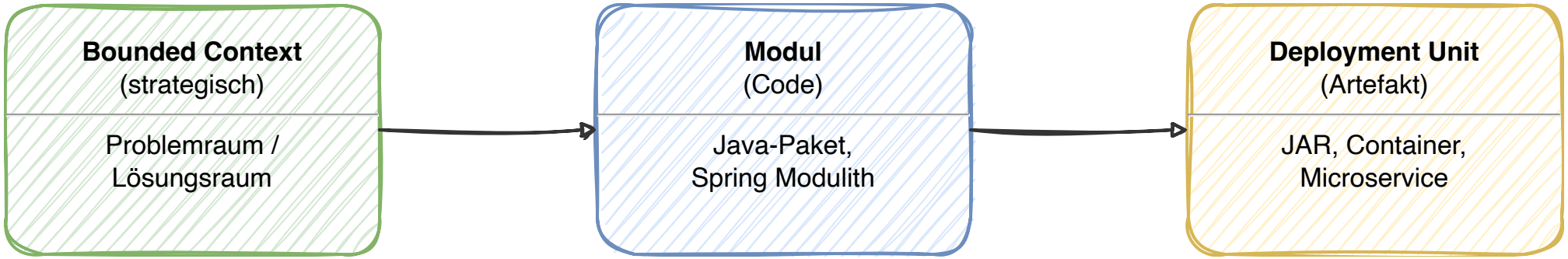
Modul 13 - Business Components & Spring Modulith

Vom Bounded Context zum modularen Monolithen

Lernziele

- Progression von Bounded Context zu Modul zu Deployment Unit verstehen
- Modular Monolith vs. Microservices bewerten
- Spring Modulith einsetzen: Module, Verifikation, Events
- Domain Events mit dem Event Collection Pattern publizieren
- Inter-Modul-Kommunikation event-basiert gestalten
- ArchUnit vs. Spring Modulith: Wann welches Werkzeug?

Von Bounded Context zu Deployment Unit



Von Bounded Context zu Deployment Unit - Übersicht

Ebene	Beschreibung	Beispiel
Bounded Context	Fachliche Grenze aus DDD	Vermittlung, Kontaktmanagement
Modul	Code-Organisation im Monolithen	`de.realestate.brokerage`
Deployment Unit	Auslieferbare Einheit	JAR, Docker Container

> Ein Bounded Context kann als Modul starten und später zum Microservice werden.

Modular Monolith vs. Microservices

Aspekt	Modular Monolith	Microservices
Deployment	Ein Artefakt	Viele Artefakte
Kommunikation	In-Process (Methodenaufruf)	Netzwerk (HTTP, Messaging)
Konsistenz	ACID-Transaktionen möglich	Eventual Consistency
Ops-Komplexität	Gering	Hoch (Monitoring, Tracing, Mesh)
Team-Autonomie	Geringer	Höher
Skalierung	Gesamtes System	Einzelne Services
Refactoring	Einfach (gleicher Prozess)	Schwer (Schnittstellen fixiert)
Fehlersuche	Stack Trace	Distributed Tracing

Monolith First (Martin Fowler)

> *"Almost all the successful microservice stories have started with a monolith that got too big and was broken up."*
>
> - Martin Fowler

Der pragmatische Weg

```
1. Starten       →  Gut strukturierter Monolith (Module = BCs)
2. Grenzen       →  Saubere Schnittstellen zwischen Modulen
3. Events        →  Inter-Modul-Kommunikation über Domain Events
4. Beobachten    →  Wo entsteht ein konkreter Engpass?
5. Extrahieren   →  Einzelnes Modul → Microservice (bei Bedarf)
```

Warum nicht sofort Microservices?

