NDC { Copenhagen }

Hybrid Caching in .NET

Jody Donetti
Coding + R&D

Jody Donetti

Coding + R&D

Doing stuff (mainly) on the web for around 30 years.

Dealt with most types of caches: memory, distributed, hybrid, HTTP caching, offline caching, CDNs.

Created FusionCache, a .NET hybrid caching library.

🏆 Google Open Source Award

🏆 Microsoft MVP Award

What now?

What now?

we are here

Show you **only 1 or 2** things, deeply.

BUT

You would **miss** a lot of interesting things.

Show you **a lot** of juicy things, less deeply.

BUT

You may deepen **later** online, if interested.

What now?

we are here

Show you **only 1 or 2** things, deeply.

BUT

You would **miss** a lot of interesting things.

Show you **a lot** of juicy things, less deeply.

BUT

You may deepen **later** online, if interested.

What now?

So I'll show you a lot of (hopefully) interesting things.

To do that, I'll go a bit fast.

The hope is you'll **discover** something new, see a lot of **interesting things** that may spark your imagination on how to apply them in **your own** projects/scenarios.

And, if so, you can always go deeper online later.

Hybrid what?

🤔 Hybrid what?

There are mainly 3 types of caches:

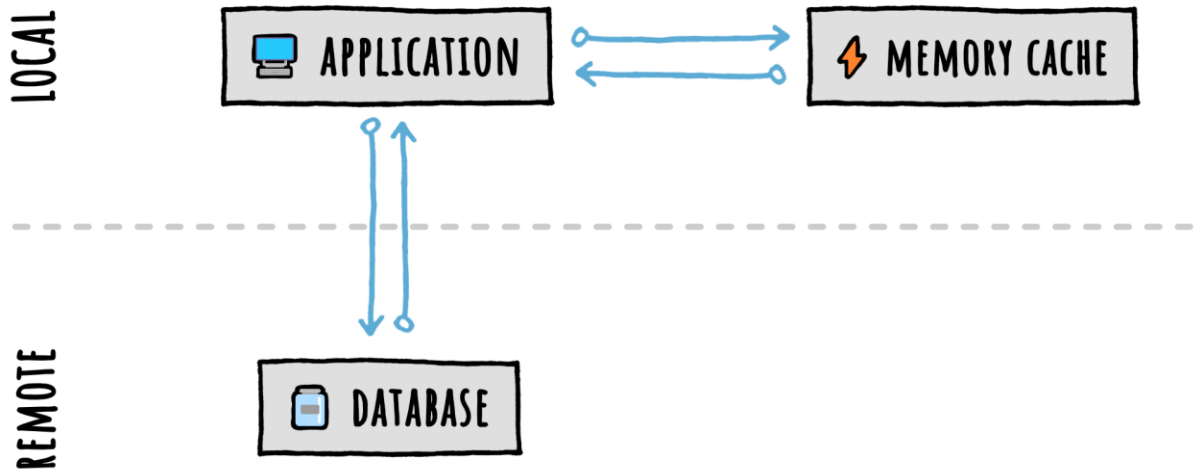
- memory caches
- distributed caches
- hybrid/multi-level caches

Memory Caches

Memory caches store data in memory.

And not just “in memory”, but in the **same** memory space as the application using it.

They are like a dictionary + some form of eviction.



Memory Caches

We can use them like this:

```
var product = cache.GetOrCreate<Product>(
    $"product:{id}",
    // WARNING: NO STAMPEDE PROTECTION
    _ => GetProductFromDb(id),
    options
);
```

⚡ Memory Caches

Some examples of **memory caches** in .NET:

📦 **BitFaster.Caching**
https://github.com/bitfaster/BitFaster.Caching

📦 **FastCache**
https://github.com/jitbit/FastCache

📦 **fast-cache**
https://github.com/neon-sunset/fast-cache

📦 **LazyCache**
https://github.com/alastairtree/LazyCache

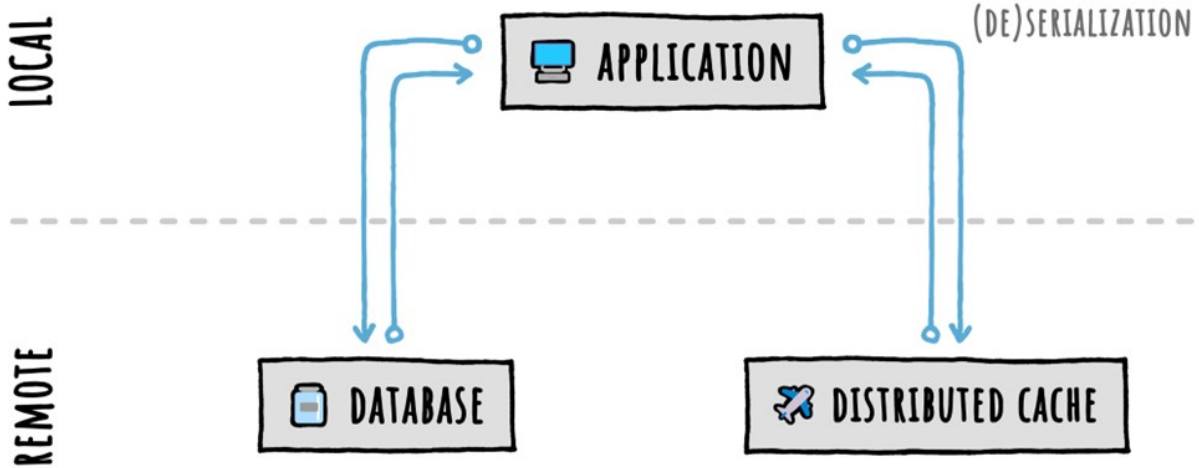
Ⓜ️ **Microsoft MemoryCache**

Distributed Caches

Distributed caches are remote key-value stores (e.g.: Redis, Memcached).

Like a database but simpler, with less features: because of that, way faster.

In .NET it's `IDistributedCache` + available implementations, and they talk `byte[]`.



Distributed Caches

We can use them like this:

```
Product product;
var payload = distributedCache.Get($"product:{id}");
if (payload is not null)
{
    product = JsonSerializer.Deserialize<Product>(payload);
}
else
{
    // WARNING: NO STAMPEDE PROTECTION
    product = GetProductFromDb(id);
    payload = JsonSerializer.Serialize<Product>(product);
    distributedCache.Set($"product:{id}", payload);
}
```

 Distributed Caches

Some examples of **distributed caches** in .NET (`IDistributedCache` implementations):

📦 **EnyimMemcachedCore** (for Memcached)
https://github.com/cnblogs/EnyimMemcachedCore

📦 **MongoDbCache** (for MongoDB)

https://github.com/outmatic/MongoDbCache

📦 **NeoSmart.Caching.Sqlite** (for SQLite)
https://github.com/neosmart/SqliteCache

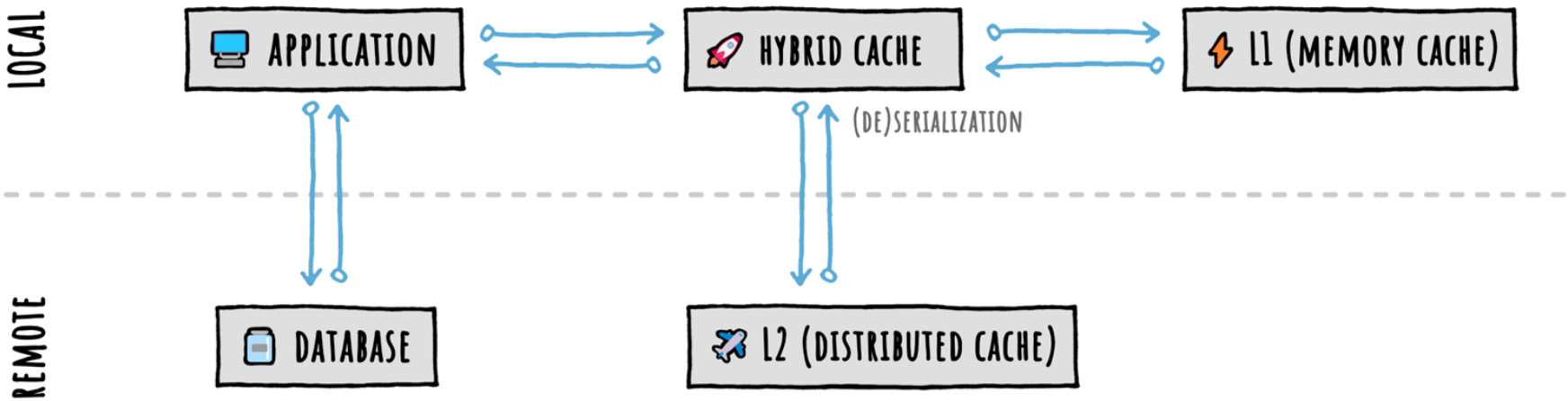
📦 **AWS.AspNetCore.DistributedCacheProvider** (for Amazon DynamoDB)
https://github.com/aws/aws-dotnet-distributed-cache-provider/

Ⓜ️ **Microsoft.Extensions.Caching.StackExchangeRedis** (for Redis)

Hybrid Caches

Hybrid caches are the most advanced caches.

They combine the best of both worlds, together: memory (L1) + distributed (L2).



Hybrid Caches

We can use them like this:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Hybrid Caches

Some examples of **hybrid/multi-level caches** in .NET:

📦 **FusionCache (hybrid)**
https://github.com/ZiggyCreatures/FusionCache

📦 **CacheTower (multi-level)**
https://github.com/TurnerSoftware/CacheTower

📦 **CacheManager (multi-level)**
https://github.com/MichaCo/CacheManager

📦 **EasyCaching (multi-level)**
https://github.com/dotnetcore/EasyCaching

Ⓜ️ **Microsoft HybridCache (hybrid)**

Libraries

In this session, we'll focus on these 2:

FusionCache

HybridCache

Ok, but why?

⚡ Memory Caches

Pros/Cons:

- ✅ **easy:** easy to use
- ✅ **data locality:** near our own code, in the same memory space
- ✅ **cost:** no remote calls, no (de)serialization
- ✅ **availability:** it is always available
- ✅ **stampede protection:** maybe there (check the library)
- ⛔ **cold starts:** at every restart the cache is empty
- ⛔ **horizontal scaling:** data is not shared between multiple nodes

✈️ Distributed Caches

Pros/Cons:

- ⛔ **easy:** less easy, we need to handle (de)serialization, etc
- ⛔ **data locality:** data is out-of-process, typically remote
- ⛔ **cost:** network calls + (de)serialization
- ⛔ **availability:** may not be always available
- ⛔ **stampede protection:** not there
- ✅ **cold starts:** at every restart the cache is already populated
- ✅ **horizontal scaling:** data is shared between multiple nodes

🚀 Hybrid Caches

Pros/Cons:

- ✅ **easy:** easy to use
- ✅ **data locality:** near our own code, in the same memory space (**L1**)
- ✅ **cost:** no remote calls, no (de)serialization (**L1**)
- ✅ **availability:** it is always available (**L1**)
- ✅ **stampede protection:** usually there (but always check the library)
- ✅ **cold starts:** at every restart L1 is empty, but **L2** is already populated
- ✅ **horizontal scaling:** data is shared between multiple nodes (**L2**)

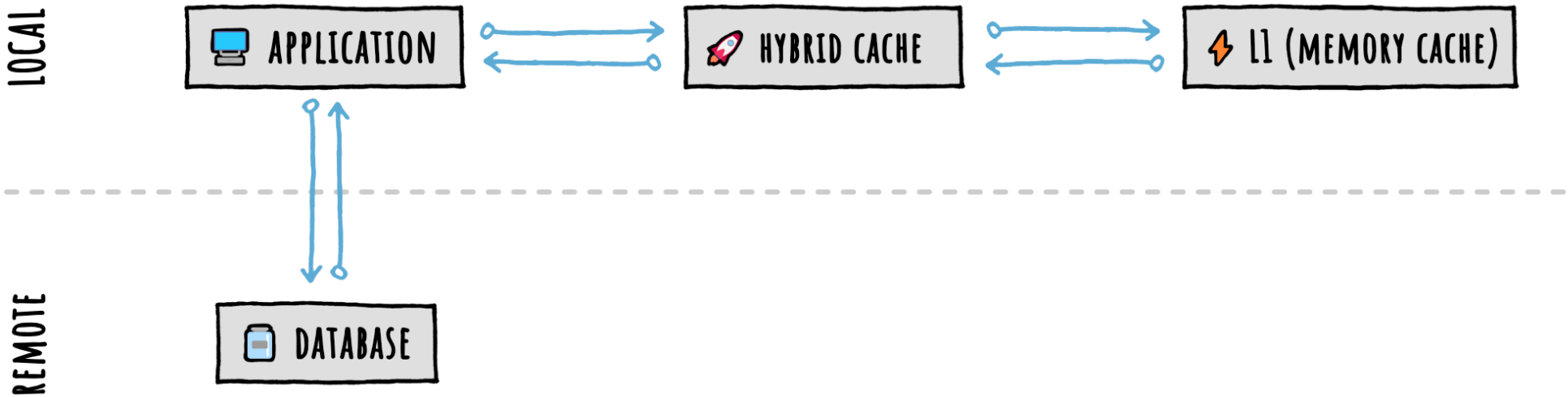
Of course when talking with L2 we have the cost for network + (de)serialization, but **only** once, when getting the first CACHE MISS on L1.

Hybrid != L1+L2

Hybrid != L1+L2

By using a hybrid cache you are **not forced** to use multiple levels (L1+L2).

You can use a hybrid cache with **only L1**:



🚀 Hybrid != L1+L2

Ok but... why?

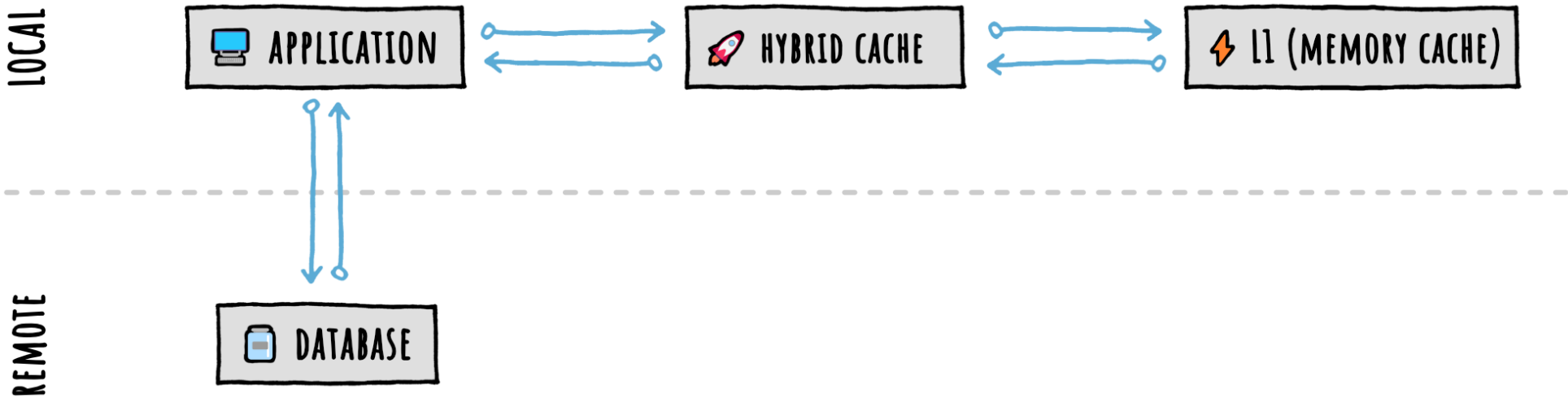
To get a super-powered memory cache with (depending on the cache):

- **stampede protection:** not always present in memory caches
- **more features:** like Eager Refresh, Factory Timeouts, Fail-Safe, etc
- **observability:** logs, traces, metrics (OTEL)

But more importantly: we can **transparently** go multi-level, with **no changes** in our code.

