Async Coding in Java

Why asynchronous calls make sense in a microservices context and a comparison of frameworks that help you do it

Petter Måhlén

Spotify®

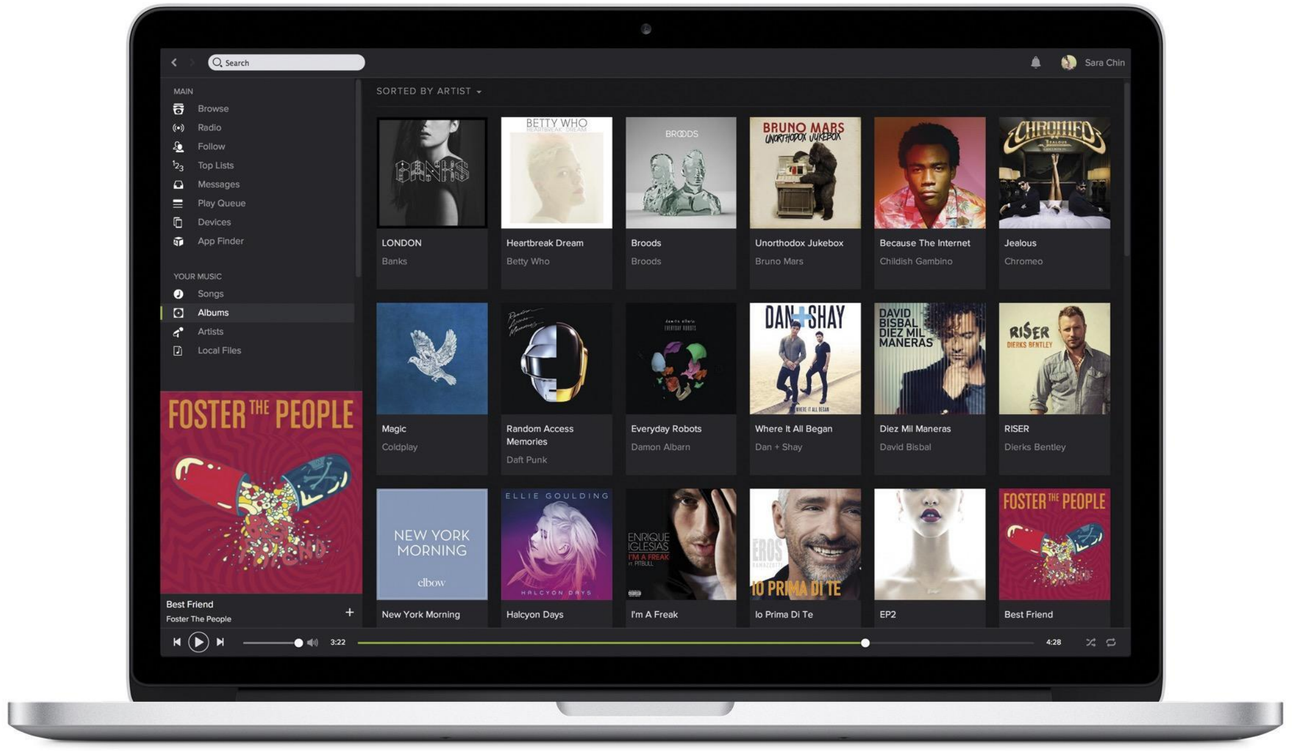
What is Spotify?

Music

Spotify brings you the right music for every moment – on your computer, your mobile, your tablet, your home entertainment system and more.

Numbers

- ❏ > 60M active users (last 30 days)
- ❏ > 1.5B playlists
- ❏ > 30M songs
- ❏ Available in 58 countries



Microservices, Async and Me

My background

- Currently building infrastructure at Spotify:
  - service discovery
  - routing infrastructure
  - service development framework
- About 6 years of microservices (4 Shopzilla, 2 Spotify)
- Similar sizes: 3-5 datacenters, a few thousand servers, more than 100 services

Async code

- Shopzilla sites: page rendering, ~10-40 service calls/page
- Shopzilla inventory: high-performance VoltDB calls
- Spotify view aggregation services, ~5-10 service calls/request

Topics covered

Why write asynchronous code?

Why not write asynchronous code?

I’m going to do it, how?

