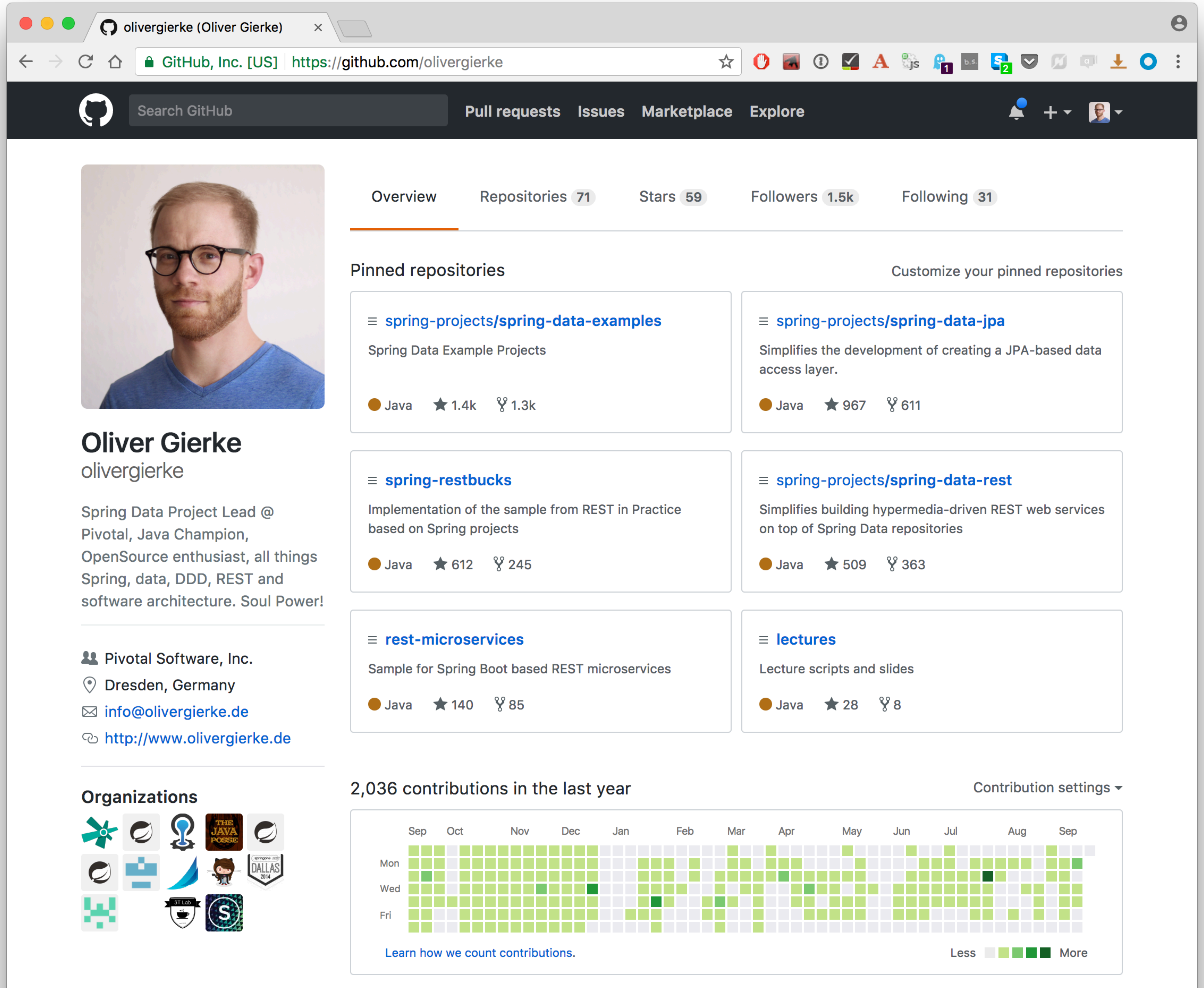
Refactoring to a System of Systems

Of monoliths, microservices and everything in between…

Oliver Gierke /olivergierke ogierke@pivotal.io



olivergierke (Oliver Gierke)

GitHub, Inc. [US] https://github.com/olivergierke

Overview Repositories 71 Stars 59 Followers 1.5k Following 31

Oliver Gierke
olivergierke

Spring Data Project Lead @ Pivotal, Java Champion, OpenSource enthusiast, all things Spring, data, DDD, REST and software architecture. Soul Power!

Pivotal Software, Inc.
Dresden, Germany
info@olivergierke.de
http://www.olivergierke.de