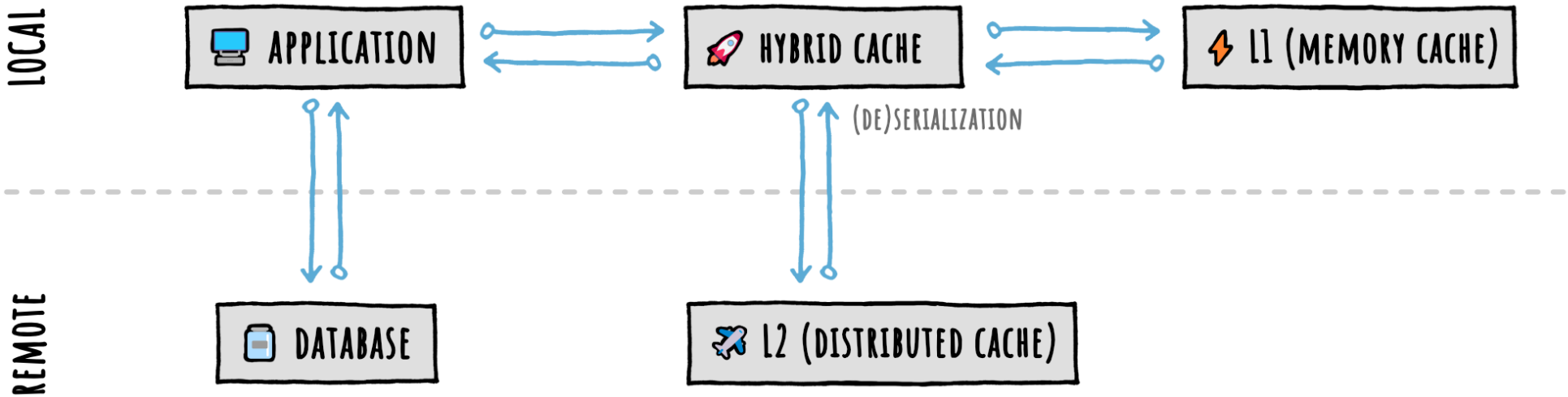
Hybrid != L1+L2

Meaning: it doesn't matter if we want **only L1**...



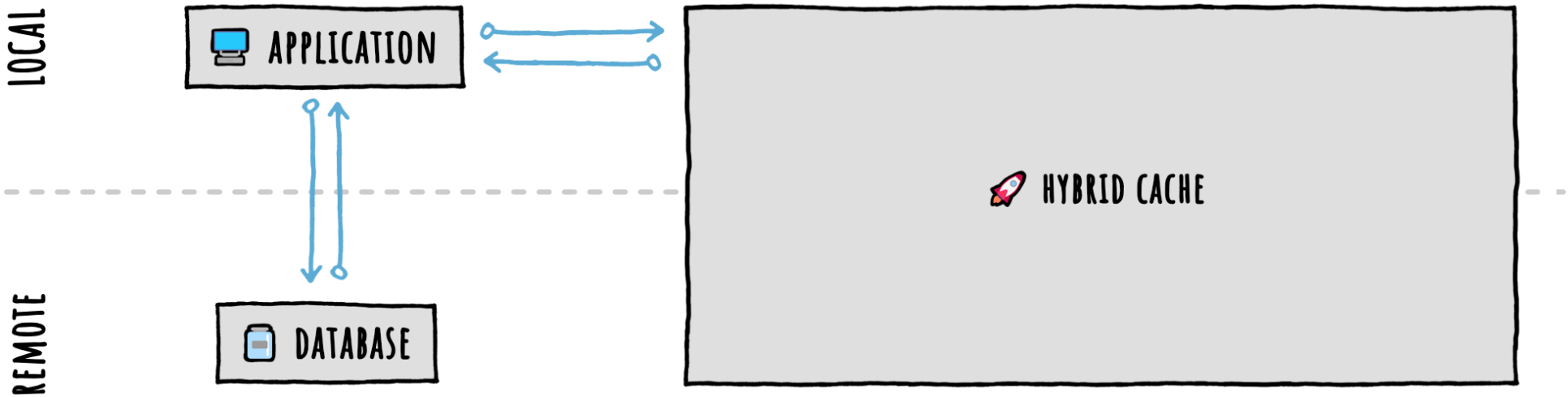
Hybrid != L1+L2

... or **L1+L2** because, either way...



Hybrid != L1+L2

... we can just **transparently** code against a single, **unified** API:



Hybrid != L1+L2

Basically, with **L1 only**:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

Hybrid != L1+L2

Or with **L1+L2**, with L2 on **Redis** + **Protobuf** serialization:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

Hybrid != L1+L2

Or with **L1+L2** again, but with L2 on **Memcached** + **JSON** serialization:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Hybrid != L1+L2

The calling code remains **always the same**.

No need to change it everywhere, only 1 line during setup.

For example:

- **L1:** use locally, during dev
- **L1+L2:** use in staging/production

Yeah, I kinda like hybrid caches 🙂

🚀 Hybrid Caches: “a hybrid cache”

Oh, one more thing.

When we talk about **«a hybrid cache»** we are talking about the **general cache type**, not necessarily `HybridCache` from Microsoft (2025).

For example, FusionCache (2020) is also «a hybrid cache».

It’s like the difference between «a memory cache» in general, and `MemoryCache`.

Hybrid VS Multi-Level

🚀 Hybrid VS Multi-Level

I am frequently using both «multi-level» and «hybrid» as if they are interchangeable.

They are somewhat similar, but different.

Hybrid VS Multi-Level

A **multi-level** cache supports any number of levels, each of any type.

As developers, we can think of them like this (no real code):

```
class MultiLevelCache
{
    List<ICacheLevel> Levels { get; }
}
```

Hybrid VS Multi-Level

A **hybrid** cache supports either L1 (memory) or L1+L2 (memory+distributed).

As developers, we can think of them like this (no real code):

```
class HybridCache
{
    IMemoryCache L1 { get; }
    IDistributedCache? L2 { get; }
}
```

🚀 Hybrid VS Multi-Level

Although hybrid caches may look more «limited» they are, in fact, more powerful.

But why?

Well:

- limitations are **pragmatic**, with no real-world impacts
- they ensure **stronger foundations** to build upon
- allow a **richer design** with **more advanced features**
- allow more **granular** control (e.g.: skip L1/L2 per-call)
- in general, they **just work**, even in complex scenarios

All in all, they are (imho) the right **balance**.

FusionCache

FusionCache

Having worked with most types of caches (memory, distributed, hybrid, etc) I faced a lot of the **challenges** of dealing with caching and had to handle a lot of the **real-world issues**.

And how to **prevent** them, when possible, or **mitigate**/**solve** them when inevitable.

FusionCache

The **OSS community** always gave me a lot: sometimes I contributed back, but not in a big way (patches, bug fixes, minor things).

So I decided to give back more, and in 2020 **FusionCache** was born.

I tried to condense all those caching experiences into one uniform and easy to use package that can help solve **real-world problems**.

FusionCache

It's a hybrid cache:

- **L1:** uses `IMemoryCache`
- **L2:** uses `IDistributedCache`

Any existing `IDistributedCache` implementation works: Redis, Memcached, SQLite, MongoDB, etc (there's a full list in the online docs).

It transparently handles:

- one or two **levels**
- one or more **instances**/**nodes**

without changing a single line of consuming code.

FusionCache

How to **register** it (after installing the package):

```
services.AddFusionCache();
```

ℹ️ **NOTE:** yes, you can also do just `new FusionCache()`

FusionCache

And you can configure a lot more, thanks to a **fluent builder**:

```
services.AddFusionCache()
    .WithOptions(...)
    .WithDistributedCache(...)
    .WithSerializer(...)
    .WithBackplane(...);
```

FusionCache

How to **use** it:

```
public class ProductController : Controller
{
    IFusionCache _cache;

    public ProductController(IFusionCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        return _cache.GetOrSet(
            $"product:{id}",
            _ => GetProduct(id),
            options => options.SetDuration(TimeSpan.FromMinutes(5))
        );
    }
}
```

FusionCache: Features

FusionCache has a lot of features, which can be divided into categories:

- **resiliency**
- **performance & scalability**
- **flexibility**
- **observability**

Let's take a very quick look.

ℹ️ **NOTE:** if you like what you'll see, you can take a deeper look on the **online docs**.

FusionCache: Resiliency

About **resiliency** we have:

- **Cache Stampede protection**
- **Fail-Safe**
- Auto-Recovery

FusionCache: Performance & Scalability

About **performance & scalability** we have:

- **L1** or **L1+L2**
- **Backplane**
- **Eager Refresh**
- **Factory Timeouts**
- Conditional Refresh
- Background Distributed Operations

FusionCache: Flexibility

About **flexibility** we have:

- Named Caches (like HTTP Named Clients)
- Keyed Services
- Tagging
- Clear
- Adaptive Caching
- **DI + Builder**
- Immutability

FusionCache: Observability

About **observability** we have:

- Logging
- OpenTelemetry (traces, metrics)
- Events

FusionCache

Also:

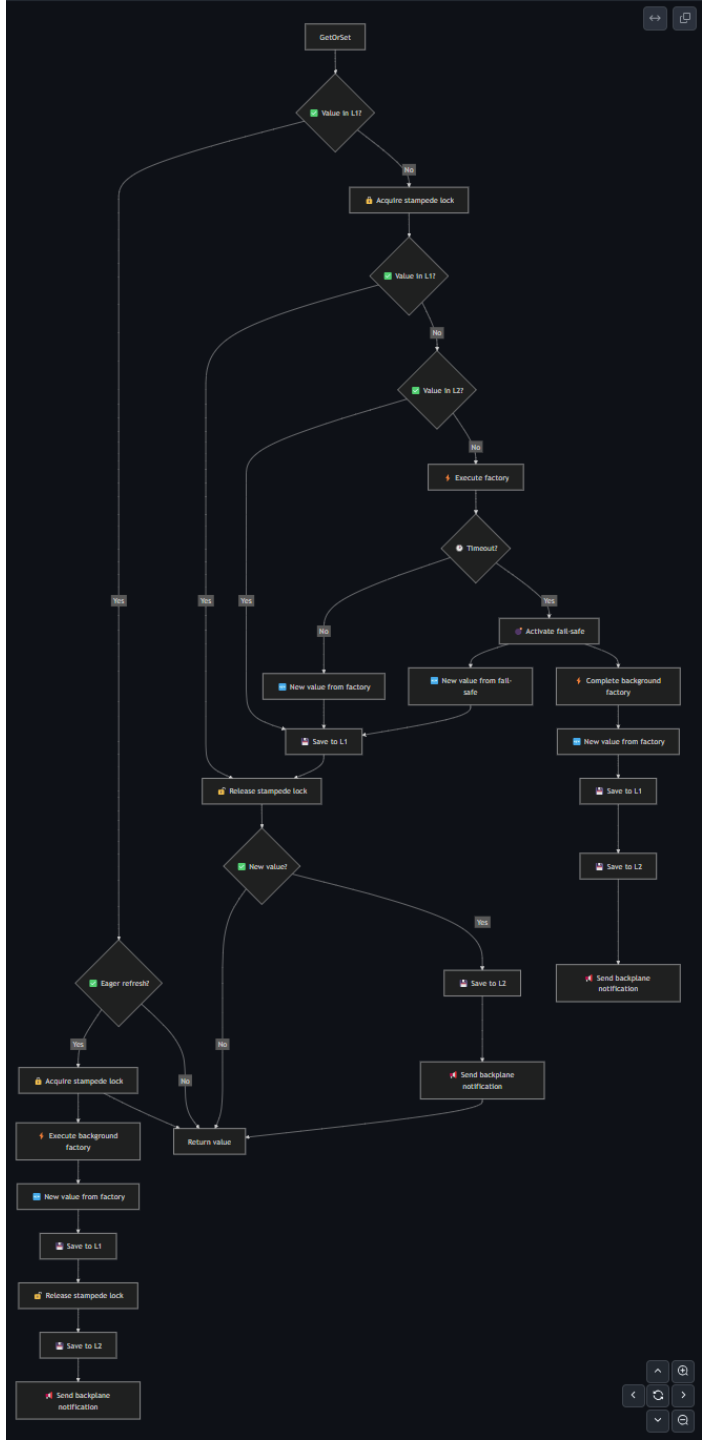
- targets .NET Standard 2.0 (runs everywhere, old + new)
- fully sync + async (no sync-over-async)
- unified serialization interface, to work with `byte[]` (for the L2)
- rich options, both global + entry + `DefaultEntryOptions` + inheritance

FusionCache: Docs

A ton of docs, both **inline** (IntelliSense) and **online**:

- getting started
- every feature, with design rationale
- examples
- diagrams
- step by step (bring some coffee)

Yeah, I care a lot about docs



FusionCache: Who's using it?

It's used by a ton of project and companies, both private and OSS.

They range from small to pretty big like **Have I Been Pwned**, **Dometrain** and even **Microsoft** itself.

And about Microsoft: FusionCache is the caching engine in **Data API Builder** (DAB).

FusionCache: License

FusionCache changing license?

No. Nada. Nein. Nope.

💰 Support

Nothing to do here.

After years of using a lot of open source stuff for free, this is just me trying to give something back to the community.

Will FusionCache one day switch to a commercial model? Nope, not gonna happen.

Mind you: nothing against other projects making the switch, if done in a proper way, but no thanks not interested. And FWIW I don't even accept donations, which are btw a great thing: that should tell you how much I'm into this for the money.

Again, this is me trying to give something back to the community.

If you really want to talk about money, please consider making 💖 **a donation to a good cause** of your choosing, and let me know about that.

Cache Stampede

Cache Stampede

Imagine this scenario.