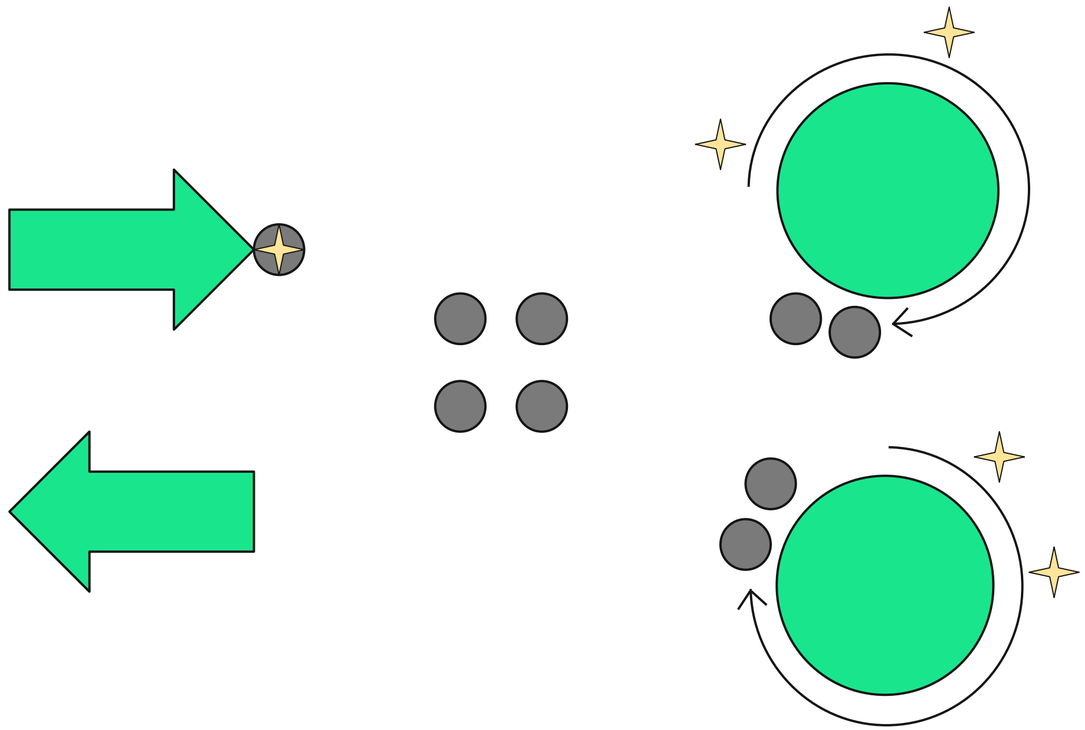
- ❏ Code examples
- ❏ Frameworks

Why Asynchronous?

Performance

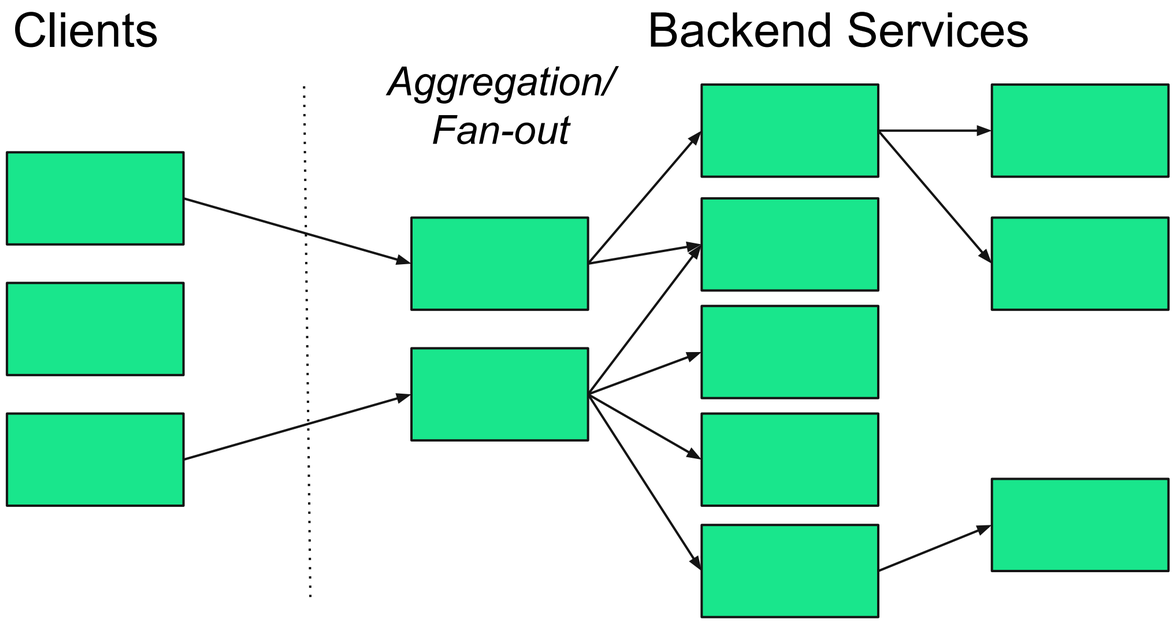
- ❏ Synchronous => throughput limited by thread/worker count
- ❏ Synchronous => resources used for the wrong things
- ❏ Asynchronous => latency improvements through parallelism
- ❏ Async means ‘less active waiting’

