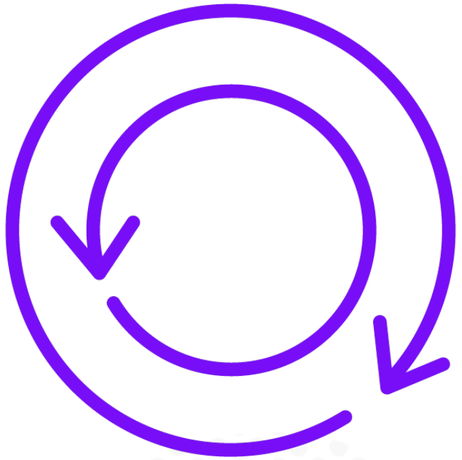
Managing Asynchronous Flow

Generators can yield promises

This allows a function to pause until the promise is resolved

This can be combined with the use of an external iterator to resume execution

The value of the promise can then be given back to the generator



Demo

Using an external iterator to resume execution

Demo

Demo without generator

Let’s wrap up

Summary

Generators and promises

Async/await with generators

Error handling for generators

Generators in enterprise frameworks

Up Next:

Practical Use Cases for Generators