

Aufgabe Makroarchitektur Teil 1 (Modulith + Theorie)

Theorie: Recherchiere zu folgenden Fragestellungen und fasse deine Erkenntnisse übersichtlich und illustrativ zusammen!

1. Was ist Softwarearchitektur? Softwarearchitektur bezeichnet die grundlegende Struktur einer Software, einschließlich der Art und Weise, wie diese Struktur ihre Verhaltensaspekte behandelt. Sie umfasst die Funktionen und Eigenschaften, die nützlich sind, wenn das Softwareprodukt ein System bildet oder ein Systemteil ist. Es handelt sich dabei um die Definition der Systemkomponenten, ihrer externen Eigenschaften und ihrer Beziehungen zueinander und zur Umgebung. Die Softwarearchitektur ist entscheidend für das Verständnis, wie ein System oder eine Softwareanwendung funktioniert. Sie bietet einen Plan und ein koordiniertes Regelwerk, das dazu beiträgt, die Komplexität eines Systems zu bewältigen. Hierbei handelt es sich um Regeln, die bei der Konstruktion des Systems helfen, etwa um Standards und Richtlinien.



(https://www.microconsult.de/blog/2022/07/tb_design-for-test/)

Im Allgemeinen kann die Softwarearchitektur verschiedene Sichten oder Perspektiven umfassen, darunter:

- Strukturansicht: Wie ist die Software in Module oder Komponenten unterteilt und wie interagieren diese miteinander?
- Verhaltensansicht: Wie verhalten sich die einzelnen Komponenten und das Gesamtsystem in Bezug auf bestimmte Eingaben oder Ereignisse?
- Praktische Ansicht: Wie wird die Software in der realen Welt eingesetzt, wie kann sie gewartet und erweitert werden, und welche nicht-funktionalen Anforderungen (z. B. Leistung, Sicherheit,

Zuverlässigkeit) muss sie erfüllen?

Die Softwarearchitektur dient auch als Kommunikationsmittel zwischen den verschiedenen Stakeholdern (wie Entwicklern, Projektmanagern, Kunden usw.), da sie einen gemeinsamen Rahmen für das Verständnis der wichtigsten Designentscheidungen bietet. Sie unterstützt zudem die Planung und Abstimmung von Aufgaben während des gesamten Softwareentwicklungslebenszyklus.

2. Wie kann man Softwarearchitektur dokumentieren? Die Dokumentation der Softwarearchitektur ist ein wichtiger Teil des Entwicklungsprozesses. Sie dient als Leitfaden für das Entwicklerteam und ermöglicht eine bessere Kommunikation mit Stakeholdern. Sie kann auch dazu beitragen, die Qualität der Software zu verbessern und die Wartung zu erleichtern. Es gibt verschiedene Ansätze zur Dokumentation der Softwarearchitektur, aber hier sind einige grundlegende Schritte und Methoden, die oft verwendet werden:

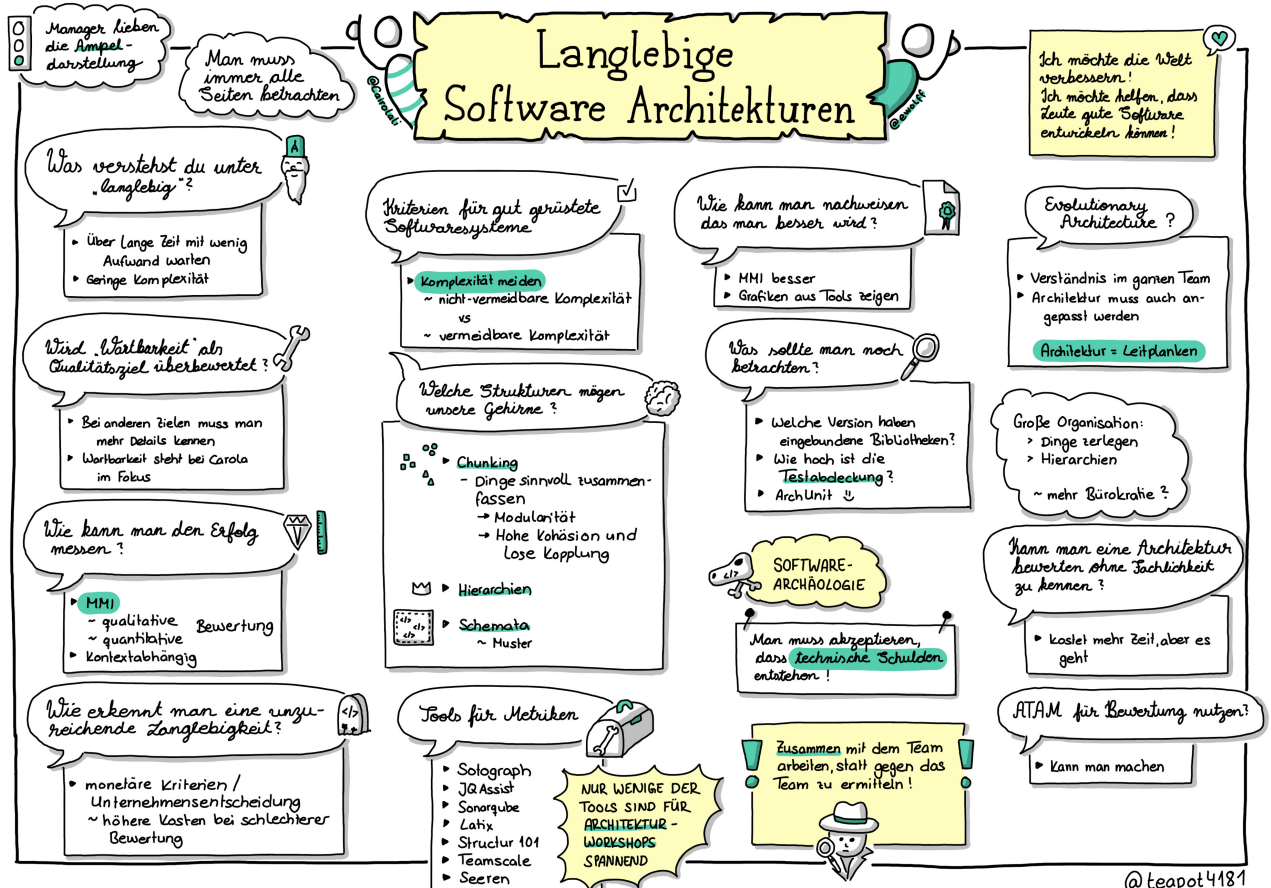
- Architekturbeschreibung: Der erste Schritt besteht darin, eine Übersicht über die Architektur zu erstellen. Dies sollte eine klare und prägnante Beschreibung der Gesamtstruktur und der einzelnen Komponenten umfassen, einschließlich ihrer Beziehungen zueinander und zur externen Umgebung.
- Diagramme: Visuelle Darstellungen können oft hilfreich sein, um die Struktur und das Verhalten der Software zu veranschaulichen. Dazu können UML-Diagramme (Unified Modeling Language) wie Klassendiagramme, Sequenzdiagramme und Zustandsdiagramme gehören.
- Technische Details: Detaillierte Informationen über die einzelnen Komponenten und ihre Beziehungen können ebenfalls hilfreich sein. Dies kann die Spezifikationen der Komponenten, die verwendeten Technologien und Plattformen, Datenstrukturen, Algorithmen und mehr umfassen.
- Nicht-funktionale Anforderungen: Diese sollten ebenfalls dokumentiert werden, einschließlich der Leistungs-, Sicherheits-, Zuverlässigkeits- und anderen Anforderungen, die die Software erfüllen muss.
- Entwurfsentscheidungen: Dokumentieren Sie die Gründe für bestimmte Entwurfsentscheidungen. Dies kann helfen, zukünftige Entscheidungen zu informieren und bietet einen Einblick in den Prozess der Softwareentwicklung.
- Richtlinien zur Wartung und Erweiterung: Diese können dazu beitragen, die langfristige Wartung und Erweiterung der Software zu erleichtern.

Es gibt auch verschiedene Standards und Rahmenwerke zur Dokumentation der Softwarearchitektur, wie das ISO/IEC/IEEE 42010:2011, das eine Anleitung zur Dokumentation der Softwarearchitektur bietet, und das "4+1" Architekturmodell, das einen Ansatz zur Darstellung von Architekturen unter Verwendung von mehreren, miteinander verbundenen Sichten bietet.

3. Welches sind die wichtigsten Eigenschaften von Langlebigen Softwarearchitekturen (Lilienthal)? Carola Lilienthal ist eine anerkannte Expertin auf dem Gebiet der Softwarearchitektur, insbesondere in Bezug auf langlebige Softwarearchitekturen und das Management technischer Schulden. In ihrem Buch "Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen" betont sie mehrere Schlüsseleigenschaften und Praktiken, die dazu beitragen, die Lebensdauer einer Softwarearchitektur zu verlängern:

- Verständlichkeit: Die Architektur sollte leicht zu verstehen und klar dargestellt sein, damit Entwickler sie effizienter warten und erweitern können.
- Konsistenz: Eine einheitliche und konsistente Architektur hilft dabei, Fehler zu reduzieren und das Verständnis des Systems zu erleichtern.
- Nachvollziehbarkeit: Entscheidungen, die die Architektur beeinflussen, sollten gut dokumentiert und begründet sein. Dies erleichtert zukünftige Wartungs- und Weiterentwicklungsaufgaben.
- Robustheit: Eine robuste Architektur sollte in der Lage sein, Änderungen in den Anforderungen oder der Umgebung zu bewältigen, ohne dass umfangreiche Umstrukturierungen oder Umbauten erforderlich sind.
- Minimierung technischer Schulden: Technische Schulden sollten so weit wie möglich minimiert werden. Dies beinhaltet die bewusste Anerkennung und Überwachung solcher Schulden und die Anstrengung, sie im Laufe der Zeit abzubauen.
- Fokus auf Qualität: Die Architektur sollte Qualitätsanforderungen wie Leistung, Sicherheit und Zuverlässigkeit von Anfang an berücksichtigen und diese nicht als nachträgliche Überlegung behandeln.
- Evolutionäre Entwicklung: Die Architektur sollte flexibel genug sein, um sich mit der Zeit weiterzuentwickeln und sich an verändernde Anforderungen anzupassen.

Diese Merkmale tragen dazu bei, dass die Softwarearchitektur langlebig ist, also eine hohe Lebensdauer und Anpassungsfähigkeit hat. Die Idee ist, eine Balance zwischen den aktuellen Anforderungen und der langfristigen Wartbarkeit und Erweiterbarkeit der Software zu finden.