(A)synchronicity in a Restaurant



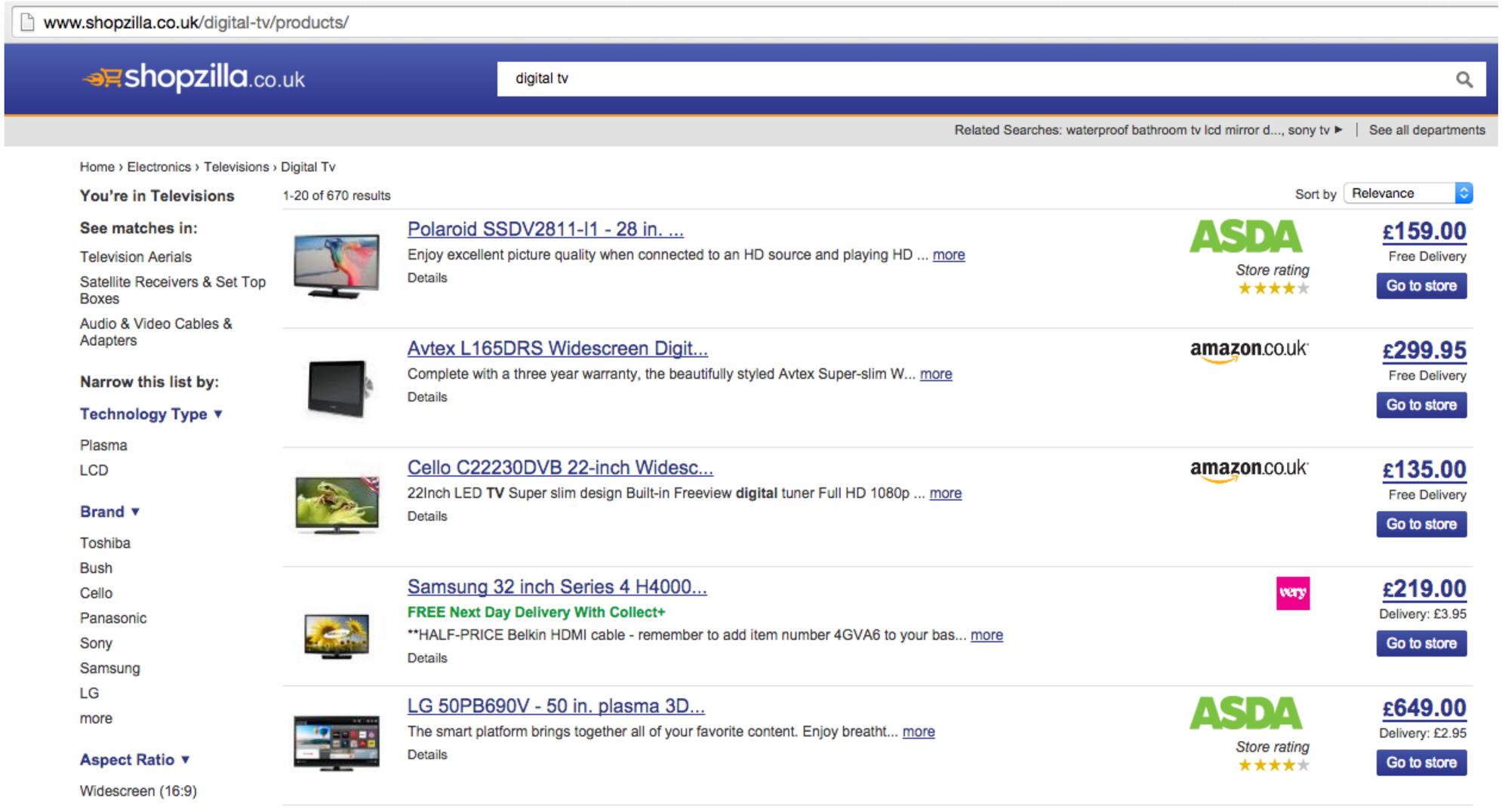
Async and Microservices

Typical microservices architecture:



Difference monolithic => microservices is latency; what used to be a method call is a remote call across the network

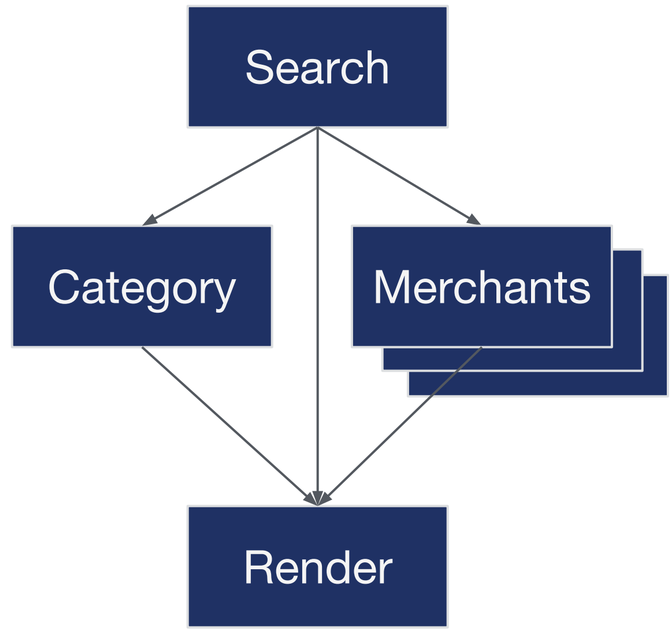
Async at Shopzilla



Async at Shopzilla

Shopzilla Async Framework

- ❏ Latency to start of render critical for revenue
- ❏ Framework put futures into a map, get actual results out
- ❏ Problems:
  - ❏ get = null - why?
  - ❏ get => block, mistakes delayed start of render
  - ❏ lack of visibility - what gets put into the map? Is it used?



Created PageFlow

- ❏ explicitly specifying call graph as data structure
- ❏ clunky syntax, tightly tied to Shopzilla infrastructure
- ❏ ‘accidentally’ moved concurrency into framework, great

Shopzilla Inventory

- ❏ Read/write logic for VoltDB databases
- ❏ In-memory, transactional, high-performance DB
- ❏ 100k+ writes/sec => async needed for performance
- ❏ Futures.transform() makes a sequence nested and harder to read

Created library for chaining invocations

- ❏ Simpler, less tied to infrastructure than PageFlow
- ❏ Just a chain, no fan-out/fan-in

