



Componentware

Transaktionen, Packaging, Architektur und Spring Boot



INHALT (1)

- Transaktionen
 - Einführung
 - Container Managed Transactions (CMT)
 - Container Managed Transactions (CMT) - @TransactionAttribute
 - Container Managed Transactions (CMT) – Container Callback Methoden
 - Bean Managed Transactions (BMT)
 - Client Managed Transactions
- Packaging
- Architektur
- Abschließendes Beispiel

INHALT (2)

- Spring-Boot
 - Überblick über Spring
 - Starter
 - Anlegen eines Spring-Boot Projekts
 - Spring Shell
 - Komponenten
 - Umsetzung der bisher kennengelernten Konzepte in Spring-Boot
 - CDI
 - Zugriff auf Datenbanken / Spring Data
 - Messaging
 - Spring Web MVC / RESTful Web Services
 - Transaktionen

EINFÜHRUNG

- Eine der wichtigsten Aufgaben des Entwicklers von Enterprise Systemen stellt die Aufrechterhaltung der Systemkonsistenz dar.
- Falls von einem Geschäftsprozess mehr als eine Aktion durchgeführt wird, muss sichergestellt sein, dass entweder alle Einzelschritte oder gar keiner der Schritte durchgeführt wird.
- Nehmen wir an, nach einer Bestellung muss zunächst der Bestand im Warenwirtschaftssystem reduziert werden und zudem muss die Bestellung dem Spediteur mitgeteilt werden.
- Beide Schritte müssen zusammen ablaufen, denn wenn der Bestand reduziert wird, ohne dass die Bestellung an den Spediteur gesendet wird, enthält das System anschließend eine falsche Bestandsanzahl, da die Ware nicht versendet wurde und sich somit noch im Lager befindet.

EINFÜHRUNG

- Falls der Spediteur den Versandauftrag bekommt, aber die Bestandsreduzierung nicht stattfindet, führt dies ebenfalls zu einem inkonsistenten Zustand, da laut System mehr Waren auf Lager sind, als tatsächlich beim Spediteur im Lager sind.
- Es müssen somit beide Schritte oder gar keiner durchgeführt werden.
- Gerade, wenn mit Datenbanken gearbeitet wird, ist die Systemkonsistenz überaus wichtig, denn es darf niemals dazu kommen, dass nach einem Fehler die Datenbankdaten nicht mehr verlässlich sind.
- Da in der Regel die einzelnen Schritte bis zur Inkonsistenz automatisiert abgelaufen sind, kann der korrekte Datenbestand in der Regel nicht oder nur mit extrem hohem Aufwand manuell wieder hergestellt werden, was dazu führt, dass die Daten praktisch wertlos wurden.

EINFÜHRUNG

- Um Inkonsistenzen zu vermeiden, stellt der Jakarta Enterprise Standard das Jakarta Transactions API (JTA) zur Verfügung, das diverse Möglichkeiten der Transaktionsverwaltung bietet.

EINFÜHRUNG

- Eine Aneinanderreihung von verschiedenen Einzelaktionen führt der Application Server zu einer sogenannten Transaktion zusammen.
- Dabei wird vor der ersten Ausführung eines Teilschrittes eine solche Transaktion eröffnet.
- Anschließend wird die Aktivität ausgeführt, wobei dafür gesorgt wird, dass der Zustand, der vor dem Start der Aktivität vorlag, zwischen gespeichert wird.
- Falls es bei der Anweisung zu einem Fehler kommt, markiert der Server die Transaktion als fehlerhaft.
- Anschließend wird die nächste Aktivität ausgeführt, usw.

EINFÜHRUNG

- Falls alle Aktionen fehlerfrei durchlaufen, bestätigt der Server die Transaktion bei allen Teilschritten als erfolgreich (commit) und es werden die zwischengespeicherten Zustände gelöscht.
- Wenn die Transaktion während des Ablaufs als ungültig markiert wurde, stellt der Application Server den Zustand, der vor der Transaktion galt, wieder her (rollback).
- In beiden Fällen ist die Transaktion danach beendet.

EINFÜHRUNG

- Solange die Aufrufe von Methoden entweder
 - vom Application Server selbst über einen Lookup,
 - durch Injection oder
 - vom Client über ein Proxy Objekterfolgen, hat der Application Server stets den Überblick über die Einzelaktionen.
- Sämtliche dabei gesammelten Informationen über die Transaktion werden dem Entwickler im sogenannten Transaktionskontext zur Verfügung gestellt.
- Der Transaktionskontext wird dabei entlang der Aufrufkette an jeden einzelnen Teilschritt weitergeleitet (propagiert).

EINFÜHRUNG

- Eine Transaktion umschließt somit die Einzelschritte eines Prozesses.
- Der Application Server garantiert dabei, dass eine Transaktion das sogenannte ACID-Prinzip erfüllt:
 - **A**tomicity: Alle Teilschritte oder keiner wird ausgeführt.
 - **C**onsistency: Nach der Ausführung ist der Datenbestand immer noch konsistent.
 - **I**solation: Mehrere Transaktionen dürfen sich nicht gegenseitig beeinflussen.
 - **D**urability: Die Ergebnisse der Transaktion müssen dauerhaft sein und dürfen zum Beispiel durch einen Neustart des Application Servers nicht verloren gehen.

EINFÜHRUNG

- Bei verteilten System gestalten sich Transaktionen sehr komplex, da oft nicht nur ein System, sondern eben viele Systeme beteiligt sind.
- Bei den bisherigen Beispielen verteilte sich der Zustand z.B. bereits auf Stateful Session Beans, die Messaging oder Web Services verwenden und ihre Daten in der Datenbank ablegen.
- Wichtig ist, dass die beteiligten Systeme Transaktionen unterstützen, denn nur so kann der Application Server die Transaktionssicherheit garantieren.

EINFÜHRUNG

- Sofern mehrere Systeme an der Ausführung beteiligt sind, erfolgt die Ausführung der Transaktion dabei durch das sogenannte 2-Phase-Commit-Verfahren.
- Dabei veranlasst der Transaktionskoordinator (in unserem Fall also der Application Server) in einer ersten Phase zunächst die Ausführung der Aktionen bei den einzelnen Systemen.
- Falls die einzelnen Schritte erfolgreich sind, gibt der Transaktionskoordinator erst in einem zweiten Schritt die endgültige Bestätigung zur Festschreibung oder im Fehlerfalle die Aufforderung zum Rollback.
- Ein potientiellies Zurückrollen erfolgt somit erst beim Auflösen des Transaktionskontexts am Ende der Transaktion.