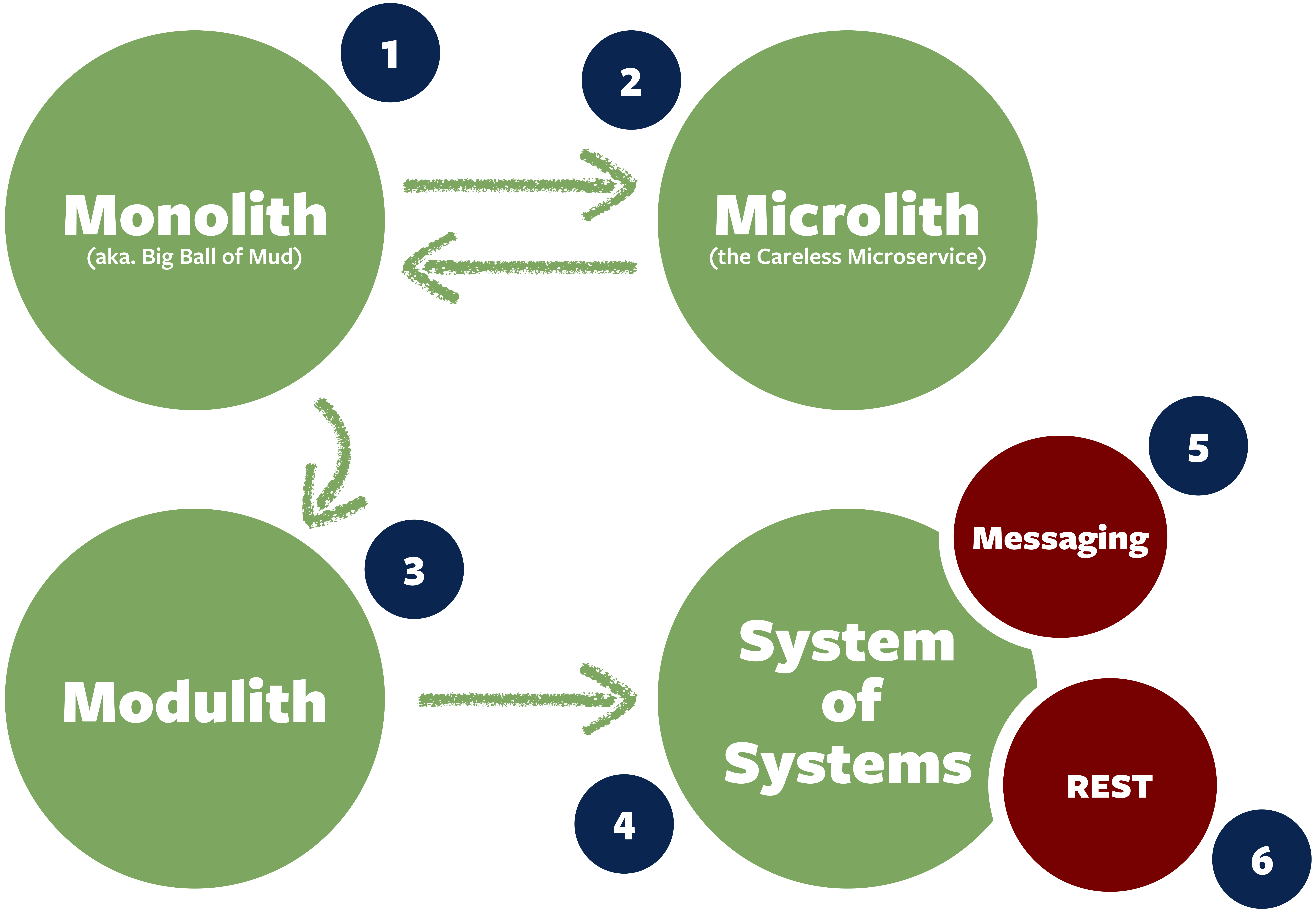
Organizations

Pinned repositories — Customize your pinned repositories

Repository	Description	Language	Stars	Forks
spring-projects/spring-data-examples	Spring Data Example Projects	Java	1.4k	1.3k
spring-projects/spring-data-jpa	Simplifies the development of creating a JPA-based data access layer.	Java	967	611
spring-restbucks	Implementation of the sample from REST in Practice based on Spring projects	Java	612	245
spring-projects/spring-data-rest	Simplifies building hypermedia-driven REST web services on top of Spring Data repositories	Java	509	363
rest-microservices	Sample for Spring Boot based REST microservices	Java	140	85
lectures	Lecture scripts and slides	Java	28	8

2,036 contributions in the last year — Contribution settings

Learn how we count contributions. Less More


1
Monolith
(aka. Big Ball of Mud)
2
Microlith
(the Careless Microservice)
3
Modulith
4
System of Systems
5
Messaging
6
REST

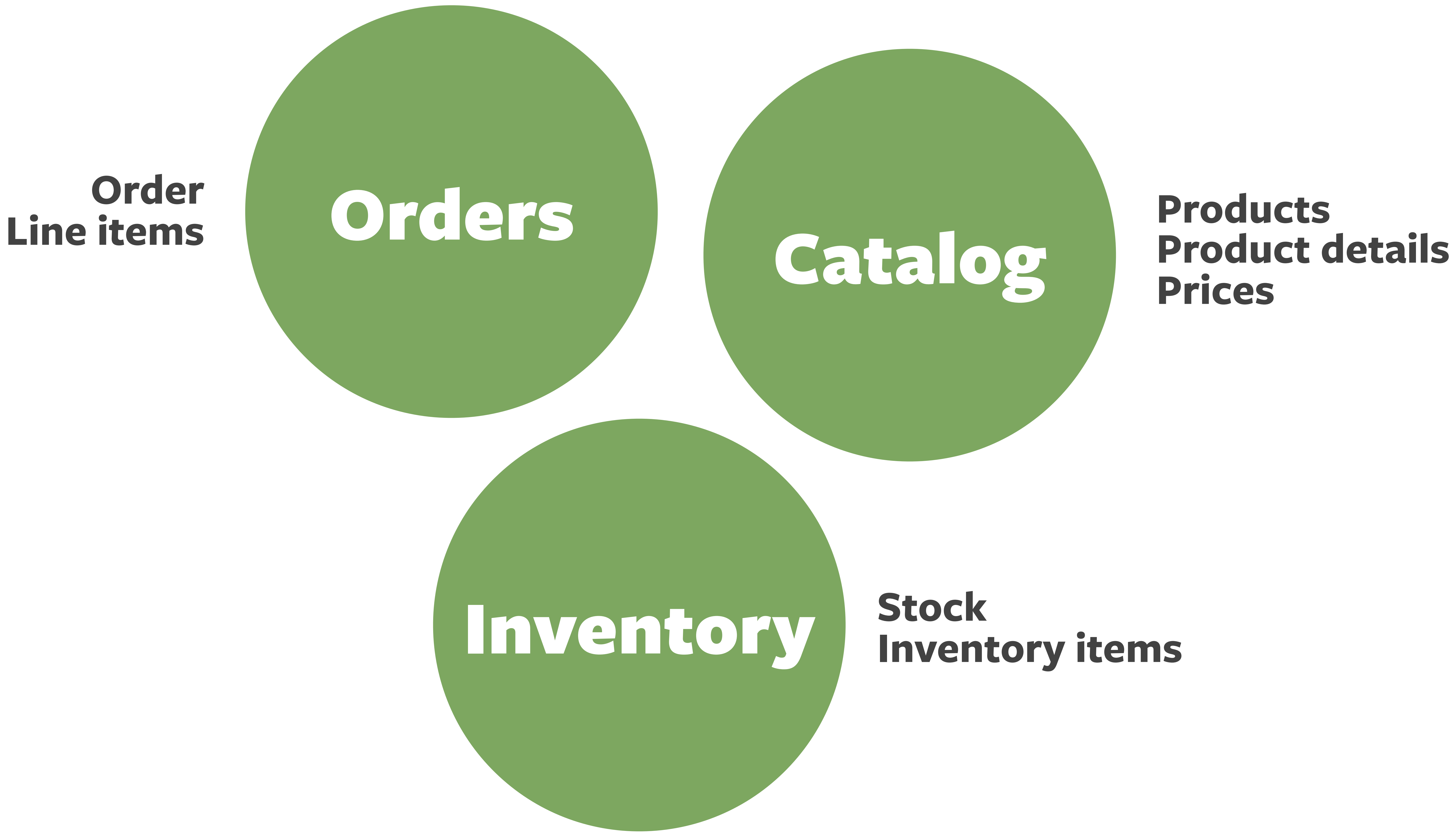
“What are typical Bounded Context interactions in a monolithic application?

“What happens if these patterns are translated 1:1 into a distributed system?

“Can we build a better monolith in the first place?

“How to translate that new approach into a distributed system?

The Domain


Order
Line items
Orders
Catalog
Products
Product details
Prices
Inventory
Stock
Inventory items

“ ***When a product is added to the catalog, the inventory needs to initialize its stock.***

“When an order is completed, inventory shall update its stock for all line items.

What do we want to focus on?

- What are commonly chosen design patterns and strategies?
- How do Bounded Contexts interact with each other?
- What types of consistency do we deal with?
- How do the systems behave in erroneous situations?
- How do the different architectures support independent evolvability?