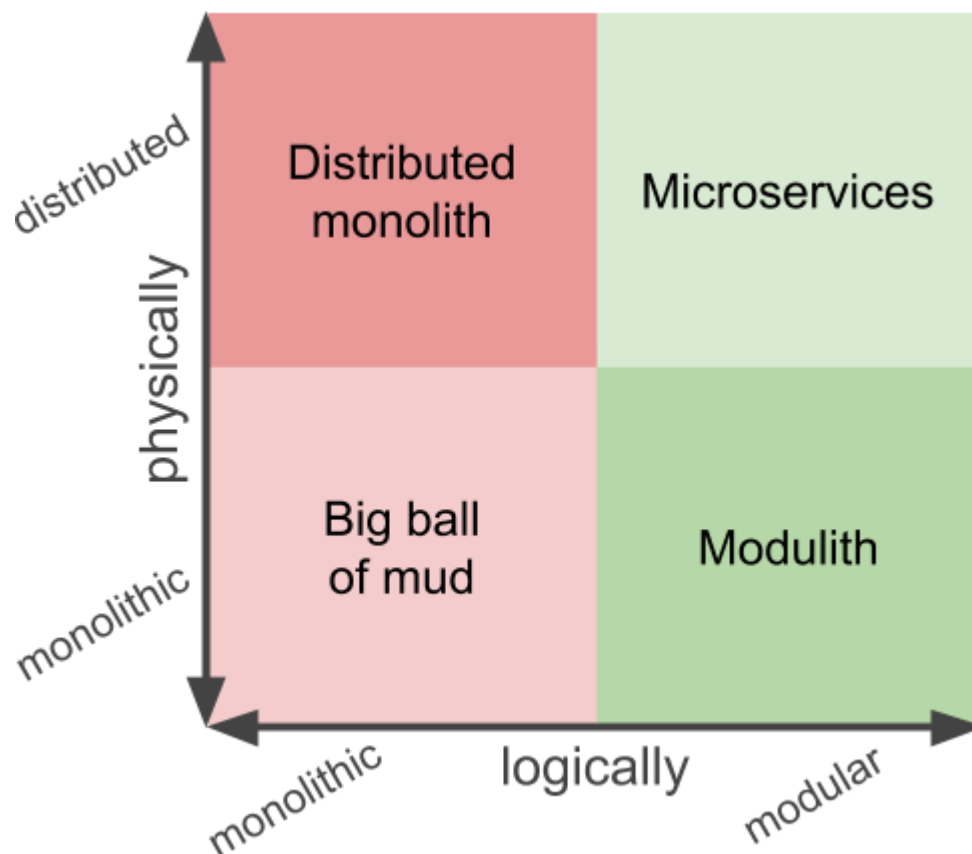
(<https://software-architektur.tv/2021/04/30/folge57.html>)

4. Was ist ein Modulith? Der Begriff "Modulith" wurde von Oliver Drotbohm geprägt und wird verwendet, um eine bestimmte Art von Softwarearchitektur zu beschreiben. Ein Modulith ist ein monolithisches System, das aber intern so strukturiert ist, dass es modulare Eigenschaften aufweist. Das heißt, es ist in eigenständige Module unterteilt, die jeweils eine bestimmte Aufgabe oder Funktion erfüllen.

Jedes Modul eines Moduliths sollte so konzipiert sein, dass es unabhängig von den anderen funktionieren kann. Die Kommunikation zwischen den Modulen erfolgt über definierte Schnittstellen. Dies ermöglicht eine klare Trennung von Zuständigkeiten und verbessert die Übersichtlichkeit und Wartbarkeit des Systems.

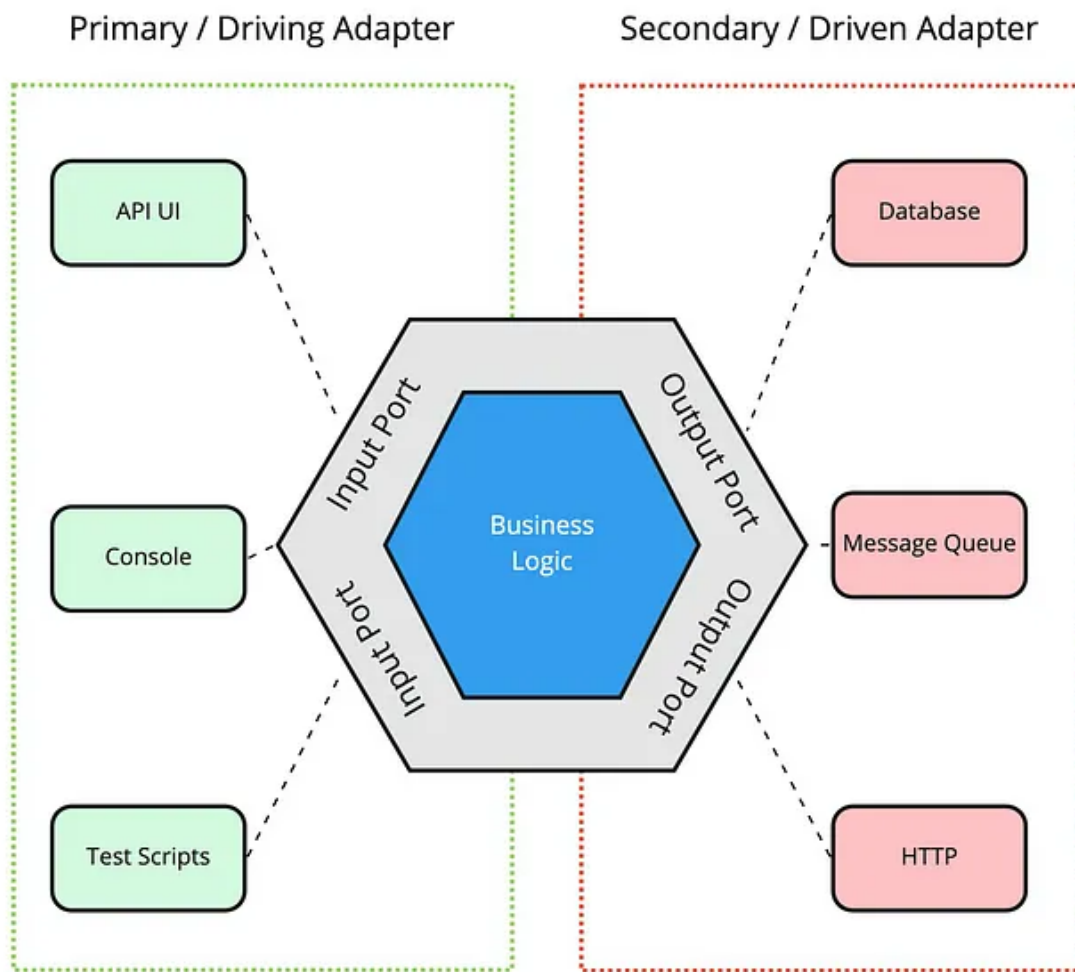
Ein Modulith ist somit ein Kompromiss zwischen einem Monolithen und einer Microservices-Architektur. Er bietet viele Vorteile der Microservices (wie die Modularität und die Isolierung der Komponenten) ohne einige der Herausforderungen, die mit der Verwaltung einer verteilten Architektur verbunden sind (wie Netzwerklatenz, Datenkohärenz und komplexe Service-Orchestrierung).

Es ist wichtig zu beachten, dass ein Modulith nicht nur eine zufällige Sammlung von Modulen ist. Es erfordert eine sorgfältige Planung und Architektur, um sicherzustellen, dass jedes Modul effektiv isoliert ist und gleichzeitig effizient mit den anderen Modulen kommunizieren kann.



(<https://blog.ttulka.com/good-and-bad-monolith/>)

5. Wie funktioniert die Ports and Adapters Architektur? Die Ports and Adapters Architektur, auch bekannt als Hexagonal Architektur, wurde von Alistair Cockburn entwickelt. Sie zielt darauf ab, die Anwendung von den Details der Infrastruktur zu entkoppeln, um die Unabhängigkeit und Austauschbarkeit zu erhöhen.



miro

(<https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0>)

Hier ist eine vereinfachte Beschreibung davon, wie die Ports and Adapters Architektur funktioniert:

- Die Anwendung im Zentrum (Hexagon): Die Geschäftslogik der Anwendung, auch als "Domain Model" bezeichnet, bildet das Zentrum der Architektur. Diese Kernkomponente ist unabhängig von den technologischen Details der Anwendung und sollte in einer idealen Welt ohne Kenntnis von Datenbanken, Benutzeroberflächen oder externen Systemen existieren können.
- Ports: Ports repräsentieren die Schnittstellen, durch die die Anwendung mit der Außenwelt interagiert. Es gibt primäre Ports, die von der Anwendung selbst genutzt werden, um ihre Funktionalität zur Verfügung zu stellen (z.B. Services, die von einer Benutzeroberfläche genutzt werden), und sekundäre Ports, die von der Anwendung genutzt werden, um auf externe Ressourcen zuzugreifen (z.B. Datenbanken, Webdienste).
- Adapters: Adapter sind Implementierungen, die zu einem Port passen und die Kommunikation zwischen der Anwendung und der Außenwelt ermöglichen. Sie übersetzen Anfragen und Daten zwischen den Formaten, die von der Anwendung und den externen Systemen verwendet werden. Zum Beispiel könnte ein Adapter eine Datenbankabfrage in einen Aufruf an die Geschäftslogik der Anwendung umwandeln.

Der Hauptvorteil dieser Architektur besteht darin, dass sie die Anwendung von der Infrastruktur entkoppelt. Dies bedeutet, dass man Teile der Infrastruktur (wie Datenbanken oder Benutzeroberflächen) ändern oder austauschen kann, ohne die Kernlogik der Anwendung zu beeinflussen. Dies erhöht die Flexibilität und erleichtert das Testen der Anwendung.

6. DDD: Was sind die wesentlichen Bausteine des modellgetriebenen Entwurfs (Taktische Pattern) aus DDD? Beim Domain-Driven Design (DDD) gibt es eine Reihe von "taktischen Mustern" oder Bausteinen, die zur Implementierung des modellgetriebenen Entwurfs verwendet werden. Die folgenden sind die wichtigsten:

- Entity (Entität): Eine Entität ist ein Objekt, das nicht nur durch seine Attribute, sondern vor allem durch eine Kontinuität und Identität über die Zeit und über verschiedene Darstellungen hinweg definiert ist. Ein Beispiel für eine Entität könnte ein bestimmter Benutzer in einem System sein.
- Value Object (Wertobjekt): Ein Value Object ist ein Objekt, das Attribute hat, aber keine eindeutige Identität benötigt. Sie sind oft unveränderlich und können frei ersetzt und zwischen Objekten ausgetauscht werden. Beispiele sind Dinge wie Farben, Geldbeträge oder Adressen.
- Aggregate: Ein Aggregate ist eine Ansammlung von Objekten, die als Einheit behandelt werden. Es gibt ein Wurzelobjekt (Aggregate Root), über das alle Interaktionen mit dem Aggregate laufen. Dieses Muster wird verwendet, um Konsistenzgrenzen zu gewährleisten und Datenintegrität zu sichern.
- Domain Event (Domänenereignis): Ein Domain Event ist etwas, das in der Domäne passiert und von Bedeutung ist. Es wird oft verwendet, um zustandsändernde Ereignisse zu modellieren, die für das Geschäft relevant sind.
- Service: Ein Service ist eine Operation, die nicht natürlich als Methode eines Value Objects oder einer Entität passt. Sie definieren Handlungen, die vom Modell ausgeführt werden können.
- Repository: Ein Repository stellt eine Sammlung von Objekten zur Verfügung, die Sie speichern und abrufen können. Es stellt oft Methoden zur Verfügung, um Objekte basierend auf bestimmten Kriterien zu finden.
- Factory: Eine Factory wird verwendet, um die Erstellung komplexer Objekte und Aggregate zu kapseln.