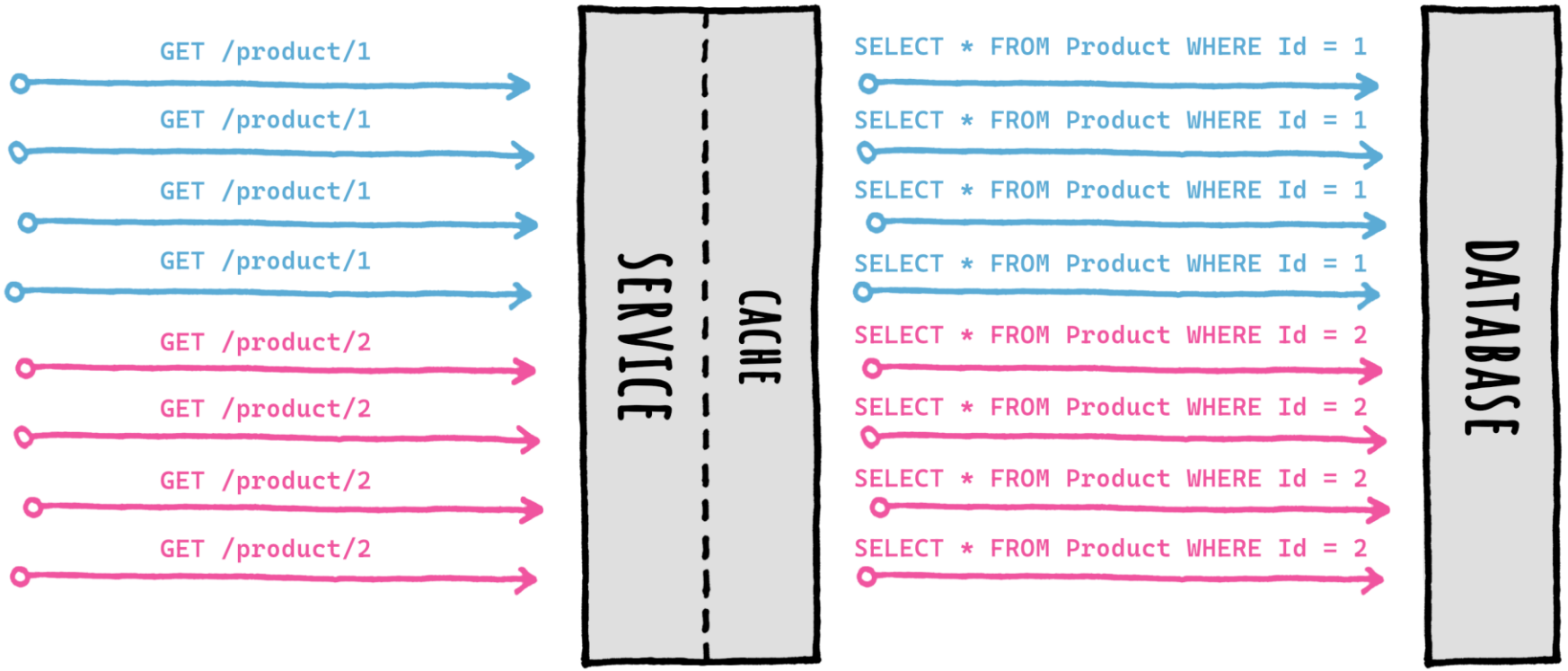
Multiple requests arrive to our app/service, all for the **same data** (not yet in the cache), all at the **same time**.

Without any special care, **all requests** would:

- **cache read:** check the cache for the data
- **cache miss:** not find anything in the cache
- **factory:** go to the database to get fresh data
- **cache write:** save the data in the cache

Cache Stampede

Basically, this:



Cache Stampede

Now: imagine the same scenario with **100** or **1,000** or even more concurrent requests.

This would be a huge **waste** of time, resources and something that can easily tear down our database... maybe during peak traffic time, maybe on Black Friday.

Because of course, right?

Nice to meet you, **Cache Stampede**.

Cache Stampede Protection

Cache Stampede Protection

Some, **but not all**, caching libraries have integrated **Cache Stampede protection**.

They **coordinate**:

- cache calls
- factory execution

for the **same cache key** at the **same time**, automatically.

And what should we do to be protected?

🛡️ Cache Stampede Protection

Instead of doing this (bad):

```
cache.Get(key)
value = factory()
cache.Set(key, value)
```

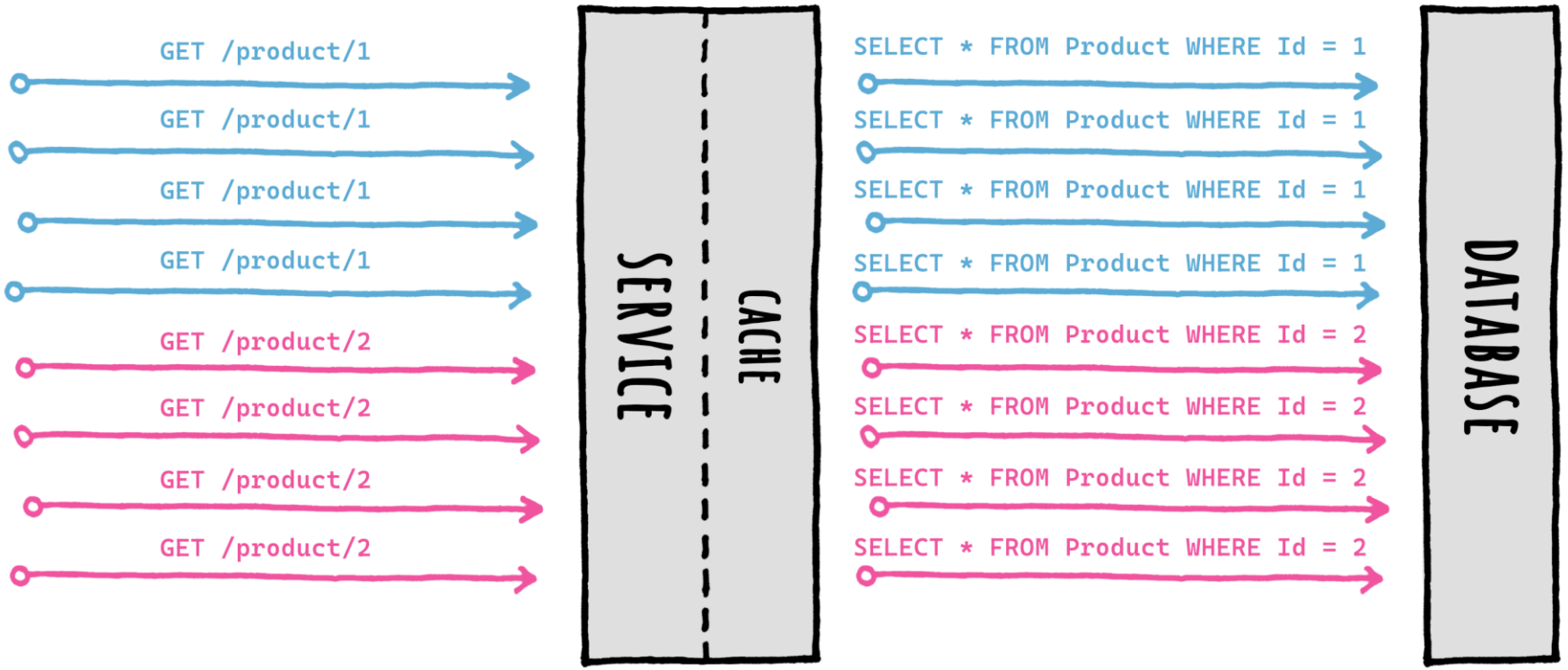
We do this (good):

```
cache.GetOrSet(key, factory)
```

ℹ️ **NOTE:** the method can be called `GetOrCreate()`, `GetOrAdd()`, etc... naming is hard.

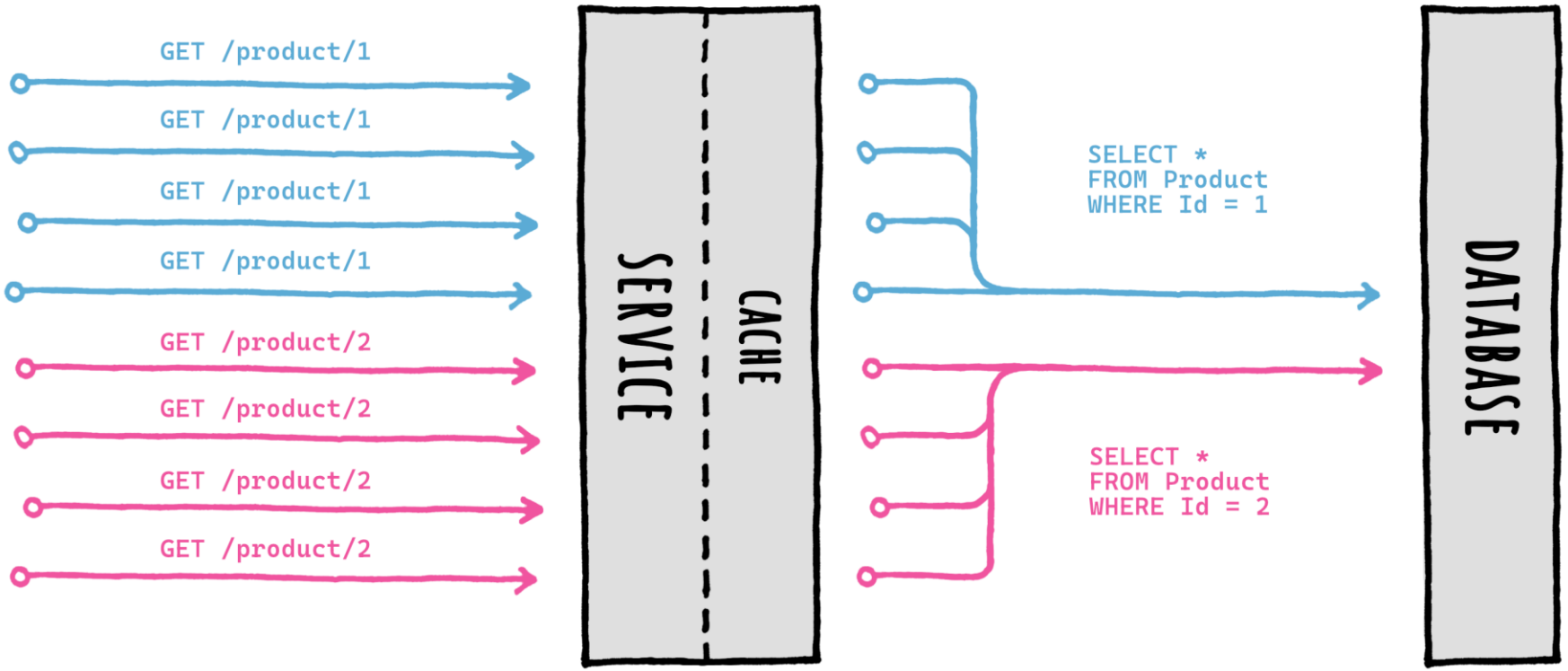
Cache Stampede Protection

It turns this...



Cache Stampede Protection

... into this:



🛡️ Cache Stampede Protection

It's a form of **request coalescing**, where multiple requests are coalesced into one.

Frequently, people think that **IF** there is a method like `GetOrSet(key, factory)` **THEN** the library has stampede protection.

Nope, not true:

- ⛔ **MemoryCache:** no protection, even with `GetOrCreate()`
- ✅ **FusionCache:** protection with `GetOrSet()`
- ✅ **HybridCache:** protection with `GetOrCreateAsync()`

Database Failures

Database Failures

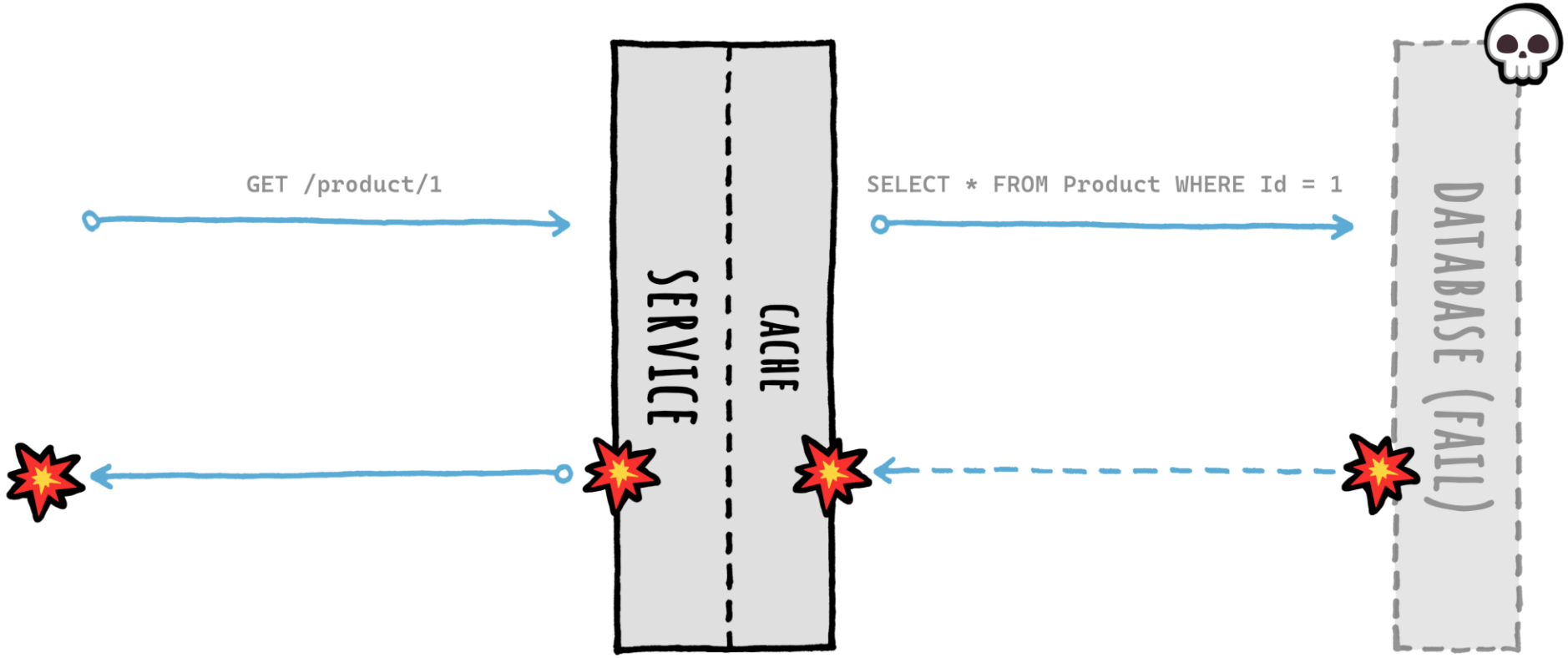
Sometimes when talking to the database we can have **transient failures**.

