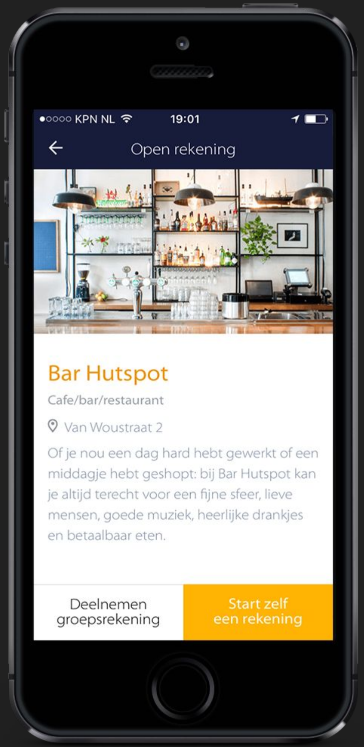
Distributed queues with RabbitMQ

IN4391 Demo Day - April 6, 2016

Jorik Oostenbrink
Dominik Harz

Tabster

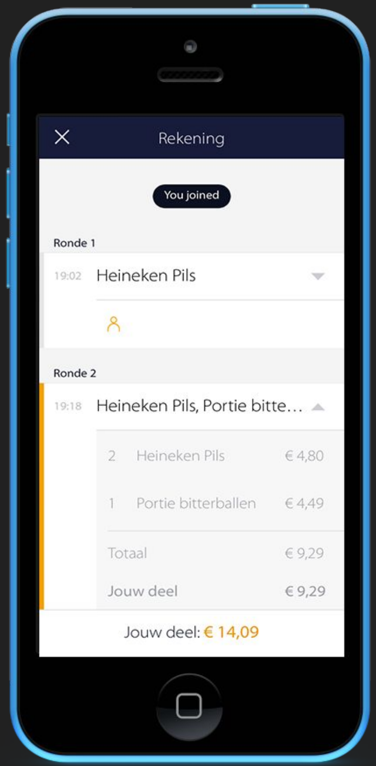


Bar Hutspot

Cafe/bar/restaurant

Van Woustraat 2

Of je nou een dag hard hebt gewerkt of een middagje hebt geshopt: bij Bar Hutspot kan je altijd terecht voor een fijne sfeer, lieve mensen, goede muziek, heerlijke drankjes en betaalbaar eten.



