Distributed Systems I

Software Architecture

Brae Webb

March 20, 2023

Mathias Verras
@mathiasverraes

There are only two hard problems in distributed systems:
2. Exactly-once delivery
1. Guaranteed order of messages
2. Exactly-once delivery

Lecture Goal: Balance a healthy love-hate relationship with distributed systems

Going forward

Investigating architectures that are *distributed*.

Distributed Systems Series

Distributed I *Reliability* and *scalability* of *stateless* systems.

Distributed II *Complexities* of *stateful* systems.

Distributed III *Hard problems* in distributed systems.

What are the benefits?

- Improved *reliability*.
- Improved *scalability*.
- Improved *latency*.

Some systems are inherently distributed.

What are the drawbacks?

- Increased *complexity*.
- Increased *attack vector*.
- Increased *latency*.
- Introduce *consistency* problems.

We'll look at a few reasons that distributed systems are *fundamentally* quite challenging

§ *Fallacies*

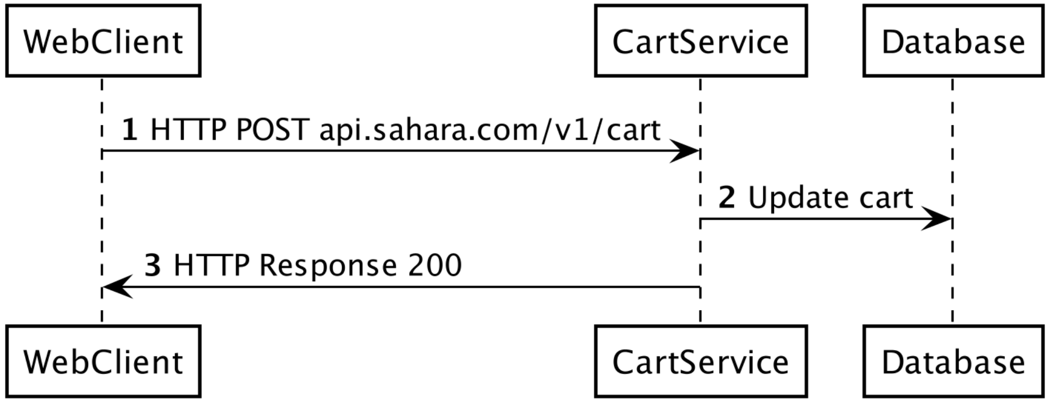
A few reasons for complexity

The Fallacies of *Distributed Computing*.

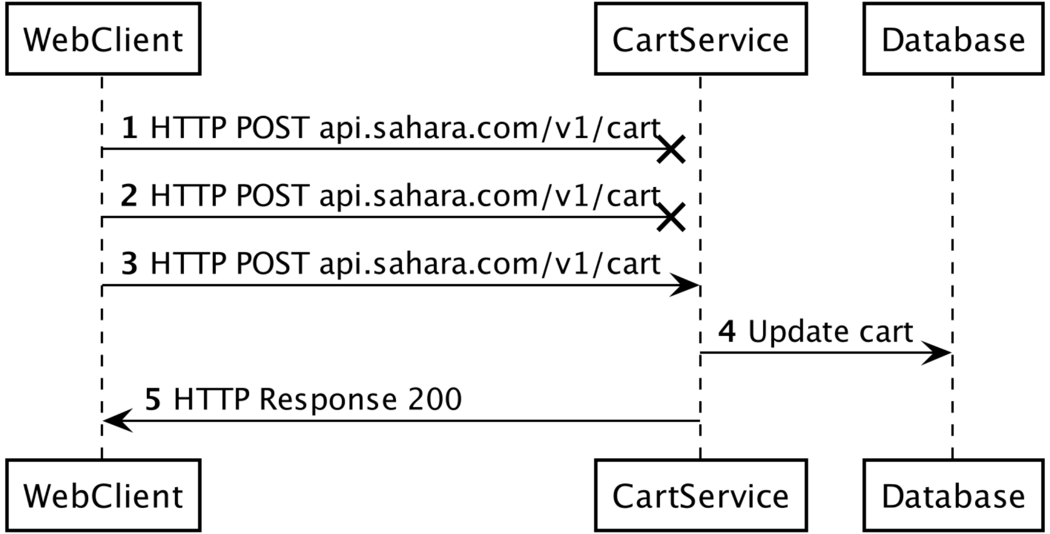
Sun Microsystems in 1994, primarily accredited to Peter Deutsch (doy-ch)