Sample code

https://github.com/olivergierke/sos

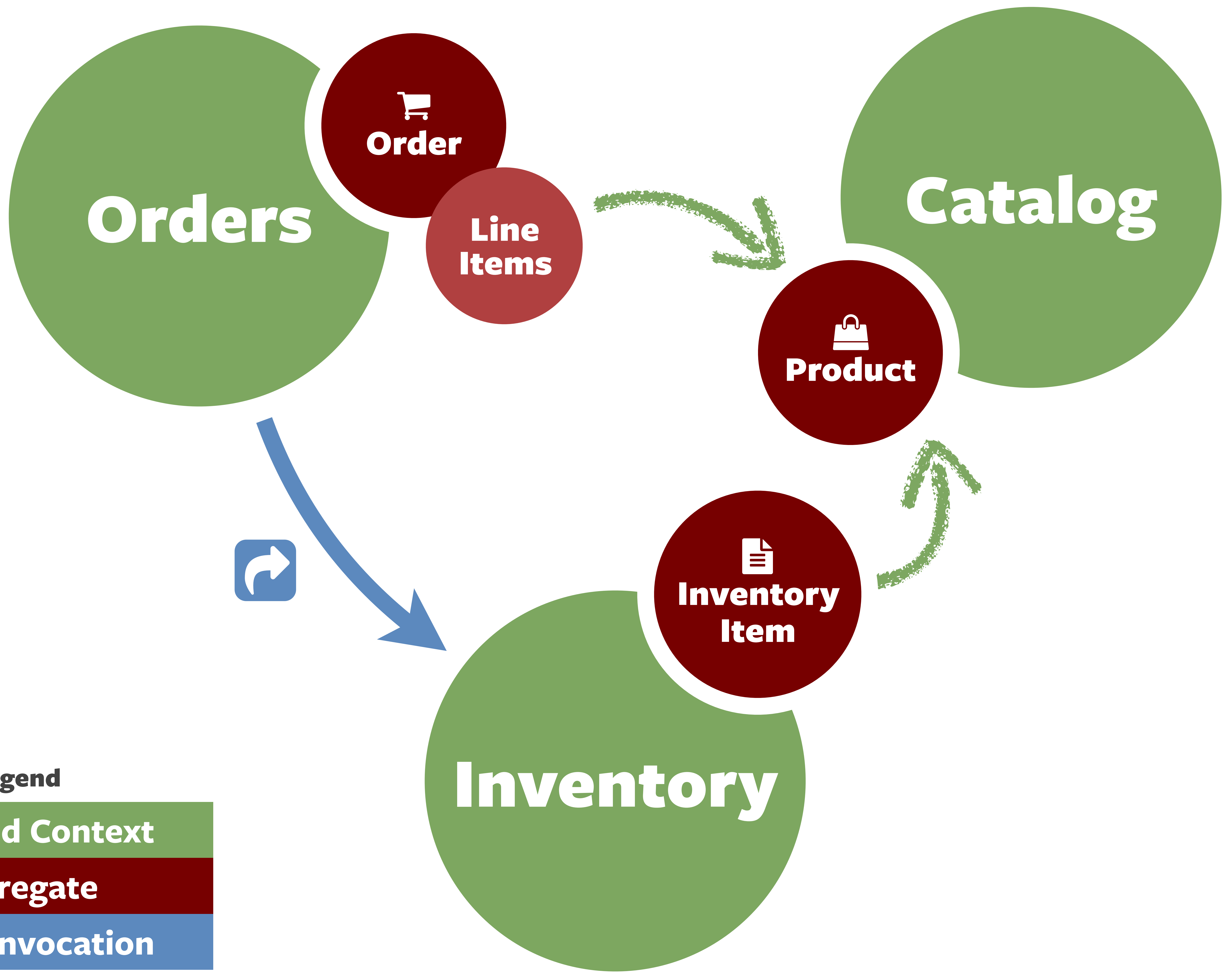
A couple of warnings…

The sample code is not a cookie cutter recipe of how to build things

The sample code is supposed to focus on showing the interaction model between Bounded Contexts, how to model aggregates and strive for immutability as much as possible. However, to not complicate the matter, certain aspects have been kept intentionally simple to avoid further complexity to blur the focus of the samples:

- Not all domain primitives are fully modeled
  - Monetary amounts are not modeled as such, but definitely should in real world projects.
  - Quantities are modeled as plain long but also should get their own value types.
- Most projects use JPA for persistence. This requires us to have default constructors and some degrees of mutability in domain types.
- Remote interaction is not fully implemented (not guarded against systems being unavailable etc.)

The Monolith



Legend

Bounded Context
Aggregate
Active invocation

The Monolith – Design Decisions

⊕ Bounded Contexts reflect into packages

A (hopefully not very) typical Spring Boot based Java web application. We have packages for individual Bounded Contexts which allows us to easily monitor the dependencies to not introduce cycles.

⊕ / ⊖ Domain classes reference each other even across Bounded Contexts

JPA creates incentives to use references to other domain types. This makes the code working with these types very simple at a quick glance: just call an accessor to refer to a related entity. However, this also has significant downsides:

- *The „domain model" is a giant sea of entities* – this usually causes problems with the persistence layer with transitive related entities as it's easy to accidentally load huge parts of the database into memory. The code is completely missing the notion of an aggregate that defines consistency boundaries.
- *The scope of a transaction grows over time* – Transactions can easily be defined using Spring's `@Transactional` on a service. It's also very convenient add more and more calls — and ultimately changes to entities — which blur the focus of the business transaction and making more likely to fail for unrelated reasons.

The Monolith – Design Decisions

➕ ↪ Inter-context interaction is process local

As the system is running as a single process, the interaction between Bounded Contexts is performant and very simple. We don't need any kind of object serialization and each call either succeeds or results in an exception. APIs can be refactored easily as IDEs can tweak calling and called code at the same time.

⛔ Very procedural implementation in order management

The design of `OrderManager.addToOrder(…)` treats domain types as pure data containers. It accesses internals of `Order`, applies some logic to `LineItems` and manipulates the `Order` state externally. However, we can find first attempts of more domain driven methods in `LineItem.increaseQuantityBy(…)`.

The Monolith – Design Decisions

⊖ Order management actively invokes code in inventory context

With the current design, services from different Bounded Contexts usually invoke each other directly. This often stems from the fact that it’s just terribly convenient to add a reference to a different managed bean via Dependency Injection and call that bean’s methods. This easily creates cyclic dependencies as the invoking code needs to know about the invoked code which in turn usually will receive types owned by the caller. E.g. `OrderManagement` knows about the `Inventory` and the `Inventory` accepts an `Order`.

A side-effect of this is that the scope of the transaction all of a sudden starts to spread multiple aggregates, even across contexts. This might sound convenient in the first place but with the application growing this might cause problems as supporting functionality might start interfering with the core business logic, causing transaction rollbacks etc.

The Monolith – Consequences

⊖ Service components become centers of gravity

Components of the system that are hotspots in business relevance („order completed“) usually become centers of dependencies and dissolve into god classes that refer to a lot of components of other Bounded Contexts. The `OrderManagement`’s `completeOrder(…)` method is a good example for that as will have to be touched to invoke other code for every feature that’s tied to that business action.

⊖ Adding a new feature requires different parts of the system to be touched

A very typical smell in that kind of design is that new features will require existing code to be touched that should not be needed. Imagine we’re supposed to introduce a rewards program that calculates bonus points for completed orders. Even if a dedicated team implements that feature in a completely separate package, the `OrderManagement` will eventually have to be touched to invoke the new functionality.

The Monolith – Consequences

⊕ Easy to refactor

The direct type dependencies allows the IDE to simplify refactorings. We just have to execute them and calling and called code gets updated. We cannot accidentally break involved third parties as there are none. Especially in scenarios where there's little knowledge about the domain, this can be very advantageous. The interesting fact to notice here is that we have strong coupling but still can refactor and evolve relatively rapidly. This is driven by the locality of the changes.