Diese taktischen Muster sind Bestandteile eines größeren "strategischen Design", das auch Kontexte, Bounded Contexts, und die Beziehungen zwischen ihnen (Kontextzuordnung) beinhaltet. DDD ist ein mächtiges Werkzeug für das Modellieren komplexer Geschäftslogik und diese taktischen Muster sind die Bausteine, mit denen dieses Modell erstellt wird.

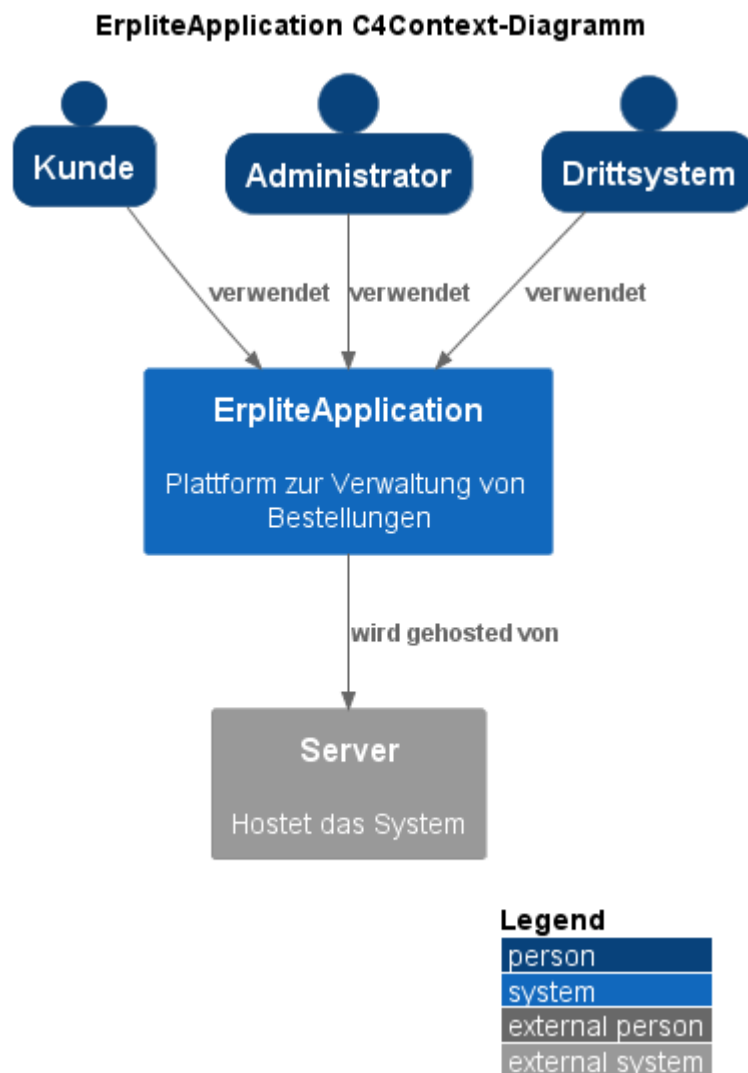
Abgabe der Architekturanalyse des bestehenden erplite-Backends

1. Dokumentation (textuelle Beschreibung, Codeauszüge, C4-Diagramme, Klassendiagramme) der Ports-Und-Adapters-Architektur und der DDD-Bestandteile (taktische Muster) von Ordermanagement anhand der gegebenen Anwendungsfälle, die schon implementiert sind:

- Bestellung aufgeben
- Bestellung auf bezahlt setzen

- Packliste generieren
- Packlistenitems abhaken

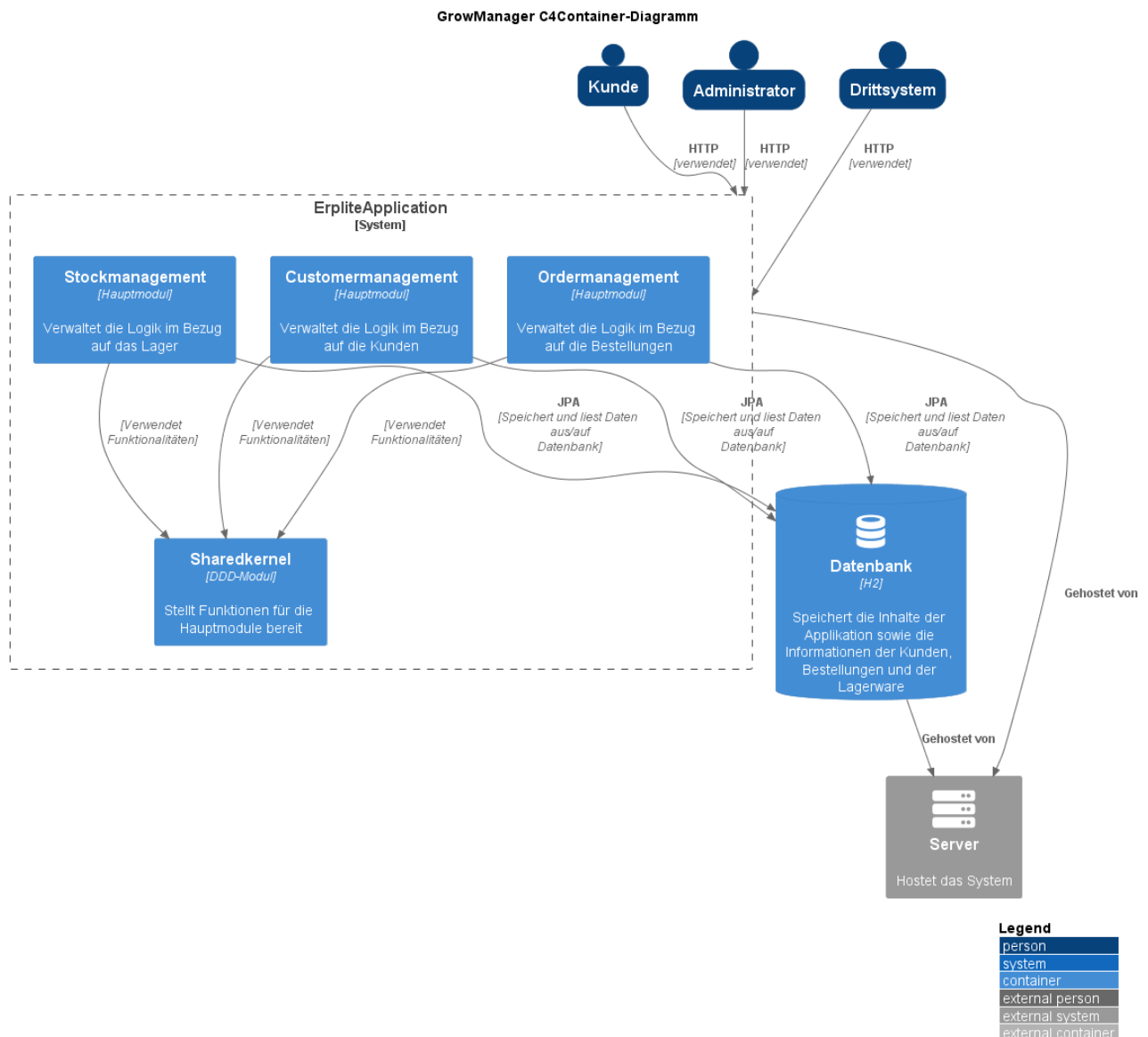
Zunächst wird der Aufbau der Anwendung mit einem C4-Diagramm näher beschrieben:



Dieses C4-Context-Diagramm visualisiert das hochrangige System und seine Interaktionen mit den beteiligten Benutzern bzw. Akteuren.

- Im Mittelpunkt steht das System "ErpliteApplication", das als Plattform zur Verwaltung von Bestellungen dient. Es gibt drei Hauptakteure, die mit dem System interagieren:
- Der Kunde: Ein Endbenutzer, der das System verwendet, wahrscheinlich um Bestellungen zu tätigen oder zu verfolgen.
- Der Administrator: Eine Art interner Benutzer, der das System verwendet, möglicherweise um Bestellungen zu verwalten, Kundeninformationen zu verwalten oder Berichte zu erstellen.
- Das Drittsystem: Ein anderes System, das mit der "ErpliteApplication" interagiert. Dies könnte ein Zahlungssystem, ein Bestandsverwaltungssystem oder ein anderes System sein, das für die ordnungsgemäße Funktion des Hauptsystems notwendig ist.

Schließlich wird das System "ErpliteApplication" von einem Server gehostet, der im Diagramm als externes System dargestellt ist.

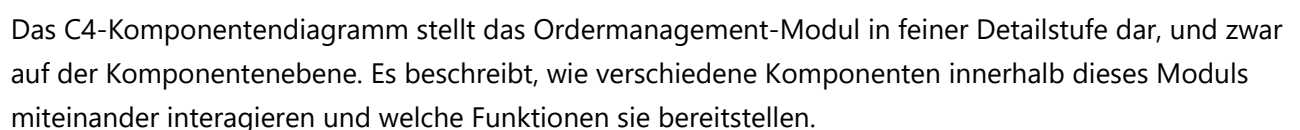


Dieses C4-Container-Diagramm stellt die Architektur der "ErpliteApplication" auf einer etwas detaillierteren Ebene dar, indem es die Hauptmodule oder "Container" innerhalb des Systems zeigt und wie sie miteinander interagieren.

Im System "ErpliteApplication" gibt es drei Hauptmodule:

- "Customermanagement": Dieses Modul verwaltet die Logik im Bezug auf die Kunden. Es könnte Funktionen wie das Hinzufügen neuer Kunden, die Aktualisierung von Kundendaten und die Verarbeitung von Kundenanfragen beinhalten.
- "Ordermanagement": Dieses Modul verwaltet die Logik im Bezug auf die Bestellungen. Es könnte Funktionen wie das Erstellen neuer Bestellungen, die Aktualisierung des Bestellstatus und die Verarbeitung von Bestellungsbezogenen Anfragen beinhalten.
- "Stockmanagement": Dieses Modul verwaltet die Logik im Bezug auf das Lager. Es könnte Funktionen wie das Hinzufügen neuer Lagerbestände, die Aktualisierung von Lagerbestandsdaten und die Verarbeitung von Lagerbezogenen Anfragen beinhalten.

