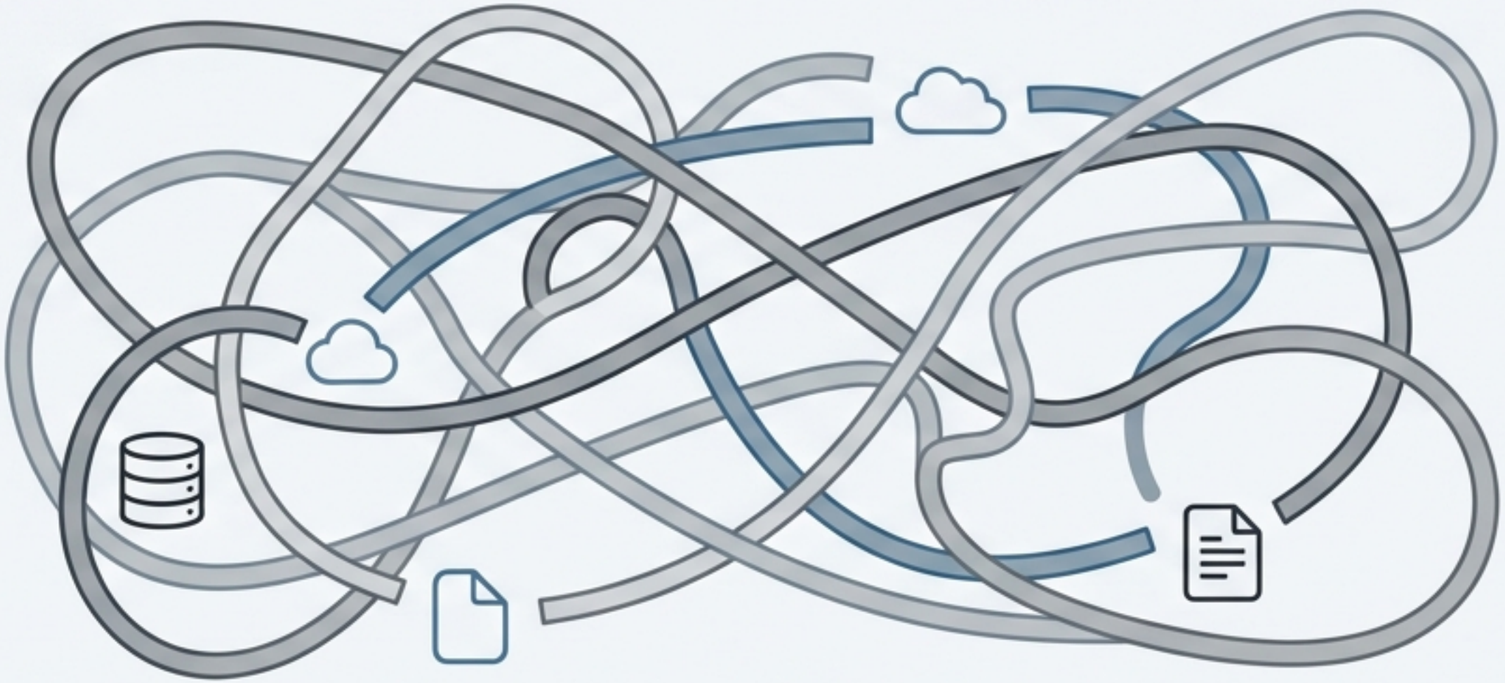
The Challenge of Asynchronous Java

Asynchronous operations are essential for modern applications, but writing clear, readable, and maintainable async code in Java can be a significant challenge. The standard `CompletableFuture` is powerful, yet its API can lead to verbose, nested, and hard-to-reason-about code, especially when handling errors and composing multiple operations.



A Better Way: Focus on Effects, Not Operations

`BetterFuture` is a thin wrapper around Java's `CompletableFuture` designed to be more consistent and easier to use. Its core principle is to let you **focus on the *effects* of your asynchronous actions—the transformations, sequences, and error** handling—instead of the low-level operational details.

"In essence, a future captures the notion of latency and error: a future is the promise of a value at some later time that might fail to compute."

Creating a Future

`BetterFuture` provides clear, intention-revealing factory methods for creating instances.