- Distributed Monolith ist schlimmer als ein Monolith
- BC-Grenzen sind anfangs oft falsch geschnitten
- Im Monolithen lassen sich Grenzen leichter verschieben
- Microservices lohnen sich erfahrungsgemäß ab ca. 5+ Teams

Spring Modulith - Einführung

- Spring-Projekt für modulare Monolithen (1.0 seit Spring Boot 3.1)
- Baut auf Spring Boot auf - keine neue Runtime

Features

Feature	Beschreibung
Modul-Konventionen	Top-Level-Packages = Module
Modul-Verifikation	Abhängigkeiten automatisch prüfen
Event-Kommunikation	`ApplicationEventPublisher` + Registry
Dokumentation	Modulstruktur-Diagramme generieren
Observability	Modul-übergreifendes Tracing

Maven-Dependency

```
<dependencies>
    <dependency>
        <groupId>org.springframework.modulith</groupId>
        <artifactId>spring-modulith-starter-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.modulith</groupId>
        <artifactId>spring-modulith-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Modul-Struktur-Konventionen

```
de.realestate                                ← @SpringBootApplication
├── brokerage/                                 ← Modul "Vermittlung"
│   ├── ViewingScheduledEvent.java                 (public → Event-API)
│   ├── adapter/                                   (Subpaket → intern)
│   │   └── web/
│   ├── application/
│   │   └── service/
│   ├── domain/
│   │   └── model/
│   └── infrastructure/
├── contact/                                   ← Modul "Kontaktmanagement"
└── RealEstateCrmApplication.java
```

- Top-Level-Packages unter der Hauptklasse = Module
- Alle Subpakete (adapter, application, domain, …) = modulintern
- Nur Klassen im Root des Moduls = öffentliche API
- Events gehören zur öffentlichen API eines Moduls

> Spring Modulith erkennt sowohl `internal/` als auch beliebige Subpakete als intern.
> Eine hexagonale Paketstruktur funktioniert genauso.

@ApplicationModule

```
// brokerage/package-info.java
@ApplicationModule(
    allowedDependencies = {
        "contact",
        "shared"
    }
)
package de.realestate.brokerage;
```

- Deklariert explizit, welche anderen Module referenziert werden dürfen
- Spring Modulith prüft, ob nur erlaubte Module verwendet werden
- Verhindert ungewollte Kopplungen zwischen Bounded Contexts

> Ohne `allowedDependencies` sind alle Module erlaubt - explizite Deklaration ist empfohlen.

ArchUnit vs. Spring Modulith - Vergleich

Aspekt	ArchUnit	Spring Modulith
Granularität	Fein (Package, Klasse, Annotation)	Modul-Ebene
Konfiguration	Code (Fluent API)	Konvention + `@ApplicationModule`
Regeln	Selbst definiert	Automatisch aus Konventionen
Events	Nicht relevant	Event-Publishing + Registry
Dokumentation	Nein	Ja (Diagramme generieren)
Spring-Kontext	Nicht nötig	Ja (für Verifikation)
Einsatz	Schicht-Regeln innerhalb eines BC	Modul-Grenzen zwischen BCs

> Empfehlung: Beide kombinieren!
> ArchUnit für feine Schicht-Regeln, Spring Modulith für Modul-Grenzen.

Modul-Verifikation als Test

```
class ModulithStructureTest {

    ApplicationModules modules =
        ApplicationModules.of(RealEstateCrmApplication.class);

    @Test
    void verifyModuleStructure() {
        // Checks: no unauthorized access between modules
        // Checks: no access to 'internal' packages from outside
        modules.verify();
    }

    @Test
    void documentModuleStructure() {
        // Generates PlantUML diagrams in target/modulith-docs/
        new Documenter(modules)
            .writeDocumentation();
    }
}
```

- `verify()` prüft alle Modul-Grenzen automatisch
- `Documenter` generiert Abhängigkeitsdiagramme als PlantUML
- Läuft als normaler JUnit-Test in der CI/CD-Pipeline

Domain Events publizieren - Event Collection Pattern

Schritt 1: Aggregate sammelt Events