Fallacy #1

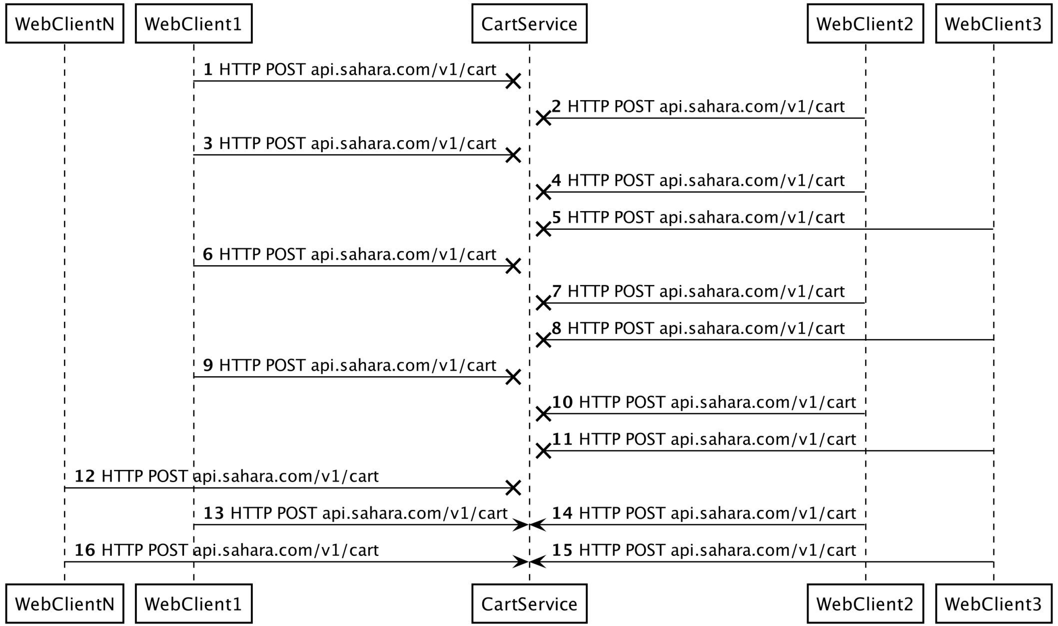
The network is reliable.


WebClient
CartService
Database
1 HTTP POST api.sahara.com/v1/cart
2 Update cart
3 HTTP Response 200
WebClient
CartService
Database