These can happen for various reasons:

- **query timeouts:** bad query, missing index
- **database restarts:** engine update, crash
- **network issues:** loss of connectivity, topology change
- etc

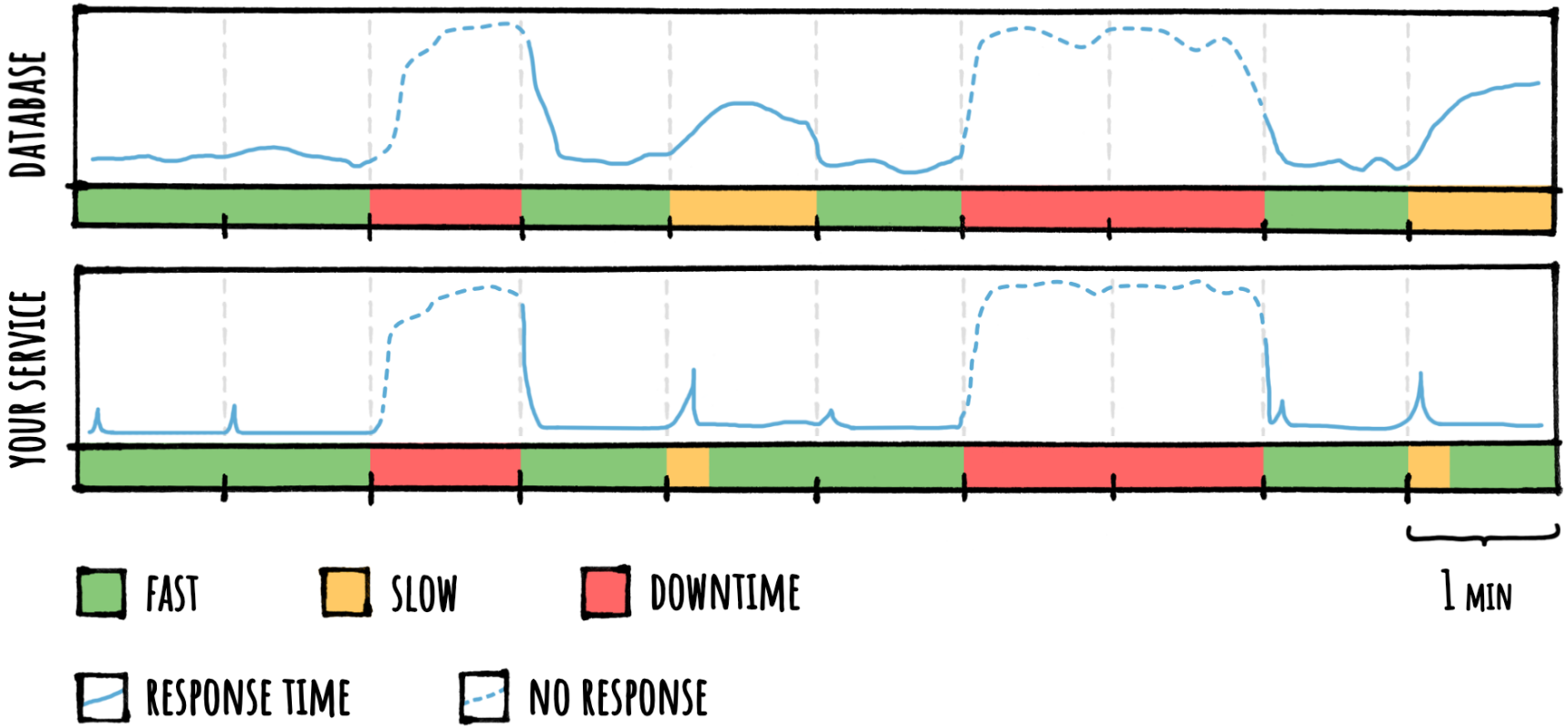
What happens then?

Database Failures



Database Failures

When the **database** throws errors, **your service** throws errors:



Database Failures

But if the data in the cache is already **expired**, and the database is **not available**, is there really something we can do?

Enter: **Fail-Safe**.

Fail-Safe (FusionCache)

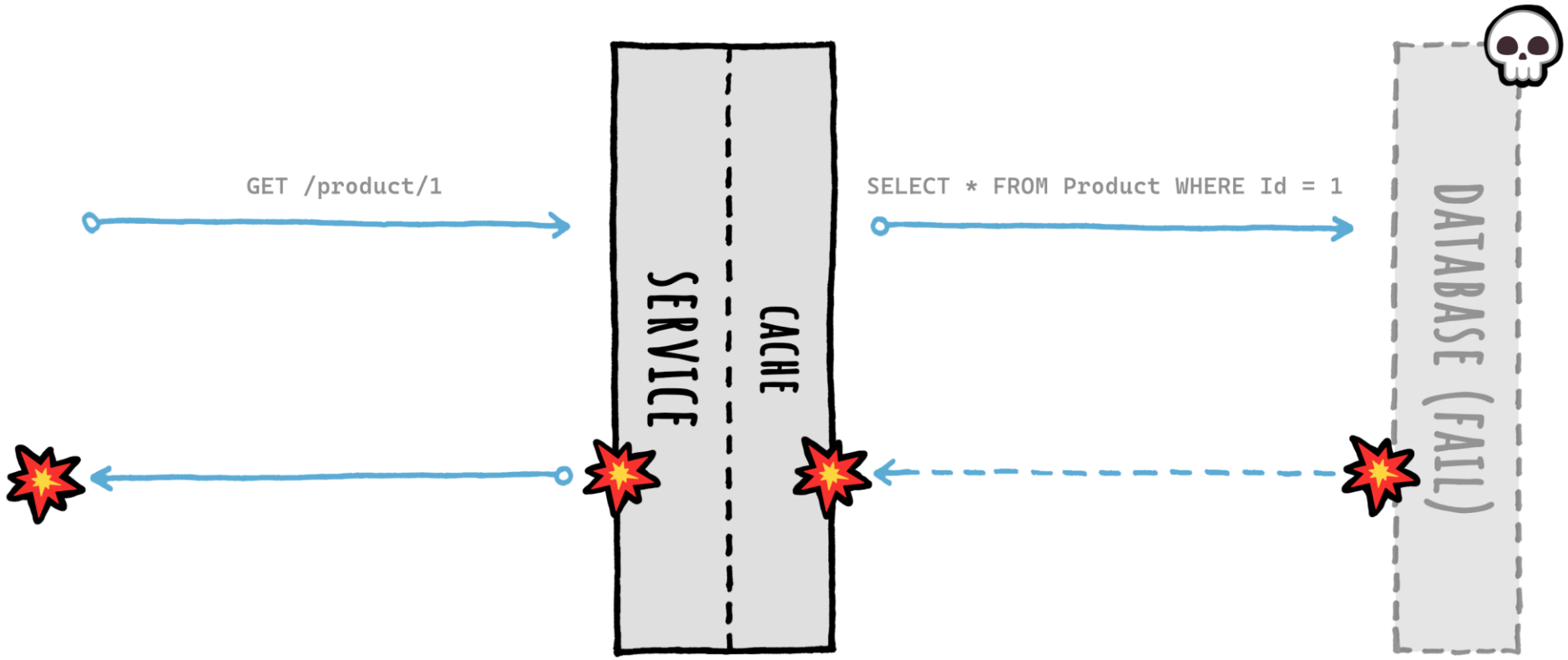
Fail-Safe

If we cached something for, say, **10 min** is it a problem to use it a **bit more** in case the database is **unavailable**?

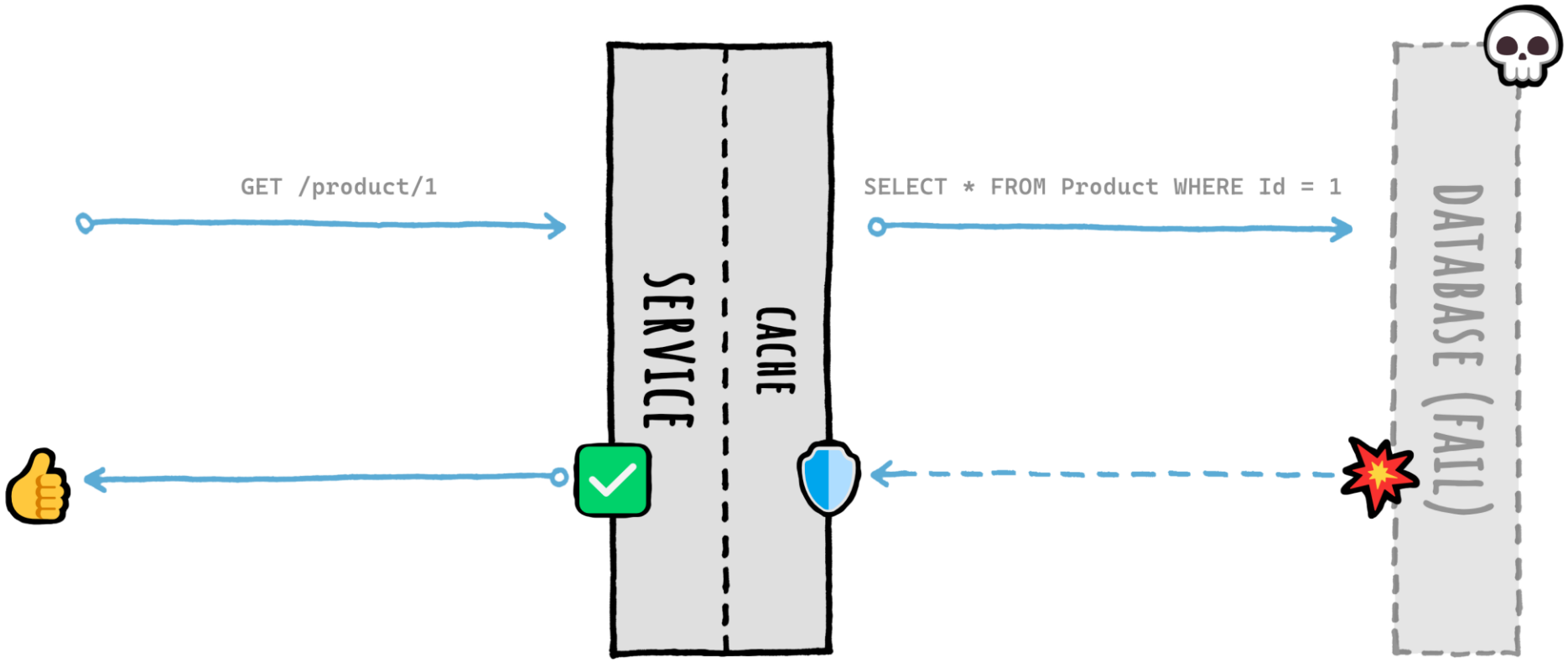
In FusionCache, the **Fail-Safe** mechanism allows us to do just that.

It's like a «second chance» for when « 💩 hit the fan».

Fail-Safe: Without



Fail-Safe: With



Fail-Safe

Let's say we have a `Duration` of **5 sec**:

- **no Fail-Safe:** after 5 sec the entry is **expired**, therefore **evicted** from the cache
- **with Fail-Safe:** after 5 sec the entry is **considered expired**, but **not evicted** from the cache

With Fail-Safe the `Duration` becomes a virtual/logical one.

In both cases after 5 sec there will be a **refresh**, but with Fail-Safe the entry will still be **available** as a **fallback**.

And in case of problems the expired entry is **temporarily** re-saved in the cache, for a bit more (all configurable).

Fail-Safe

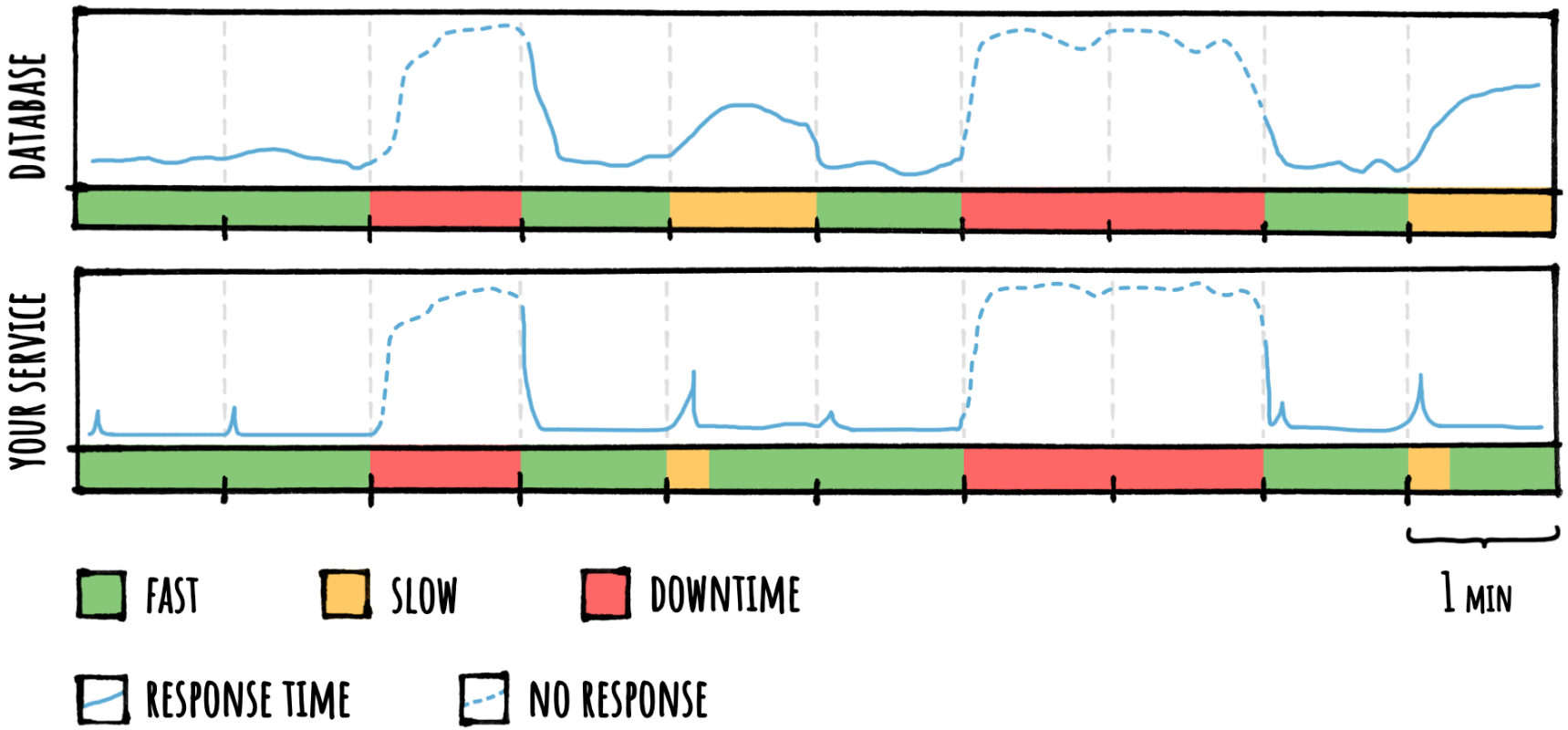
Here's how to use it:

```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // FAIL-SAFE
    .SetFailSafe(true, TimeSpan.FromHours(24), TimeSpan.FromSeconds(30))
);
```