EINFÜHRUNG

- In Java kann die Durchführung von Transaktionen an den Application Server abgegeben werden (Container Managed Transactions -> CMT).
- Auf Wunsch kann sich das Programm jedoch auch selbst um die Transaktionsverwaltung kümmern (Bean Managed Transactions -> BMT).
- Zudem besteht auch die Möglichkeit, Transaktionen durch den Client zu verwalten (Client verwaltete Transaktionen)
- CMT ist dabei die Voreinstellung und wird somit automatisch genutzt, wenn EJBs verwendet werden.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Bei Container Managed Transactions kümmert sich der Container selbst um das Eröffnen der notwendigen Transaktionen.
- Der Container überwacht dabei auch selbständig den Erfolg oder Misserfolg von Transaktionen und leitet das entsprechende Commit oder einen Rollback ein.
- Eine Bean kann für CMTs vorgesehen werden, indem die Klasse mit `@TransactionManagement (`
`TransactionManagementType.CONTAINER)`
annotiert wird.
- CMT ist dabei die Voreinstellung für `@TransaktionManagement`.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Containerverwaltete Transaktionen werden vom Container bei einem Aufruf einer Methode durch den Client gestartet.
- Mit dem Ende der Methode endet dabei in der Regel auch die Transaktion.
- Während der Transaktion benutze Ressourcen, wie zum Beispiel JMS-Verbindungen oder Datenbankverbindungen, werden dabei ebenfalls in den Transaktionskontext aufgenommen.
- Daher werden z.B. JMS-Nachrichten oder Datenbankstatements erst am Ende der Transaktion versendet.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Auch wenn Transaktionen verwendet werden und die Transaktionen vom Container verwaltet werden, kann es dennoch zu Inkonsistenzen bei der Abarbeitung kommen.
- Um dieses nachvollziehen zu können, ist es wichtig zu verstehen, wann der Container eine Transaktion als erfolgreich und wann eben nicht, bewertet.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Ein Rollback wird in folgenden Fällen ausgelöst:
 1. Die ausgeführte Methode wird mit einer System-Exception verlassen.
 - Im EJB-Umfeld ist eine System-Exception als eine Exception, die von `RuntimeException` oder `java.rmi.RemoteException` abgeleitet wurde, definiert.
 - Hierbei handelt es sich oft um technische Probleme, die vom Entwickler schwerlich behandelt werden können, wie z.B. eine `NullPointerException`.
 - Solche Exceptions werden vom Application Server nicht direkt weitergeleitet, sondern in eine Application-Exception (= checked Exception) verpackt und in dieser Form an den Client geleitet.
 - Die ursprüngliche System-Exception kann dabei der *caused by* Klausel entnommen werden.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Ein Rollback kann bei System-Exceptions verhindert werden, indem bei der (`RuntimeException` erweiternden) Exceptionklasse die Annotation `@ApplicationException` angegeben wird.
- Falls es in einer Stateful oder Stateless Session Bean zu einer System-Exception kommt, wird die Bean im Anschluss zerstört.
- Bei Stateless Session Beans ist dies nicht sofort ersichtlich, da die Bean Instanzen einem Pool entnommen werden.
- Die Instanz einer Stateful Session Bean kann jedoch nach einer System-Exception nicht mehr verwendet werden, so dass ein Aufruf einer Methode auf dem Beanobjekt zu einer Exception führt.

CONTAINER MANAGED TRANSACTIONS (CMT)

2. Die ausgeführte Methode wird mit einer Exception verlassen, die zwar nicht von `RuntimeException` abgeleitet ist, aber mit der Annotation `@ApplicationException(rollback=true)` versehen wurde.
 - Als Application-Exception werden im EJB-Umfeld checked Exceptions, also Exceptions, die von `Exception`, aber nicht von `RuntimeException` oder `java.rmi.RemoteException` abgeleitet wurden, bezeichnet.
 - Bei Application-Exceptions kommt es nicht zu einem Rollback, es sei denn, sie wurden mit der obigen Annotation versehen.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Da es sich bei Application Exceptions um checked-Exceptions handelt, müssen sie im Methodenkopf angegeben und entsprechend vom Entwickler berücksichtigt werden.
- In der Regel werden solche Exceptions gefangen und bearbeitet, daher spricht man hierbei auch von fachlichen Fehlern.
- Durch die Angabe der Annotation `@ApplicationException` bei selbst erstellten Runtime-Exceptions, werden auch Runtime-Exceptions zu Application-Exceptions, wodurch der Rollback verhindert wird.
- Die catch-or-throws-Regel hat natürlich dabei trotzdem keine Gültigkeit, da es sich auch mit der Annotation immer noch um eine Runtime-Exception handelt.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Application-Exceptions werden im Gegensatz zu System-Exceptions vom Application Server direkt an den aufrufenden Client weitergeleitet.

CONTAINER MANAGED TRANSACTIONS (CMT)

3. In der Methode wird die Transaktion durch den Entwickler als Misserfolg markiert.

- Hierzu kann der Entwickler die Methode `setRollbackOnly()` auf dem `SessionContext`-Objekt aufrufen.
- Der Session Context kann in eine Bean injiziert werden, indem ein Attribut vom Typ `SessionContext` vereinbart wird, das mit `@Resource` annotiert wird (keine Attribute notwendig).