WebClient
CartService
1 HTTP POST api.sahara.com/v1/cart
WebClient
CartService



Solve it by resending it

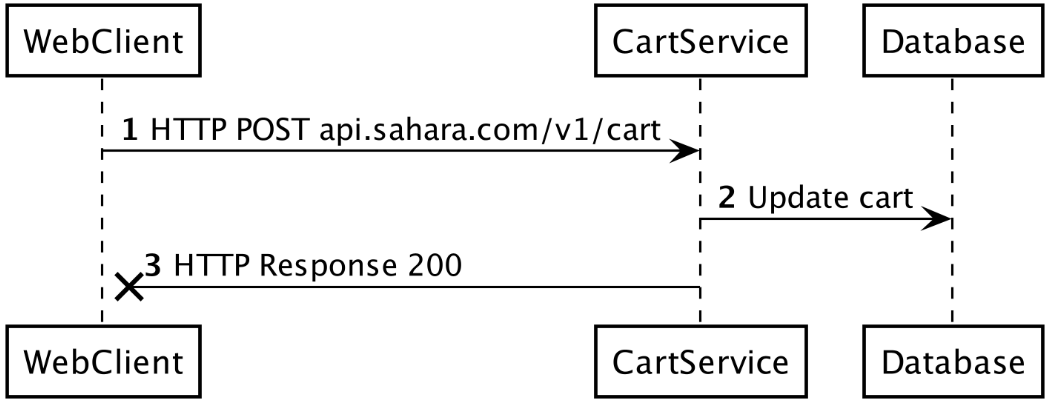


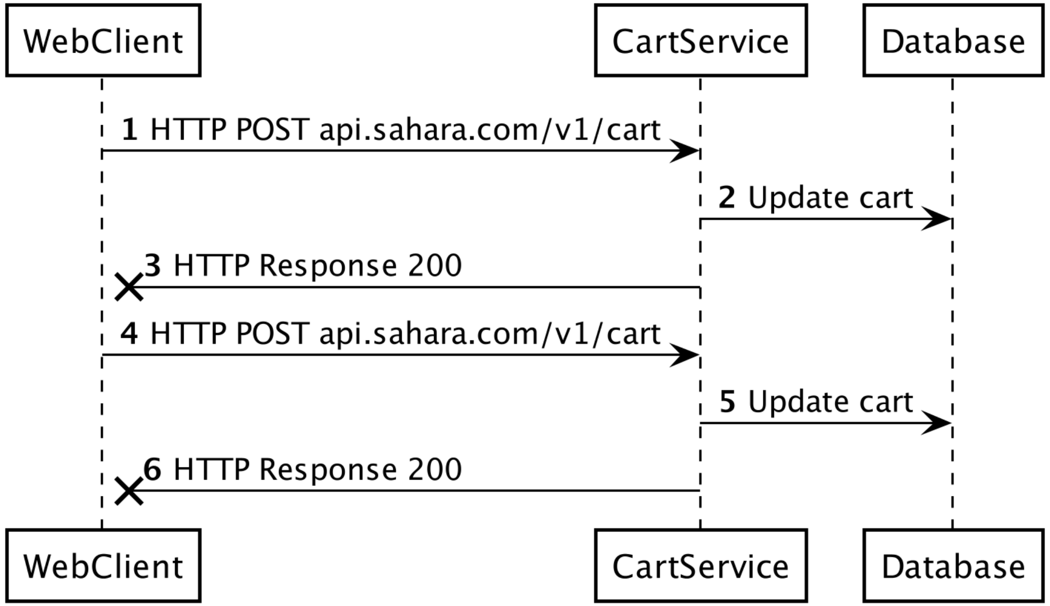
If the service goes down and all clients are re-trying, the service is in for a shock when it comes back, we solve this with *exponential backoff*

Exponential backoff

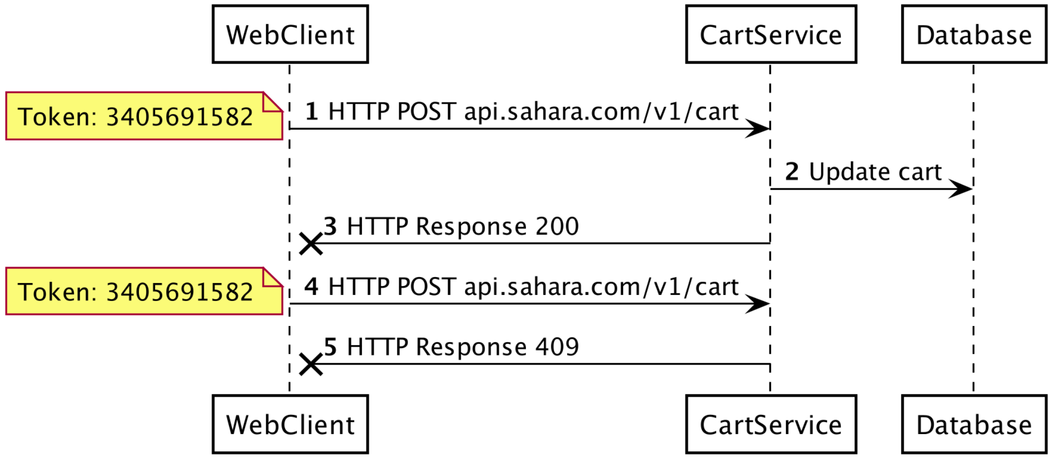
```
retry = True
do:
    status = service.request()

    if status != SUCCESS:
        wait(2 ** retries)
    else:
        retry = False
while (retry and retries < MAX_RETIRES)
```


WebClient
CartService
Database
1 HTTP POST api.sahara.com/v1/cart
2 Update cart
3 HTTP Response 200
WebClient
CartService
Database



Causes duplicate actions, problem for ordering/payments



Use tokens to prevent duplicates.

Fallacy #2

Latency is zero.

Home to UQ

Home to us-east-1

EC2 to EC2

Network Statistics

Home to UQ	20.025ms
Home to us-east-1	
EC2 to EC2	

Network Statistics

Home to UQ	20.025ms
Home to us-east-1	249.296ms
EC2 to EC2	

Home to UQ	20.025ms
Home to us-east-1	249.296ms
EC2 to EC2	0.662ms

Be mindful when designing distributed systems. Network call much slower then local call.

Fallacy #3

Bandwidth is infinite.