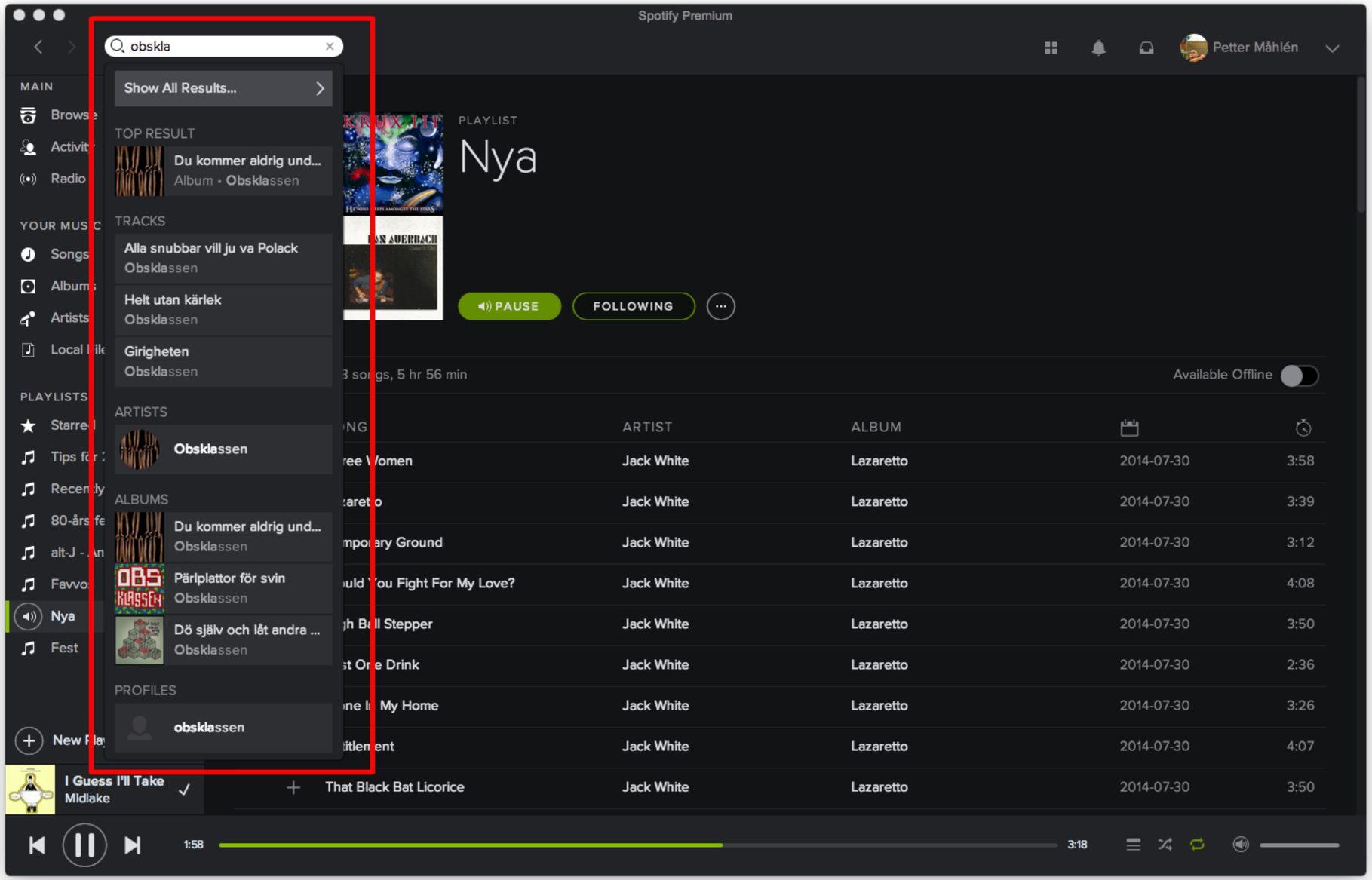
Async at Spotify

Thinner Clients

- Move logic from clients to backend
  - Easier, faster deployment
  - More mobile-friendly
- “View aggregation services”
  - Many downstream service invocations, more complex graphs
  - Use of ListenableFutures makes code complex

Created Trickle with Rouzbeh Delavari

- Open source (https://github.com/spotify/trickle/)
- Explicit graph like PageFlow
- Generic like the VoltDB library


Spotify Premium
obskla
Show All Results...
TOP RESULT
Du kommer aldrig und...
Album • Obsklassen
TRACKS
Alla snubbar vill ju va Polack
Obsklassen
Helt utan kärlek
Obsklassen
Girigheten
Obsklassen
ARTISTS
Obsklassen
ALBUMS
Du kommer aldrig und...
Obsklassen
Pärlplattor för svin
Obsklassen
Dö själv och låt andra ...
Obsklassen
PROFILES
obsklassen
MAIN
Browse
Activity
Radio
YOUR MUSIC
Songs
Albums
Artists
Local Files
PLAYLISTS
Starred
Tips for 2
Recently
80-årsfe
alt-J - An
Favvos
Nya
Fest
New Play
I Guess I'll Take
Midlake
Petter Måhlén
PLAYLIST
Nya
PAUSE
FOLLOWING
3 songs, 5 hr 56 min
Available Offline
ARTIST
ALBUM
Jack White
Lazaretto
2014-07-30
3:58
3:39
3:12
4:08
3:50
2:36
3:26
4:07
3:50
That Black Bat Licorice
1:58
3:18