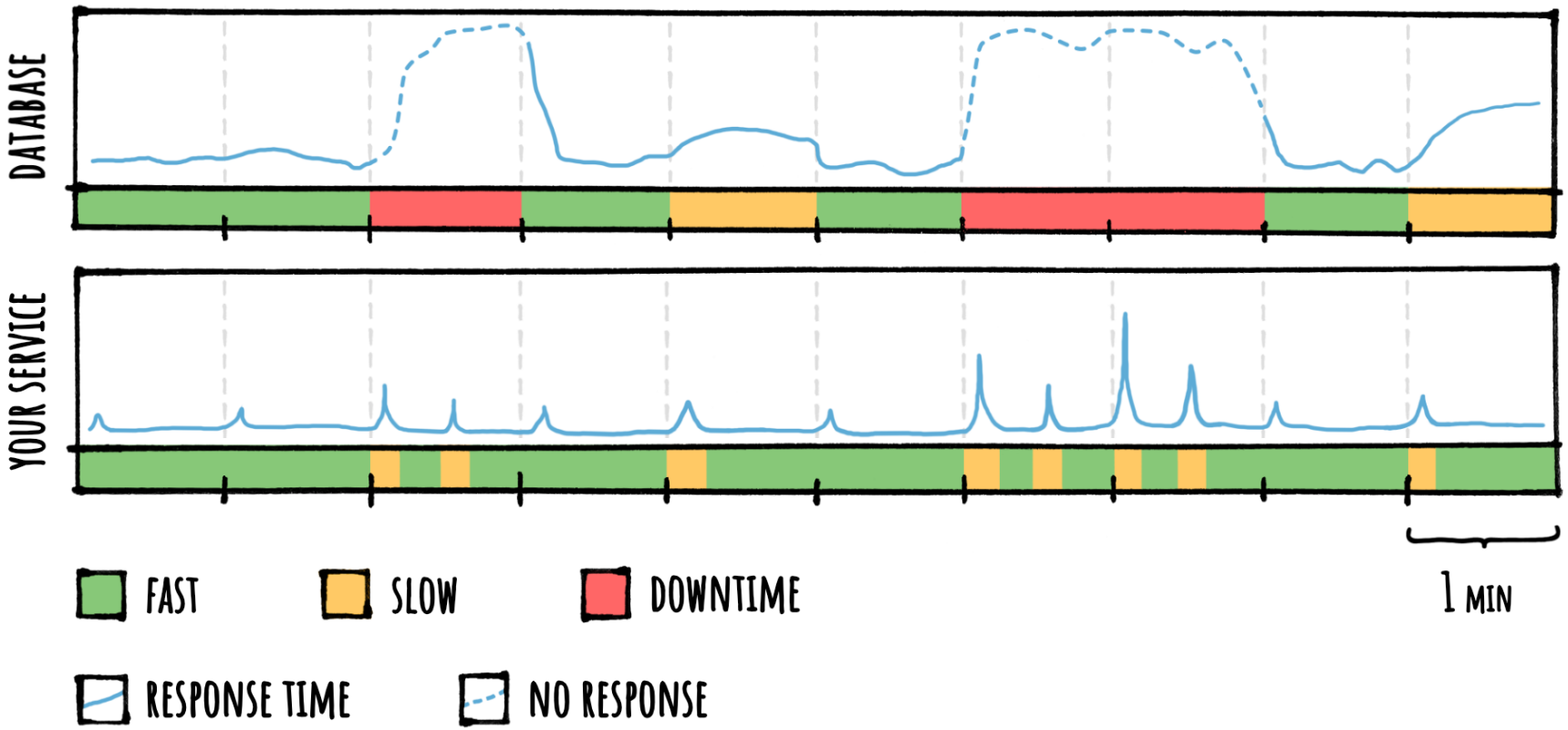
Fail-Safe: Without

Your service will **reflect** the database errors:



Fail-Safe: With

Your service will be **shielded** from database errors:



❤️ Fail-Safe: Sidney Lumet

Year: 1964

Directed by: Sidney Lumet
Cinematographer: Gerald Hirschfeld

Cast:

- Edward Binns
- Walter Matthau
- Henry Fonda
- Dan O'Herlihy
- Fritz Weaver
- Janet Ward
- Frank Overton
- Dana Elcar

Database Slowdowns

🐌 Database Slowdowns

Sometimes the database is not completely **down**, is just **slow**.

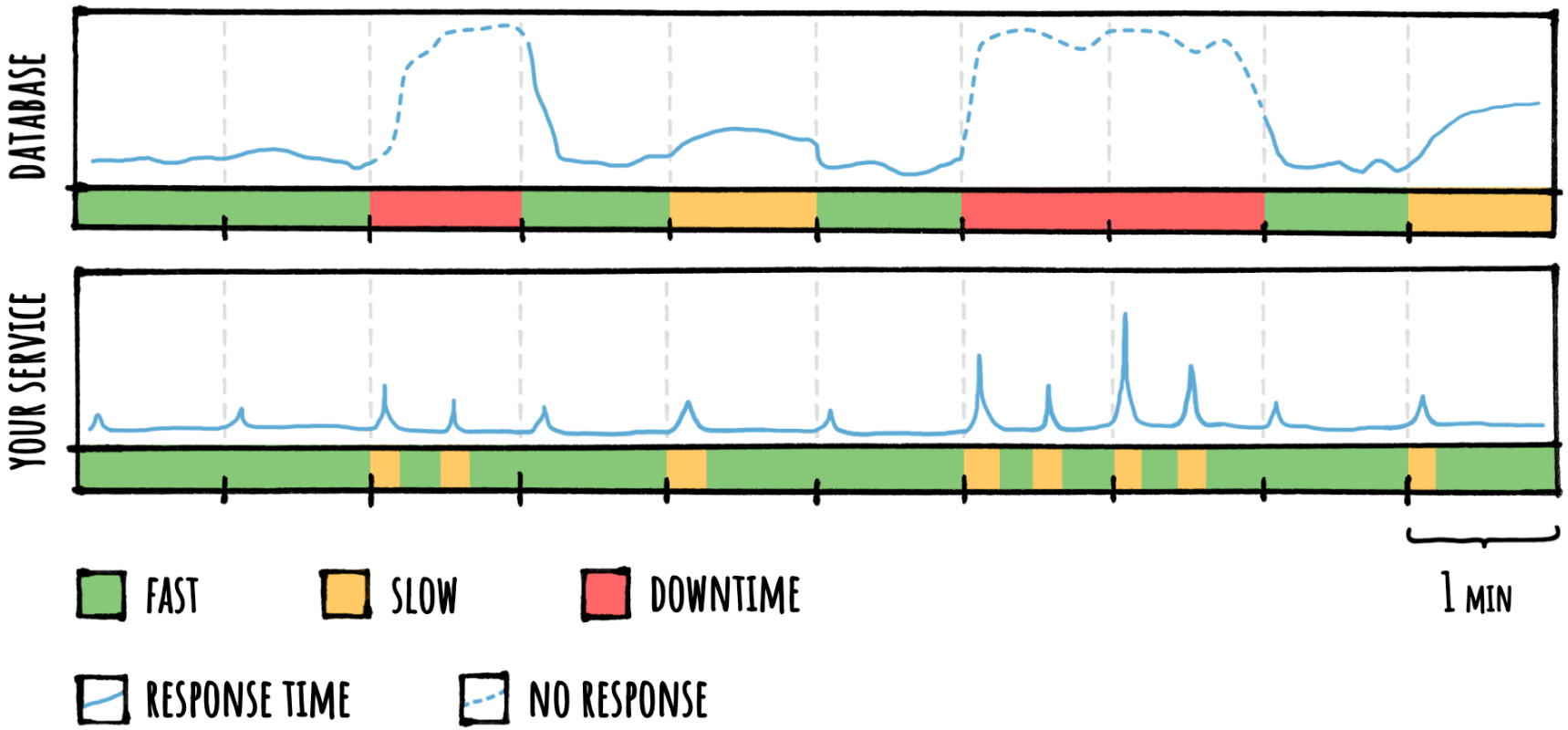
That can happen for various reasons:

- **high workload:** the database is currently overloaded
- **bad query:** sometimes we don't write the most spectacular queries
- **missing index:** we forgot an index
- **network issues:** congestion, topology change
- etc

What happens then?

🐌 Database Slowdowns

When the **database** is slow, **your service** is slow:



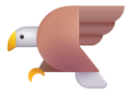
🐌 Database Slowdowns

But if the data is already **expired**, and the database is **slow**, what can we do?

Maybe... act **proactively**?

Enter: **Eager Refresh**.

Eager Refresh (FusionCache)



Eager Refresh

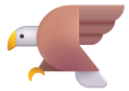
With **Eager Refresh** we can start refreshing a value **before** it expires, but only **after** a certain **threshold**.

The threshold is a **percentage** of the `Duration`, expressed as a value between 0.0 and 1.0, where 0.5 is 50%, 0.75 = 75%, etc.

When a request is made to the cache **after** the threshold, Eager Refresh will be **activated**.

When activated:

- the cached value is **immediately** returned (since it's **still valid**)
- the factory is eagerly executed in the **background** (in a **non-blocking** way)



Eager Refresh

Here's how to enable it:

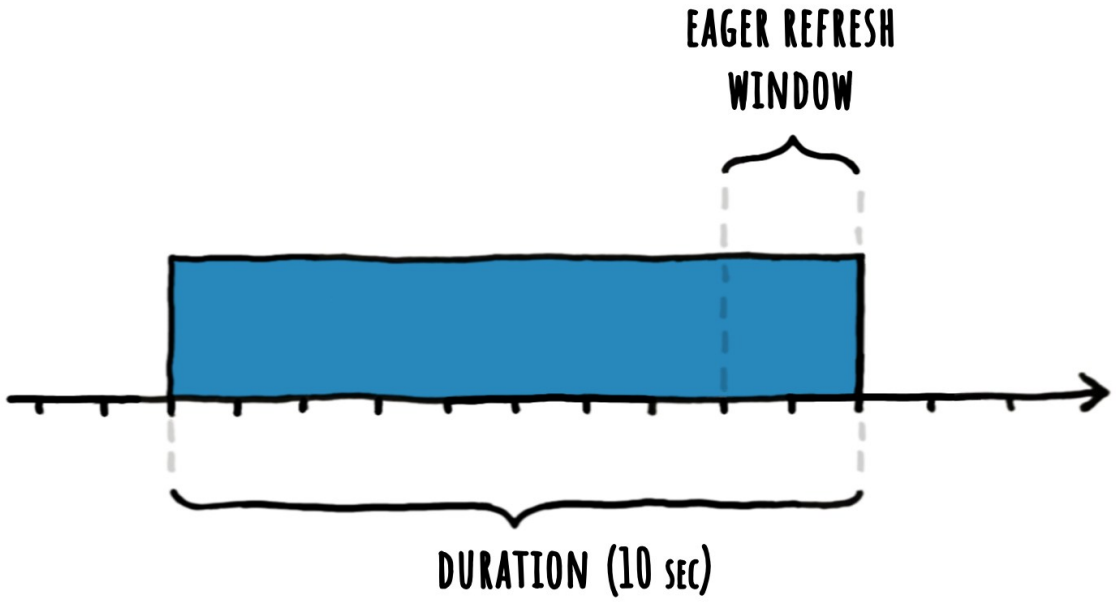
```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // EAGER REFRESH
    .SetEagerRefresh(0.9f)
);
```

Eager Refresh

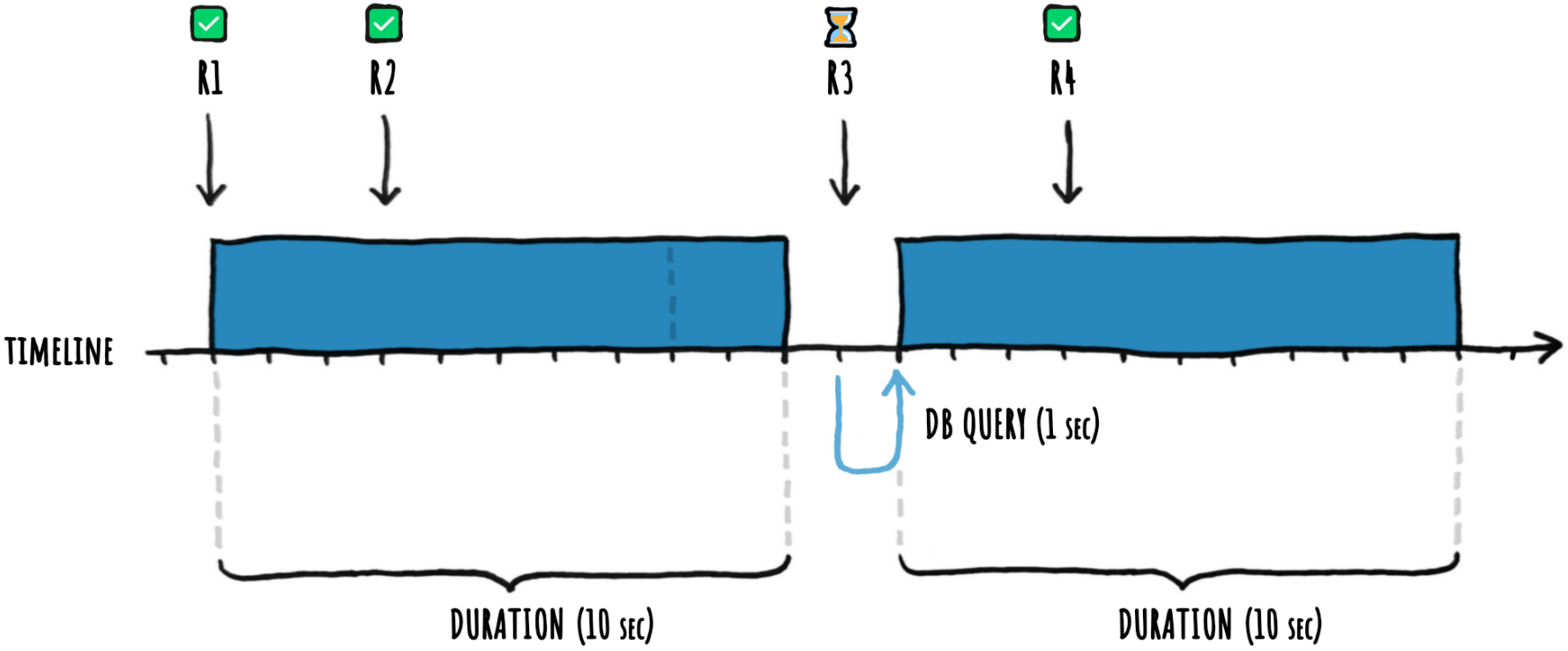
But Eager Refresh can help us **only** if there's a request:

- **after** the threshold
- but **before** the expiration



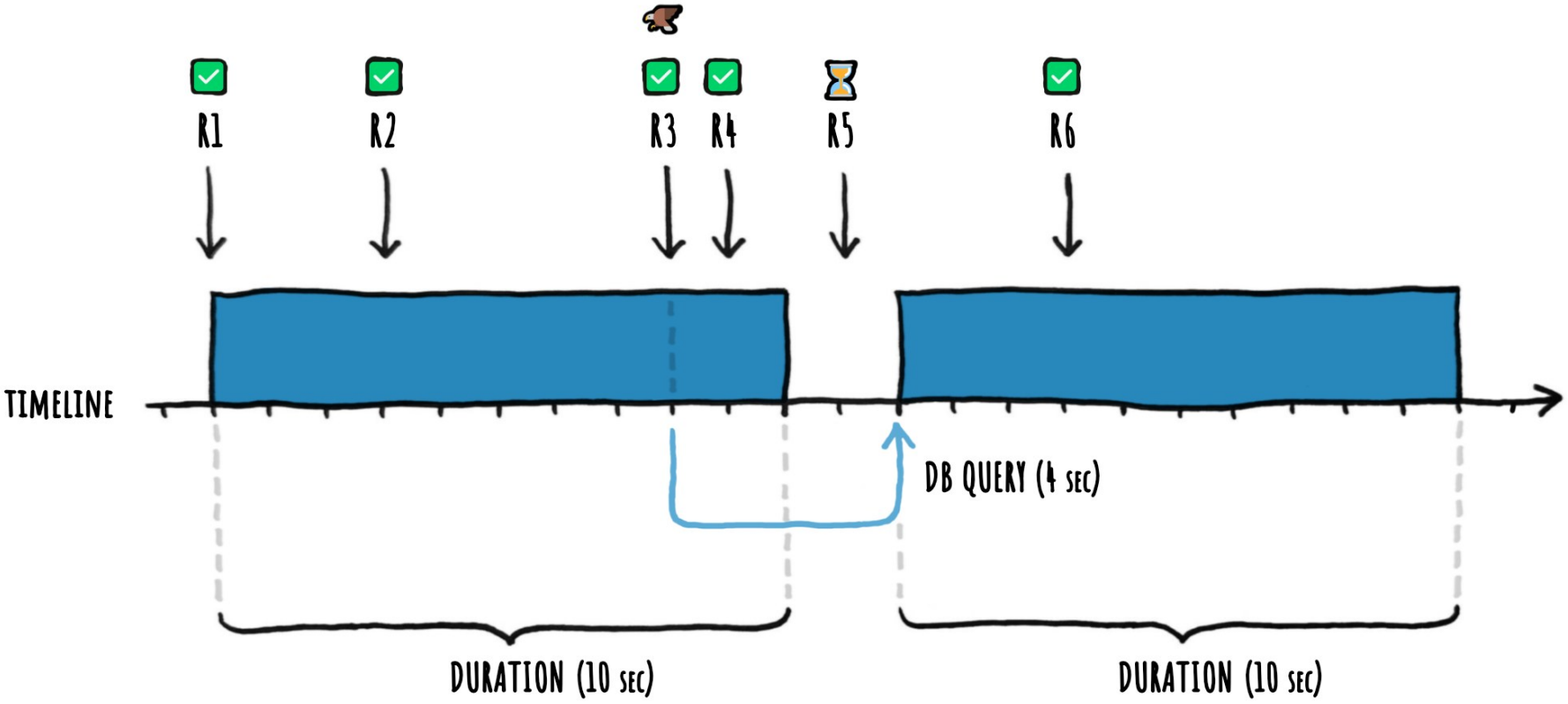
Eager Refresh

But what happens if there are no requests **inside** of that window?