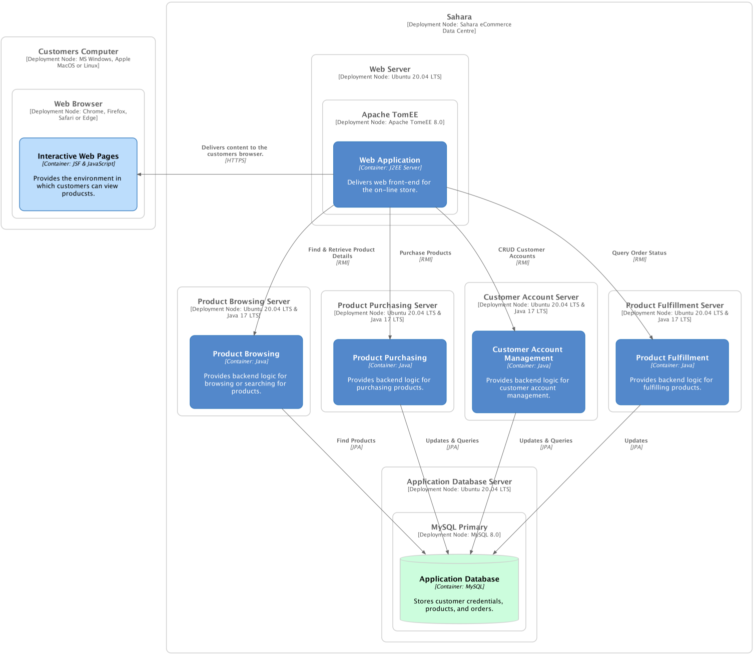
Similar to previous fallacy, be mindful, distributed calls clog up network.

Definition 1. Stamp Coupling

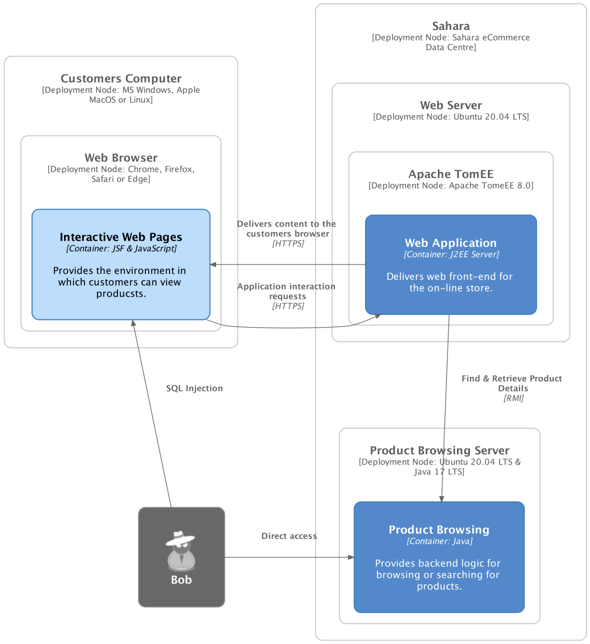
Components which share a composite data structure.

Fallacy #4

The network is secure.



Authentication only occurs when entering Sahara data centre



Bad actor gets access via one insecure node, network is compromised. Practice defence in depth.

Fallacy #5

The topology never changes.

Topology changes all the time, cloud has just made this easier. Don't rely on static IPs. Don't assume consistent latency.

Fallacy #6

There is only one administrator.

Things spontaneously break. Who can help you?

Fallacy #7

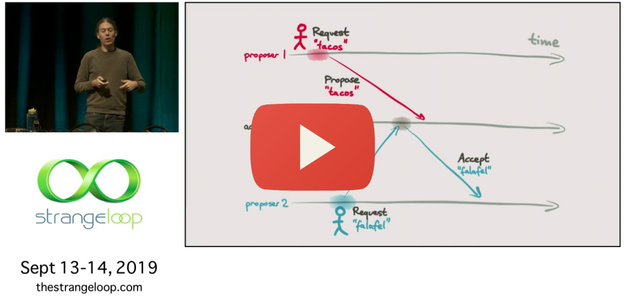
Transport cost is zero.

Remember

Distributed systems are *hard*.
The choice to use them should be *well considered*.

Can often introduce more problems than they solve

When you need to, maybe prove it?



https://youtube.com/watch?v=7w4KC6i9Yac

Or, more realistically,

Use existing algorithms and software.

Distributed Systems Series

Distributed I *Reliability* and *scalability* of *stateless* systems.

Distributed II Complexities of stateful systems.

Distributed III Hard problems in distributed systems.

Stateless vs. Stateful Systems

Stateless Does *not* utilize *persistent data*.

Stateful Does utilize *persistent data*.

Question

What makes software *reliable*?

Definition 2. Reliable Software

Continues to work, even when things go wrong.

Definition 3. Fault

Something goes wrong.

Death, taxes, and computer system failure are all inevitable to some degree.
Plan for the event.