```
// domain.model (no Spring!)
public class BrokerageProcess {
    private final List<Object> domainEvents = new ArrayList<>();

    public ViewingId scheduleViewing(ContactId prospect,
                                     LocalDateTime appointmentDate) {
        var viewing = new Viewing(
            ViewingId.generate(), prospect, appointmentDate);
        viewings.add(viewing);

        domainEvents.add(new ViewingScheduledEvent(
            id, viewing.getId(), prospect, appointmentDate));

        return viewing.getId();
    }

    public List<Object> domainEvents() {
        return Collections.unmodifiableList(domainEvents);
    }

    public void clearDomainEvents() { domainEvents.clear(); }
}
```

Domain Events publizieren - Event Collection Pattern

Schritt 2: Application Service dispatched Events nach dem Speichern

```
@Service
public class ScheduleViewingService implements ScheduleViewing {

    private final BrokerageProcessRepository repository;
    private final ApplicationEventPublisher eventPublisher;

    public ScheduleViewingService(
            BrokerageProcessRepository repository,
            ApplicationEventPublisher eventPublisher) {
        this.repository = repository;
        this.eventPublisher = eventPublisher;
    }

    @Transactional
    @Override
    public ViewingId schedule(ScheduleViewingCommand cmd) {
        var process = repository.findById(cmd.processId())
            .orElseThrow(() -> new ProcessNotFound(cmd.processId()));
        var id = process.scheduleViewing(cmd.prospectId(), cmd.appointmentDate());
        repository.save(process);
        process.domainEvents().forEach(eventPublisher::publishEvent);
        process.clearDomainEvents();
        return id;
    }
}
```

> Der `ApplicationEventPublisher` ist hier akzeptabel - er ist ein Spring-Interface in der Application-Schicht, nicht in der Domain.

Domain Event als Record - Öffentliche Modul-API

```
// Located in the module root (NOT in internal/) → public API
package de.realestate.brokerage;

public record ViewingScheduledEvent(
    UUID processId,
    UUID viewingId,
    UUID prospectId,
    LocalDateTime appointmentDate
) {}
```

- Events verwenden einfache, standardisierte Typen (UUID, String), keine Value Objects
- Andere Module sollen keine Domain-internen Typen kennen
- Events sind immutable - Records eignen sich perfekt
- Lose Kopplung: Sender kennt die Empfänger nicht

Events konsumieren - @EventListener

```
// In a different module: contact
package de.realestate.contact.internal;

@Component
class ViewingNotification {

    @EventListener
    public void onViewingScheduled(ViewingScheduledEvent event) {
        log.info("New viewing for process {} on {}",
            event.processId(), event.appointmentDate());
        // Notify prospect via email
    }
}
```

- `@EventListener` = synchrone Verarbeitung
- Läuft in der gleichen Transaktion wie der Publisher
- Fehler im Listener → Rollback der gesamten Transaktion

> Verwende `@EventListener` nur, wenn der Listener Teil der gleichen Business-Transaktion sein soll.

@TransactionalEventListener - Side Effects