Ronde 1

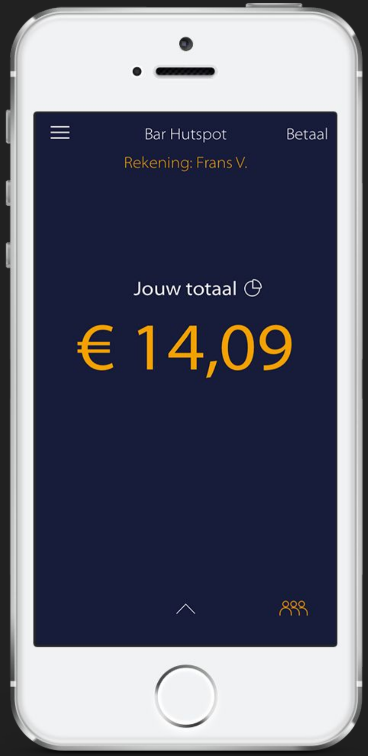
19:02 Heineken Pils

Ronde 2

19:18 Heineken Pils, Portie bitte…

2	Heineken Pils	€ 4,80
1	Portie bitterballen	€ 4,49
	Totaal	€ 9,29
	Jouw deel	€ 9,29

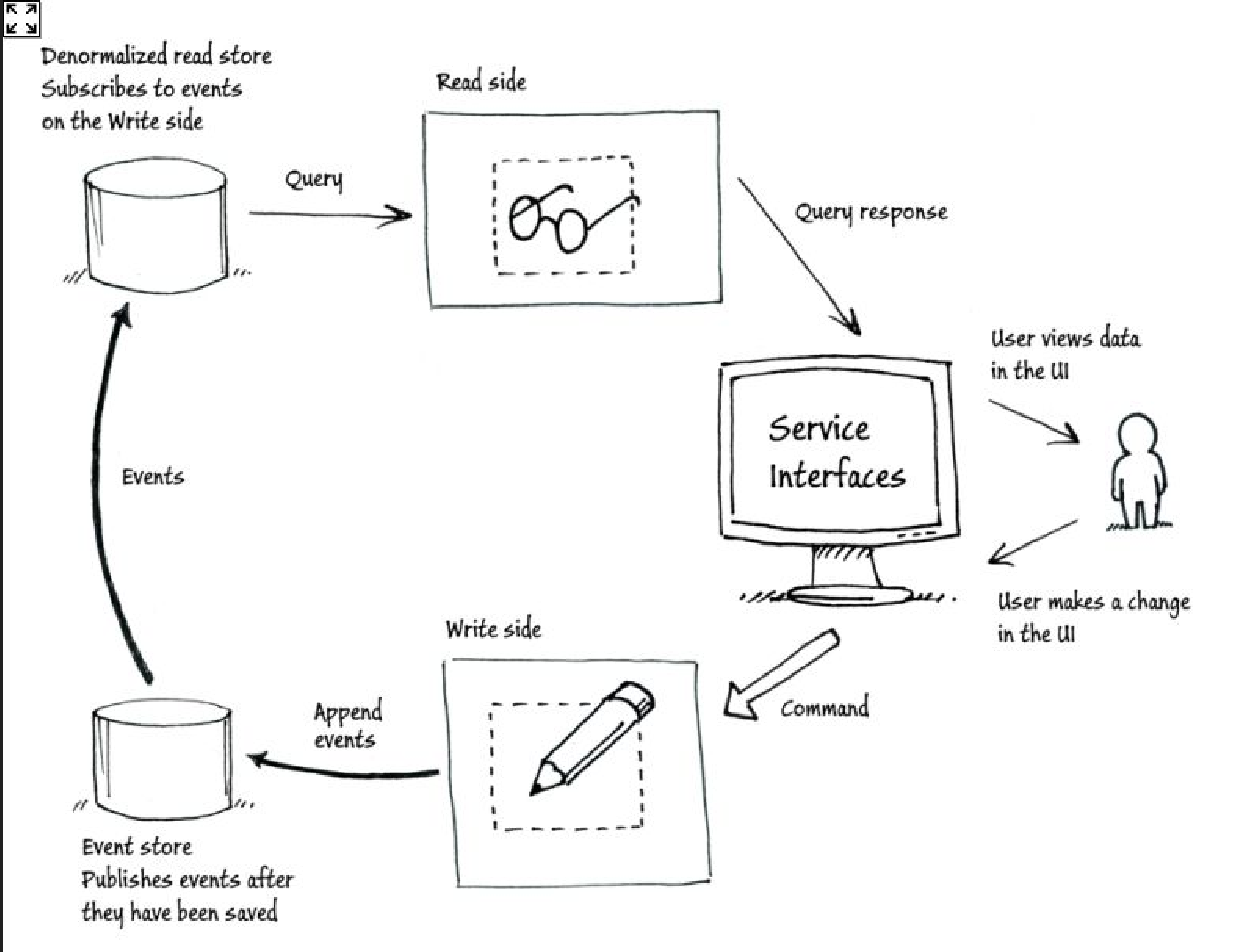
Jouw deel: € 14,09



Jouw totaal

€ 14,09

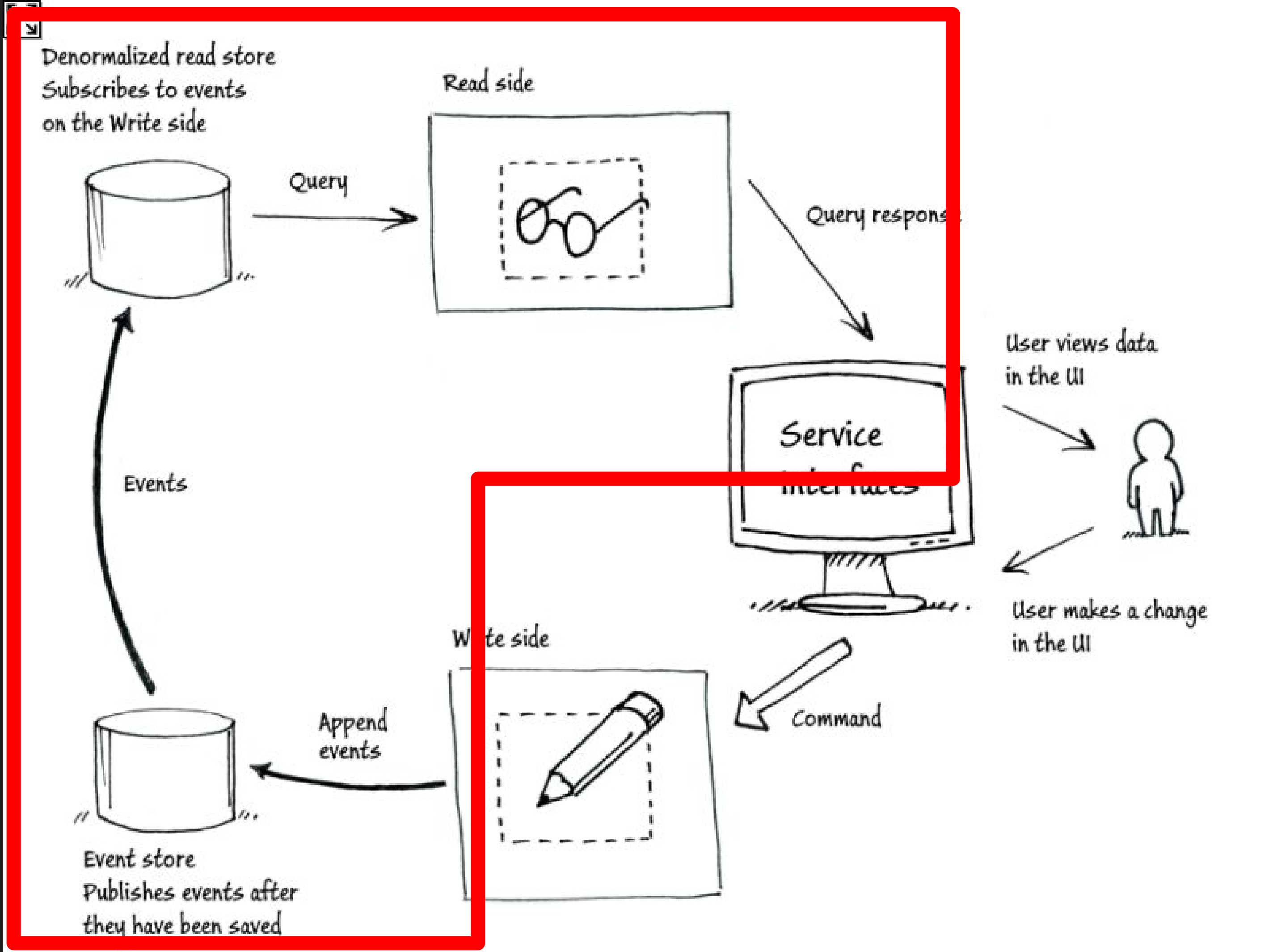
Event based architecture



Command and Query Responsibility Segregation (CQRS)