⊕ / ⊖ Strong consistency

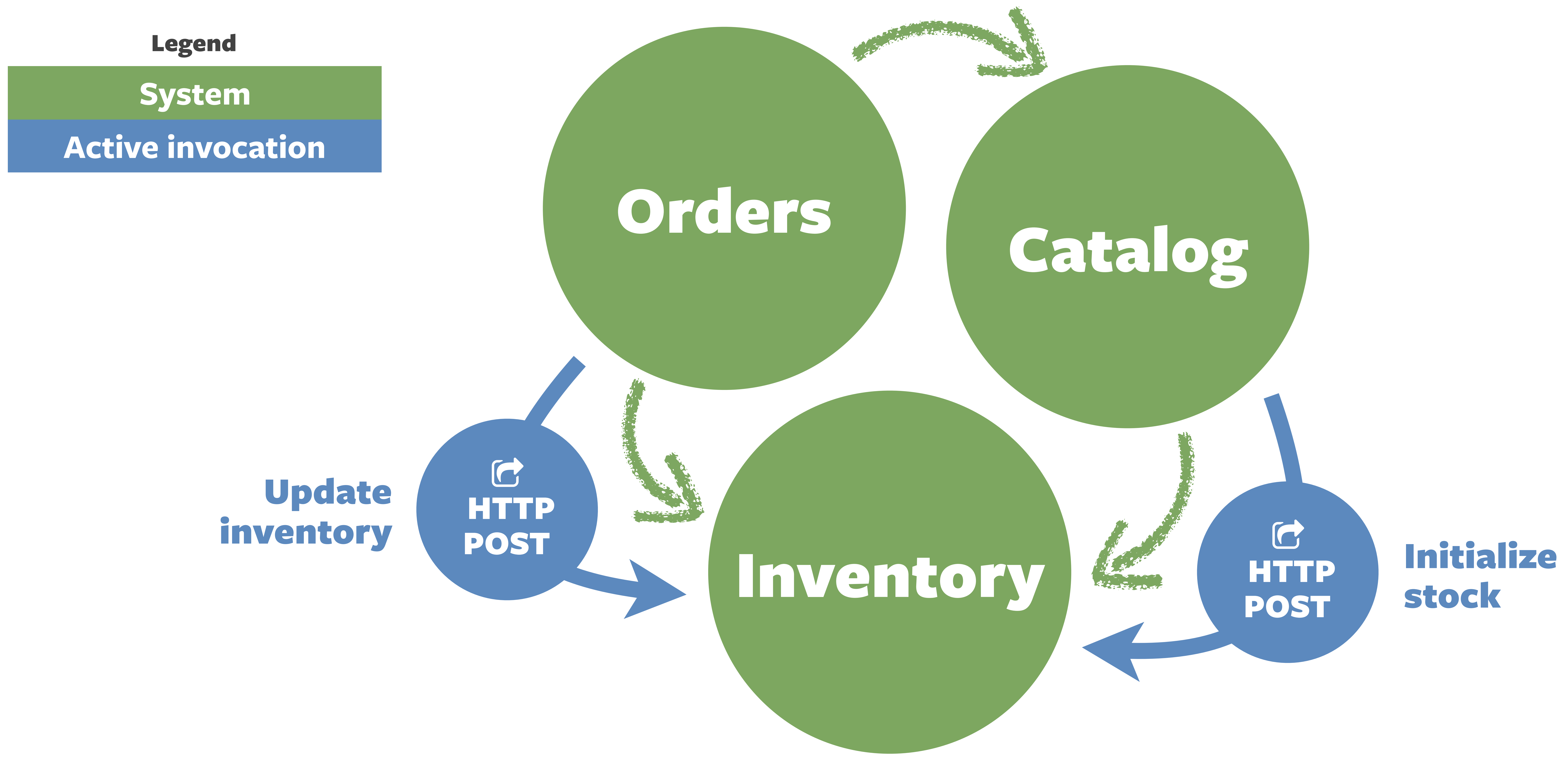
JPA creates incentives to use references to other domain types. This usually leads to code that attempts to change the state of a lot of different entities. In conjunction with `@Transactional` it's very easy to create huge chunks of changes that spread a lot of entities, which seems simple and easy in the first place. The lack of focus on aggregates leads to a lack of structure that significantly serves the erosion of architecture.

The Monolith – Consequences

⊖ **Order management becomes central hub for new features**

The lack of structure and demarcation of different parts usually manifests itself in code that implements key business cases to get bloated over time as a lot of auxiliary functionality being attached to it. In most cases it doesn’t take more than an additional dependency to be declared for injection and the container will hand it into the component. That makes up a convenient development experience but also bears the risk of overloading individual components with too many responsibilities.

The Microlith


Legend
System
Active invocation
Orders
Catalog
Inventory
Update inventory
HTTP POST
Initialize stock
HTTP POST

The Microlith – Problems

⊕ / ⊖ Simple, local transactional consistency is gone

The business transaction that previously could use strong consistency is now spread across multiple systems which means we have two options:

- Stick to strong consistency and use XA transactions and 2PC
  - *„Starbucks doesn't use two-phase commit"* – Gregor Hohpe, 2004
- Switch to embracing eventual consistency and idempotent, compensating actions

⊖ Interaction patterns of the Monolith translated into a distributed system

What had been a local method invocation now involves network communication, serialization etc. usually backed by very RPC-ish HTTP interaction. The newly introduced problems usually solved by adding *more* technology to the picture to implement well-known patterns of remote systems interaction, like bulkheads, retries, fallbacks etc. Typically found technology is Netflix Hystrix, Resilience4j, Spring Cloud modules etc.

The Microlith – Problems

⛔ Remote calls executed while serving user request

As this interaction pattern usually accumulates a lot of latency (especially if the called system calls other systems again) the execution module needs to switch to asynchronous executions and reactive programming, further complicating the picture.

⛔ Individual systems need to know the systems they want to invoke

While the location of the system to be called can be abstracted using DNS and service discovery, systems following that architectural style tend to ignore hypermedia and hard-code resource locations to interact with into client implementations. This creates a rather strong coupling as it limits the servers ability to change it’s APIs.

⛔ Running the system requires upstream systems to be available or mocked

As the invocation of other systems is a fundamental part of the execution of main business logic, these upstream systems need to be available when a system is run. This complicates testing as these systems usually need to be stubbed or mocked.

The Microlith – Problems

⊖ Strong focus on API contracts

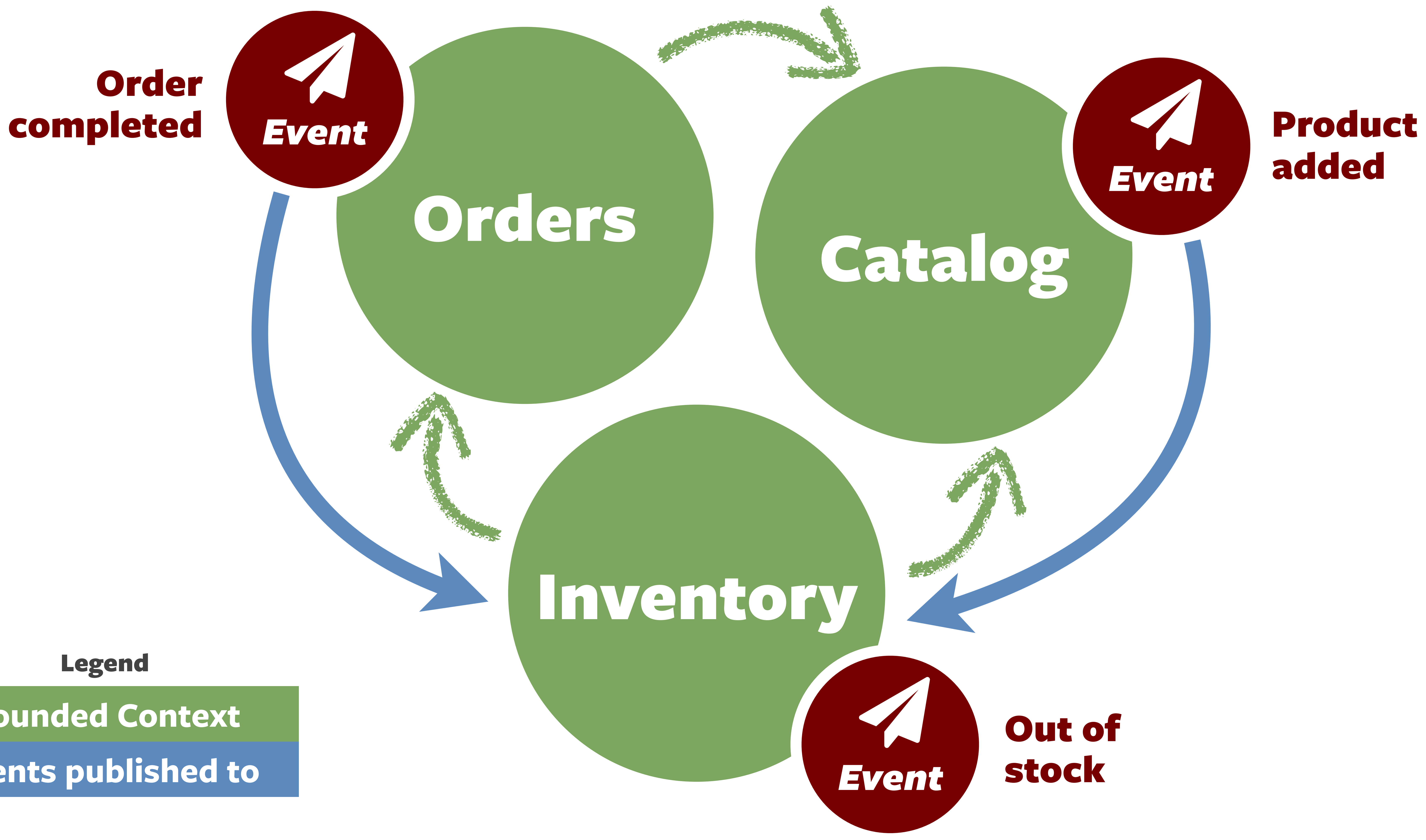
As the interaction pattern between the systems is a 1:1 copy of the one practiced in the monolith, usually the same API definition techniques and practices are used. This usually oversees that this creates the same strong coupling between the communicating parties and evolvability severely suffering as the communicating parties are located at much greater distance than in the monolith.