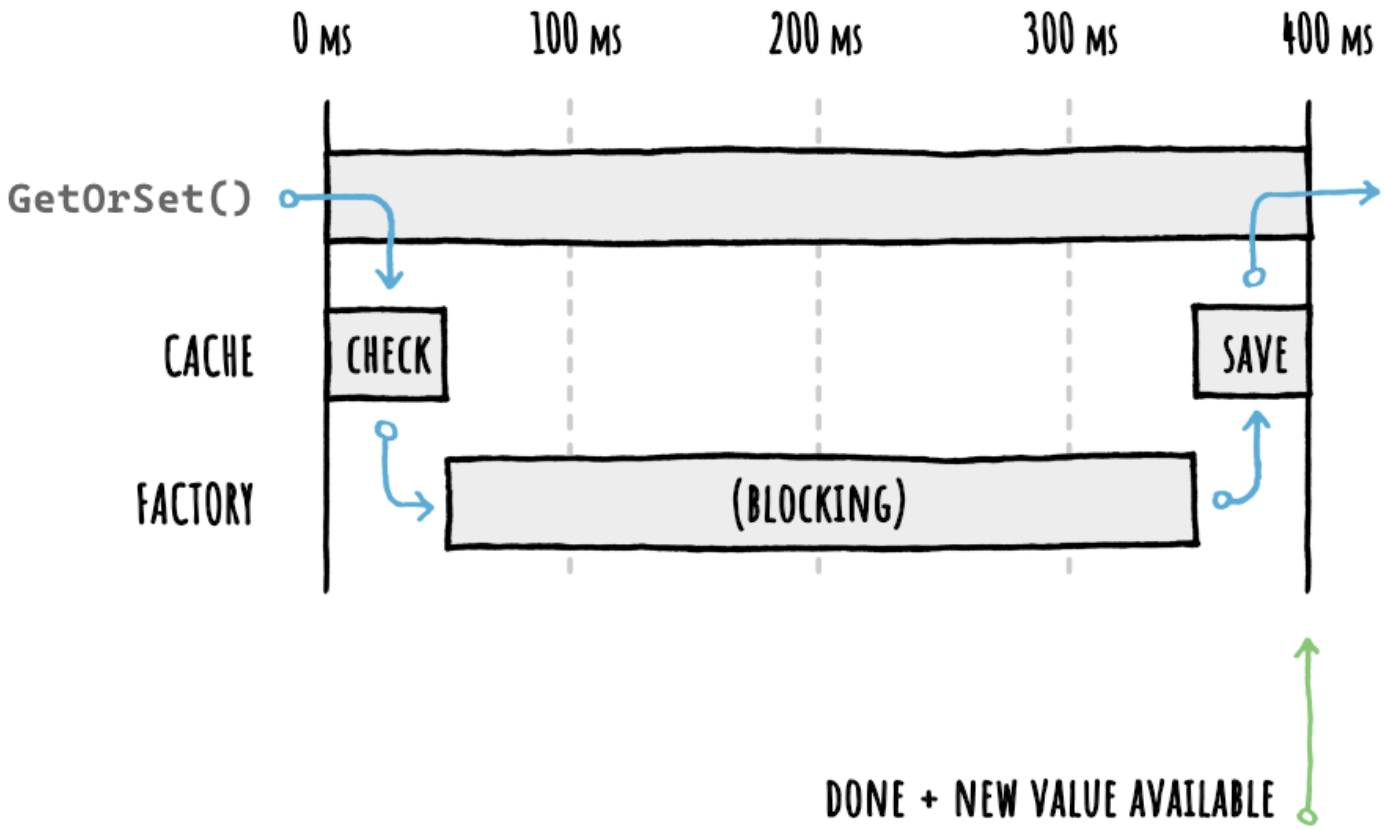
Eager Refresh

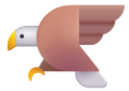
Or what if Eager Refresh has been triggered on time, but the factory is **particularly slow**?



Eager Refresh

The problem is that the factory execution on a cache miss is a **blocking** operation:





Eager Refresh

But here's the thing: if the data in the cache is **expired**, the factory execution is a **blocking** operation, and the database is **slow**, is there really something we can do?

(I think you can see where I'm going)

Enter: **Factory Timeouts**.

Factory Timeouts (FusionCache)

Factory Timeouts

If we cached something for, say, **5 min** is it a problem to use it a bit more in case the database is **slow**?

Sounds familiar, right? Yep, like Fail-Safe.

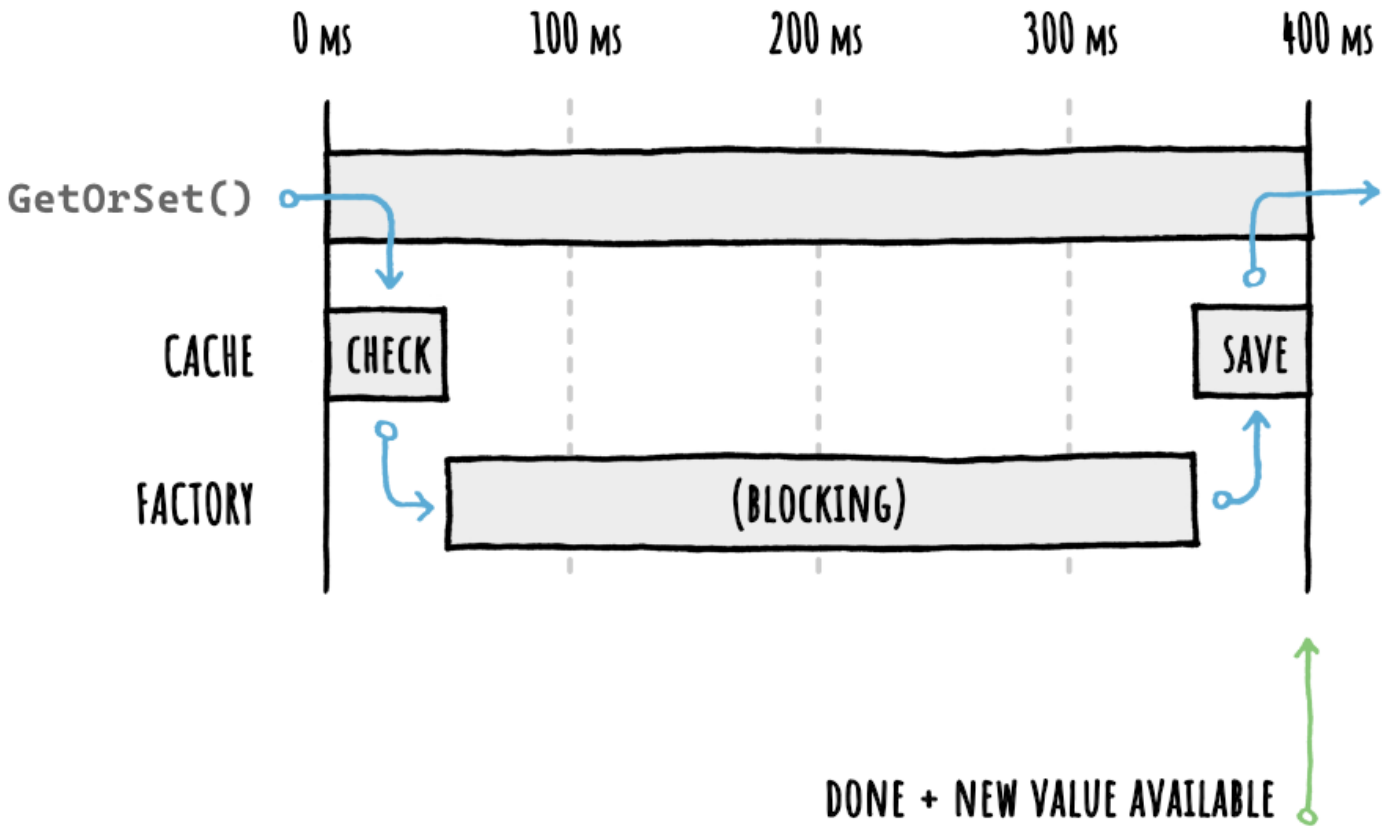
So the idea is that we can set a special **timeout** for the factory execution.

If hit, it will:

- **use** the **stale** value, available thanks to Fail-Safe
- **complete** the factory in the **background**

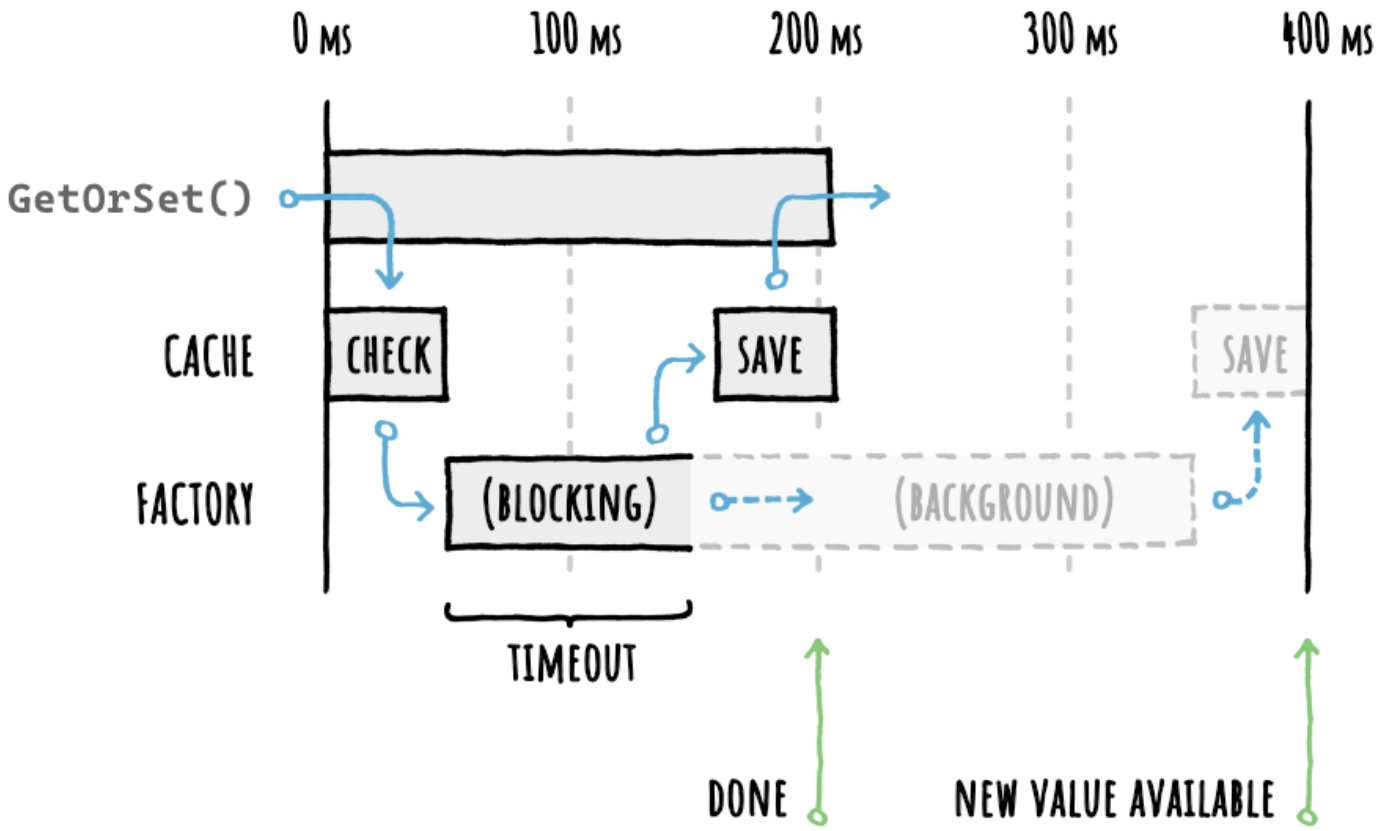
Factory Timeouts: Without

Before:



Factory Timeouts: With

After:



Factory Timeouts

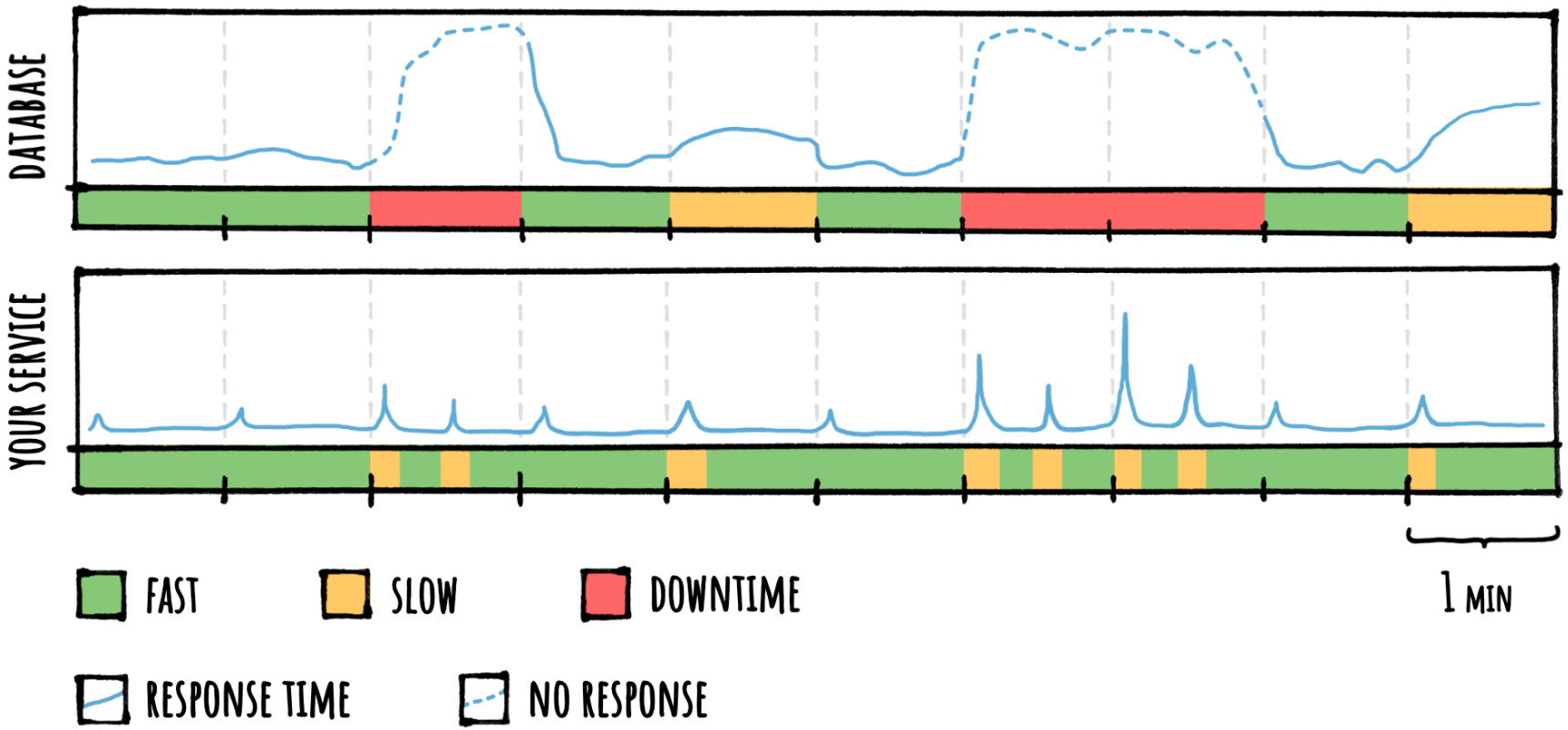
Here's how to use it:

```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // FACTORY TIMEOUTS
    .SetFactoryTimeouts(TimeSpan.FromMilliseconds(100), TimeSpan.FromSeconds(1))
);
```

