Fail-fast design principle

What is a bad service?

- does not work (low availability/uptime)
- does not know how to withstand faults (not fault tolerant)
- does not know how to quickly recover from failures (not resilient)
- does not always return accurate results (not reliable)
- loses data from time to time (not durable)
- does not know how to scale quickly (not elastic)
- returns unpredictable results (supports a weak consistency model)
- hard to maintain (does not follow operational excellence guidelines)
- poorly tested (low unit/functional/integration/performance test coverage)
- not secure (violates CIA triad rules)
- …
- slow (performance degrades sometimes)

we have covered many design principles that help deal with these problems

and more yet to come…

Fail-fast design principle

fail immediately and visibly when faults occur in the system

object initialization

```
public class SomeClass {
  private final String username; //immutable object
  public SomeClass(@NotNull String userName) {
    this.username = username;
  }
  …
}
```

precondition

```
public static double sqrt(double value) {
  Preconditions.checkArgument(value >= 0.0);
  …
}
```

configuration validation

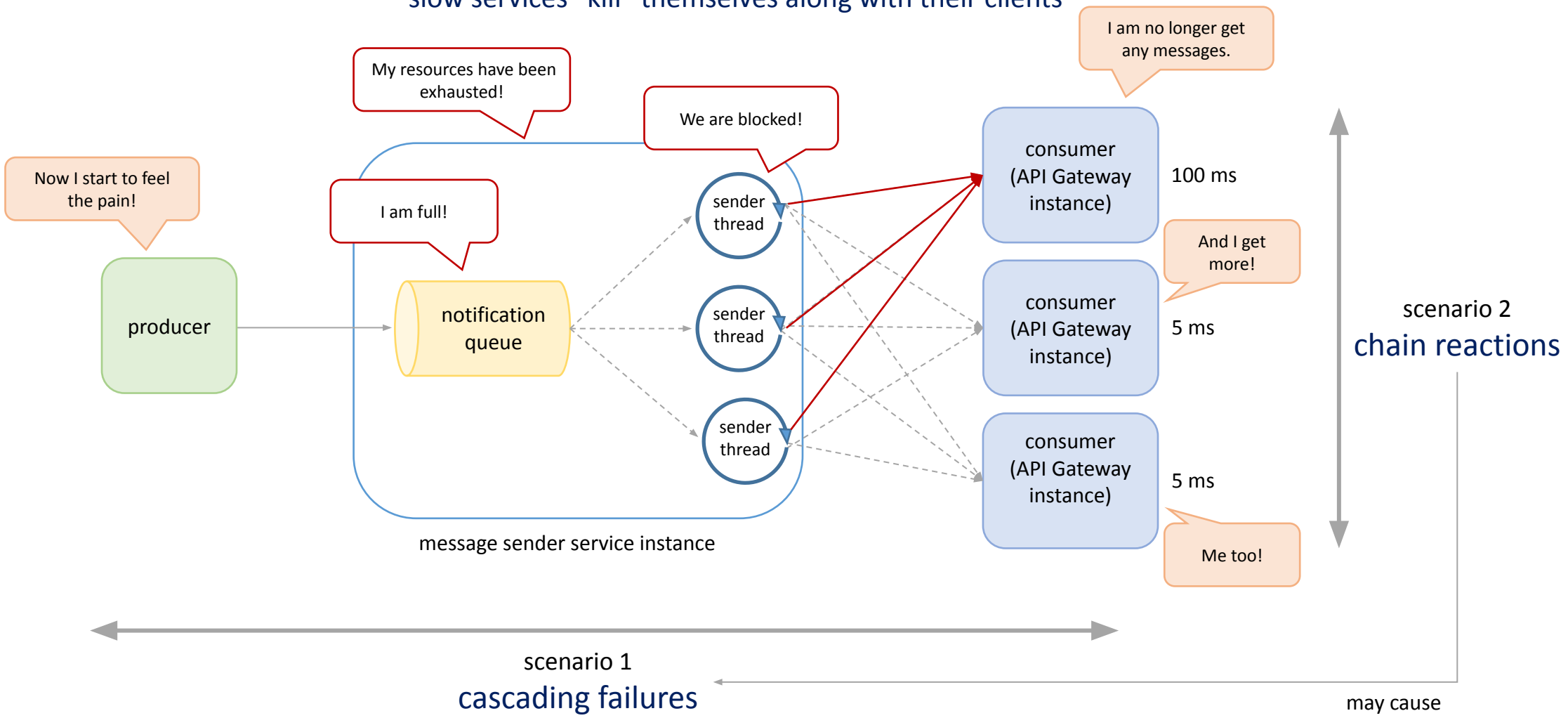
```
public int maxQueueSize() {
  String property = Config.getProperty("maxQueueSize");
  if (property == null) {
    throw new IllegalStateException("…");
  }
  …
}
```

request validation

```
private void validateRequestParameter(String param) {
  if (param == null) {
    throw new IllegalArgumentException("…");
  }
}
```

Fail-fast design principle

slow services “kill” themselves along with their clients



Fail-fast design principle

How to avoid?

cascading failures

clients need to protect themselves

- timeouts
  helps to minimize blocking time
- circuit breaker
  helps to identify and isolate a bad dependency
- health checks
  helps to identify and isolate a bad dependency
- bulkhead
  helps to minimize impact of a bad dependency

chain reactions

servers need to protect themselves

- load shedding
  to protect remaining servers from overload
- autoscaling
  to quickly add redundant capacity
- monitoring
  to quickly identify and replace failed servers
- chaos engineering
  to constantly test server failures
- bulkhead
  to minimize impact of a bad client