- In allen anderen Fällen wird die Transaktion als erfolgreich bewertet.
- Nochmal: Checked-Exceptions führen in der Regel nicht zu einem Rollback.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Zu Inkonsistenzen kann es nun insbesondere dann kommen, wenn es innerhalb einer Transaktion eine Application-Exception gibt, die von der Methode an den Aufrufer weitergeworfen wird.
- In diesem Fall kommt es seitens des Application Servers nicht zum Rollback.
- Falls danach unachtsam von Entwicklern eine weitere Aktion ausgeführt wird, kann es sein, dass die erste Aktion nicht ausgeführt wurde, aber die zweite trotzdem durchgeführt wird.
- Siehe:
Componentware_Kapitel6_Transaktionen_Demo/konto/*.java
und
Componentware_Kapitel6_Transaktionen_Demo_TestClient/clients/KontoClient.java

CONTAINER MANAGED TRANSACTIONS (CMT)

- Hier bestand das Problem insbesondere darin, dass es sich bei der Exception um eine Application-Exception handelte.
- Da der Application Server ein Rollback nur bei System-Exceptions vorsieht, kommt es hier nicht zum Zurückrollen der Transaktion.
- Es gibt nun mehrere Möglichkeiten das fehlerhafte Verhalten zu beseitigen:
 - Die `KontoException` wird statt von `Exception` nun von `RuntimeException` abgeleitet.

Hierdurch wird jedoch die catch-or-throws-Regel außer Kraft gesetzt, es entfällt also die Notwendigkeit für die Entwickler sich um die Exception zu kümmern.

CONTAINER MANAGED TRANSACTIONS (CMT)

- Die Methode kann im Fehlerfall die Transaktion selbst für einen Rollback vorsehen.

Dazu wird die Methode `setRollbackOnly()` am injizierten `SessionContext` aufgerufen.

- Die Exception wird mit der Annotation `@ApplicationException` versehen, die das Attribut `rollback=true` erhält.

Dadurch wird die Exception einen Rollback verursachen.

```
@ApplicationException(rollback=true)
public class KontoException extends Exception{
    ...
}
```

Dies führt zudem bei `RuntimeExceptions` dazu, dass die Exception nicht mehr in eine EJB-Exception verpackt wird, sondern direkt zum Client gesendet wird.

CMT - @TRANSACTIONATTRIBUTE

- Neben dem Remote Client können auch lokale Clients Methoden einer Bean aufrufen.
- Falls bei einem lokalen Aufruf bereits ein Transaktionskontext existiert, wird dieser Kontext der aufgerufenen Bean übergeben (propagiert), so dass diese den Kontext weiter verwenden kann.
- Da aber nicht für jede Methode unbedingt eine Transaktion notwendig ist, kann das Transaktionsverhalten einzelner Methoden angepasst werden.
- Dazu wird die Annotation `@TransactionAttribute` verwendet:
`@TransactionAttribute(TransactionAttributeType.<VALUE>)`
- Bei Lifecycle-Callback-Methoden ist eine solche Angabe nicht erlaubt.

CMT - @TRANSACTIONATTRIBUTE

- Als Value stehen dabei folgende Werte zur Verfügung:

- REQUIRED

Dieses Attribut drückt aus, dass die Methode einen Transaktionskontext benötigt.

Falls zuvor ein Kontext für eine vorhergehende Methode erzeugt wurde, wird dieser weiter verwendet.

Sofern noch kein Transaktionskontext existierte, wird ein neuer angelegt.

Dieses Attribut wird meistens verwendet und ist daher die Voreinstellung für `@TransactionAttribute`.

CMT - @TRANSACTIONATTRIBUTE

- REQUIRED (Fortsetzung)

Der Einsatz dieses Parameters ist immer dann sinnvoll, wenn die Methode Teil eines größeren Geschäftsprozesses ist, der bereits eine Transaktion verwaltet.

Ein neuer Kontext für jede Methode innerhalb dieses Prozesses würde die Konsistenz und Integrität der Transaktion gefährden.

CMT - @TRANSACTIONATTRIBUTE

- REQUIRED (Fortsetzung)

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void zumWarenkorbHinzufügen(Produkt produkt) {
    // Artikel wird dem Warenkorb hinzugefügt
    warenkorb.add(produkt);
}
```

Wenn z. B. in einem Online-Shop ein Artikel zu einem Warenkorb hinzugefügt wird, geschieht dies eventuell im Rahmen einer anderen Methode, die bereits eine Transaktion verwaltet.

Ein neuer Kontext für jede Methode innerhalb dieses Prozesses würde die Konsistenz und Integrität der Transaktion gefährden, da die Methode dann ein eigenes Rollbackverhalten hätte.

CMT - @TRANSACTIONATTRIBUTE

- `REQUIRES_NEW`

Dieses Attribut drückt aus, dass die Methode einen eigenen Transaktionskontext benötigt.

Falls die vorhergehende Methode also bereits einen Kontext mitbringt, wird dieser pausiert und für die aufgerufene Methode ein neuer Kontext verwendet.

Unabhängig vom Ergebnis des neuen Transaktionskontexts wird nach dem Ende der Transaktion ein eventuell pausierter Kontext wieder aktiviert.

CMT - @TRANSACTIONATTRIBUTE

- `REQUIRES_NEW` (Fortsetzung)

Der Einsatz dieses Parameters ist immer dann sinnvoll, wenn eine eigene, unabhängige Transaktion benötigt wird, die nicht vom Zustand der äußeren Transaktion abhängt.

Wenn eine Methode z. B. Fehlerprotokollierung oder Logging in einer eigenen Transaktion durchführen muss, unabhängig davon, ob die Haupttransaktion fehlschlägt oder erfolgreich ist, ist dieser Parameter sinnvoll.

CMT - @TRANSACTIONATTRIBUTE

- REQUIRES_NEW (Fortsetzung)

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void loggeAbbuchung(Konto konto, double betrag) {
    // Transaktion wird immer in einer neuen Transaktion
    // gespeichert
    logger.log("Abbuchung von Konto " +
               konto.getNummer() + ":" + betrag);
}
```

Bei einer Banküberweisung, bei der eine Haupttransaktion für das Abbuchen von einem Konto und das Hinzufügen auf einem anderen Konto verantwortlich ist, könnte das Logging der Transaktion in einer separaten Methode mit dem Attribut `REQUIRES_NEW` ausgeführt werden.

Falls das Protokollieren fehlschlägt, wird die Haupttransaktion nicht beeinträchtigt.

CMT - @TRANSACTIONATTRIBUTE

- MANDATORY

Dieses Attribut drückt aus, dass die Methode einen bereits vorhandenen Kontext erwartet.

Falls noch kein Kontext existiert, wird kein neuer erstellt, sondern es kommt zu einer `TransactionRequiredException`.

CMT - @TRANSACTIONATTRIBUTE

- MANDATORY (Fortsetzung)

Diese Option ist sinnvoll, wenn die Methode auf jeden Fall innerhalb einer Transaktion ausgeführt werden soll.