For reference, see Jim Weirich's talk on Connascence:

„As the distance between software elements increases, use weaker forms of connascence.“

Ignoring that rule produces tightly coupled distributed systems preventing independent evolution of the individual systems, a core goal of a microservice architecture in the first place.

The Modulith


Order completed
Event
Orders
Catalog
Event
Product added
Inventory
Event
Out of stock
Legend
Bounded Context
Events published to

The Modulith – Fundamental differences

✚ Focus of domain logic implementation has moved to the aggregate

The aggregates become the focus point of domain logic. Key state transitions are implemented as methods on the aggregate. Some of them register even dedicated events.

✚ Integration of Bounded Contexts is implemented using events

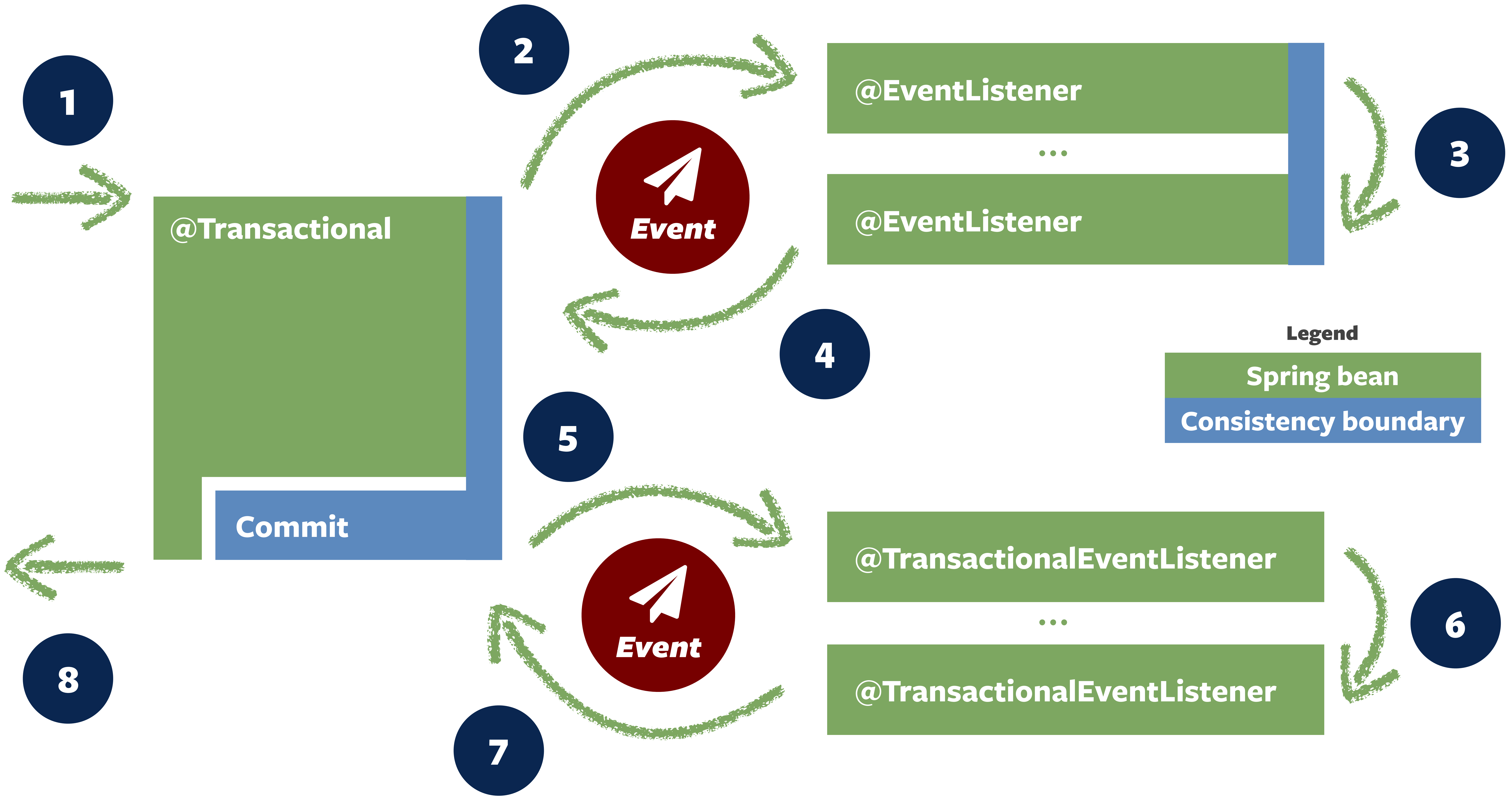
The events produced by an aggregate are automatically published on repository interaction via Spring's application event mechanism. This allows to define event listeners in other interested Bounded Contexts.

✚ Invert invocation dependencies between Bounded Contexts

Previously code within a Bounded Context actively reached out to other contexts and invoked operations that change state within that context. These state transitions can now be triggered by consuming events published by other Bounded Contexts.

Detour:

Events and Consistency


1
2
3
4
5
6
7
8
@Transactional
Commit
Event
Event
@EventListener
...
@EventListener
@TransactionalEventListener
...
@TransactionalEventListener
Legend
Spring bean
Consistency boundary

Application events in a Spring application

1. We enter a transactional method

Business code is executed and might trigger state changes on aggregates.

2. That transactional method produces application events

In case the business code produces application events, standard events are published directly. For each transactional event listener registered a transaction synchronization is registered, so that the event will eventually be published on transaction completion (by default on transaction commit).

3. Event listeners are triggered

By default, event listeners are synchronously invoked, which means they participate in the currently running transactions. This allows listeners to abort the overall transaction and ensure strong consistency. Alternatively, listeners can be executed asynchronously using `@Async`. They then have to take care of their transactional semantics themselves and errors will not break the original transaction.

Application events in a Spring application

4. Service execution proceeds once event delivery is completed

Once all standard event listeners have been invoked, the business logic is executed further. More events can be published, further state changes can be created.

5. The transaction finishes

Once the transactional method is done, the transaction is completed. Usually all pending changes (created by the main business code or the synchronous event listeners) are written to the database. In case inconsistencies or connection problems, the transaction rolls back.

6. Transactional event listeners are triggered

Listeners annotated with `@TransactionalEventListener` are triggered when the transaction commits, which means they can rely on the business operation the event has been triggered from having succeeded. This allows the listeners to read committed data. Listeners can be invoked asynchronously using `@Async` in case the functionality to be invoked might be long-running (e.g. sending an email).