Drei Arten von Akteuren interagieren mit dem System: Der Kunde, der Administrator und das Drittsystem. Sie verwenden alle das HTTP-Protokoll, um mit der "ErpliteApplication" zu interagieren.

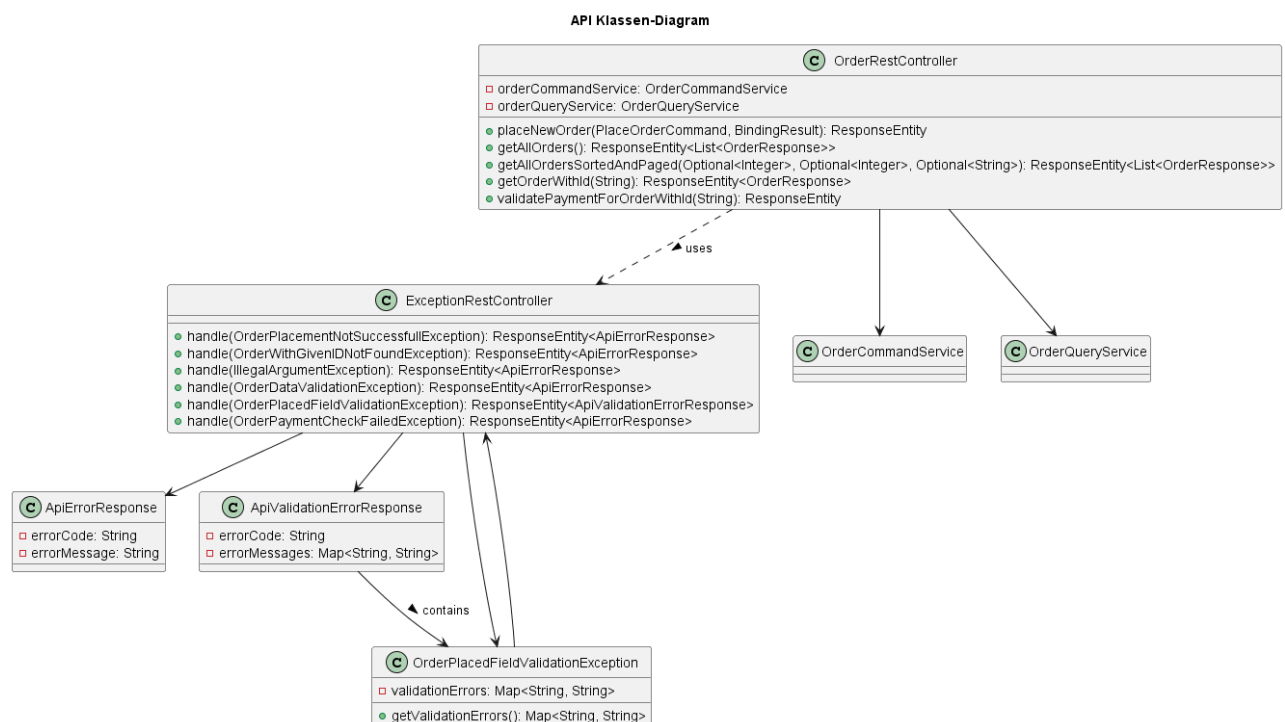


- 9 / 31

- "DB" oder "Datenbank-Entitäten": Diese Komponente umfasst die Datenbank-Entitäten für Bestellungen und deren Details. Sie beinhaltet auch einen Service für die Umwandlung zwischen Datenbank- und Domänen-Entitäten sowie eine JPA-Repository-Implementierung zur Durchführung von Datenzugriffsoperationen.
- "Service" oder "Anwendungsdienste": Diese Komponente beinhaltet die Implementierungen von Anwendungsdiensten für das Ausführen von Befehlen und Abfragen auf Bestellungen, das Empfangen von Nachrichten, das Mapping von Antworten und benutzerdefinierte Ausnahmen, die spezifische Fehlerbedingungen repräsentieren.
- "Datenmodelle" oder "Entity-Klassen": Diese Komponente enthält die Hauptgeschäftslogik und Datenmodelle der Anwendung, einschließlich wichtiger Geschäftsereignisse, Wertobjekte und Hauptentitäten wie Kunde, Artikel und Bestellung.
- "Ports" oder "Ports and Adapters": Diese Komponente definiert die Grenzen und Verträge für eingehende und ausgehende Interaktionen der Anwendung, einschließlich ausführender und lesender Operationen auf Bestellungen sowie eingehender und ausgehender Nachrichten.
- "Messaging Spring" oder "Nachrichtenereignisse": Diese Komponente ist verantwortlich für das Handling von eingehenden und ausgehenden Nachrichtenereignissen in der Anwendung, insbesondere bezogen auf Bestellungen, unter Verwendung des Spring Frameworks.

Alle diese Komponenten interagieren und arbeiten zusammen, um die Funktionalitäten des Ordermanagement-Moduls bereitzustellen.

Darüber hinaus interagiert das Ordermanagement-Modul auch mit dem "Sharedkernel"-Modul, dem "Stockmanagement"-Modul, dem "Customermanagement"-Modul und der "H2"-Datenbank, um die erforderlichen Daten abzurufen oder zu speichern.

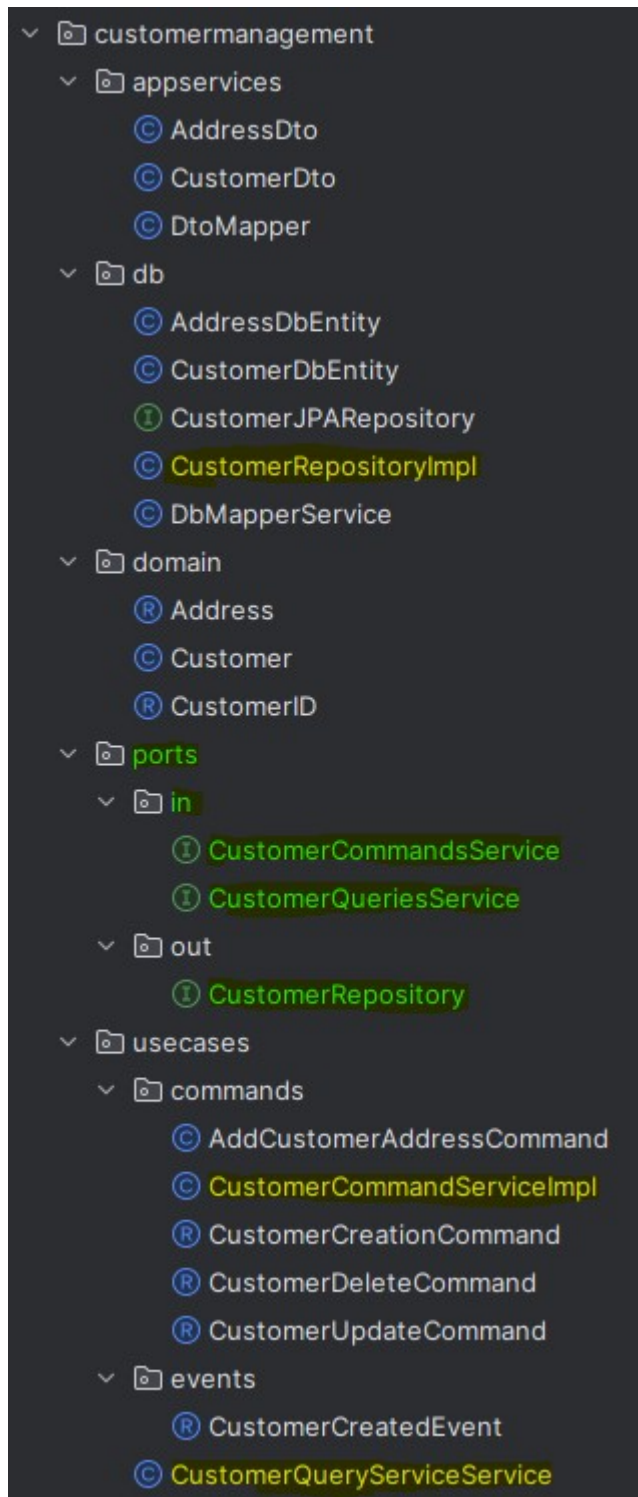


Dieses Klassendiagramm zeigt die Beziehungen zwischen verschiedenen Klassen im Kontext der API-Komponente des vorangegangenen Diagramms.

- `ApiResponse` ist eine Klasse, die ein `Error-Code`- und `Error-Message`-Feld enthält. Diese Klasse wird in der Regel verwendet, um eine standardisierte Fehlerantwort für die API zu liefern.
- `ApiValidationErrorResponse` ist eine spezielle Art von Fehlerantwort, die für Validierungsfehler verwendet wird. Sie enthält neben dem `Error-Code` ein `Map`-Objekt, das eine Liste von Fehlermeldungen enthält.
- `OrderPlacedFieldValidationException` ist eine spezielle Ausnahme, die auftritt, wenn bei der Validierung einer platzierten Bestellung Fehler auftreten. Es enthält eine `Map` von Validierungsfehlern, die von der Methode `getValidationErrors()` abgerufen werden können.
- `ExceptionRestController` ist eine Klasse, die zum Handling von Ausnahmen (Exceptions) verwendet wird. Sie definiert mehrere Methoden, um verschiedene Arten von Ausnahmen zu behandeln und gibt dabei jeweils eine `ResponseEntity` mit einer `ApiResponse` oder `ApiValidationErrorResponse` zurück.
- `OrderRestController` ist die Hauptcontroller-Klasse, die die Anfragen der API bearbeitet. Sie hat zwei abhängige Dienste: `OrderCommandService` und `OrderQueryService`. Diese Klasse definiert mehrere Methoden zur Verwaltung von Bestellungen, wie das Platzieren einer neuen Bestellung, das Abrufen aller Bestellungen, das Abrufen einer Bestellung mit einer bestimmten ID und die Validierung der Zahlung für eine Bestellung mit einer bestimmten ID.

Die Beziehungen zwischen den Klassen sind durch die Linien dargestellt, wobei die Pfeilspitze die Richtung der Abhängigkeit anzeigt. Die gestrichelte Linie zwischen `OrderRestController` und `ExceptionRestController` bedeutet, dass `OrderRestController` die `ExceptionRestController` Klasse verwendet.