```
@Component
class ViewingNotification {

    @TransactionalEventListener(phase = AFTER_COMMIT)
    public void onViewingScheduled(ViewingScheduledEvent event) {
        // Only executed AFTER successful commit
        emailService.sendInvitation(event.prospectId(), event.appointmentDate());
    }
}
```

Phase	Wann?	Use Case
`AFTER_COMMIT`	Nach erfolgreichem Commit	E-Mail, Notification
`AFTER_ROLLBACK`	Nach Rollback	Fehler-Logging
`AFTER_COMPLETION`	Immer nach Abschluss	Cleanup

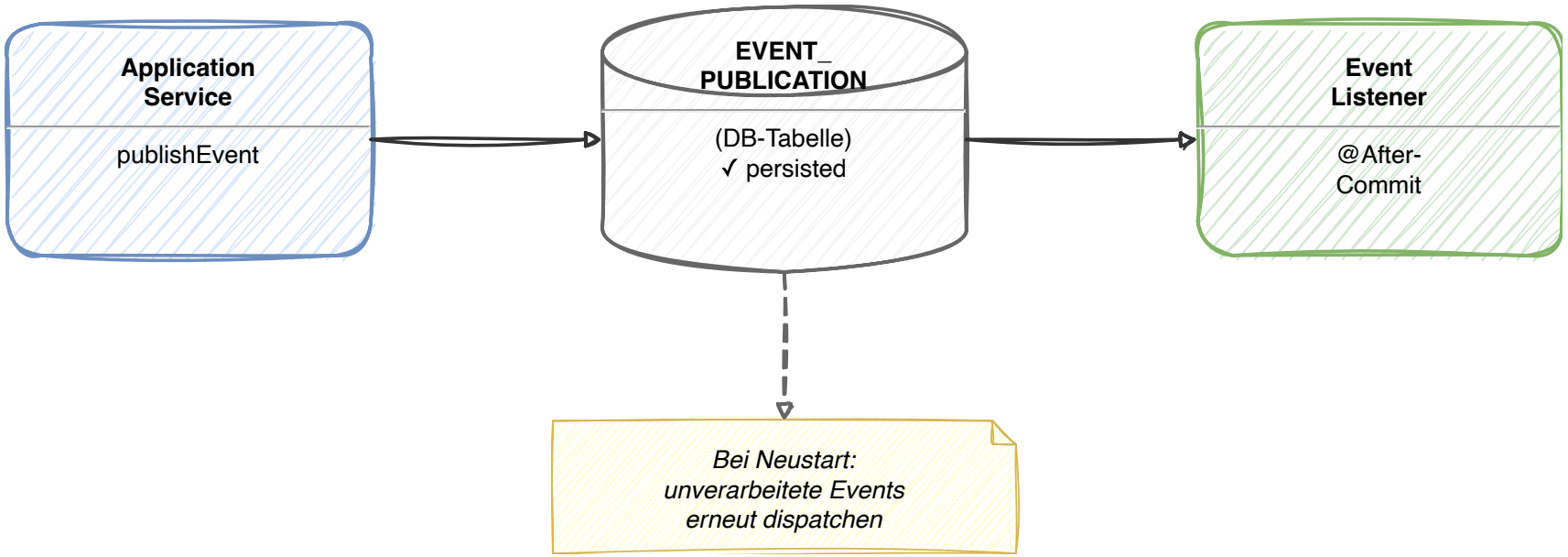
> Empfehlung: `AFTER_COMMIT` für alle Side Effects (E-Mail, Benachrichtigung, externe API-Calls).

Event Publication Registry - At-Least-Once Delivery

Problem: Was passiert, wenn der Listener nach dem Commit abstürzt?

```
<!-- Enable Event Publication Registry -->
<dependency>
    <groupId>org.springframework.modulith</groupId>
    <artifactId>spring-modulith-events-jdbc</artifactId>
</dependency>
```

Problem: Was passiert, wenn der Listener nach dem Commit abstürzt?



- Events werden in der gleichen Transaktion in eine DB-Tabelle geschrieben
- Nach Verarbeitung: Event wird als completed markiert
- Bei Crash: unvollständige Events werden beim Neustart erneut dispatched

@Async - Asynchrone Event-Verarbeitung

```
@Component
class ViewingStatistics {

    @Async
    @TransactionalEventListener(phase = AFTER_COMMIT)
    public void onViewingScheduled(ViewingScheduledEvent event) {
        // Runs in its own thread, its own transaction
        statisticsService.recordViewing(event);
    }
}
```

Voraussetzung: `@EnableAsync`