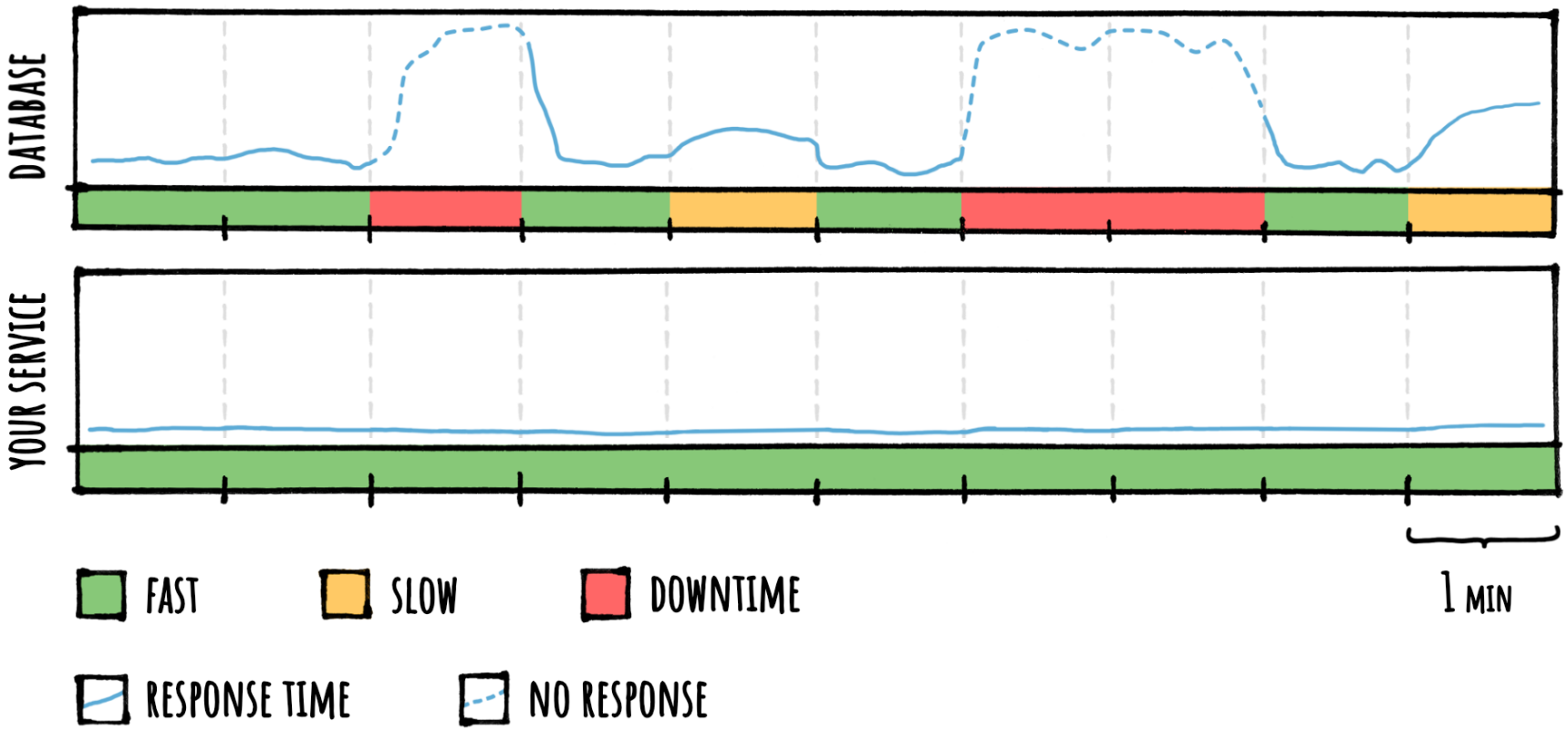
Eager Refresh + Factory Timeouts: Without

In the end, we can go from this:



⏱️ Eager Refresh + Factory Timeouts: With

To this:



More Problems

🥶 Cold Starts

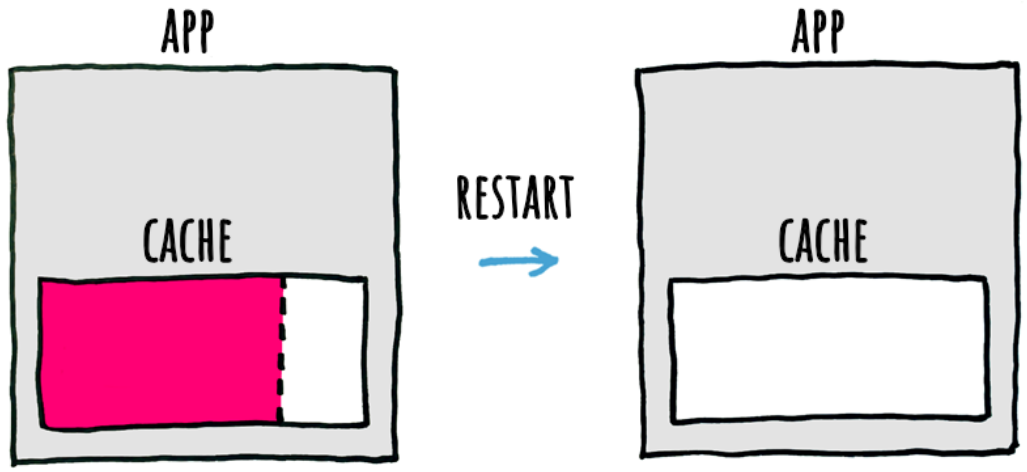
Say you are using a **memory cache** or a **hybrid cache** with **only L1**.

What happens when the app **restarts**?

Remember: a memory cache (or an L1) is just an **in-memory dictionary**.

🥶 Cold Starts

Well, this happens:



The memory cache is now **empty**.

And it needs to be **re-populated** again, from the **database** (more queries).

Horizontal Scalability

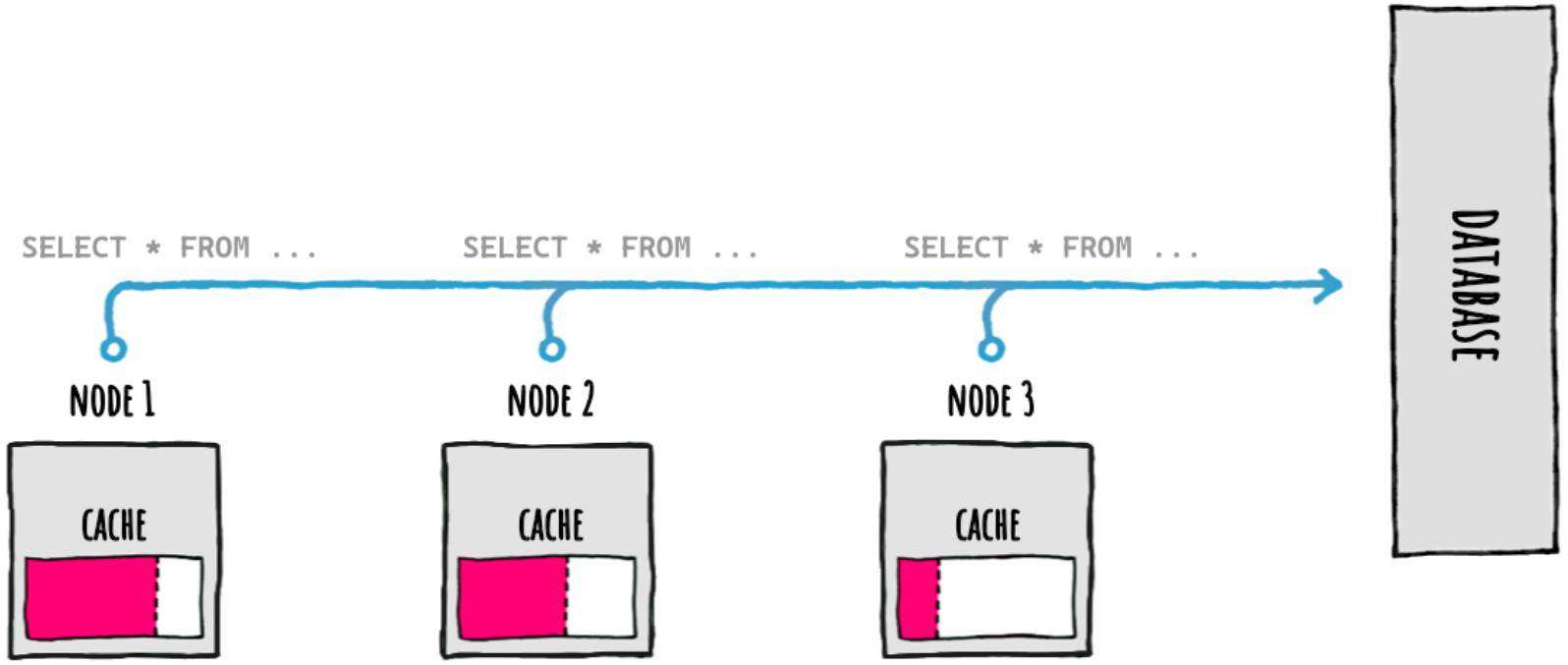
Or, what happens with:

- **multiple nodes:** same app on multiple nodes
- **multiple apps:** different apps working on the same data

Again, remember: a memory cache (or an L1) is just an **in-memory dictionary**.

Horizontal Scalability

Well, this happens:



Each instance/node has **its own** memory cache.

And each will be **populated** from the **database**, with no sharing (more queries).

Distributed Cache?

✈️ Distributed Cache?

We can **share** cached data between nodes/apps, thanks to a **distributed cache**.

But... if we were working **directly** with a **memory cache**, it means changing this:

```
var product = cache.GetOrCreate<Product>(
    $"product:{id}",
    // WARNING: NO STAMPEDE PROTECTION
    _ => GetProductFromDb(id),
    options
);
```

 Distributed Cache?

... to this (**everywhere** 😳):

```
Product product;
var payload = distributedCache.Get($"product:{id}");
if (payload is not null)
{
    product = JsonSerializer.Deserialize<Product>(payload);
}
else
{
    // WARNING: NO STAMPEDE PROTECTION
    product = GetProductFromDb(id);
    payload = JsonSerializer.Serialize<Product>(product);
    distributedCache.Set($"product:{id}", payload);
}
```

Also, extra costs (network, serialization, etc).

L1+L2

🔀 L1+L2

Instead, with a **hybrid cache**, we just **enable L2** during setup:

```
services.AddFusionCache()
    .WithSerializer(
        new FusionCacheSystemTextJsonSerializer()
    )
    .WithDistributedCache(
        new RedisCache(new RedisCacheOptions { ... })
    )
;
```

ℹ️ **NOTE:** again, any `IDistributedCache` implementation will work.

🔀 L1+L2

All the dance between the **2 levels**, L1 (memory) and L2 (distributed) is taken care of by the hybrid cache, **automatically**.

For example:

- a cache miss on L1 followed by a cache hit on L2 will **automatically** copy the value on L1, so the **next request** will be a cache hit on L1
- an **update** to the cache is **automatically propagated** to L1 and L2
- etc

We can just keep using “a cache”, that’s it.

Cache Coherence

😨 Cache Coherence

When in L1+L2 setup, what happens when there's an **update** to a cache entry?

The hybrid cache writes on both **L1+L2**, but... only on the L1 where **the update** happened.

So, if other nodes already have that entry cached (in L1, memory), what happens?

We would need to wait their **expiration**.
And in that time window the cache, as a whole, becomes **incoherent**.

And this is **bad**.

So? What can we do?

Backplane (FusionCache)

📢 Backplane

We can simply use a **backplane**, which is like a lightweight message bus so each node can communicate with the others.

Like this (again, only during setup):

```
services.AddFusionCache()
    .WithBackplane(
        new RedisBackplane(new RedisBackplaneOptions { ... })
    )
;
```

Done 🥳

Microsoft HybridCache

HybridCache

You probably already heard about **HybridCache**, right?

Well, it’s not necessarily what you think it is.

> You keep using that word.
> I do not think it means
> what you think it means.
>
> Íñigo Montoya

HybridCache

Early 2024, Microsoft announced their own multi-level/hybrid cache:

Let's see:

- can be **L1** (`IMemoryCache`) or **L1+L2** (`IDistributedCache`)
- unified serialization interface, to work with `byte[]`
- global options + entry options, with `DefaultEntryOptions`

It transparently handles:

- one or two **levels**
- one or more **nodes**... kind of (*)

FusionCache design validated 🥳

HybridCache

Features:

- L1 or L1+L2 (*)
- Cache Stampede protection (*)
- Tagging (*)
- Immutability
- targets .NET Standard 2.0 (runs everywhere, old + new)

HybridCache: Abstraction

But most importantly, it's both:

- a **shared abstraction**
- a **default implementation**, by Microsoft

In fact:

- **abstraction:** `public abstract class HybridCache`, part of .NET 9
- **implementation:** `internal class DefaultHybridCache`, separate package

HybridCache: Abstraction

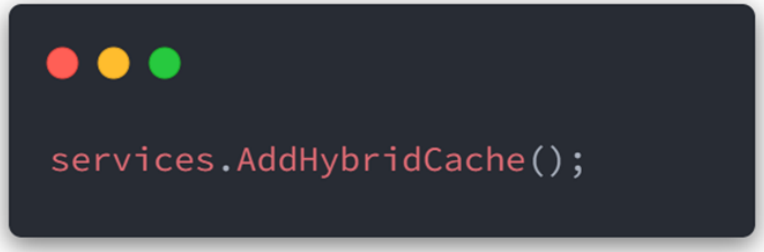
So HybridCache it's not just a Microsoft-provided **implementation**: it's first and foremost an **abstraction**.

