HSLU Hochschule Luzern

Software Architecture and Techniques

Architecture Trends II

Marcel Baumann, tangly llc

Truths (1/2)

- *Make or Die* is the new game
- Vendor anti-pattern – *the missing 10% -*
- The main difference between your company and your competitors is the **software applications** your customers use to interact with you

Truths (2/2)

- The smaller the quantum size of the architecture the more evolvable it will be
- The smaller the quantum size of the architecture the more difficult it is to debug, log, and monitor
- Two phases commit transactions are a huge pain
  Database transactions act as strong nuclear force, binding quant together

Assumptions (1/3)

- SOA is dead, long live micro-services
- JEE is dead, long live micro-profile
- Micro services are only viable using Docker, Helm and domain driven design
- Enterprise micro services are only viable using technologies such as Kubernetes and Helm (historically OpenShift)
- Lambda functions is the next evolution iteration

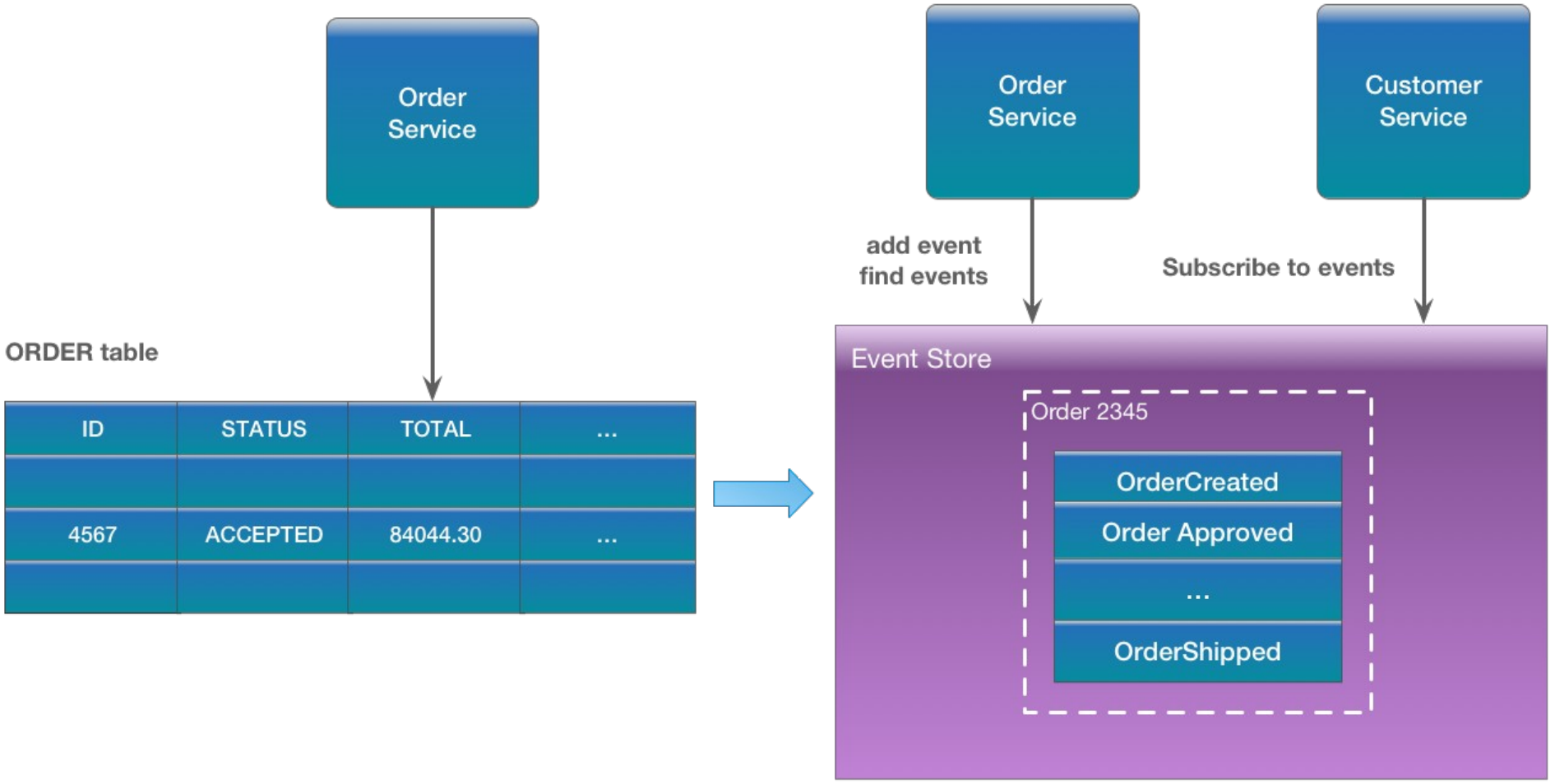
Assumptions (2/3)