Sehen wir uns nun die Ports and Adapters Architektur der Anwendung anhand des "customermanagement"-Packages an:

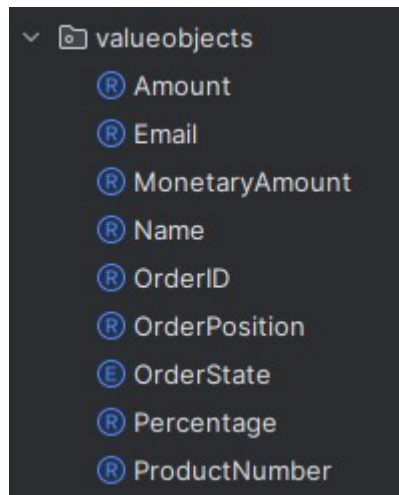


Die Ports, die meist nur Interfaces sind (im Bild mit Grün markiert), des Packages "customermanagement" sind im Package "ports" zu finden. Diese unterscheiden sich in "Türen", die hinein führen und "Türen", die hinausführen. Die Ausimplementierungen (im Bild mit Gelb markiert), sind in anderen Packages zu finden. Die Funktionen werden dann jeweils über die Interfaces aufgerufen.

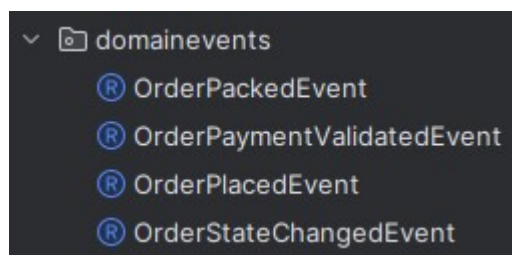
DDD-Bestandteile

Jetzt werden die einzelnen Bestandteile des DDD im Package "ordermanagement" aufgelistet: In der gegebenen Paketstruktur sind die wesentlichen Komponenten und Prinzipien von Domain-Driven Design (DDD) wie folgt repräsentiert:

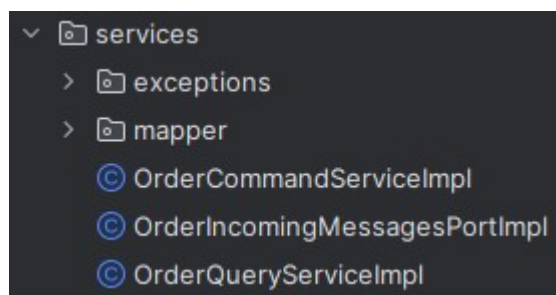
- Domain Layer: Das domain-Paket repräsentiert den Domain Layer. Es enthält alle Entitäten, Wertobjekte und Domänenevents, die die Kerngeschäftslogik und -regeln der Anwendung darstellen.
- Entities: Die Klasse Order repräsentiert eine Entität.
 - Value Objects: Klassen wie Amount, Email, MonetaryAmount, Name, OrderID, OrderPosition, Percentage, und ProductNumber repräsentieren Value Objects.



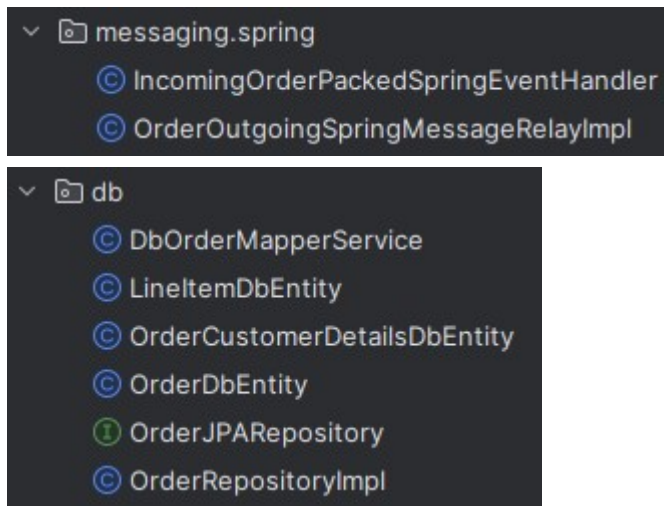
- Domain Events: Klassen wie OrderPackageEvent, OrderPaymentValidatedEvent, OrderPlacedEvent, und OrderStateChangedEvent repräsentieren Domain Events.



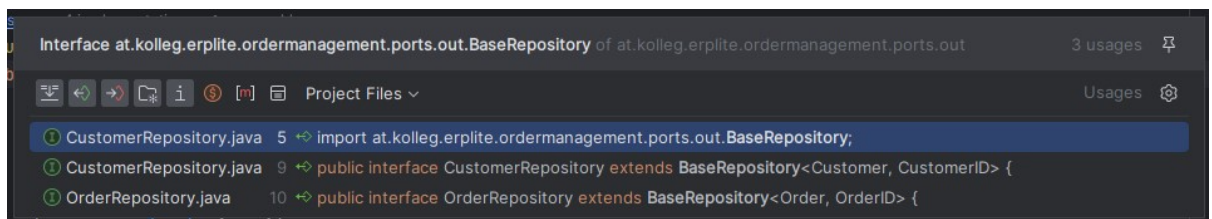
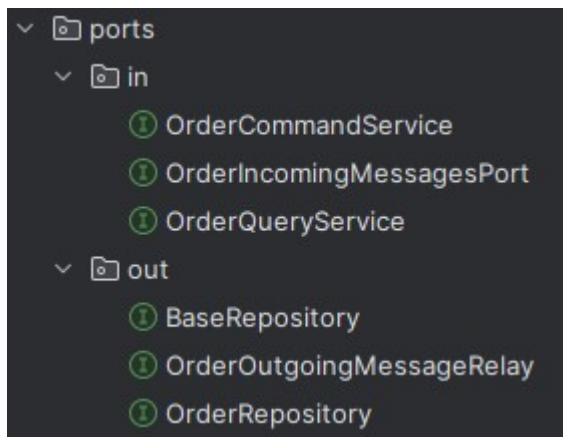
- Application Layer: Das services-Paket repräsentiert den Application Layer. Hier befinden sich Anwendungsdienste wie OrderCommandServiceImpl und OrderQueryServiceImpl, die die Interaktion zwischen den Infrastruktur-, Domänen- und UI-Layern orchestrieren.



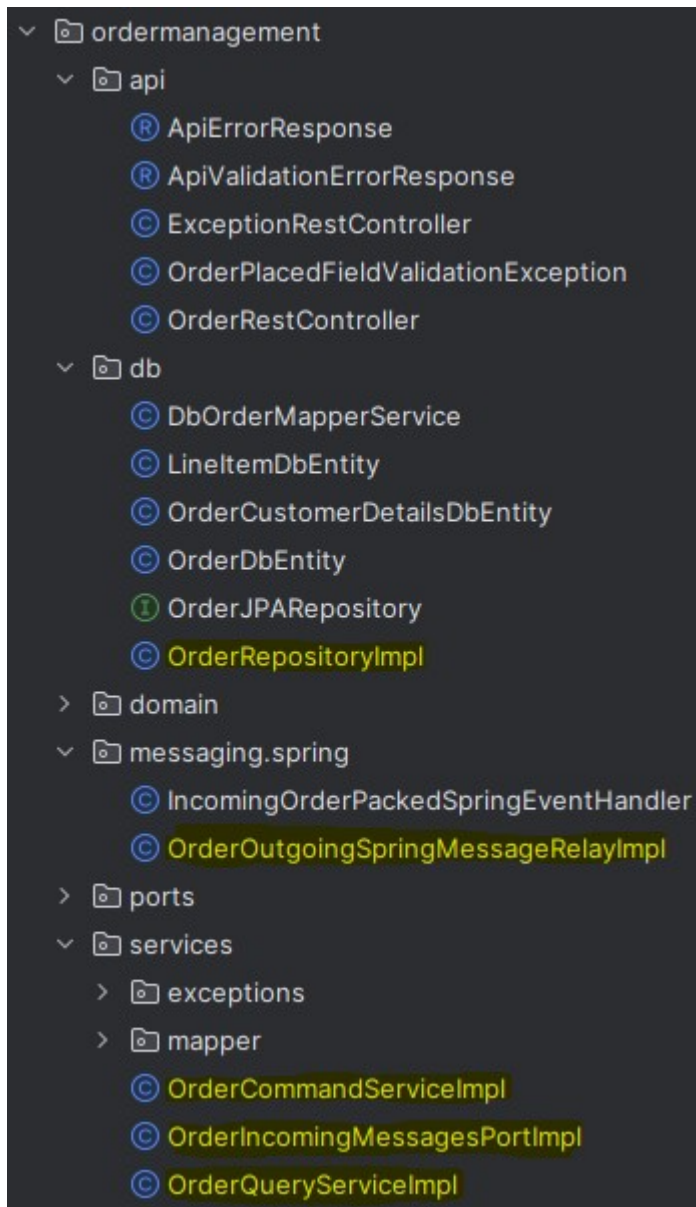
- Infrastructure Layer: Die db- und messaging.spring-Pakete repräsentieren den Infrastructure Layer. Sie beinhalten die Klassen, die sich mit der Persistenz (wie OrderJPRepository, OrderRepositoryImpl, OrderDbEntity usw.) und der Kommunikation mit anderen Systemen (wie OrderOutgoingSpringMessageReplayImpl und IncomingOrderPackedSpringEventHandler) befassen.



- Interfaces (Ports): Die ports-Pakete repräsentieren die verschiedenen Ports in DDD. Die in-Interfaces definieren die Funktionalitäten, die für eingehende Anfragen erforderlich sind, während die out-Interfaces die Funktionalitäten definieren, die für ausgehende Anfragen benötigt werden.



- Adapters: Die Ausimplementierungen der Interfaces im Package "ports" repräsentieren die Adapter in DDD. Diese sind verantwortlich für die Kommunikation zwischen der Anwendung und externen Diensten oder Nutzern.