```
@SpringBootApplication
@EnableAsync
public class RealEstateCrmApplication { }
```

- `@Async` + `@TransactionalEventListener` = eigener Thread-Pool, eigene Transaktion
- Fehler beeinflussen den Publisher nicht
- Ohne Event Publication Registry: fire-and-forget
- Mit Registry: fehlgeschlagene Events werden beim Neustart erneut verarbeitet

@Externalized - Events nach außen leiten

```java
@Externalized("viewings::#{#this.viewingId()}")
public record ViewingScheduledEvent(
    UUID processId,
    UUID viewingId,
    UUID prospectId,
    LocalDateTime appointmentDate
) {}
```

```xml
<!-- e.g. Kafka integration -->
<dependency>
    <groupId>org.springframework.modulith</groupId>
    <artifactId>spring-modulith-events-kafka</artifactId>
</dependency>
```

- `@Externalized` leitet Events an Messaging-Infrastruktur weiter
- Unterstützt: Kafka, RabbitMQ, Amazon SQS, JMS
- Routing Key über SpEL definierbar
- Vorbereitung für spätere Microservice-Extraktion

Wann einen Microservice extrahieren?

Extrahieren bei konkretem Grund

- Unabhängiges Deployment: Team will unabhängig deployen
- Skalierung: Ein Modul hat andere Lastanforderungen
- Technologie: Ein Modul braucht einen anderen Tech-Stack
- Organisatorisch: Eigenes Team, eigener Lifecycle

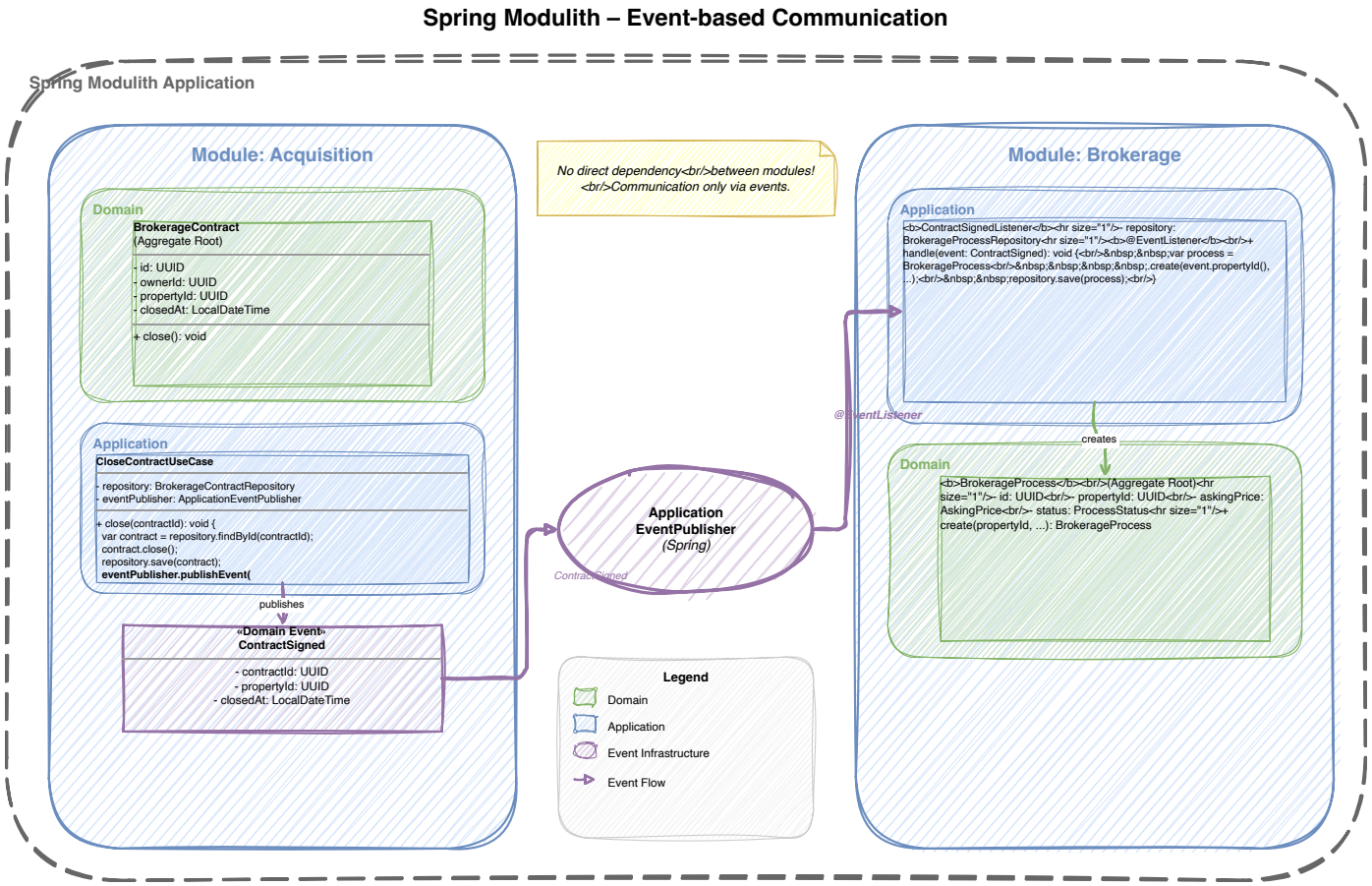
NICHT extrahieren wegen

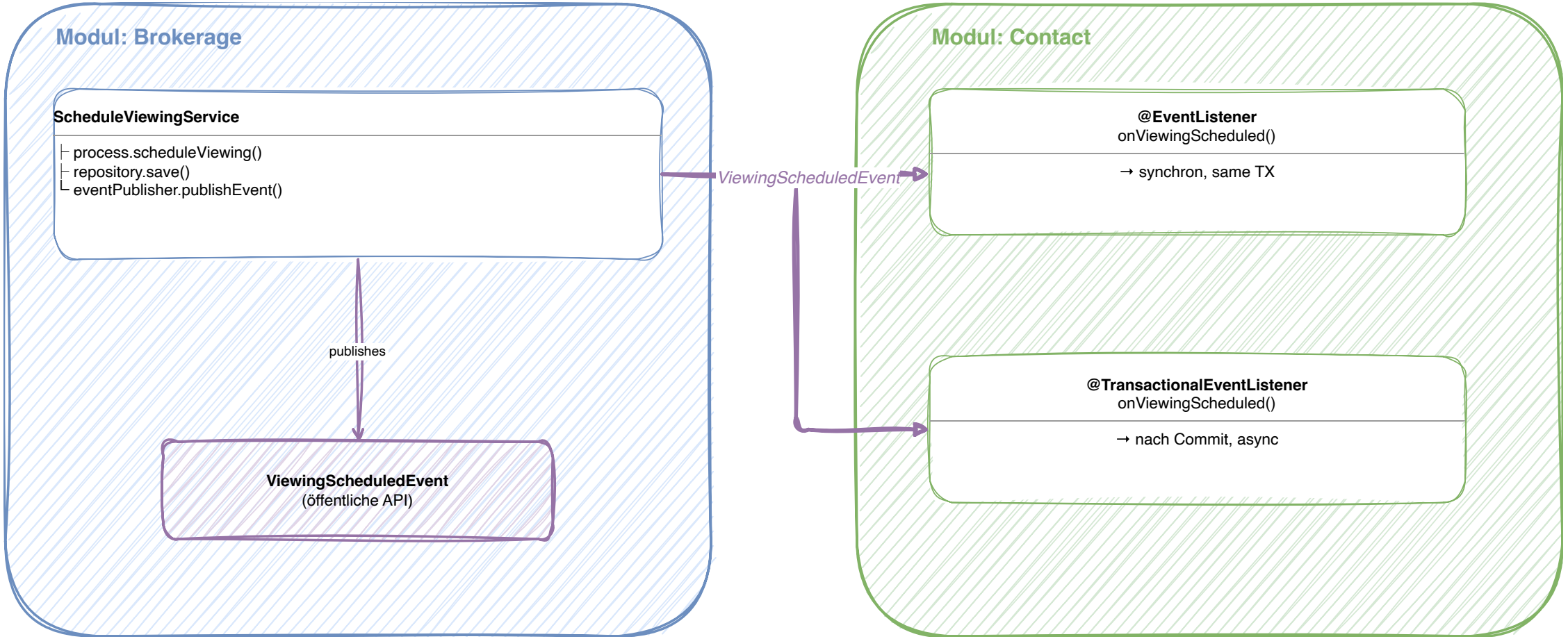
- "Das machen alle so" / Microservice-Hype
- Vermutete zukünftige Anforderungen
- "Microservices sind moderner"

Extraktionsstrategie