- You should use docker images, virtual machines are dead
- You should use hybrid cloud if you calculate the project investment and **running costs**
- Lean thinking will win in the long term
- Ignoring security is your professional death

Assumptions (3/3)

- **Eventual consistency** is the norm
- **Idempotent** is highly valuable
- **Immutability** is the new kid on the block
- Events and commands are an important design pattern *CQRS*
- Event streaming and event store are worth a thought

Event Sourcing



Services

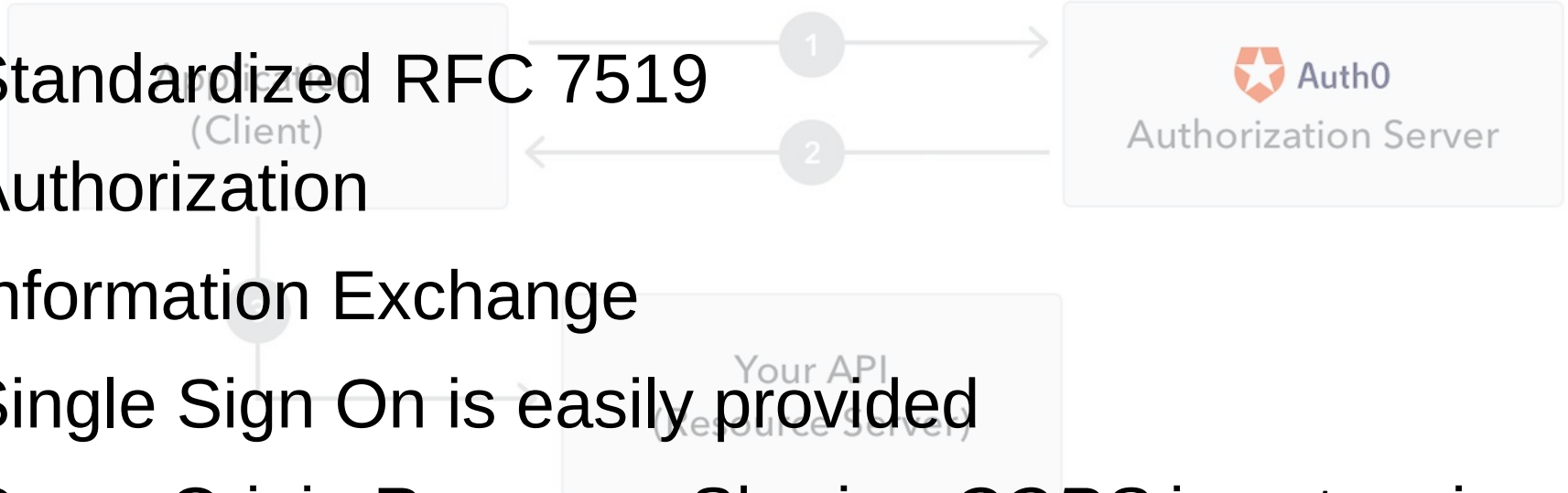
- JSON and JSON type
  - XML is dead
- GraphQL
  - Optimize latency
  - Provide type security and release capability
  - *Aggregator of backend services for front-end applications*
- API Mesh

Reactive Systems

- Reactive is a set of design principles – *similar to REST approach -*
- Is asynchronous
- Understand difference between event and message
- Use the event stream as communication fabric