- Repositories: In DDD sind Repositories verantwortlich für den Zugriff auf Daten. Hier sind das OrderJpaRepository-Interface und seine Implementierung OrderRepositoryImpl zu finden.

```
@RepositoryMarker
@Repository
class OrderRepositoryImpl implements OrderRepository {

    @Autowired
    private OrderJpaRepository orderJpaRepository;

    @Override
    public Optional<Order> insert(Order order) {
        if (order == null) throw new IllegalArgumentException("Order to be
inserted must not be null!");
        OrderDbEntity orderDbEntity = DbOrderMapperService.toOrm(order);
        OrderDbEntity insertedEntity =
orderJpaRepository.save(orderDbEntity);
        if (insertedEntity == null) return Optional.empty();
        return Optional.of(DbOrderMapperService.toDomain(insertedEntity));
    }
}
```



```
@Override
public Optional<Order> getById(OrderID id) {
    if (id == null) throw new IllegalArgumentException("OrderID for
Order to get from db must not be null!");
    Optional<OrderDbEntity> orderEntityOptional =
this.orderJpaRepository.findById(id.id());
    if (!orderEntityOptional.isPresent()) return Optional.empty();
    return
Optional.of(DbOrderMapperService.toDomain(orderEntityOptional.get()));
}

@Override
public List<Order> getAll() {
    List<OrderDbEntity> list = this.orderJpaRepository.findAll();
    if (list == null) return Collections.emptyList();
    return list.stream().map(dbEntity ->
DbOrderMapperService.toDomain(dbEntity)).toList();
}

@Override
public void deleteById(OrderID id) {
    this.orderJpaRepository.deleteById(id.id());
}

@Override
public List<Order> getAllSortedAndPaged(int page, int pageSize, String
sortedBy) {
    Pageable sortedAndPagedOrders = PageRequest.of(page, pageSize,
Sort.by(sortedBy));
    Page<OrderDbEntity> resultPage =
this.orderJpaRepository.findAll(sortedAndPagedOrders);
    if (resultPage.isEmpty()) return Collections.emptyList();
    return resultPage.stream().map(dbEntity ->
DbOrderMapperService.toDomain(dbEntity)).toList();
}

@Override
public void updateOrderWithNewState(Order order) { //first read the
order, then change the entity object, then persist the order with save-call
(transaction management included)
    Optional<OrderDbEntity> optOrderDbEntity =
this.orderJpaRepository.findById(order.getOrderID().id());
    if (optOrderDbEntity.isPresent()) {
        OrderDbEntity orderDbEntity = optOrderDbEntity.get();
        orderDbEntity.setState(order.getState()); //set state
        //further field could be updated if necessary (in further
methods or in one Method containing all changes)
        this.orderJpaRepository.save(orderDbEntity);
    }
}
}
```


- Aggregates: In dieser Codebasis könnte das Order-Objekt und alle mit ihm verbundenen Objekte als ein Aggregat betrachtet werden, obwohl das aus dem gegebenen Code nicht klar hervorgeht, da die genaue Organisation von Aggregaten von der spezifischen Geschäftslogik abhängt.

```
@AggregateMarker
public final class Order {

    //ID
    private final OrderID orderID;

    //Customer-Data
    private CustomerData customerData;

    //Line-Items
    private final List<LineItem> lineItems;

    //Order-State and calculated values
    private OrderState state;
    private final MonetaryAmount taxTotal;
    private final MonetaryAmount netTotal;
    private final MonetaryAmount grossTotal;
    private final LocalDateTime date;

    public Order(OrderID orderID, CustomerData customerData,
                 LocalDateTime date, List<LineItem> lineItems, OrderState
status) {
        if (orderID == null) throw new IllegalArgumentException("OrderID
must not be null!");
        this.orderID = orderID;
        if (customerData == null)
            throw new IllegalArgumentException("Customer-Data for order
invalid!");
        this.customerData = customerData;
        if (lineItems == null)
            throw new IllegalArgumentException("LineItems-List must not be
null, at least an empty List is needed!");
        this.lineItems = lineItems;
        if (status == null) throw new IllegalArgumentException("State must
not be null!");
        this.state = status;
        this.taxTotal = this.calculateOrderTax();
        this.netTotal = this.calculateOrderNetTotal();
        this.grossTotal = this.calculateOrderGrossTotal();
        this.date = date;
    }

    public List<LineItem> getLineItems() {
        return Collections.unmodifiableList(this.lineItems);
    }
}
```

```
public void orderStateTransitionTo(OrderState newState) {
    switch (newState) {
        case CANCELED -> {
            if (this.state == OrderState.IN_DELIVERY || this.state ==
OrderState.DELIVERED)
                throw new OrderStateChangeNotPossibleException("Order in
State " + this.state + " cannot be canceled");
            this.state = newState;
        }
        case PAYMENT_VERIFIED -> {
            if (this.state != OrderState.PLACED)
                throw new OrderStateChangeNotPossibleException("Order
must be in state PLACED for transition to PAYMENT VERIFIED!");
            this.state = newState;
        }
        case PREPARING_FOR_DELIVERY -> {
            if (this.state != OrderState.PAYMENT_VERIFIED)
                throw new OrderStateChangeNotPossibleException("Order
must be in state PAYMENT VERIFIED for transition to PREPARING FOR
DELIVERY!");
            this.state = newState;
        }
        case IN_DELIVERY -> {
            if (this.state != OrderState.PREPARING_FOR_DELIVERY)
                throw new OrderStateChangeNotPossibleException("Order
must be in state PREPARING FOR DELIVERY for transition to IN DELIVERY!");
            this.state = newState;
        }
        case DELIVERED -> {
            if (this.state != OrderState.IN_DELIVERY)
                throw new OrderStateChangeNotPossibleException("Order
must be in state IN DELIVERY for transition to DELIVERED!");
            this.state = newState;
        }
    }
}

private MonetaryAmount calculateOrderTax() {
    MonetaryAmount orderTax = new MonetaryAmount(new BigDecimal(0));
    for (LineItem lineItem : this.lineItems) {

        orderTax = orderTax.add(lineItem.getTotalTaxLine());
    }
    return orderTax;
}

private MonetaryAmount calculateOrderNetTotal() {
    MonetaryAmount orderNetSum = new MonetaryAmount(new BigDecimal(0));
    for (LineItem lineItem : this.lineItems) {
        orderNetSum = orderNetSum.add(lineItem.getTotalNetLine());
    }
    return orderNetSum;
}
```

```

private MonetaryAmount calculateOrderGrossTotal() {
    return this.netTotal.add(this.taxTotal);
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Order)) return false;

    Order order = (Order) o;

    return getOrderID().equals(order.getOrderID());
}

@Override
public int hashCode() {
    return getOrderID().hashCode();
}
}

```

Dieser Überblick zeigt, wie die DDD-Komponenten in dieser Codebasis repräsentiert sind. Es ist wichtig zu beachten, dass die DDD-Strukturierung und -Nomenklatur je nach Anwendung und Team variieren kann.

Sehen wir uns den Aufbau nun anhand bereits implementierter Funktionen an!

- Bestellung aufgeben

Zunächst wird über einen Post-Request die Methode "placeNewOrder()" im OrderRestController aufgerufen:

```

//OrderRestController.java
@PostMapping("/orders/")
public ResponseEntity placeNewOrder(@RequestBody @Valid
PlaceOrderCommand placeOrderCommand, BindingResult bindingResult) {

    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Handling place new order api request ...");

    HashMap<String, String> errors = new HashMap<>();

    if (bindingResult.hasErrors()) {
        Logger.getLogger(this.getClass().getName()).log(Level.WARNING,
"Errors in placeOrderCommand detected!");
        for (FieldError fieldError : bindingResult.getFieldErrors()) {
            errors.put(fieldError.getField(),
fieldError.getDefaultMessage());
        }
        throw new OrderPlacedFieldValidationException("Validation

```

```

        errors for order placement!", errors);
    }

    OrderResponse orderResponse =
orderCommandService.handle(placeOrderCommand);

    String resourceLocation =
ServletUriComponentsBuilder.fromCurrentContextPath().build().toUriString() +
"/orders/" + orderResponse.orderID();
    try {
        return ResponseEntity.created(new
URI(resourceLocation)).body(orderResponse);
    } catch (URISyntaxException e) {
        return ResponseEntity.noContent().build();
    }
}

```

Beinhaltet die mitgegebene Bestellung keine Fehler, so wird eine neue OrderResponse erstellt, indem die Methode "handle()" über die Port-Klasse "OrderCommandService" aufgerufen wird. Der dazugehörige Adapter, die ausimplementierte Klasse "OrderCommandServiceImpl", tritt dann in Aktion.

```

//OrderQueryServiceImpl.java
@Transactional
public OrderResponse handle(PlaceOrderCommand placeOrderCommand) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Handle
place order command ...");
    List<String> errors = validatePlaceOrderCommand(placeOrderCommand);
    if (errors.size() != 0) throw new
OrderDataValidationException(errors);

    List<LineItem> lineItemList = new ArrayList<>();
    int i = 1;
    for (CartItem cartItem : placeOrderCommand.cartItems()) {
        lineItemList.add(new LineItem(
            new OrderPosition(i),
            new ProductNumber(cartItem.productNumber()),
            new Name(cartItem.productName()),
            new MonetaryAmount(new
BigDecimal(cartItem.priceNet()),
            new Percentage(cartItem.tax()),
            new Amount(cartItem.amount())
        ));
        i++;
    }

    Order orderToInsert = new Order(
        new OrderID("ONR" +
UUID.randomUUID().toString().substring(0, 7)),
        new CustomerData(

```

```

        new CustomerID(placeOrderCommand.customerID()),
        new Name(placeOrderCommand.customerFirstname()),
        new Name(placeOrderCommand.customerLastname()),
        new Email(placeOrderCommand.customerEmail()),
        placeOrderCommand.customerStreet(),
        placeOrderCommand.customerZipcode(),
        placeOrderCommand.customerCity(),
        placeOrderCommand.customerCountry()
    ),
    LocalDateTime.now(),
    lineItemList,
    OrderState.PLACED
);

Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Insert
new order in DB ...");
Optional<Order> orderOptional =
this.orderRepository.insert(orderToInsert);

if (orderOptional.isPresent()) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Publishing order placed domain event ...");
    orderOutgoingMessageRelay.publish(new
OrderPlacedEvent(OrderResponseMapper.toResponseFromDomain(orderOptional
.get())));
    return
OrderResponseMapper.toResponseFromDomain(orderOptional.get());
} else {
    throw new
OrderPlacementNotSuccessfulException("OrderQueryServiceImpl: Order
could not be placed!");
}
}

```

Zunächst wird wieder überprüft, ob die Bestellung Fehler beinhaltet. Ist dies nicht der Fall, werden neue LineItems erstellt (pro bestelltem Artikel ein LineItem). Anschließend wird eine neue Bestellung (DDD-Bestandteil Aggregates - darin werden die ValueObjects verwendet) inkl. neuem Kunden, den zuvor erstellten LineItems und dem Status "OrderState.PLACED" erstellt. Die neue Bestellung wird in die Datenbank eingefügt und in ein Optional gespeichert. Hat das Einfügen in die Datenbank funktioniert und das Optional "isPresent()", wird die Bestellung veröffentlicht und an den Aufrufer zurückgegeben.

```

//OrderRepositoryImpl.java
@Override
public Optional<Order> insert(Order order) {
    if (order == null) throw new IllegalArgumentException("Order to be
inserted must not be null!");
    OrderDbEntity orderDbEntity = DbOrderMapperService.toOrm(order);
    OrderDbEntity insertedEntity =
orderJpaRepository.save(orderDbEntity);
    if (insertedEntity == null) return Optional.empty();
}

```

```
        return Optional.of(DbOrderMapperService.toDomain(insertedEntity));
    }
}
```