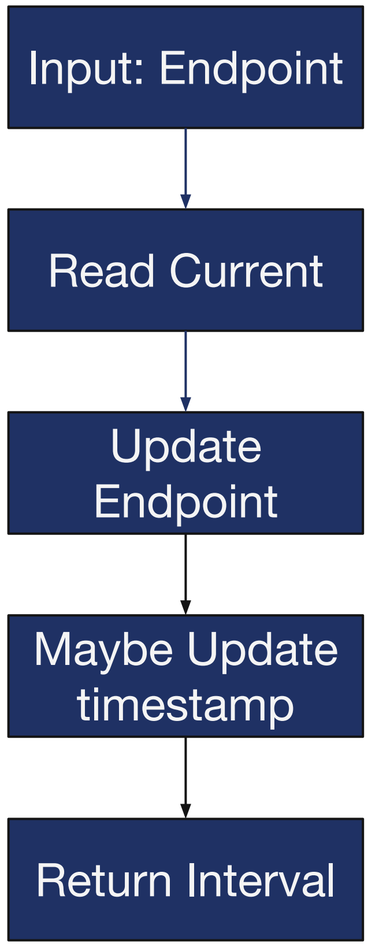
Why not asynchronous?

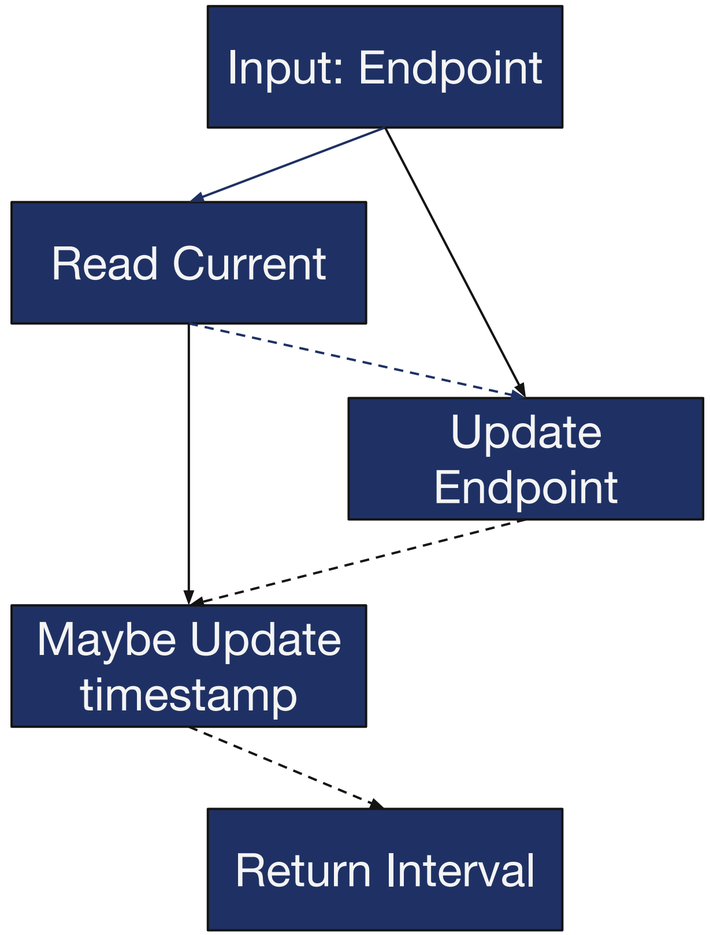
Because it’s harder to write, read, test and reason about

- ❏ Business logic obscured by concurrency management overhead
- ❏ Concurrency primitives can be invasive. What if somebody by accident does a get() instead of a transform?
- ❏ Typesafe fan-in hard (Futures.allAsList(), FuncN, BiConsumer/BiFunction, etc.)
- ❏ Testing - flakiness, exception handling, more execution paths
- ❏ Understanding errors/call stacks

- ❏ (Graceful degradation in case of errors)

Code Examples!