Detour:

Application Events with Spring (Data)

Application events with Spring (Data)

- **Powerful mechanism to publish events in Spring applications**
  - Application event – either a general object or extending `ApplicationEvent`
  - `ApplicationEventPublisher` – injectable to manually invoke event publication

- **Spring Data event support**
  - Spring Data's focus: aggregates and repositories
  - Domain-Driven Design aggregates produce application events
  - `AbstractAggregateRoot<T>` – base class to easily capture events and get them published on `CrudRepository.save(…)` invocations.
  - No dependency to infrastructure APIs

- **Integration with messaging technology via event listeners**

```java
// Super class contains methods with
// @DomainEvents und @AfterDomainEventPublication
class Order extends AbstractAggregateRoot<Order> {

  Order complete() {

    registerEvent(OrderCompletedEvent.of(this));
    return this;
  }
}
```

```java
@Component
class OrderManagement {

  private final OrderRepository orders;

  @Transactional
  void completeOrder(Order order) {
    orders.save(order.complete());
  }
}
```

Application events – Error Scenarios

A synchronous event listener fails

In case a normal event listener fails the entire transaction will roll back. This enables strong consistency between the event producer and the listeners registered but also bears the risk of supporting functionality interfering with the primary one, causing the latter to fail for less important reasons. The tradeoff here could be to move to a transactional event listener and embrace eventual consistency.

An asynchronous event listener fails

The event is lost but the primary functionality can still succeed as the event is handled in a separate thread. Retry mechanisms can (should?) be deployed in case some form of recovery is needed.

Application events – Error Scenarios

The transactional service execution fails

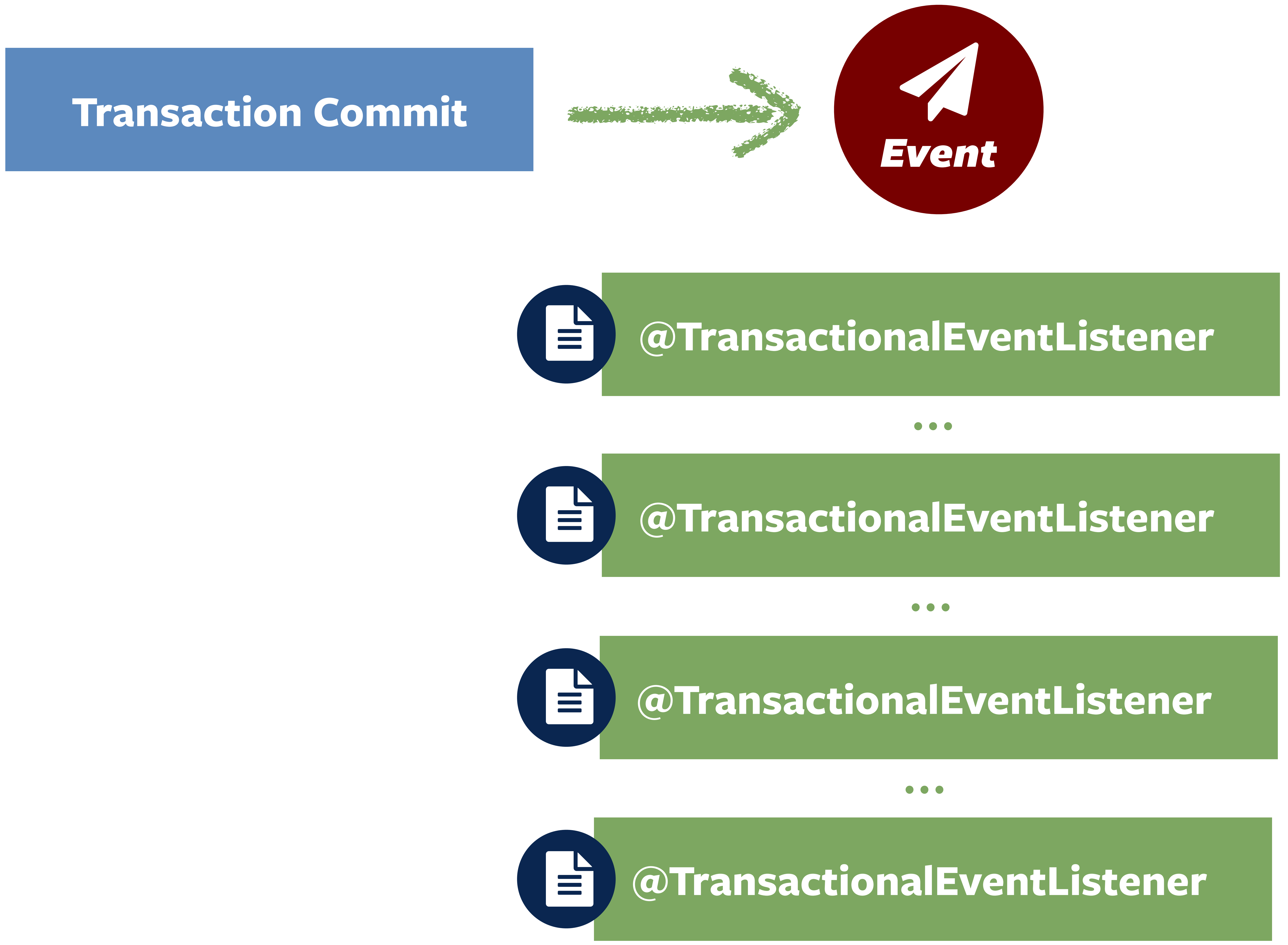
Assuming the event listeners also execute transactional logic, the local transaction is rolled back and the system is still in a strongly consistent state. Transactional event listeners are not invoked in the first place.