Wenn keine Transaktion vorhanden ist, würde die Methode nicht ausgeführt werden, was zu einem Fehler führt.

Das ist z. B. sinnvoll für kritische Geschäftsprozesse, bei denen die Methode garantiert Teil einer Transaktion sein muss, da ohne Transaktion keine Konsistenz gewährleistet werden kann.

CMT - @TRANSACTIONATTRIBUTE

- MANDATORY (Fortsetzung)

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public void ueberweise(Konto von, Konto an, double betrag) {
    // Diese Methode muss in einer bestehenden
    // Transaktion ausgeführt werden
    if (von.getSaldo() < betrag) throw new SaldoException();
    von.zahleAus(betrag);
    an.zahleEin(betrag);
}
```

Diese Option ist sinnvoll, wenn die Methode auf jeden Fall innerhalb einer Transaktion ausgeführt werden soll.

Nehmen wir an diese Methode wird stets im Rahmen einer anderen Methode aufgerufen.

Wenn für diese andere Methode keine Transaktion vorhanden wäre, darf die hier implementierte Methode nicht ausgeführt werden, da sonst gar keine Transaktion vorhanden wäre.

CMT - @TRANSACTIONATTRIBUTE

- SUPPORTS

Dieses Attribut zeigt an, dass die Methode keinen Kontext benötigt, es sie jedoch auch nicht stört, wenn ein bereits existierender Kontext propagiert wird.

Dies ist nützlich für Methoden, die keine Transaktion benötigen, aber in einer bestehenden Transaktion ausgeführt werden können, wenn diese vorhanden ist.

Das ist ideal für Optionen, bei denen Transaktionssicherheit gewünscht wird, aber nicht erforderlich ist.

CMT - @TRANSACTIONATTRIBUTE

- SUPPORTS (Fortsetzung)

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public Order getBestellDetails(long bestellnummer) {
    return bestellverwaltung.liesBestellung(bestellnummer);
}
```

Eine Methode in einem Online-Shop, die Bestellinformationen anzeigt, könnte mit SUPPORTS annotiert sein.

Wenn bereits eine Transaktion besteht (z. B. um den Bestellstatus zu aktualisieren), wird die Lesemethode innerhalb dieser Transaktion ausgeführt, andernfalls wird sie ohne Transaktion ausgeführt.

CMT - @TRANSACTIONATTRIBUTE

- NOT_SUPPORTED

Diese Angabe drückt aus, dass die Methode keinen Kontext benötigt.

Falls von der vorhergehenden Methode ein bereits existierender Kontext propagiert wurde, wird dieser unterbrochen.

Typische Szenarien:

- Performance-intensive Methoden, die keine atomare Ausführung benötigen,
- oder Methoden, die keinen Einfluss auf die Transaktionskonsistenz haben (z. B. das Cachen von Informationen).

CMT - @TRANSACTIONATTRIBUTE

- NOT_SUPPORTED (Fortsetzung)

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public String ermittleConfigWert(String key) {
    return configCacheBean.ermittleWert(key);
}
```

Nehmen wir an, es gäbe eine externe Datenbank zur Speicherung von Konfigurationseinstellungen, z. B. welche Features aktiv sind oder in der UI-Konfigurationen gespeichert sind.

Diese Daten ändern sich selten und sollen in einem lokalen Cache zwischengespeichert werden, um Datenbanklast zu vermeiden.

Da das Lesen und Zwischenspeichern von Konfigurationsdaten keinen transaktionalen Kontext benötigt und nicht durch ein mögliches Rollback betroffen sein soll, ist NOT_SUPPORTED sinnvoll.

CMT - @TRANSACTIONATTRIBUTE

- NOT_SUPPORTED (Fortsetzung)

Das Caching erfolgt unabhängig von laufenden Transaktionen.

Wenn die Methode innerhalb einer aktiven Transaktion aufgerufen wird (z. B. während der Verarbeitung einer Bestellung), wird die Transaktion ausgesetzt, um unnötige Transaktionsverwaltungskosten zu vermeiden.

Der Zugriff auf die Konfiguration soll nicht zurückgerollt werden, etwa bei einem Fehler in einer späteren Datenbankoperation.

CMT - @TRANSACTIONATTRIBUTE

- NEVER

Diese Angabe drückt aus, dass die Methode keinen Kontext akzeptiert.

Falls von der vorhergehenden Methode ein bereits existierender Kontext propagiert wurde, kommt es zu einer `EJBException`.

CMT - @TRANSACTIONATTRIBUTE

- NEVER (Fortsetzung)

Das ist dann sinnvoll, wenn sicher ausgeschlossen werden soll, dass die Methode versehentlich in einer Transaktion ausgeführt wird – etwa weil die Methode:

- Unterschiedliches Verhalten bei doppeltem Aufruf hat,
- oder externe Systeme aufruft, die ihrerseits keine Transaktionen unterstützen (z. B. ältere REST- oder SOAP-Dienste).

Das Verhalten dient als Sicherheitsnetz, um Transaktionen explizit zu verbieten und bei Zuwiderhandlung einen Fehler zu erzeugen.

CMT - @TRANSACTIONATTRIBUTE

- NEVER (Fortsetzung)

```
@Stateless
@TransactionAttribute(TransactionAttributeType.NEVER)
public boolean prüfeLizenz(String lizenzschluessel) {
    // Aufruf eines externen Lizenzprüfungsdienstes
    return extererLizenzdienst.prüfe(lizenzschluessel);
}
```

Stellen wir uns ein System vor, das eine externe Lizenzprüfung durchführt.

Der Lizenzserver ist ein externer Dienst, der keine Transaktionen unterstützt und bei doppeltem Aufruf Lizenzen mehrfach zählt.

Durch den zwei Phasen Commit wird es jedoch zu einem doppelten Aufruf kommen.

CMT - @TRANSACTIONATTRIBUTE

- Sobald eine Methode mit einem Fehler endet, markiert der Application Server den Transaktionskontext als ungültig.
- Ein weiterer Aufruf einer Methode mit dem gleichen Kontext führt auch bei Angabe des Attributs `REQUIRED` zu einer `TransactionRequiredException`, da zwar ein Kontext vorhanden ist, dieser aber ungültig ist.
- Da bereits ein Kontext vorhanden ist, wird auch bei `REQUIRED` kein neuer Kontext angelegt.

CMT - @TRANSACTIONATTRIBUTE

- Der Container und damit auch der Transaktionskontext registriert nur Methodenaufrufe, die über lookups, injections oder ein Proxy-Objekt erfolgen.
- Falls eine EJB-Methode jedoch eine Methode der gleichen Klasse aufruft, z. B. über `this`, kann der Container dies nicht registrieren.
- Sofern bei einem solchen Aufruf die aufgerufene Methode zudem noch ein anderes Transaktionsattribut hat, kann dies zu fehlerhaftem Verhalten führen.

CMT - @TRANSACTIONATTRIBUTE

- Beispiel:

```
@Stateless
public class KontoVerwaltung implements KontoVerwaltungRemote {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void ueberweise(Konto von, Konto an, double betrag) {
        if (von.getSaldo() < betrag) throw new SaldoException();
        von.zahleAus(betrag);
        this.loggeAbbuchung(von, betrag);
        an.zahleEin(betrag);
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void loggeAbbuchung(Konto konto, double betrag) {
        logger.log("Abbuchung von Konto " +
                    konto.getNummer() + ":" + betrag);
    }
}
```

CMT - @TRANSACTIONATTRIBUTE

- Offenbar soll `loggeAbbuchung()` in einer separaten Transaktion ablaufen.
- Da der Container den Aufruf jedoch nicht steuert, kann für `loggeAbbuchung()` auch keine neue Transaktion erzeugt werden.
- In einem solchen Fall muss somit dafür gesorgt werden, dass der Container den Aufruf registriert.
- Dies kann geschehen, indem der Aufruf nicht über `this`, sondern über ein registriertes Proxy Objekt erfolgt.
- In diesem Fall muss eine andere Bean verwendet werden, deren Objekt z. B. über `@EJB` injiziert wird.

CMT - @TRANSACTIONATTRIBUTE

- Beispiel:

```
@Stateless
public class LogService {
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void loggeAbbuchung(Konto konto, double betrag) {
        logger.log("Abbuchung von Konto " +
            konto.getNummer() + ":" + betrag);
    }
}

@Stateless
public class KontoVerwaltung implements KontoVerwaltungRemote {
    @EJB
    private LogService logService;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void ueberweise(Konto von, Konto an, double betrag) {
        if (von.getSaldo() < betrag) throw new SaldoException();
        von.zahleAus(betrag);
        logService.loggeAbbuchung(von, betrag);
        an.zahleEin(betrag);
    }
}
```

CMT - @TRANSACTIONATTRIBUTE

- Alternativ zum Aufruf der Methode einer anderen Bean, kann auch die Methode `getBusinessObject(<NameDerKlasse>.class)` verwendet werden, die auf einem Objekt des Session Context aufgerufen wird.

```
@Stateless
public class KontoVerwaltung implements KontoVerwaltungRemote {

    @Resource
    private SessionContext sessionContext;