Think of the `HybridCache` abstract class as the `IDistributedCache` interface.

Meaning: an **abstraction** for generic hybrid caches.

HybridCache

How to **register** it (after installing the package):



HybridCache

How to **use** it:

```
public class ProductController : Controller
{
    HybridCache _cache;

    public ProductController(HybridCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> Get(int id)
    {
        return await _cache.GetOrCreateAsync<Product>(
            $"product:{id}",
            async _ => GetProduct(id),
            options
        );
    }
}
```

Microsoft + OSS

I shared with the team suggestions, ideas, gotchas, etc and they **listened**: awesome.

I think the HybridCache effort is a great example of what it *may* look like when Microsoft and the OSS community have a constructive **dialogue**.

Thanks **Marc Gravell** & team!

HybridCache Limitations & Issues (mid 2025)

HybridCache Limitations & Issues

Since the current Microsoft implementation is their very **first version**, it still has some limitations and issues.

Note that most of them are **not** related to the **abstraction**, but **only** to their current **implementation**.

Some are minor, some are more severe: it's important to know them.

Let's look at some of them.

HybridCache: no new()

As we saw the concrete class is **internal** and therefore it's not possible to directly create an instance of it via `new()`.

There's a builder, but that's also **internal**.

This means relying only on the **DI** approach: maybe good, but important to know.

HybridCache: no DI control

About the DI approach, we **cannot** specify what to do with:

- L1 (memory level)
- L2 (distributed level)

Both picked up automatically, no control over it.

For example:

- if an `IDistributedCache` is registered, it **will** be used
- if an `IDistributedCache` is not registered, **nothing** can be used

HybridCache: single instance

Again, about DI: we also **cannot** register multiple named caches.

This means every consuming code in our app will share the same instance: we should be careful about potential **cache key collisions**.

It also means **no different configurations**, since there's only one instance.

HybridCache: async only

The API surface area is **async only**.

This means we can't use HybridCache in non-async call sites, unless we do really bad things like `.Result` or `.GetAwaiter().GetResult()` which, I mean.

HybridCache: no get methods

There are no read-only methods, so it's not possible to just "get a value".

At first it may look like it would be possible to create a custom extension method that simulates it by calling `GetOrCreateAsync()` with some specific entry options:

https://github.com/dotnet/extensions/issues/5688#issuecomment-2692247434

Problems are:

- stampede protection is **non-deterministic** (see next)
- in a **L1+L2** setup it would **not work** correctly

HybridCache: non-deterministic

This is pretty subtle.

When calling `GetOrCreateAsync(key, factory)`, the expectation is:

- if the value **is** in the cache (cache hit) just **return it**
- if the value **is not** in the cache (cache miss) **execute the factory**

But: currently, a factory may **not run** even when a value **is not** in the cache.

And there's no way to know it.

Basically, the behavior is **non-deterministic** (on a cache miss).

HybridCache: non-deterministic

For example, to simulate a get-only method we can do this:

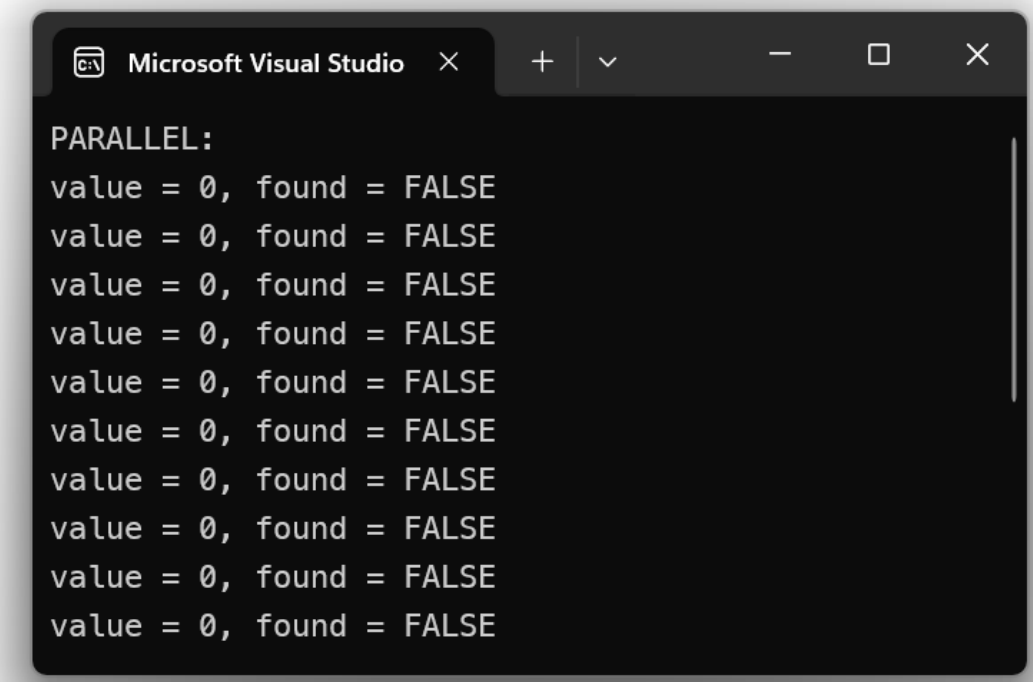
- declare a variable `bool found = true`
- call `GetOrCreateAsync<T>()`
- pass a factory that, when run sets `found` to `false` and returns `default(T)`
- pass some options to disable writing to L1 & L2

Basically, if the factory runs it means there was a cache miss.

Makes sense, right?

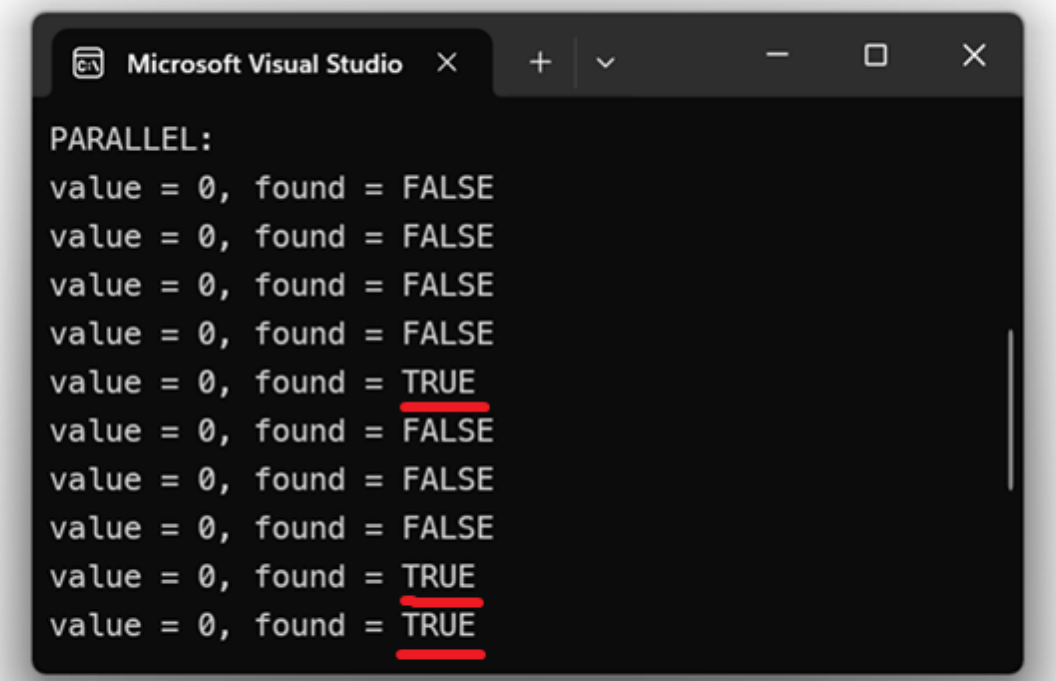
HybridCache: non-deterministic

When executed **in parallel** you would expect this:



HybridCache: non-deterministic

... but instead, you'll get this:



HybridCache: distributed issues

HybridCache currently doesn't work well with multi-nodes/instances.

Since it does not have something like the **Backplane** in FusionCache, it cannot notify the other nodes/instances: so, updates, removes, tagging, etc are not propagated.

In short: because of these limitations, the current Microsoft implementation of HybridCache is, in practice, not usable with more than 1 instance/node (imho).

ℹ️ **NOTE:** you may use a lower `LocalCacheExpiration` to mitigate.

HybridCache: A Solution?

But we said that:

- HybridCache is also an **abstraction**
- so, other **implementations** are possible
- and **FusionCache** is a hybrid cache

So...

FusionCache as HybridCache

FusionCache as HybridCache

FusionCache is its own, independent thing, and this doesn't change.

But there may be **value** in being able to work with a **shared** 1st party Microsoft abstraction.

So FusionCache can **also** be used as an implementation of the new `HybridCache` abstraction, while keeping the **extra features** of FusionCache.

And all thanks to a small **adapter**.

How?

FusionCache as HybridCache

Like this:

```
services.AddFusionCache()
  .AsHybridCache(); // MAGIC
```

No code changes required, anywhere.

FusionCache as HybridCache

> Super easy, barely an inconvenience.

Screenwriter Guy

FusionCache as HybridCache

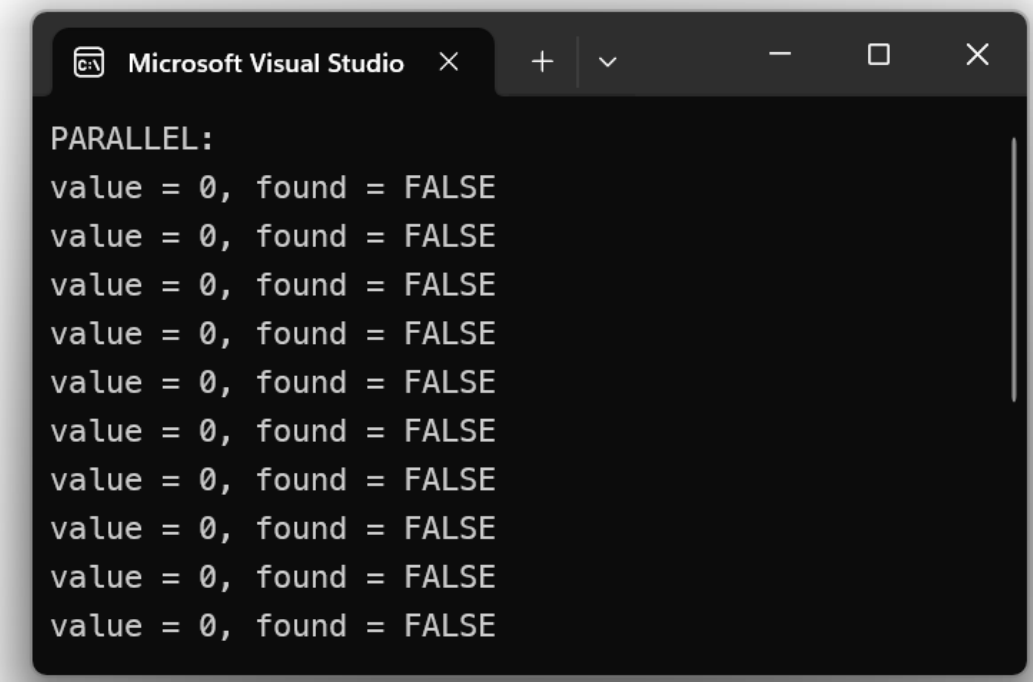
Remember those (*) ? Gone.

Using FusionCache as an implementation of HybridCache allows:

- **features:** Fail-Safe, Eager Refresh, Factory Timeouts, Auto-Recovery, etc
- **together:** use FusionCache and HybridCache at the same time
- **stampede protection:** deterministic + unified + sync/async
- **control:** total control over L1/L2 setup
- **backplane:** multi-nodes support with instant updates
- **no distributed issues:** bye bye out-of-sync nodes and cache incoherence
- **multiple caches:** we can use Named Caches + Keyed Services

FusionCache as HybridCache

About the non-deterministic problem, now it's like this:

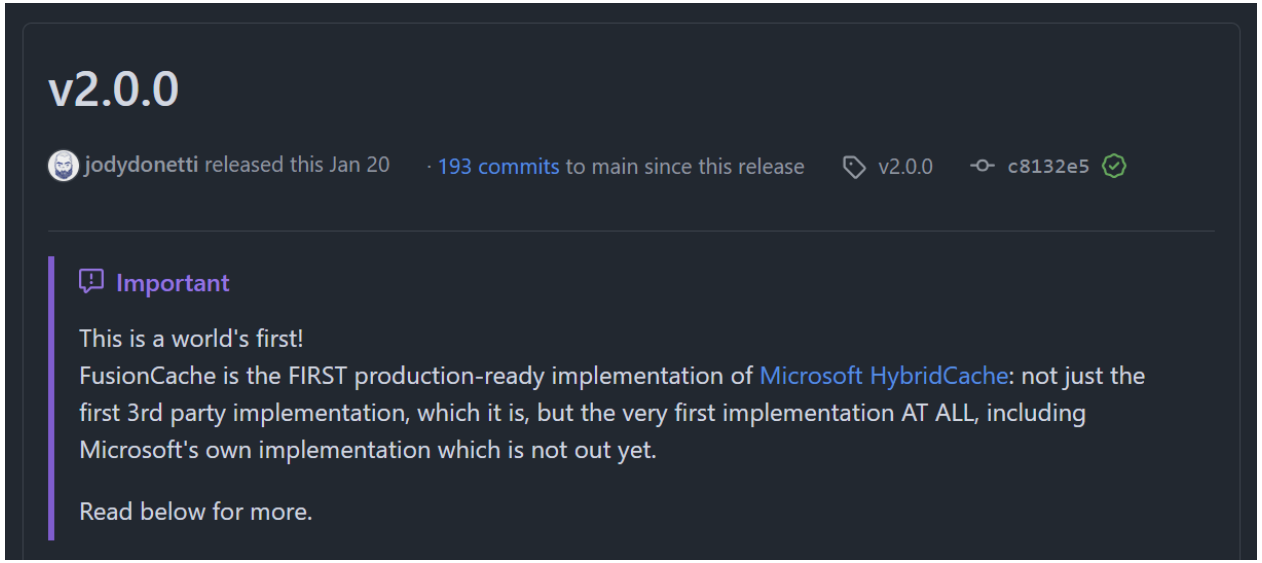


 FusionCache as HybridCache

The timeline:

- 🗓️ **Nov 2024:** HybridCache abstraction released (with .NET 9)
- 🗓️ **Jan 2025:** FusionCache v2 released (with HybridCache adapter)
- 🗓️ **Mar 2025:** HybridCache default implementation by Microsoft released

World's first 🥳



Let's wrap up

👋 Let's wrap up

We saw how **hybrid caches** are awesome.

They combine the best of both worlds:

- memory caches (L1)
- distributed caches (L2)

all in one **unified** and **transparent** API.

And they can have **more features**.

👋 Let's wrap up

To go **hybrid**, we can use:

- **FusionCache**, directly
- **HybridCache**, with the **default** implementation by Microsoft (*)
- **HybridCache**, with the **FusionCache** implementation

We have choices, let’s make the most out of them 🥲

Thanks!

github.com/jodydonetti

twitter.com/jodydonetti

linkedin.com/in/jody-donetti