- Howard and LeBlanc

Reliable software is

Fault *tolerant*.

John von Neumann built fault tolerant hardware in the 1950s.

Problem

Individual computers fail *all the time*

10-50 years hard-drive lifetime. 10,000 disks will fail daily. Google last had 2.5 million servers.

Solution

Spread the risk of faults over *multiple computers*, or, *nodes*.

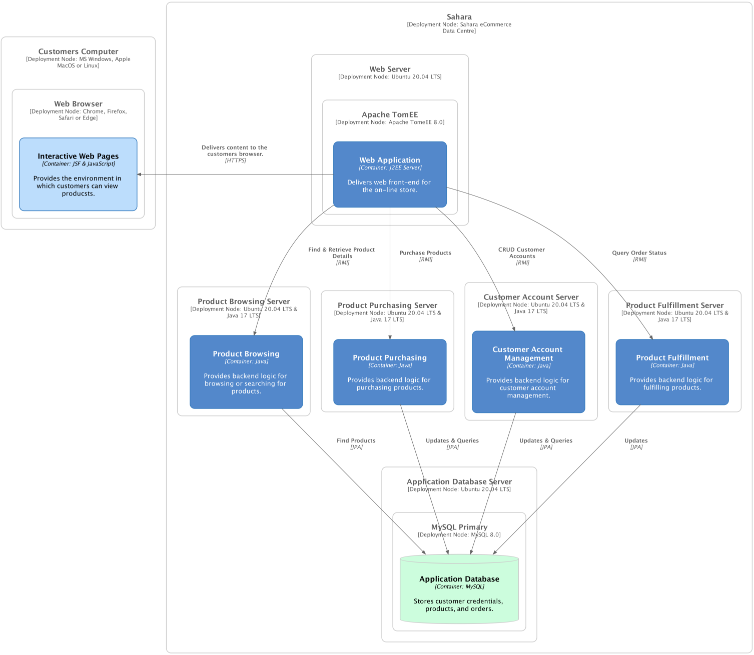
Spreading Risk

If you have software that works with *just one* computer, spreading the software over *two* computers *halves* the risk that your software will fail.

Spreading Risk

If you have software that works with *just one* computer, spreading the software over *two* computers *halves* the risk that your software will fail.

Adding *100* computers reduces the cuts the risk by *100*.



- Why is this software somewhat reliable?
- Any individual service can go down and the rest still work.
- Can we do better?
- Can a service go down but have that service still work?

Question

Who has used *auto-scaling*?

Auto-scaling terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Auto-scaling terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Capacity Amount of instances *currently* in an auto-scaling group.

Auto-scaling terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Capacity Amount of instances *currently* in an auto-scaling group.

Desired Capacity Amount of instances *we want to have* in an auto-scaling group.

Auto-scaling terminology

Auto-scaling group A *collection of instances* managed by auto-scaling.

Capacity Amount of instances *currently* in an auto-scaling group.

Desired Capacity Amount of instances *we want to have* in an auto-scaling group.

Scaling Policy How to determine the desired capacity.

Auto-scaling terminology

Auto-scaling group
: A *collection of instances* managed by auto-scaling.

Capacity
: Amount of instances *currently* in an auto-scaling group.

Desired Capacity
: Amount of instances *we want to have* in an auto-scaling group.

Scaling Policy
: How to determine the desired capacity.

Minimum/Maximum Capacity
: *Hard limits* on the minimal and maximum amount of instances.

What we really want

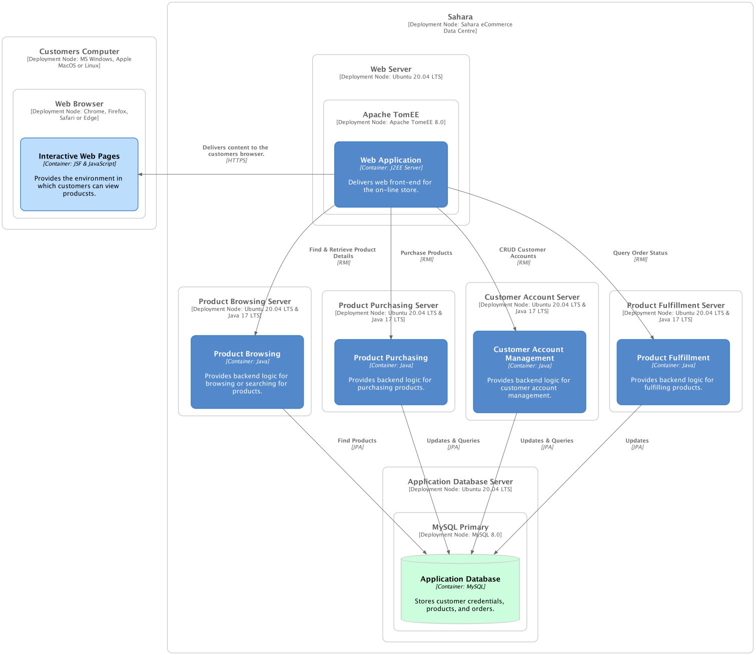
Desired Capacity Amount of *healthy* instances we want to have in an auto-scaling group.