    public void ueberweise(Konto von, Konto an, double betrag) {
        if (von.getSaldo() < betrag) throw new SaldoException();
        von.zahleAus(betrag);
        KontoVerwaltung self=
            sessionContext.getBusinessObject(KontoVerwaltung.class);
        self.loggeAbbuchung(von, betrag);
        an.zahleEin(betrag);
    }
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void loggeAbbuchung(Konto konto, double betrag) {
        logger.log("Abbuchung von Konto " +
            konto.getNummer() + ":" + betrag);
    }
}
```

CMT - @TRANSACTIONATTRIBUTE

- Wie oben geschildert, ist bei Lifecycle-Callback-Methoden eine Angabe der Annotation `@TransactionAttribute` nicht erlaubt.
- Falls nun aber ein solches Attribut notwendig ist, müssen die Anweisungen der Callback-Methode an eine Bean-Methode delegiert werden.
- Hierzu ist somit ebenfalls ein beaninterner Aufruf notwendig, der dann wiederum über ein Proxy-Objekt erfolgen muss.

CMT - @TRANSACTIONATTRIBUTE

- Da Message Driven Beans nicht direkt vom Client, sondern asynchron aufgerufen werden, können diese Beans niemals einen schon vorher definierten Transaktionskontext erhalten.
- Daher ist bei MDBs nur die Verwendung von `REQUIRED`, `REQUIRES_NEW` und `SUPPORTS` möglich.

CMT - CONTAINER CALLBACK METHODEN

- Bei Stateful Session Beans existiert die Bean in der Regel länger als für die Dauer einer Transaktion.
- Dies kann dazu führen, dass z.B. in Abhängigkeit von dem Ergebnis einer Transaktion noch Aufgaben zu erledigen sind.
- Zum Beispiel könnte es sein, dass ein zwischengespeicherter Attributswert nach einem Rollback wiederhergestellt werden soll.
- Beispiel: Bei einem Online-Shop kann die Bestellverwaltung über Stateful Session Beans abgewickelt werden.

Hierbei sollte auch nach einem Fehler im Bestellablauf der Warenkorb noch vorhanden sein.

Siehe: [Componentware_Kapitel6_Transaktionen_Demo/warenkorb1](#)

CMT - CONTAINER CALLBACK METHODEN

- Hierbei wird in der Methode `schliesseKaufAb()` sowohl der Versand beauftragt, als auch eine Bestätigungsemail an den Kunden gesendet.
- Nachdem die Produkte an den Lieferanten versendet wurden, erfolgt eine Löschung der Waren im Warenkorb.
- Sofern es keinen Fehler gibt, sind nachher alle Versandaufträge für die Produkte beim Lieferanten, die Mail wurde versendet und der Bestand ist wieder gelöscht.

CMT - CONTAINER CALLBACK METHODEN

- Falls es beim Versenden zum Lieferanten zu einem Fehler kommt, wird die Transaktion zurückgerollt.
- Da der Warenkorb erst nach dem kompletten Versand als letztes gelöscht wird, steht er also nach einem Abbruch des Versands noch zur Verfügung.
- Dies ist genau das gewünschte Verhalten.

CMT - CONTAINER CALLBACK METHODEN

- Falls es jedoch in der Methode `sendeVersandbestaetigung()` zu einem Fehler kommt, ist die Methode `sendeZumLieferanten()` bereits komplett durchlaufen worden.
- In dieser Methode wurde der Warenkorb jedoch gelöscht.
- Auch wenn es zu einem Rollback kommt, ist der Warenkorb nachher geleert.
- Es wäre also gut, wenn wir uns den Inhalt der Produktliste zwischenspeichern könnten und diesen Inhalt dann im Fehlerfall wiederherstellen könnten.
- Dazu müssten wir jedoch den Fehlerfall “signalisiert” bekommen.

CMT - CONTAINER CALLBACK METHODEN

- Um ein solches Verhalten zu ermöglichen, existiert **für Stateful Session Beans** die sogenannte Session Synchronisierung.
- Die Session Synchronisierung ermöglicht, zu bestimmten Zeitpunkten über den Zustand von Transaktionen informiert zu werden.
- So können Methoden erstellt werden, die automatisch durch den Application Server zu bestimmten Zeitpunkten aufgerufen werden (Container-Callback-Methoden).
- So kann zum Beispiel eine Methode programmiert werden, die vor dem Ende einer Transaktion durch den Application Server aufgerufen wird.
- Eine andere Methode kann aufgerufen werden, sobald die Transaktion beendet wurde, usw.

CMT - CONTAINER CALLBACK METHODEN

- Dazu kann eine Bean entweder das Interface `SessionSynchronization` mit den zugehörigen Methoden implementieren, oder es können Annotationen an Methoden beliebigen Namens geschrieben werden, wobei die Namen der Annotationen den Bezeichnern der Methoden des Interfaces entsprechen.

CMT - CONTAINER CALLBACK METHODEN

- Das Interface enthält folgende Methoden oder Annotationen:

- `public void afterBegin() / @AfterBegin`

Diese Methode wird nach dem Beginn der Transaktion aufgerufen.

Alternativ wird die Methode aufgerufen, wenn eine Instanz der implementierenden Bean erzeugt wurde und es bereits zuvor eine Transaktion gab.

- `public void beforeCompletion() /
@BeforeCompletion`

Diese Methode wird in der ersten Phase des Zwei-Phasen-Commits aufgerufen.

CMT - CONTAINER CALLBACK METHODEN

- `public void afterCompletion(boolean commit) /
@AfterCompletion`

Diese Methode wird in der zweiten Phase des Zwei-Phasen-Commits aufgerufen.

Der Parameter enthält `true`, wenn es ein finales commit gab und er enthält `false`, wenn es zu einem Rollback gekommen ist.

In dieser Methode muss der Zustand wieder hergestellt werden, den die Bean vor dem Aufruf von `afterBegin()` hatte.

Falls dieses Verhalten durch eine annotierte Methode gestaltet werden soll, muss die entsprechende Methode ebenfalls einen boolean-Parameter erhalten.

- Siehe dazu
[Componentware_Kapitel6_Transaktionen_Demo/warenkorb2](#)

BEAN MANAGED TRANSACTIONS (BMT)

- Alternativ zu Container verwalteten Transaktionen kann der Entwickler die Steuerung der Transaktionen auch selbst vornehmen.
- Dabei bestimmt der Entwickler, wann eine Transaktion beginnt und ob und wann eine Transaktion erfolgreich bestätigt oder ggf. aufgrund eines Fehlers zurückgerollt wird.
- Dieses Verfahren wird Bean Managed Transaction (BMT) genannt.
- Eine Bean kann für BMTs vorgesehen werden, indem die Klasse mit `@TransactionManagement(TransactionManagementType.BEAN)` annotiert wird.

BEAN MANAGED TRANSACTIONS (BMT)

- Um Transaktionsklammern zu setzen, stellt das JTA das Interface `UserTransactions` bereit, welches diverse Methoden zur Transaktionssteuerung enthält.
- Um Zugriff auf die Methoden des Interface zu erhalten, kann in Beans mit BMT auf dem `SessionContext`-Objekt die Methode `getUserTransaction()` aufgerufen werden.
- Falls die Methode aufgerufen wird, nachdem von der Bean bereits ein Transaktionskontext angelegt wurde, liefert die Methode erneut den zuvor angelegten Kontext zurück.

BEAN MANAGED TRANSACTIONS (BMT)

- Das Interface `UserTransactions` enthält insbesondere die folgenden Methoden, die alle eine `SystemException` werfen können:

- `begin()`

Die Methode startet eine neue Transaktion.

Zu beachten ist, dass Bean verwaltete Transaktionen nicht an Transaktionen teilnehmen können, die der Methode propagiert werden.

Daher startet die Methode jedes mal eine neue Transaktion.

Umgekehrt kann eine durch eine Bean erzeugte Transaktion jedoch an eine Methode einer CMT-Bean übergeben werden.

Wenn `begin()` aufgerufen wird und bereits eine Transaktion besteht, kommt es zu einer `NotSupportedException`.

BEAN MANAGED TRANSACTIONS (BMT)

- `commit()`

Die Methode beendet die aktuelle Transaktion und bestätigt die Änderungen.

Falls keine Transaktion besteht, wird eine `IllegalStateException` geworfen.

- `rollback()`

Die Methode beendet die aktuelle Transaktion und verwirft die Änderungen.

Nach dem Aufruf dieser Methode darf nicht im weiteren Verlauf noch die Methode `commit()` aufgerufen werden.

Falls keine Transaktion besteht, wird eine `IllegalStateException` geworfen.

BEAN MANAGED TRANSACTIONS (BMT)

- Nachdem eine Transaktion über `commit()` oder `rollback()` beendet wurde, kann die Transaktion nicht erneut geöffnet werden.
- Stattdessen muss in diesem Fall über `getUserTransaction()` ein neuer Kontext angelegt werden.

BEAN MANAGED TRANSACTIONS (BMT)

- Ein hohes Fehlerpotential stellen BMTs in Stateful Session Beans dar.
- Da Transaktionen in Stateful Session Beans auch über mehrere Methoden hinweg geöffnet bleiben können, muss ein besonderes Augenmerk darauf gelegt werden, die Transaktion auch wieder zu beenden.

CLIENT MANAGED TRANSACTIONS

- Neben den vom Container und den Beans verwalteten Transaktionen, kann auch ein Client eine Transaktion starten.
- Bei jedem Client-Aufruf einer Methode wird bei Container verwalteten Transaktionen zunächst eine Transaktion gestartet, die in der Regel mit der Methode endet.
- Wenn nun ein Client mehrere Methodenaufrufe in einer Transaktion durchführen möchte, geht das mit den bislang beschriebenen Mitteln nicht.
- Ein weiteres Problem stellt sich, wenn ein Client Aufrufe auf mehreren Servern durchführen möchte, die alle zusammen in einer Transaktion ablaufen.
- Um dies zu erreichen, müsste somit der Client die Transaktion starten.

CLIENT MANAGED TRANSACTIONS

- Vom Ablauf her funktioniert das Verwalten von Transaktionen durch den Client genauso wie bei BMT.
- Zunächst muss wieder einer Instanz der UserTransaction ermittelt werden.
- Da sich im Client jedoch nicht einfach eine Variable vom Typ SessionContext injizieren lässt, muss hier der Aufruf über `lookup(..)` erfolgen.