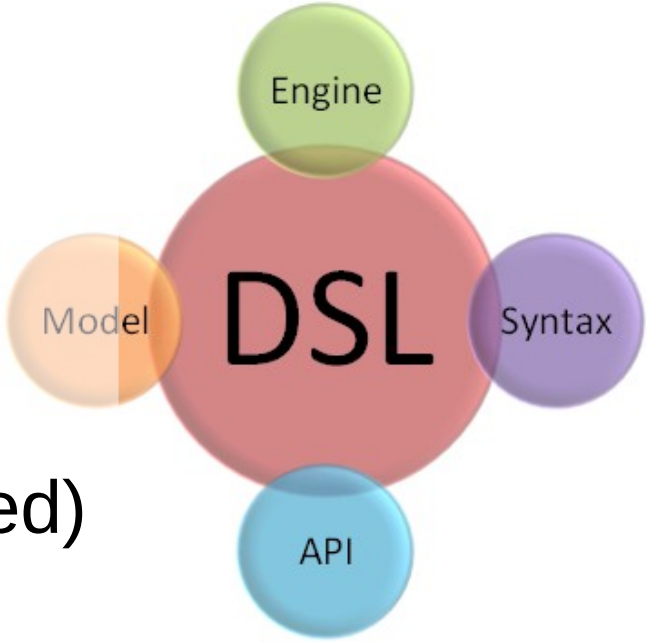
Security with JSON Web Tokens

- Stateless
- Standardized RFC 7519
- Authorization
- Information Exchange
- Single Sign On is easily provided
- Cross-Origin Resource Sharing *CORS* is not an issue



Domain Specific Languages

- Give the approach a try
- Start with embedded DSL
  - **Fluent API**
  - **Builder** pattern
  - **Gradle DSL** (Groovy or Kotlin based)



Monitoring

- Automatic monitoring
- Alarming, Tracing, Logging
- Fitness Functions
- Debugging
- e.g. ELK ElasticSearch, LogStash, Kibana

Central Logging

- Logging such as **slf4j** or **log4j2**
- Logging coding guidelines and architecture rules
- Central repository of logs
- Architecture should identify flows of related events and commands

Logging Concepts (1/3)

The Java logging API consists of three core components:

- **Loggers** are responsible for capturing events - called Log records - and passing them to the appropriate Appender.
- **Appenders** - also called Handlers in some logging frameworks - are responsible for recording log events to a destination. Appenders use Layouts to format events before sending them to an output.
- **Layouts** - also called Formatters in some logging frameworks - are responsible for converting and formatting the data in a log event. Layouts determine how the data looks when it appears in a log entry.

Logging Concepts (2/3)

- Logger declaration
- Logging Level
- Structure through layout and message syntax

```
private static final Logger logger =
LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

logger.atError().setCause(e).log("invoices: Error during
invoice asciiDoc generation {}", invoicePath);
```

Logging Concepts (3/3)

- Fluent Interface with log4j2 or slf4j
  - Efficient and flexible logging
- Mapped Diagnostic Context MDC
  - Information {key, value} per thread or process
  - Central feature for multi-threaded logging

Auditing

- Auditing is always a **compliance** component
- Learn the laws and recommendations
- Audit must be **human readable**
- Audit must be **tamper proof**
- *Audit are worthless for the application functions*

Persistence

- Relational Database
  - Embedded Database HSQLDB
  - Java Persistence API JPA
  - JOOQL
- Non-Relational Database
  - Document Database
  - MicroStream
  - Serialization (*currently moribund)*

12 Factors (1/2)

I) Codebase – One codebase tracked in revision controls, many deploys

II) Dependencies – Explicitly declare and isolate dependencies

III) Configuration – store in the environment

IV) Backing services – treat backing services as attached resources

V) Build, release, run – strictly separate build and run stages

VI) Processes – Execute the app as one or more stateless processes

12 Factors (2/2)

VII) Port binding – Export services via port binding

VIII) Concurrency – Scale out via the process model

IX) Disposability – Maximize robustness with fast startup and graceful shutdown

X) Dev and prod parity - Keep development, staging, and production as similar as possible

XI) Logs – Treat logs as event streams

XII) Admin processes – Run admin and management tasks os on-off processes

Hypothesis

Given a choice start with a **monolithic modular** and **domain driven design** approach

Move to a distributed solution when growing

Once you make real money ponder if you should move to a micro-architecture solution

Quote

In any moment of decision, the best thing you can do is the right thing, the next best thing you can do is the wrong thing, the worst thing you can do is nothing.

– Theodore Roosevelt

Architecture Styles

- Layered Architecture
- Event-driven Architecture
- Microkernel Architecture
- Micro-services Architecture
- Client-Server Architecture
- Master-Slave Architecture
- Pipe-Filter Architecture
- Broker Architecture
- Peer-to-Peer Architecture
- SOA Architecture

Workshop

- Discuss architecture problems you found in your projects
- Take the exercise and design
  - CRM
  - Incentive program
  - 360 degree view
  - *Build it small, think big*