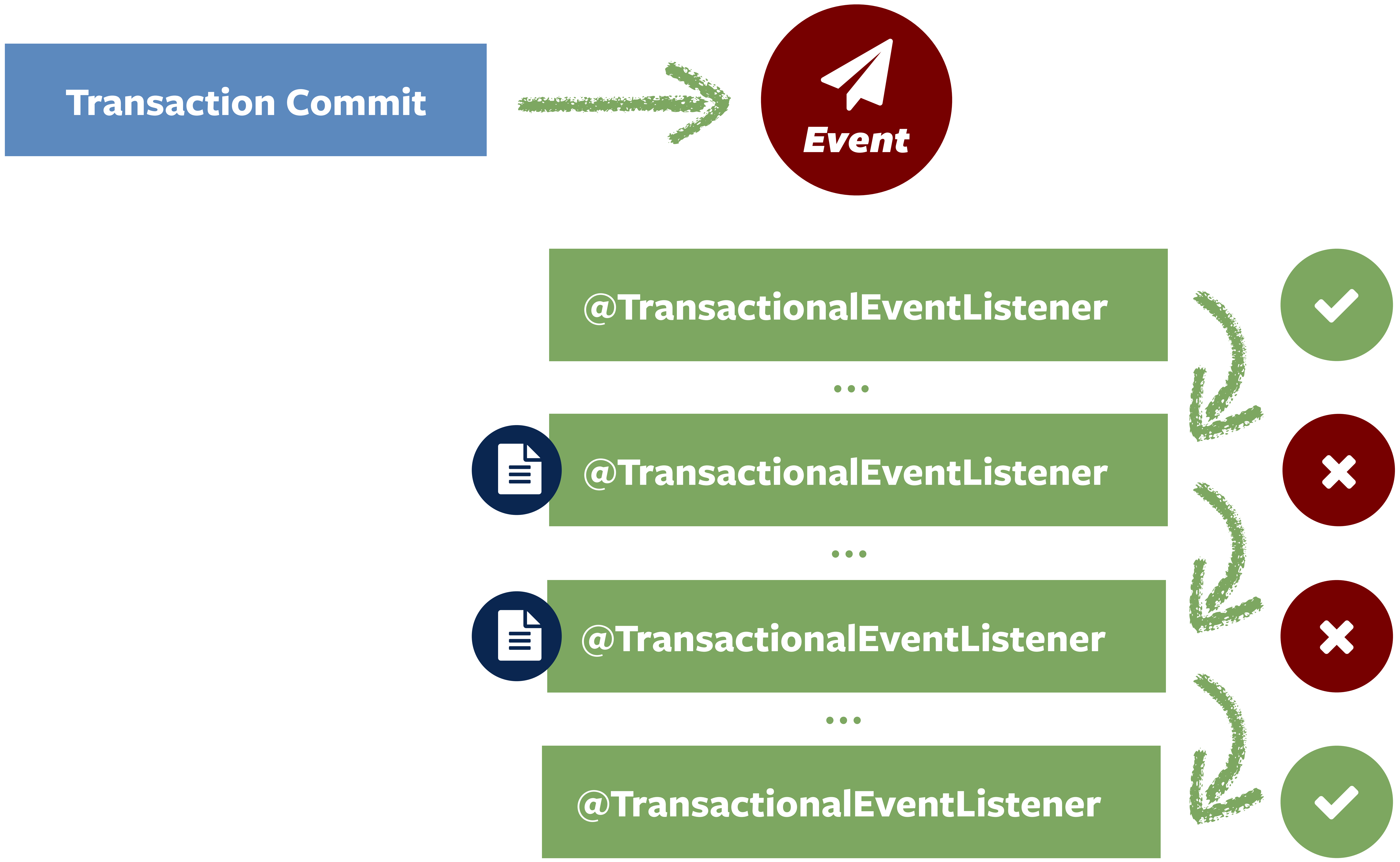
A transactional event listener fails

In case a transactional event lister fails or the application crashes while transactional event listeners are executed, the event is lost and functionality might not have been invoked.

Detour:

Event Publication Registry


Transaction Commit
Event
@TransactionalEventListener
…
@TransactionalEventListener
…
@TransactionalEventListener
…
@TransactionalEventListener


Transaction Commit
Event
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener
...
@TransactionalEventListener

Event Publication Registry

1. Write application event publication log for transactional listeners

On application event publication a log entry is written for every event and transactional event listener interested in it. That way, the transaction remembers which events have to be properly handled and in case listener invocations fail or the application crashes events can be re-published.

2. Transaction listeners are decorated to register successful completion

Transactional event listeners are decorated with an interceptor that marks the log entry for the listener invocation on successful listener completion. When all listeners were handled, the log only contains publication logs for the ones that failed.

3. Incomplete publications can be retried

Either periodically or during application restarts.

Sample code

https://github.com/olivergierke/spring-domain-events

Summary

Events for Bounded Context interaction

Spring’s application events are a very light-weight way to implement those domain events. Spring Data helps to easily expose them from aggregate roots. The overall pattern allows loosely coupled interaction between Bounded Contexts so that the system can be extended and evolved easily.

Externalize events if needed

Depending on the integration mechanism that’s been selected we can now write separate components to translate those JVM internal events into the technology of choice (JMS, AMQP, Kafka) to notify third-party systems.

The System of Systems

Messaging

REST

Integration options

“

What is needed for a single system to run?

“ **_What happens if a system goes down?_**

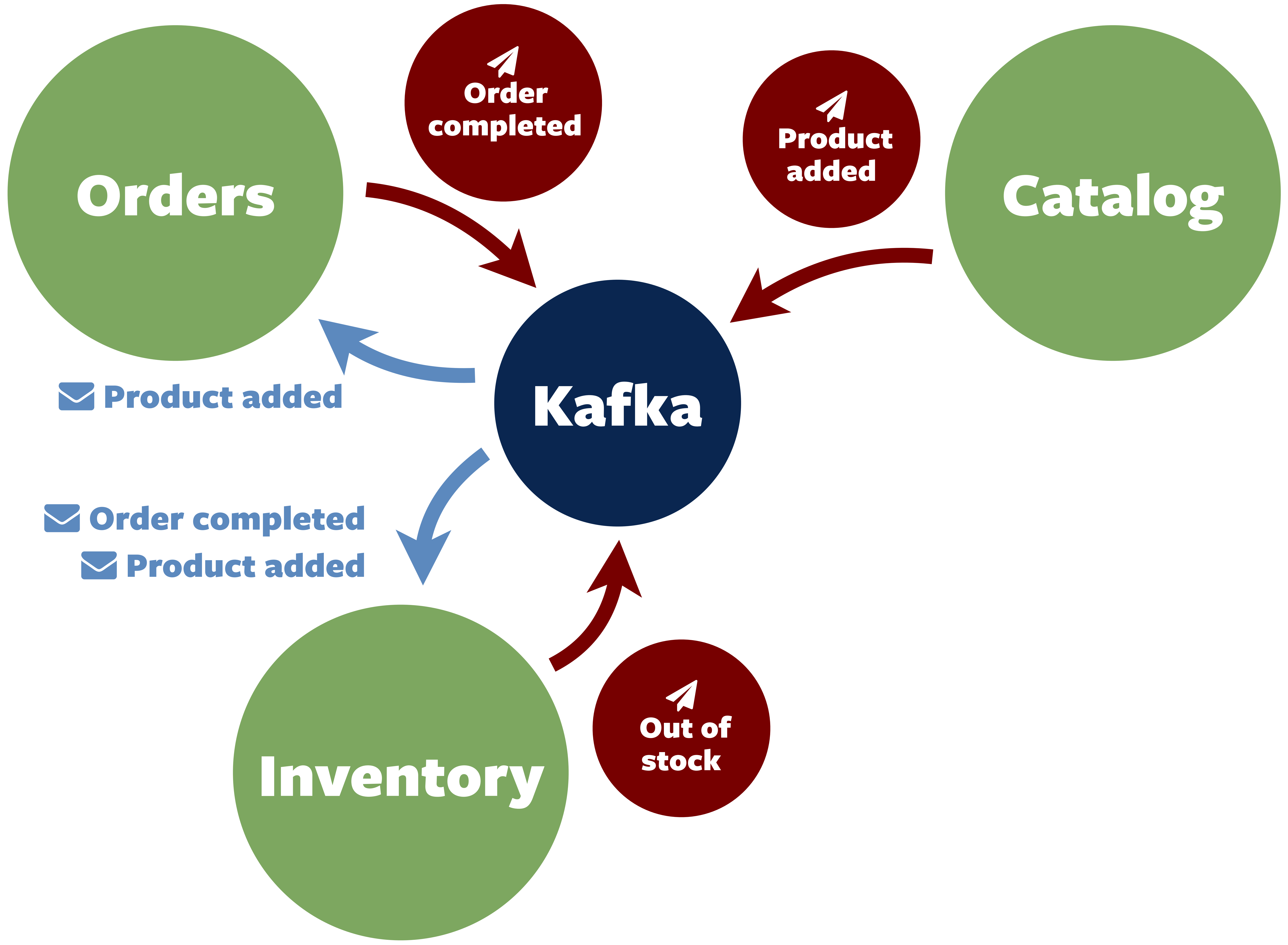
“

What happens if a failed system comes up again?