```
InitialContext ctx = new InitialContext(..);  
  
UserTransaction tx =  
    (UserTransaction) ctx.lookup("UserTransaction");
```

CLIENT MANAGED TRANSACTIONS

- Wie bei BMT kann die Transaktionsverwaltung nun über die Methoden
`begin()`
`commit()`
`rollback()`
erfolgen.

CLIENT MANAGED TRANSACTIONS

- Der Jakarta EE Standard empfiehlt den Serverherstellern einen Aufruf durch den Client zu ermöglichen.
- Da dies jedoch keinen Zwang darstellt, kann die Möglichkeit eines Clientaufrufs unter Umständen von Plattform zu Plattform unterschiedlich sein.

Packaging



PACKAGING

- In den vergangenen Kapiteln wurde für die Jakarta Enterprise Beans stets ein Jakarta EE Library Projekt angelegt und darin ein EJB Application Archive als Artifact erstellt.
- Dies hat dazu geführt, dass IntelliJ für dieses Modul ein Java-Archive (jar) erstellte, welches in das Deploy-Verzeichnis des Servers kopiert wurde.
- Dieses Vorgehen wird als Packaging & Deployment bezeichnet.
- Für den Stand-Alone-Client wurde in IntelliJ ein „einfaches“ Java SE-Projekt angelegt.
- Dieses kann zum Beispiel in ein Executable-Jar verpackt oder als Batchdatei mittels des `java` Kommandozeilenbefehls ausgeführt werden.

PACKAGING

- Falls ein Web-Client vorliegt, wird für diesen in IntelliJ ein Jakarta EE Web Application Projekt angelegt und darin ein Web Application Archive als Artifact erstellt.
- Dies führt dazu, dass IntelliJ ein Web-Archive (war) anlegt und dieses dann deployed.
- Da nun zwei Projekte angelegt wurden, müssen diese miteinander verknüpft werden.
- Am saubersten ist diese Verknüpfung, wenn das EJB-Projekt, für das eine jar-Datei erstellt wurde und das Web-Projekt, für das eine war-Datei erstellt wurde, in einem Enterprise-Archive (ear) zusammengefasst werden.

PACKAGING

- In diesem Fall wird die jar-Datei und die war-Datei zusammen mit einem Deploymentdeskriptor in eine Datei kopiert, die gemäß dem Zip-Algorithmus gepackt wird.
- Diese Datei erhält den Namen *<Projektname>.ear* und kann in das Deploy-Verzeichnis des Application Servers kopiert werden.

jar-Datei (EJB-Projekt)

war-Datei (Web-Projekt)

<Projektname>.ear

PACKAGING

- Um eine solche Struktur mit IntelliJ zu erstellen, sind folgende Schritte notwendig:
 1. Erstellung eines Projekts gemäß der Anleitung "Gemeinsames Projekt für REST Web Service und EJB mit JPA anlegen (IntelliJ 20nn, Java EE n)“

Für die ejb- und war-Module dabei jedoch nicht das Archiv im Deployments-Ordner als "Output directory" erstellen, sondern im Temp-Ordner belassen.