Health check

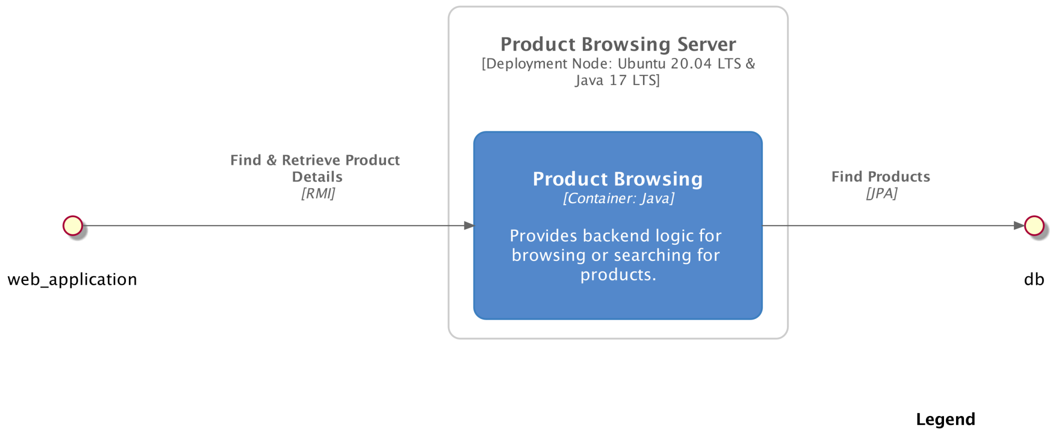
Mechanism to determine whether an instance is *healthy*.