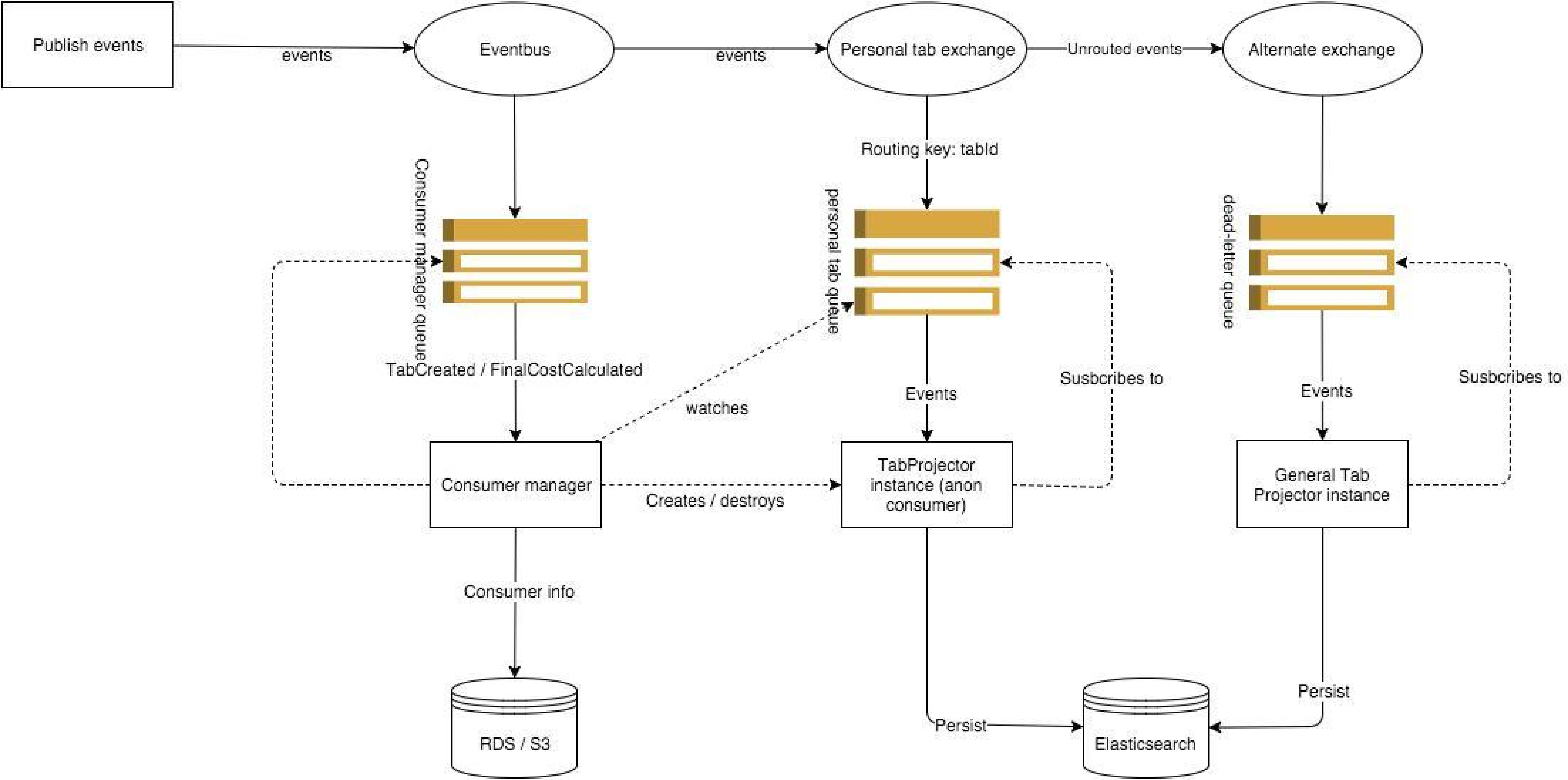
https://msdn.microsoft.com/en-us/library/dn568103.aspx

Event based architecture



Key requirements

- A failure of one queue or worker does not affect other system components
- All system events must be logged in the order they occur
- The properties of the Tabster system must be demonstrated when it contains
    - 10 tabs with at least 100 events per tab
    - 2 consumer managers
    - workload consists of at least 1000 events in total


Publish events
events
Eventbus
events
Personal tab exchange
Unrouted events
Alternate exchange
Routing key: tabId
Consumer manager queue
personal tab queue
dead-letter queue
TabCreated / FinalCostCalculated
watches
Events
Susbcribes to
Events
Susbcribes to
Consumer manager
Creates / destroys
TabProjector instance (anon consumer)
General Tab Projector instance
Consumer info
Persist
Persist
RDS / S3
Elasticsearch

Features

- Eventual consistency
    - Separate read and write models
    - Processes events from separate tabs simultaneously
    - Processes events from a single tab in the order they arrived
- Fault tolerance
    - Queues get recreated after failing (set to durable)
    - Workers are monitored for failures
    - Elasticsearch configured as a cluster with replication
    - API configured as a HA cluster with nginx as load-balancer

Demo time!

Next steps

1. Execute RabbitMQ as a cluster with multiple nodes
2. Place the system on AWS
3. Execute stress testing
   a. Optimize distributed system parameters
   b. Compare to existing system

Conclusion

- Performance increase due to distributed queues and workers?
- CQRS can be implemented in a distributed way
    - Application: PHP and Symfony2
    - Database: MySQL
    - Eventbus: RabbitMQ
    - Eventstore: Elasticsearch
- Lessons learned
    - Making an existing solution distributed is kind of hard (loads of dependencies)
    - Symfony2 is not designed for clustering RabbitMQ
    - Eventual consistency is tricky to test