For Asynchronous Tasks

Use `BetterFuture.future()` to wrap a `Callable`. The future automatically fails if the `Callable` throws an exception.

```
// Executes asynchronously
BetterFuture<Data> data =
BetterFuture.future(API::fetchDa
ta);
```

For Already Completed Values

Use `succeeded()` and `failed()` for futures that are resolved from the start.

```
// Instantly successful
BetterFuture<String> name =
BetterFuture.succeeded("Alice");

// Instantly failed
BetterFuture<Integer> error =
BetterFuture.failed(new
IOException());
```

For Manual Completion

Use the constructor for futures that you will complete later.

```
// To be completed manually
var manualFuture = new
BetterFuture<Boolean>();
manualFuture.succeed(true);
```

Consuming a Future's Result

Once a future is created, you can register callbacks to react to its completion or block execution to wait for its result.

Reacting to Completion (Non-Blocking)

Register callbacks to handle success, failure, or any completion state without blocking the current thread.

```
future.onSuccess(value ->
System.out.println("Got: " + value));

future.onFail(error -> log.error("Task
failed", error));
```

Awaiting Completion (Blocking)

Poll the future's status or block until the result is available. Useful in specific contexts like testing or at the end of a process.

```
// Block until complete
User user = userFuture.get();

// Check status without blocking
if (userFuture.isSucceeded()) { ... }
```

Transforming a Value with `.map()`

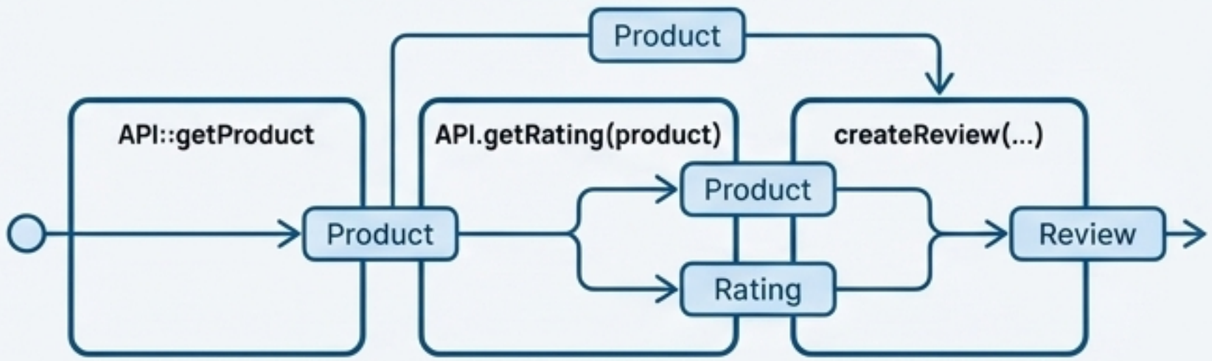
The `.map()` method allows you to apply a function to a future's successful result, producing a new future of a different type. It lifts a simple function into the asynchronous world.

```
// We have a future that will eventually contain a String
BetterFuture<String> stringFuture = BetterFuture.succeeded("42");

// We map it to a future that will contain an Integer
BetterFuture<Integer> intFuture = stringFuture.map(Integer::parseInt);
```

Sequencing Dependent Tasks with .andThen()

When your next asynchronous operation depends on the result of the previous one, use `.andThen()`. This chains futures together in a clear, linear sequence, avoiding nested callbacks.