Es wird oben wieder ein Adapter über den Port "OrderOutgoingMessageRelay" aufgerufen.

```
//OrderOutgoingSpringMessageRelayImpl.java
@Override
public void publish(final OrderPlacedEvent orderPlacedEvent) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
        "Handling order placed domain event published for order# " +
        orderPlacedEvent.orderResponse().orderId());
    OrderPlacedSpringEvent orderPlacedSpringEvent = new
    OrderPlacedSpringEvent(this, orderPlacedEvent.orderResponse());
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
        "Publishing order placed spring event for order# " +
        orderPlacedEvent.orderResponse().orderId());
    applicationEventPublisher.publishEvent(orderPlacedSpringEvent);
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Order
    placed spring event for order# " +
        orderPlacedEvent.orderResponse().orderId() + " published!");
}
}
```

- Bestellung auf bezahlt setzen

```
//OrderRestController.java
@PostMapping("/orders/checkpayment/{orderid}")
public ResponseEntity validatePaymentForOrderWithId(@PathVariable
String orderid) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
        "Handling check payment for order api request ...");
    this.orderCommandService.handle(new
    OrderPaymentCheckCommand(orderid));
    return ResponseEntity.accepted().body("Order payment check
    executed. Order payment ok!");
}
}
```

Um die Bestellung auf bezahlt zu setzen, muss zunächst wieder ein HTTP-Request abgesetzt werden, indem die Bezahlung überprüft wird. Das Prüfen der Bezahlung wird wieder an den Port "OrderCommandService" und damit an den Adapter "OrderCommandServiceImpl" übergeben.

```
//OrderCommandServiceImpl.java
@Override
@Transactional
public void handle(OrderPaymentCheckCommand orderPaymentCheckCommand)
throws OrderPaymentCheckFailedException {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
```

```

"Handling order payment check command ...");
    if (orderPaymentCheckCommand == null)
        throw new OrderPaymentCheckFailedException("Empty command for
order payment check!");
    if (!OrderID.isValid(orderPaymentCheckCommand.orderID()))
        throw new OrderPaymentCheckFailedException("Order ID for order
payment check not valid!");
    Optional<Order> optionalOrderToCheck =
this.orderRepository.getById(new
OrderID(orderPaymentCheckCommand.orderID()));
    if (optionalOrderToCheck.isPresent()) {
        Order order = optionalOrderToCheck.get();
        try {
            order.orderStateTransitionTo(OrderState.PAYMENT_VERIFIED);
            this.orderRepository.updateOrderWithNewState(order);
            this.orderOutgoingMessageRelay.publish(new
OrderPaymentValidatedEvent(OrderResponseMapper.toResponseFromDomain(ord
er)));
            Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Payment validated event published!");
        } catch (OrderStateChangeNotPossibleException
orderStateChangeNotPossibleException) {
            throw new OrderPaymentCheckFailedException("Order payment
check not possible. Order in wrong state! " +
orderStateChangeNotPossibleException.getMessage());
        }
    } else {
        throw new OrderPaymentCheckFailedException("Order with Id " +
orderPaymentCheckCommand.orderID() + " not found for payment check!");
    }
}

```

Es wird geprüft, ob eine Bezahlung vorhanden ist und ob die Bezahlung gültig ist. Ist dies der Fall, wird die Bestellung aus der Datenbank geholt, geprüft, ob die Bestellung vorhanden ist und anschließend wird die Bestellung wieder in das DDD-Aggregat "Order" gespeichert. Nun wird versucht, den Zustand auf "PAYMENT_VERIFIED" zu setzen, die Bestellung in der Datenbank zu aktualisieren und ein neues Event (OrderPaymentValidatedEvent) wird abgesetzt.

```

//OrderRepositoryImpl.java
@Override
public void updateOrderWithNewState(Order order) { //first read the
order, then change the entity object, then persist the order with save-
call (transaction management included)
    Optional<OrderDbEntity> optOrderDbEntity =
this.orderJpaRepository.findById(order.getOrderID().id());
    if (optOrderDbEntity.isPresent()) {
        OrderDbEntity orderDbEntity = optOrderDbEntity.get();
        orderDbEntity.setState(order.getState()); //set state
        //further field could be updated if necessary (in further
methods or in one Method containing all changes)
        this.orderJpaRepository.save(orderDbEntity);
    }
}

```

```

    }

}

```

- Packliste generieren & Packingliste anlegen

```

//StockIncommingMessageHandler.java
@Override
@Async("threadPoolTaskExecutor")
public void onApplicationEvent(OrderPaymentValidatedSpringEvent event)
{
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Handling order payment validated spring event ...");

    OrderResponse orderResponse = event.getOrderResponse();

    Packing packingToSaveToDb =
        Packing.builder()
            .id(null)
            .orderId(orderResponse.orderID())
            .deliveryData(new DeliveryData(
                orderResponse.customerFirstname() +
" " + orderResponse.customerLastname(),
                orderResponse.customerStreet(),
                orderResponse.customerZipcode(),
                orderResponse.customerCity(),
                orderResponse.customerCountry()
            )
            ).packingItemList(null) // List is generated down
under
            .build();

    List<PackingItem> packingItemList = new ArrayList<>();
    for (LineItemResponse lineItemResponse :
orderResponse.orderLineItems()) {
        packingItemList.add(
            new PackingItem(
                null,
                lineItemResponse.productNumber(),
                lineItemResponse.productName(),
                lineItemResponse.amount(),
                false,
                packingToSaveToDb
            )
        );
    }
    packingToSaveToDb.setPackingItemList(packingItemList);
    this.packingRepository.save(packingToSaveToDb);
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "New
packing list created and saved in db ...");
}

```


Wenn die Bestellung auf bezahlt gesetzt wird, das Event "OrderPaymentValidatedEvent" abgesetzt. Die Methode "onApplicationEvent()" aufgerufen. In dieser Methode werden die neue Bestellung sowie die bestellten Artikel zur PackingList hinzugefügt sowie die PackingList in die Datenbank gespeichert.

- Packlistenitems abhaken & Packingitems als verpackt markieren

```
//PackingRestController.java
@PostMapping("/setPackedForPacking/{packingItemId}")
public void setPackingItemPackedForPacking(@PathVariable Long
packingItemId) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
    "Handling packing for item# " + packingItemId);

    Optional<PackingItem> optionalPackingItem =
    this.packingItemRepository.findById(packingItemId);
    if (optionalPackingItem.isPresent()) {
        PackingItem packingItem = optionalPackingItem.get();
        packingItem.setPacked(true);
        packingItemRepository.save(packingItem);

        Long packingId = packingItem.getPacking().getId();

        Optional<Packing> packing =
        this.packingRepository.findById(packingId);

        boolean allpacked = true;
        for (PackingItem item : packing.get().getPackingItemList()) {
            if (!item.isPacked()) allpacked = false;
        }
        if (allpacked) {
            Logger.getLogger(this.getClass().getName()).log(Level.INFO,
            "All items for order# " + packing.get().getOrderId() + "packed.
            Publishing event ...");

            this.stockMessagePublisher.publishOrderPackedSpringEventForOrderId(pack
            ing.get().getOrderId());
        }
    }
}
```

Um ein PackListItem abzuhaken, muss zunächst ein POST-Request auf "/setPackedForPacking/{packingItemId}" abgesetzt werden. Es wird dann die Methode "setPackingItemPackedForPacking()" aufgerufen. Das PackingItem wird aus der Datenbank geholt und auf Anwesenheit geprüft. Das PackingItem wird dann auf "setPacket(true)" gesetzt und wieder in die Datenbank gespeichert. Wenn alle PackingItems auf "packed" gesetzt sind, wird die Methode "publishOrderPackedSpringEventForOrderId()" aufgerufen.

```
//StockMessagePublisher.java
public void publishOrderPackedSpringEventForOrderId(String orderId) {
    OrderPackedSpringEvent orderPackedSpringEvent = new
OrderPackedSpringEvent(this, orderId);
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Publishing order fully packed event ...");
    applicationEventPublisher.publishEvent(orderPackedSpringEvent);
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Order
fully packed event published!");
}
```

In dieser Methode wird ein neues "OrderPackedSpringEvent()" erstellt und abgesetzt.

- Bestellung auf IN_DELIVERY setzen wenn alle Packlistenitems gepackt sind

```
//IncomingOrderPackedSpringEventHandler.java
@Override
@Async("threadPoolTaskExecutor")
public void onApplicationEvent(OrderPackedSpringEvent event) {
    Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Order
packed event received for order# " + event.getOrderId());
    this.orderIncomingMessagesPort.handle(new OrderPackedEvent(new
OrderId(event.getOrderId())));
}
```

Wenn das Event "OrderPackedSpringEvent()" abgesetzt wird, über die Klasse "IncomingOrderPackedSpringEventHandler", die den "ApplicationListener<'OrderPackedSpringEvent>" implementiert, aufgerufen. Die Methode "onApplicationEvent()" gibt das Event dann mit "orderIncomingMessagePort().handle()" an den Port "OrderIncomingMessagePort" weiter.

```
//OrderIncomingMessagePortImpl.java
@Transactional
public void handle(OrderPackedEvent orderPackedEvent) {
    //Materialize object into Memory, place changes, and forward the
domain object to repository
    //this ensures, that businesslogic will be executed und object is
in consistent state.
    Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Handling order packed event ...");
    Optional<Order> optionalOrderToCheck =
this.orderRepository.findById(orderPackedEvent.orderId());
    if (optionalOrderToCheck.isPresent()) {
        Order order = optionalOrderToCheck.get();
        try {
            order.orderStateTransitionTo(OrderState.PREPARING_FOR_DELIVERY);
            this.orderRepository.updateOrderWithNewState(order);
        }
    }
}
```

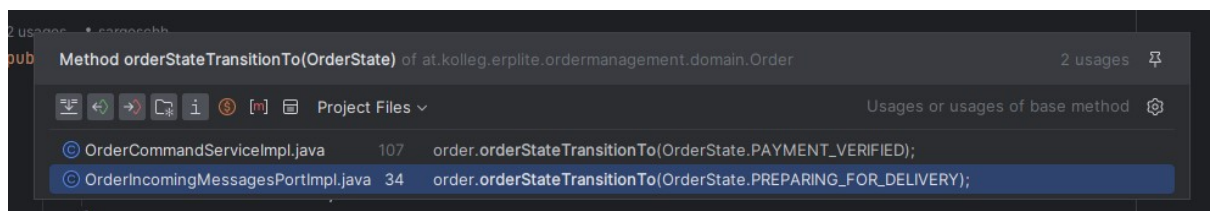
```

        Logger.getLogger(this.getClass().getName()).log(Level.INFO,
"Order state changed to preparing_for_delivery, changed order
persisted!");
    } catch (OrderStateChangeNotPossibleException
orderStateChangeNotPossibleException) {
        throw new OrderPaymentCheckFailedException("Order state
change to preparing for delivery not possible! " +
orderStateChangeNotPossibleException.getMessage());
    }
    } else {
        throw new OrderWithGivenIDNotFoundException("Order with Id " +
orderPackedEvent.orderId().id() + " not found for state change to
preparing for delivery!");
    }
}
}

```

Hier wird die Bestellung, sollte sie vorhanden sein, auf "PREPARING_FOR_DELIVERY" gesetzt und mit aktuellem Status wieder in die Datenbank gespeichert.