```
1. Modul im Monolith sauber geschnitten  → Spring Modulith verify() [OK]
2. Events für Kommunikation              → @Externalized vorbereitet
3. API definiert                         → REST / gRPC Schnittstelle
4. Eigene Datenbank                      → Shared DB auflösen
5. Eigenes Deployment                    → Container, CI/CD Pipeline
```

Spring Modulith Events - Übersicht





Zusammenfassung

- Bounded Context → Modul → Deployment Unit: schrittweise Progression
- Monolith First: Starten mit modularem Monolith, extrahieren bei Bedarf
- Spring Modulith: Module, Verifikation, Event-basierte Kommunikation
- Event Collection Pattern: Domain sammelt Events, Application dispatched
- `@EventListener` für synchrone, `@TransactionalEventListener` für Side Effects
- Event Publication Registry für At-Least-Once Delivery
- `@Externalized` als Brücke zur Messaging-Infrastruktur
- ArchUnit für Schicht-Regeln, Spring Modulith für Modul-Grenzen
- Microservice-Extraktion nur mit konkretem Grund

Integration-Tests mit der Scenario-API

```
@ApplicationModuleTest
class BrokerageIntegrationTest {

    @Test
    void viewingScheduledTriggersNotification(Scenario scenario) {
        scenario.publish(new ViewingScheduledEvent(
                processId, viewingId, prospectId, LocalDateTime.now()))
            .andWaitForEventOfType(NotificationSentEvent.class)
            .toArriveAndVerify(event ->
                assertThat(event.prospectId()).isEqualTo(prospectId));
    }
}
```

- `Scenario` testet event-basierte Interaktionen zwischen Modulen
- Wartet asynchron auf Folge-Events - kein `Thread.sleep()` nötig
- Ersetzt komplexe Mocking-Setups für Modul-Kommunikation

Hands-on: Lab-10

Aufgabe

Strukturiert das Immobilien-CRM mit Spring Modulith:

1. Module nach Bounded Contexts schneiden (Top-Level-Packages)
2. `package-info.java` mit `@ApplicationModule` anlegen
3. Modul-Verifikationstest schreiben (`modules.verify()`)
4. Domain Event mit Event Collection Pattern publizieren
5. Event in einem anderen Modul mit `@TransactionalEventListener` konsumieren
6. Event Publication Registry aktivieren (JDBC)

> Dauer: ca. 45 Minuten
> Details und Aufgabenstellung im Lab-10

Diskussion

> Modulith oder Microservices - was passt zu eurem Kontext?

- Wie groß ist euer Team / eure Organisation?
- Wie oft müsst ihr einzelne Teile unabhängig deployen?
- Habt ihr die Ops-Kapazität für Microservices (Monitoring, Tracing, Service Mesh)?
- Welche Erfahrungen habt ihr mit verteilten Systemen (Eventual Consistency, Partial Failures)?
- Wo liegen eure Bounded-Context-Grenzen aktuell?
- Könntet ihr Events zwischen euren Modulen identifizieren?