Auto-scaling

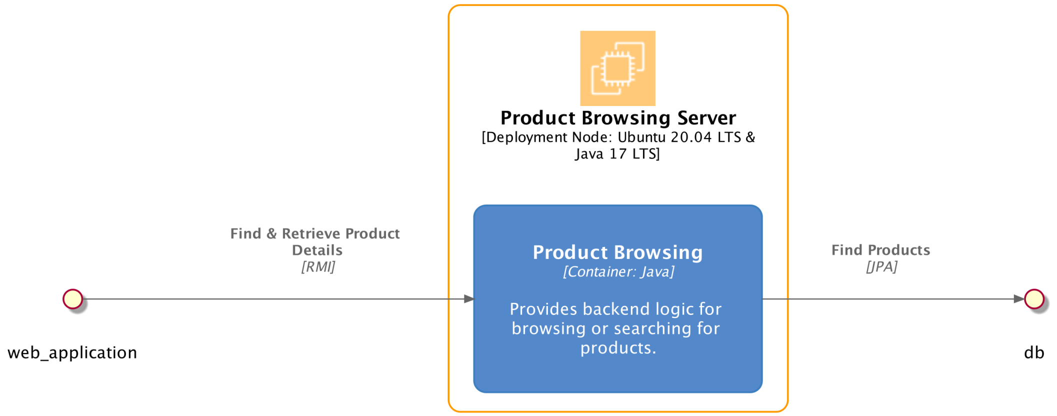
An example



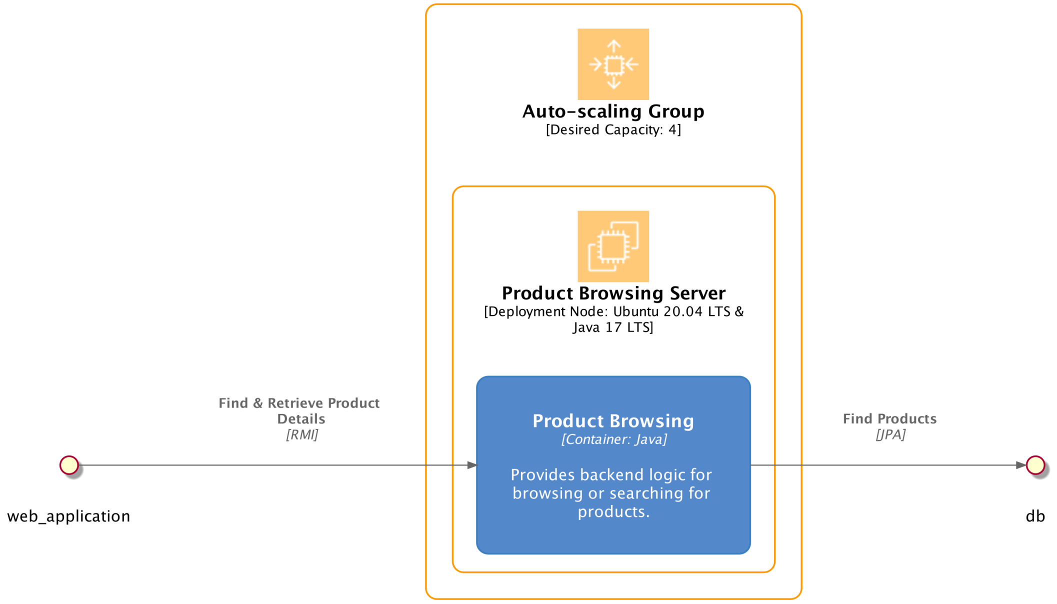
Product service keeps going down



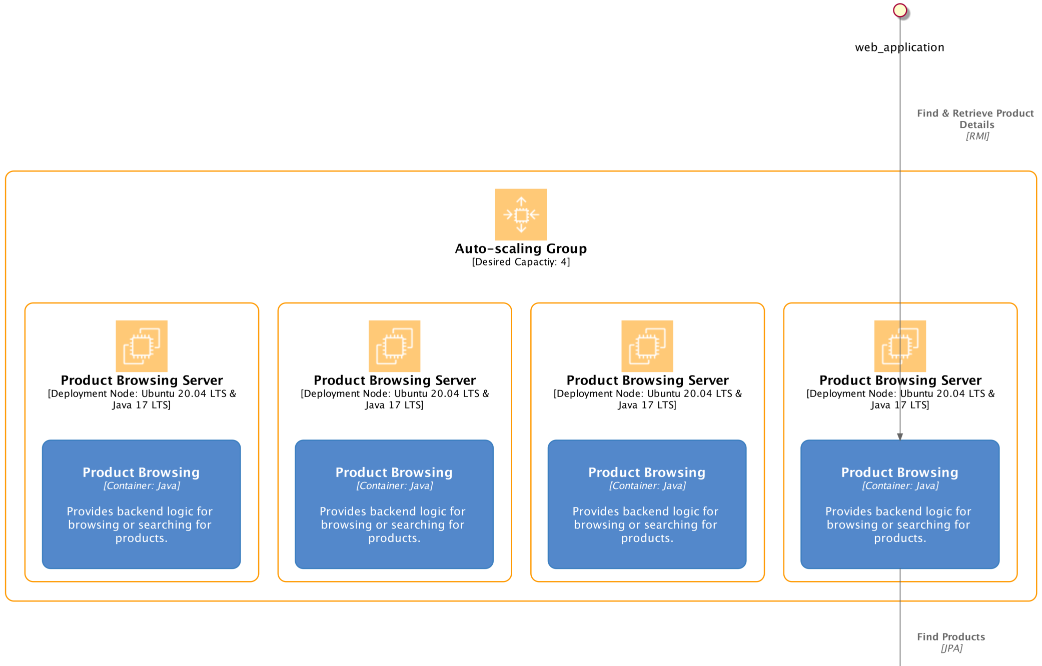
We might expect product service to have a much higher load than other services


Product Browsing Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Find & Retrieve Product Details
[RMI]
Find Products
[JPA]
web_application
db