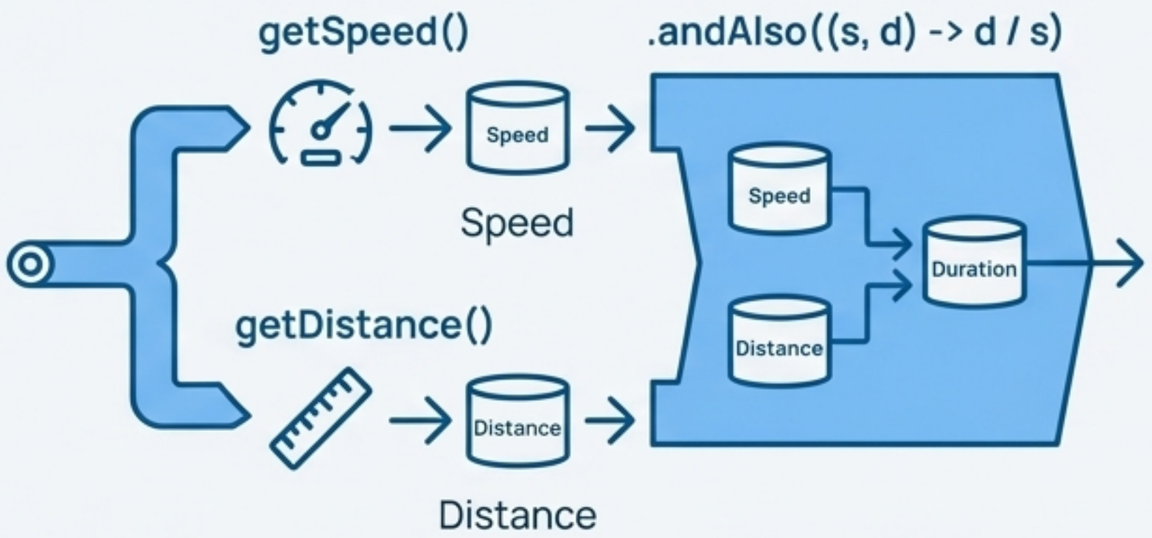
```
// 1. Get a product asynchronously
BetterFuture<Product> productFuture =
BetterFuture.future(API::getProduct);

// 2. Use the product to get its rating, then create
a review
BetterFuture<String> reviewFuture =
productFuture.andThen(product ->
    BetterFuture.future(() -> API.getRating(product))
        .map(rating -> createReview(product, rating))
);
```



Executing Independent Tasks in Parallel with .andAlso()

When two asynchronous tasks don't depend on each other, they can be executed in parallel for maximum efficiency. **.andAlso()** runs another future alongside the current one and provides a way to combine their results when both are complete.

```
// Run both in parallel and combine results
to calculate duration

BetterFuture<Integer> speedFuture =
BetterFuture.future(API::getSpeed);
BetterFuture<Integer> distanceFuture =
BetterFuture.future(API::getDistance);

BetterFuture<Integer> durationFuture =
speedFuture.andAlso(
    distanceFuture,
    (speed, distance) -> distance / speed
);
```



Recovering Gracefully from Failure

Failed futures can disrupt your entire execution chain. `BetterFuture` provides `recover` methods that allow you to handle a failure and transform it back into a successful future, preventing catastrophic failure.

.recover()

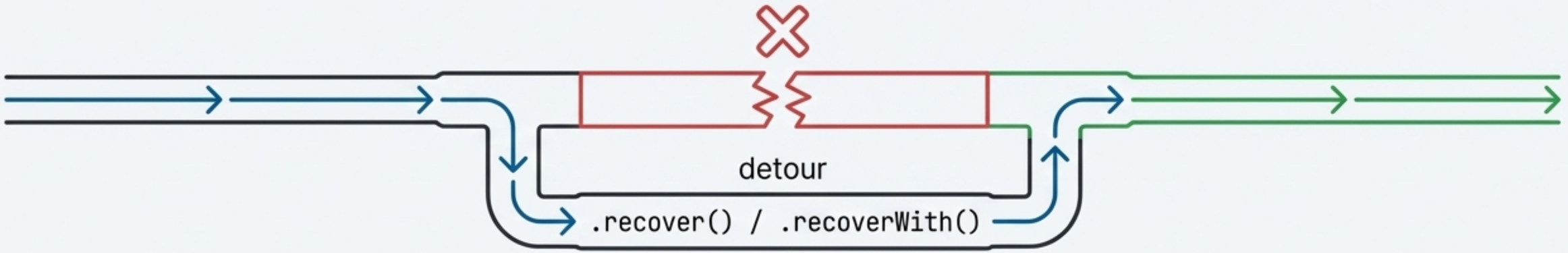
Provide a default value if the future fails.
Corresponds to `.map()` for the failure case.

```
// Returns "default.json" if the API call fails
BetterFuture<String> config = BetterFuture
    .future(API::fetchConfig)
    .recover(error -> "default.json");
```

.recoverWith()

Execute another asynchronous operation as a fallback.
Corresponds to `.andThen()` for the failure case.