Subjective Comparison:
ListenableFutures

https://code.google.com/p/guava-libraries/wiki/ListenableFutureExplained

Pros

- ❏ low-level: not much magic
- ❏ (mostly) familiar concepts
- ❏ nice and small API
- ❏ good interoperability with other frameworks since futures are so common

Cons

- ❏ verbose
- ❏ concurrency management obscures business logic
- ❏ low-level: concurrency is in your face, easier to make mistakes
- ❏ fan-in is messy

Subjective Comparison:

RxJava

Rx = Reactive Extensions

http://reactivex.io/

Pros

- ❏ feature-rich, especially for streams of data
- ❏ separates concurrency from business logic
- ❏ easy to combine results, do fallbacks, etc.
- ❏ clean code

Cons

- ❏ unfamiliar concepts/high learning threshold
- ❏ large and clumsy API (cf #methods on Observable interface)
- ❏ “everything is a collection”

Subjective Comparison:
Trickle

https://github.com/spotify/trickle/

Pros

- ❏ separates concurrency from business logic
- ❏ nice error handling + reporting support
- ❏ developer-friendly API
- ❏ good interoperability with regular Futures/other frameworks

Cons

- ❏ weird to do graph wiring in data
- ❏ not in widespread use

Many subjective comparisons

Result of engineers at Spotify coding up a pretty small async graph

Technology	Get going	Focus on core	Cleanness
ListenableFutures	4.0	3.6	2.7
RxJava	2.8	3.7	3.1
Trickle	3.9	3.8	4.4

Let’s get more data: try it yourself at https://github.com/pettermahlen/async-shootout and fill in the form!

Choices, choices

There's more:

- ❏ Akka
  - ❏ actors
  - ❏ cool, but sort of all-or-nothing - greenfield only?
- ❏ CompletionStage in Java 8
  - ❏ allows chaining of asynchronous calls
  - ❏ fan-in is harder than Rx or Trickle
- ❏ Disruptor
  - ❏ Not just super-high-performance; allows constructing call graphs
  - ❏ also all-or-nothing, at least within single service

Picking your Framework

Consider:

- ❏ your migration path, if any
- ❏ how to integrate with third-party tools
- ❏ the learning curve
- ❏ the expected level of concurrency expertise of devs

… and above all, make sure you need it!

Questions?