“ ***What happens if a new system enters the scene?***

The System of Systems

Messaging


Order completed
Product added
Orders
Catalog
Kafka
Product added
Order completed
Product added
Inventory
Out of stock

Demo

- **Start broker**
- **Start individual services**
  - Show HAL browser, APIs
- **Show systems interaction**
  - Add Product -> show InventoryItem and ProductInfo being created
  - Trigger shipment -> show amount in InventoryItem increasing
  - Trigger order creation -> show amount in InventoryItem decreasing
  - Trigger further order creations -> show Inventory publishing OutOfStock event

Key characteristics

Integration via a central broker

- Shared infrastructure
- Some business decisions (TTL of events) in shared component
- Broker knows about all messages of all systems (potentially forever)
- Technology built for scale

Events published as messages

Key characteristics

❗ Different broker technologies have different replay characteristics

- JMS — durable subscription (requires initial registration with the broker)
- AMQP — fanout exchanges (requires initial registration with the broker)
- Kafka — topics / partitions, log compaction

❗ Coupling via message serialization format

Key characteristics

❗ Different broker technologies have different replay characteristics

- JMS — durable subscription (requires initial registration with the broker)
- AMQP — fanout exchanges (requires initial registration with the broker)
- Kafka — topics / partitions, log compaction

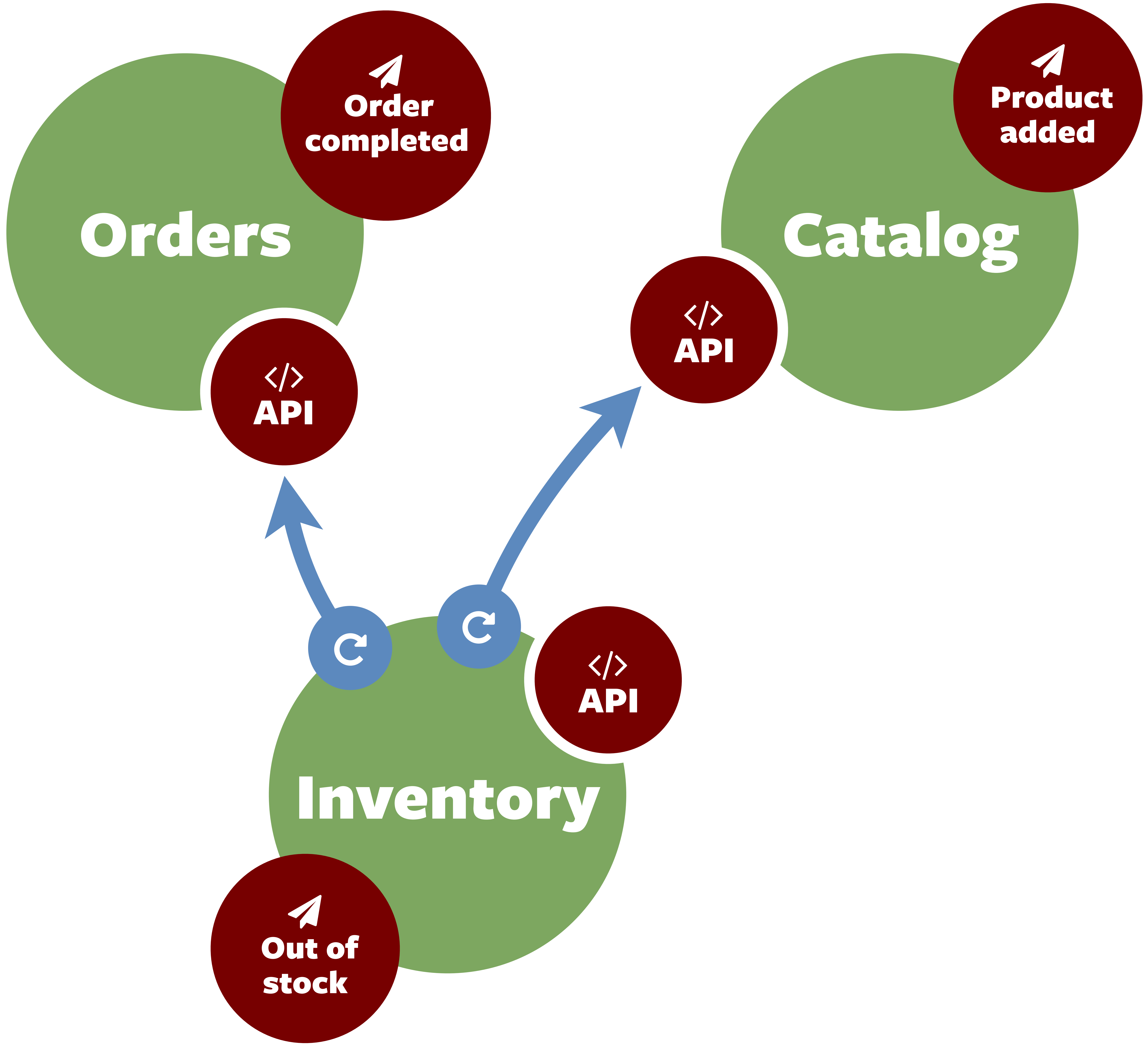
❗ Coupling via message serialization format

❗ Transactional semantics

- 2PC or compensating messages

The System of Systems

</> REST


Order completed
Orders
API
Product added
Catalog
API
API
Inventory
Out of stock

Event publication via HTTP resources

HTTP resources for events

Events are considered application state and expose HTTP resources for client consumption

Typical design aspects

- Collection resource filterable by:
  - Event type — as a replacement for topics
  - Publication date after — to see event
- Pagination — to allow clients to define the pace at which they want to see events
- Caching & conditional requests — to avoid load on the application

Typical media types used

- Atom feeds (XML)
- HAL
- Collection/JSON

Event consumption via polling

Clients regularly poll event resources

Clients interested in events of other systems discover producing system and events resource

Client under control of the consistency gap

- Trade polling frequency over

Typical design aspects

- Low coupling through service- and resource discovery
  - Focus on link relations and URI templates, `http://…/events{?since,type}`
- Polling frequency as key actuator for integration
  - Back-off strategies if the remote system is unavailable
  - Purposely bigger consistency window in case of heavy load

Key characteristics

➕ No (additional) centralized infrastructure component needed

We don't need to connect to a central, shared resource to run the system. This is eases testing. Also, HTTP interaction is usually well understood and already used in the system anyway. A lot of HTTP based technology available to help constructing the overall system (caches, proxies etc.)

➕ Event publication is part of a local transaction

Event publication does not involve interaction with an additional resource. It's basically an additional row in the database table.

➕ Publishing systems controls event lifecycle / security

The publishing system completely controls the lifecycle of the events published. Changes in e.g. the TTL do not involve reconfiguration of infrastructure. Security implications (who is allowed to see which events?) are handled on the API level.

Key characteristics

⊕ Events stay with the publishing system

As events are application state, there's no single component in the overall system that can get flooded by one system potentially flooding the overall system with events.

⊖ Bigger consistency gap because of polling

The pull-model of course creates a bigger consistency gap than message listener invocation. As systems have to cope with eventual consistency anyway, this might not be a big problem.

⊖ Doesn't scale too well for high-volume event publications

In case of a lot of events per single aggregate, the aforementioned consistency gap might be unacceptable.

The System of Systems

Summary

Key aspects

- **Limited remote interaction**
  - „I like monoliths so much that I'd like to build many of them." — Stefan Tilkov
- **Separation of user requests and state synchronization**
  - Data duplication to avoid the need to synchronously reach out to 3rd-party systems
  - Implicit anti-corruption layer to map models

Resources

Resources

Self-Contained Systems

- https://scs-architecture.org

Connascence

- https://en.wikipedia.org/wiki/Connascence_(computer_programming)

The Grand Unified Theory – Jim Weirich, 2009

- https://www.youtube.com/watch?v=NLT7Qcn_PmI

Software Architecture for Developers – Simon Brown

- https://leanpub.com/b/software-architecture

Starbucks doesn’t use two-phase commit – Gregor Hohpe, 2004

- http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

Resources – Domain-Driven Design

Domain-Driven Design – Eric Evans, 2003

- https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215

Implementing Domain-Driven Design – Vaughn Vernon, 2013

- https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/

Domain-Driven Design Distilled – Vaughn Vernon, 2016

- https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420
- https://www.amazon.de/Domain-Driven-Design-kompakt-Vaughn-Vernon/dp/3864904390