Nach meiner Analyse kann ich jedoch nicht feststellen, dass die Bestellung an irgendeiner Stelle der Applikation auf "IN_DELIVERY" gesetzt.



Wie in dieser Abbildung ersichtlich ist, wird die Methode, die für das Ändern des Status verantwortlich ist, nur zweimal aufgerufen.

```

//Order.java
public void orderStateTransitionTo(OrderState newState) {
    switch (newState) {
        case CANCELED -> {
            if (this.state == OrderState.IN_DELIVERY || this.state ==
OrderState.DELIVERED)
                throw new OrderStateChangeNotPossibleException("Order
in State " + this.state + " cannot be canceled");
            this.state = newState;
        }
        case PAYMENT_VERIFIED -> {
            if (this.state != OrderState.PLACED)
                throw new OrderStateChangeNotPossibleException("Order
must be in state PLACED for transition to PAYMENT VERIFIED!");
            this.state = newState;
        }
        case PREPARING_FOR_DELIVERY -> {
            if (this.state != OrderState.PAYMENT_VERIFIED)
                throw new OrderStateChangeNotPossibleException("Order

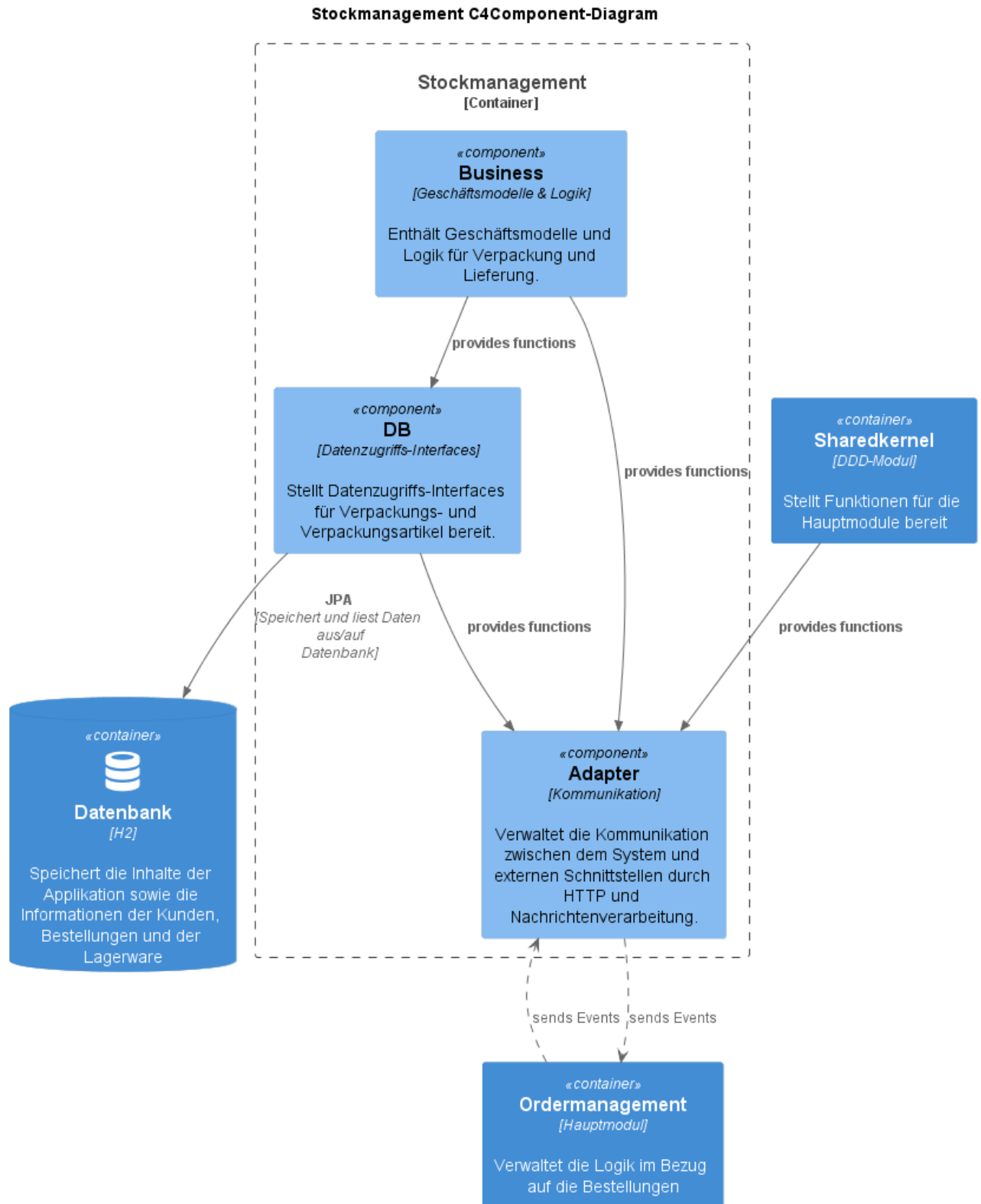
```

```
must be in state PAYMENT VERIFIED for transition to PREPARING FOR
DELIVERY!");
        this.state = newState;
    }
    case IN_DELIVERY -> {
        if (this.state != OrderState.PREPARING_FOR_DELIVERY)
            throw new OrderStateChangeNotPossibleException("Order
must be in state PREPARING FOR DELIVERY for transition to IN
DELIVERY!");
        this.state = newState;
    }
    case DELIVERED -> {
        if (this.state != OrderState.IN_DELIVERY)
            throw new OrderStateChangeNotPossibleException("Order
must be in state IN DELIVERY for transition to DELIVERED!");
        this.state = newState;
    }
}
}
```

Dies ist die Methode, die für das Ändern des Status verantwortlich ist.

2. Dokumentation (textuelle Beschreibung, Codeauszüge, Diagramme, C4-Diagramme, Klassendiagramme) der "Architektur" von Stockmanagement anhand der gegebenen Anwendungsfälle, die schon implementiert sind:

Das C4-Context Diagramm und das C4-Container Diagramm sind ident mit dem Diagrammen vom Ordermanagement. Nun folgt das C4-Component Diagramm des Stockmanagements:



Das vorliegende C4-Component-Diagramm beschreibt die Struktur des "Stockmanagement"-Systems und zeigt die verschiedenen Komponenten und deren Beziehungen zueinander.

Zunächst gibt es drei "Container"-Elemente:

- "Ordermanagement": Dies ist das Hauptmodul, das die Logik im Zusammenhang mit Bestellungen verwaltet.
- "Sharedkernel": Ein Modul im Stil von Domain-Driven Design (DDD), das Funktionen für die Hauptmodule bereitstellt.

- "Datenbank": Eine H2-Datenbank, die die Inhalte der Anwendung sowie Informationen über Kunden, Bestellungen und Lagerbestände speichert.

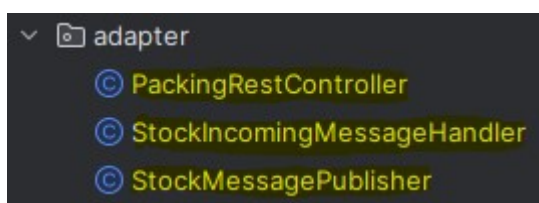
Im "Stockmanagement" Container gibt es drei Komponenten:

- "Adapter": Dieser kümmert sich um die Kommunikation zwischen dem System und externen Schnittstellen über HTTP und Nachrichtenverarbeitung. Er erhält Ereignisse vom "Ordermanagement" und sendet Ereignisse an das "Ordermanagement". Zudem nutzt er Funktionen, die von den Komponenten "Business", "DB" und "Sharedkernel" bereitgestellt werden.
- "Business": Diese Komponente enthält Geschäftsmodelle und Logik für Verpackung und Lieferung. Sie stellt Funktionen für den "Adapter" und die "DB" bereit.
- "DB": Dieses Modul stellt Datenzugriffs-Interfaces für Verpackungs- und Verpackungsartikel bereit. Es nutzt Funktionen, die von der "Business"-Komponente bereitgestellt werden und interagiert mit der Datenbank.

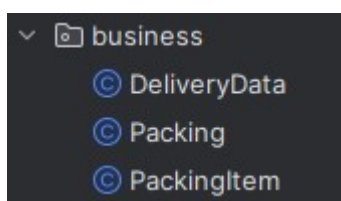
Die "DB"-Komponente interagiert mit der "Datenbank"-Container, indem sie JPA (Java Persistence API) verwendet, um Daten zu speichern und von der Datenbank zu lesen. Die Beziehungen im Diagramm sind teilweise gestrichelt, was auf eine lose Kopplung oder eine asynchrone Kommunikation hinweisen kann.

In Bezug auf Ports und Adapters und Domain-Driven Design (DDD) kann das gegebene "stockmanagement" Paket wie folgt betrachtet werden:

- Adapter (Infrastructure Layer): Das Paket "adapter" repräsentiert die Adapter in der Architektur von Ports und Adapters. Adapter sind dafür verantwortlich, die Kommunikation zwischen der Anwendung und externen Agenten (wie Benutzerschnittstellen, Datenbanken oder externen Systemen) zu handhaben. In diesem Fall enthält das Paket "adapter" die Klassen `PackingRestController`, `StockIncomingMessageHandler` und `StockMessagePublisher`, die alle als Adapter dienen könnten, um mit verschiedenen externen Systemen zu kommunizieren.

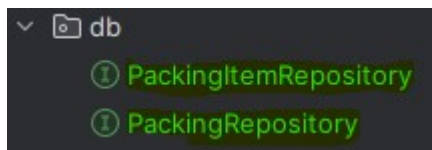


- Business (Domain Layer): Das Paket "business" repräsentiert die Domain Layer in DDD und die Anwendung in Ports und Adapters. Es enthält die Klassen `DeliveryData`, `Packing` und `PackingItem`, die wahrscheinlich Geschäftsobjekte oder Entitäten im Sinne von DDD repräsentieren. Diese Klassen kapseln die Geschäftslogik und Regeln der Anwendung.



- Db (Infrastructure Layer): Das Paket "db" kann als Teil der Infrastrukturschicht in DDD und Ports und Adapters gesehen werden. Es enthält Interfaces `PackingItemRepository` und

PackingRepository, die wahrscheinlich die Datenzugriffsfunktionen zur Interaktion mit einer Datenbank bereitstellen. In der Ports und Adapters Architektur könnten diese Interfaces als "Ports" fungieren, durch die die Anwendung mit der Datenbank kommuniziert.



Da die Anwendungsfälle "Packingliste anlegen & Packingitems als verpackt markieren" eng mit dem Ordermanagement verbunden sind, wurden die Codebeispiele bereits weiter oben eingebaut.