Spotify®

```java
public class SynchronousHeartbeat {
  private final SynchronousStore store;
  private final long heartbeatIntervalMillis;

  public SynchronousHeartbeat(SynchronousStore store, long heartbeatIntervalMillis) {
    this.store = store;
    this.heartbeatIntervalMillis = heartbeatIntervalMillis;
  }

  public long heartbeat(final Endpoint endpoint) {
    // fetch what we currently know about the endpoint
    RegistryEntry previous = store.get(endpoint);

    // no matter what, flag it as known to be UP right now
    store.put(createEntry(endpoint, UP));

    // if it wasn't known to be up before, that's a change to the current state of things,
    // which means we need to update the last change timestamp.
    if (!isPresent(previous) || !isUp(previous)) {
      store.updateLastChangeTimestamp();
    }

    // let the caller know when he's next expected to be in touch
    return heartbeatIntervalMillis;
  }
}
```

```
public class ListenableFutureHeartbeat {
  private final Store store;
  private final long heartbeatIntervalMillis;

  public ListenableFutureHeartbeat(Store store, long heartbeatIntervalMillis) {
    this.store = store;
    this.heartbeatIntervalMillis = heartbeatIntervalMillis;
  }

  public ListenableFuture<Long> heartbeat(final Endpoint endpoint) {
    ListenableFuture<RegistryEntry> previousFuture = store.get(endpoint);

    return Futures.transform(
            previousFuture,
            (AsyncFunction<RegistryEntry, Long>) previous -> {
              final ListenableFuture<Boolean> upFuture = store.put(createEntry(endpoint, UP));

              ListenableFuture<Void> serialNumberFuture =
                      Futures.transform(upFuture,
                                        (AsyncFunction<Boolean, Void>) IGNORED -> {
                                          if (!isPresent(previous) || !isUp(previous)) {
                                            return store.updateLastChangeTimestamp();
                                          }
                                          return Futures.immediateFuture(null);
                                        });

              return Futures.transform(serialNumberFuture,
                                       (AsyncFunction<Void, Long>) IGNORED ->
                                               Futures.immediateFuture(heartbeatIntervalMillis));
            });
  }
}

// NOTE:
// - noise level
// - 'pretend' transforms
// - ignoring inputs to handle chaining
```

```scala
class ScalaHeartbeat(store: ScalaStore, heartbeatIntervalMillis: Long) {

  def createEntry(endpoint: Endpoint, state: State): RegistryEntry = ???
  def isPresent(entry: RegistryEntry): Boolean = ???
  def isUp(entry: RegistryEntry): Boolean = ???


  def heartbeat(endpoint: Endpoint): Future[Long] = {
    for {
      previous <- store.get(endpoint)
      _ <- store.put(createEntry(endpoint, UP))
      _ <- if (!isPresent(previous) || !isUp(previous))
        store.updateLastChangeTimestamp()
      else
        Future.successful(())
    } yield heartbeatIntervalMillis
  }
}

trait ScalaStore {
  def get(endpoint: Endpoint):Future[RegistryEntry] = ???
  def put(entry: RegistryEntry): Future[Boolean] = ???
  def updateLastChangeTimestamp():Future[Void] = ???
}
```

```java
public class RxHeartbeat {
  private final ObservableStore store;
  private final long heartbeatIntervalMillis;

  public RxHeartbeat(ObservableStore store, long heartbeatIntervalMillis) {
    this.store = store;
    this.heartbeatIntervalMillis = heartbeatIntervalMillis;
  }

  public Observable<Long> heartbeat(Endpoint endpoint) {
    Observable<RegistryEntry> previous = store.get(endpoint);
    Observable<Boolean> up = previous.flatMap(IGNORED -> store.put(createEntry(endpoint, UP)));

    return previous
            .zipWith(up, (RegistryEntry previousEntry, Boolean IGNORED) -> {
              if (!isPresent(previousEntry) || !isUp(previousEntry)) {
                return store.updateLastChangeTimestamp();
              }
              return null;
            })
            .map(IGNORED -> heartbeatIntervalMillis);
  }
}

// NOTE:
// - rx = reactive extensions
// - reduced noise level
// - 'functional' method names
// - ignoring inputs to handle chaining
// - method count on Observable => discoverability
```

```java
public class TrickleHeartbeat {
  private final Input<Endpoint> endpointInput;
  private final Graph<Long> heartbeatGraph;

  public TrickleHeartbeat(Store store, long heartbeatIntervalMillis) {
    endpointInput = Input.named("endpoint");

    Graph<RegistryEntry> previous = call(store::get).with(endpointInput);
    Graph<Boolean> up = call((Endpoint endpoint) -> store.put(createEntry(endpoint, UP)))
            .with(endpointInput)
            .after(previous);
    Graph<Void> timestamp = call(
            (RegistryEntry previousEntry) -> {
              if (!isPresent(previousEntry) || !isUp(previousEntry)) {
                return store.updateLastChangeTimestamp();
              }
              return immediateFuture(null);
            })
            .with(previous)
            .after(up);

    heartbeatGraph = call(() -> immediateFuture(heartbeatIntervalMillis)).after(timestamp);
  }

  public ListenableFuture<Long> heartbeat(final Endpoint endpoint) {
    return heartbeatGraph.bind(endpointInput, endpoint).run();
  }
}

// NOTE:
// - noisier than Rx
// - what's with all the 'Graph':s?
// - optional names
// - IDE help building graphs
```

Calling an asynchronous method

- exceptions
- timing

Testing tips

Some code

```
public ListenableFuture<Gherkin> serve() {
  ListenableFuture<Integer> count = counter.count();
  return Futures.transform(count, new Function<>() {
     public Gherkin apply(Integer count) {
       if (count == 0) {
         froobishes.delete();
       }
       return new Gherkin(count);
     }
  });
}
```

A Test

```
public void shouldDeleteFroobishesWhenCountIsZero()
    throws Exception {

  when(counter.count()).thenReturn(intFuture(0));
  service.serve().get(); // <--- terminate the future

  verify(froobishes).delete();
}
```