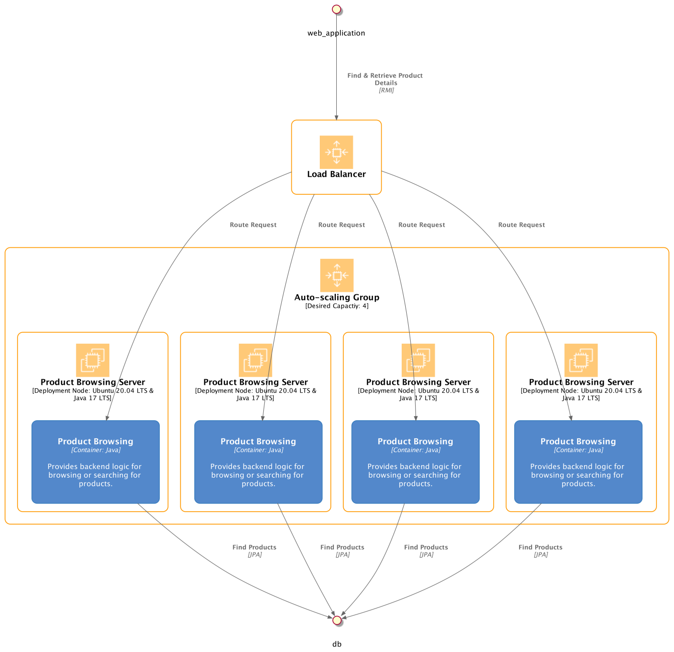
Legend



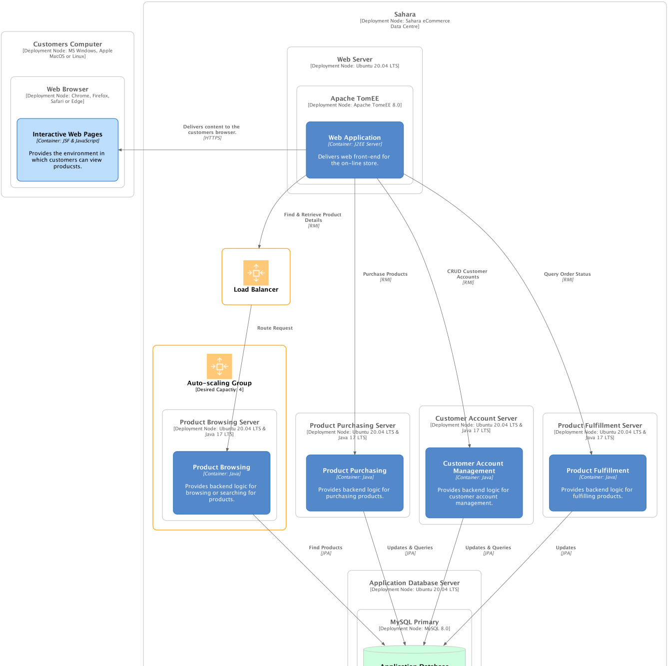
Use an auto-scaling group to replicate the service



What’s the problem?



Traffic was all sent through the one instance, load balancer routes to all



Sahara
[Deployment Node: Sahara eCommerce Data Centre]
Customers Computer
[Deployment Node: MS Windows, Apple MacOS or Linux]
Web Browser
[Deployment Node: Chrome, Firefox, Safari or Edge]
Interactive Web Pages
[Container: JSF & JavaScript]
Provides the environment in which customers can view producsts.
Delivers content to the customers browser.
[HTTPS]
Web Server
[Deployment Node: Ubuntu 20.04 LTS]
Apache TomEE
[Deployment Node: Apache TomeEE 8.0]
Web Application
[Container: J2EE Server]
Delivers web front-end for the on-line store.
Find & Retrieve Product Details
[RMI]
Load Balancer
Purchase Products
[RMI]
CRUD Customer Accounts
[RMI]
Query Order Status
[RMI]
Route Request
Auto-scaling Group
[Desired Capactiy: 4]
Product Browsing Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Product Purchasing Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Purchasing
[Container: Java]
Provides backend logic for purchasing products.
Customer Account Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Customer Account Management
[Container: Java]
Provides backend logic for customer account management.
Product Fulfillment Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Fulfillment
[Container: Java]
Provides backend logic for fulfilling products.
Find Products
[JPA]
Updates & Queries
[JPA]
Updates & Queries
[JPA]
Updates
[JPA]
Application Database Server
[Deployment Node: Ubuntu 20.04 LTS]
MySQL Primary
[Deployment Node: MySQL 8.0]

In Summary

- Simplicity
- Reliability
- Scalability

In Summary

Simplicity *Minimal network communication* (compared to other distributed systems), less impacted by fallacies.

Reliability

Scalability

In Summary

Simplicity *Minimal network communication* (compared to other distributed systems), less impacted by fallacies.

Reliability Traffic is spread to various services, still *partially operational* if one goes down. Auto-scaling allows for *basic replication*.

Scalability

In Summary

Simplicity *Minimal network communication* (compared to other distributed systems), less impacted by fallacies.

Reliability Traffic is spread to various services, still *partially operational* if one goes down. Auto-scaling allows for *basic replication*.

Scalability Auto-scaling and load balancing allows *individual services to scale*. However, the *database is a bottle-neck*.

database is a bottle-neck is foreshadowing