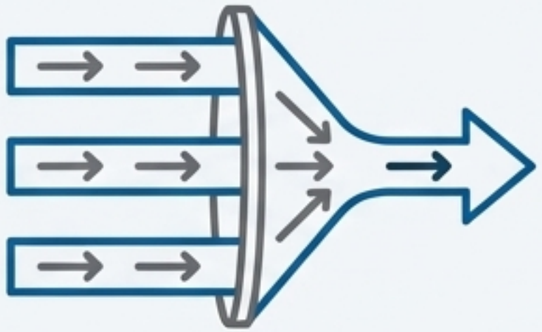
```
// Tries a backup API if the primary one fails
BetterFuture<Data> data = BetterFuture
    .future(API::fetchPrimary)
    .recoverWith(error ->
      BetterFuture.future(API::fetchBackup));
```



Composing Streams of Futures

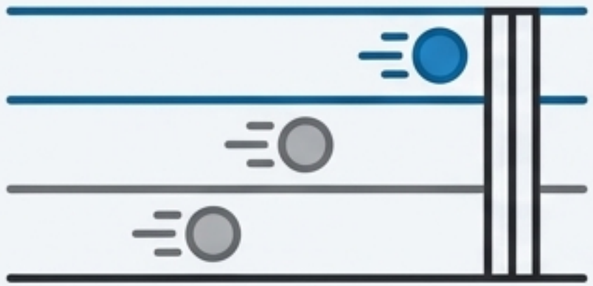
`BetterFuture` provides powerful combinators for working with multiple futures at once, enabling complex patterns like parallel reduction and fan-in collection with simple, declarative code.

BetterFuture.reduce()



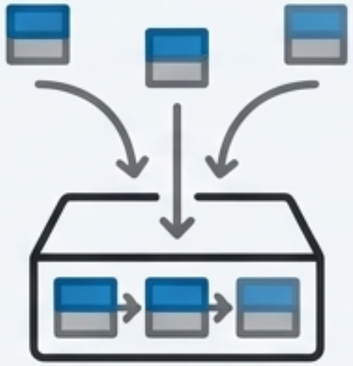
Reduces a `Stream<BetterFuture<T>>` to a `BetterFuture<Stream<T>>`. All futures run in parallel. The result fails immediately if any of the input futures fail.

BetterFuture.first()



Given a stream of futures, returns a new future that completes with the result of the *first* instance in the stream to complete.

BetterFuture.collect()



Collects a stream of futures into a blocking queue, ordered by their completion time, not their position in the original stream.

Bringing It All Together: A Complete Flow

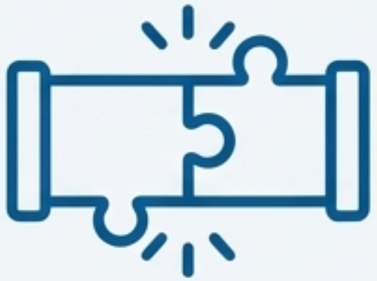
Let's revisit the product review example. Notice how creation, sequencing, and transformation chain together into a single, declarative block of code. This flow is easy to read, reason about, and maintain.


```
BetterFuture<String> productReview = BetterFuture.future(API::getProduct) // <-- 1. Create Future
    .andThen(product -> ———————————————————————————————————————— // <-- 2. Sequence
        BetterFuture.future(() -> API.getRating(product))
            .map(rating -> newProductReview(product, rating)) ———————— // <-- 3. Transform
    );
```

The `BetterFuture` Advantage

Simpler Reasoning

By focusing on effects, the API allows you to write code that clearly expresses your intent without getting lost in operational details.



Powerful Composition

Elegantly handle sequential, parallel, and recovery flows with a fluent, chainable API.

Consistent API

Provides a clean and consistent wrapper over `CompletableFuture`, smoothing out its rough edges and adding useful functionality like `future(Callable)`.

Readable & Maintainable

The declarative style results in code that is easier to read, test, and maintain over time.

Start Building a Better Future

`BetterFuture` is an open-source project ready for you to explore and use in your own applications. Find the complete source code, documentation, and examples on GitHub.

github.com/mduerig/better-future

Apache-2.0 License