PACKAGING

2. Unter Artifacts zusätzlich ein "Java EE Application: Archive" erstellen.
 - Name wählen
 - Als "Output directory" das Deployverzeichnis des WildFly wählen.
 - "Include in project build" anhaken.
 - Das EJB- und das WAR-Artifact hinzufügen. Diese sind rechts unter "Available Elements" (Artifacts) aufgeführt und können per drag & drop auf den Namen des ear gezogen werden.

PACKAGING

Abschließendes Beispiel:

Componentware_Kapitel6_Transaktionen_Demo_Mitarbeiter
bzw.

Componentware_Kapitel6_EAR

Aufruf:

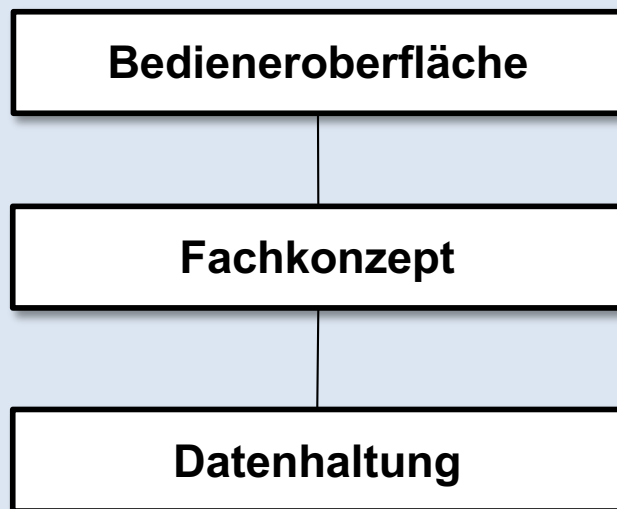
http://localhost:8080/Componentware_Kapitel6_Transaktionen_Demo_Mitarbeiter_Web/view/mitarbeiterListe.xhtml

Architektur



ARCHITEKTUR

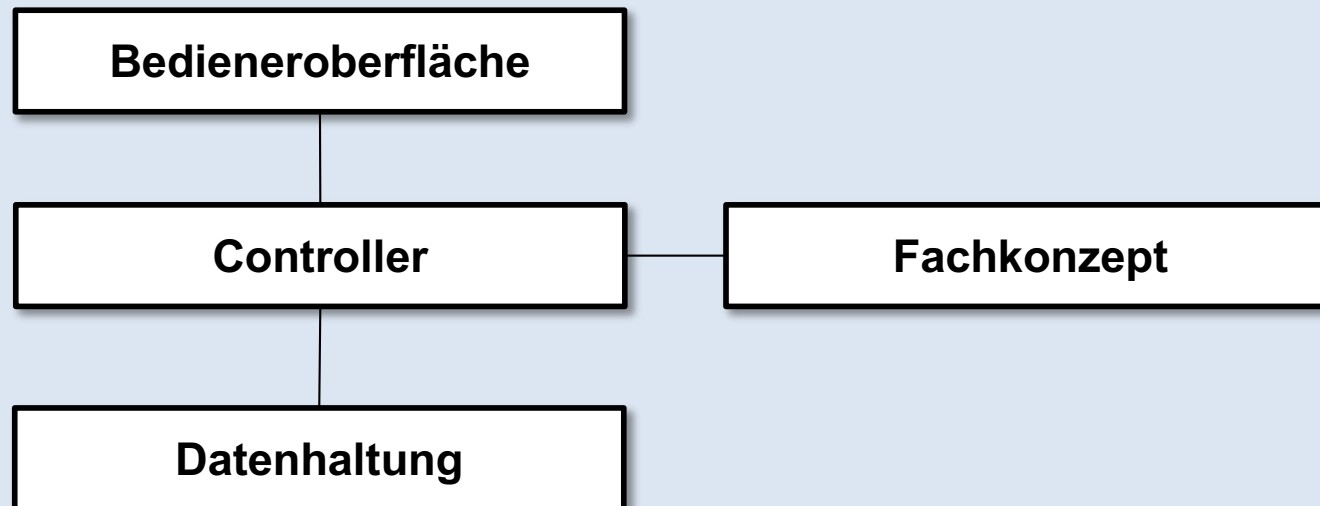
- Klassische dreischichtigen Architekturen enthalten folgende Schichten:



- Dadurch wird insbesondere eine lose Koppelung und somit eine hohe Wiederverwendbarkeit bewirkt.

ARCHITEKTUR

- Bei Weboberflächen wird der Controller in der Regel gemäß Modell-View-Controller-Muster separiert.



ARCHITEKTUR

- In den vorangegangenen Kapiteln wurde dargestellt, wie die Geschäftsprozesse in Form von Jakarta Enterprise Beans abgebildet werden können.
- In diesem Fall kommt eine weitere Schicht zu den bislang genannten Schichten hinzu.
- Diese Schicht enthält die Jakarta Enterprise Beans und wird im Allgemeinen als Middleware bezeichnet.

ARCHITEKTUR

- Abhängig von der Art des Clients (z.B. Web-Frontend oder Stand-Alone-Client) befindet sich die GUI-Schicht dabei auf dem Server (Web) oder auf dem Client (alleinstehendes Programm).
- Bei Web-GUIs stellt zwar ein Browser den Client dar, aber die eigentliche Implementierung befindet sich jedoch auf dem Server.
- So besteht die GUI-Schicht z.B. im Fall von JSF-Projekten aus XHTML-Dateien, die den View darstellen und aus Managed-Beans, die den Controller bilden.

ARCHITEKTUR

- Die Middleware stellt für den serverbasierten und transaktionalen Zugriff Data-Access-Objekte (DAO) je Fachkonzeptobjekt bereit.
- Nach einer Aktivität im Client wird nun der Controller (im Beispiel von JSF in Form der Managed Bean) die notwendigen Daten zusammenstellen und über das Data-Access-Objekt an die Serverschicht weiterleiten.
- Die Serverschicht kann nun die Daten gemäß Geschäftsprozess verarbeiten.

ARCHITEKTUR

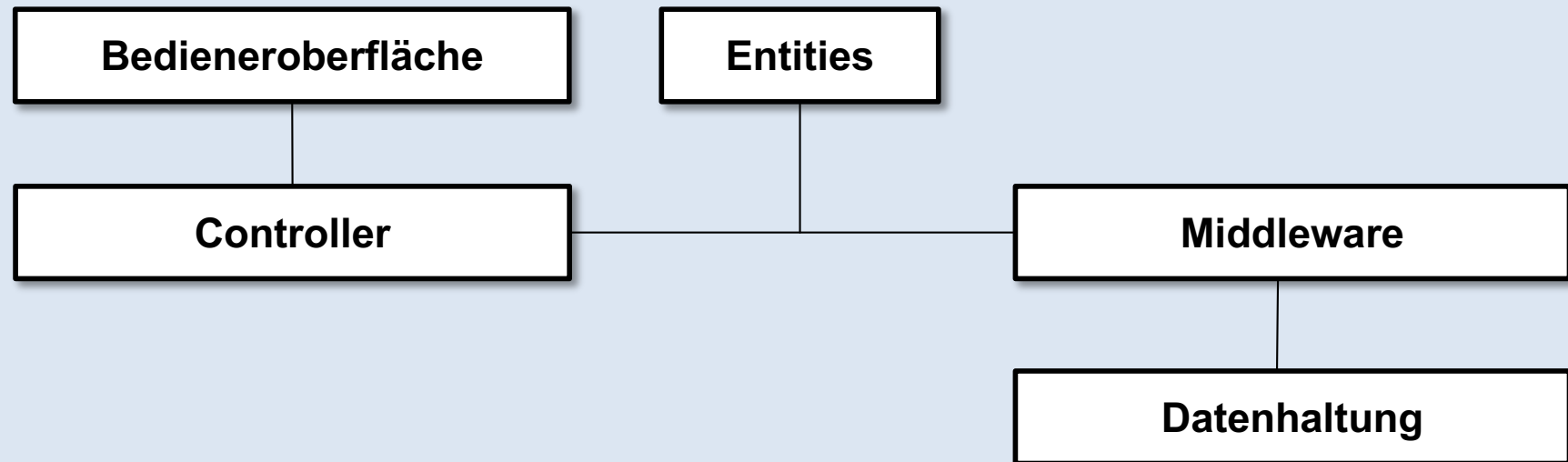
- Beim Aufruf der DAO-Methoden werden für den Transport der Nutzdaten in der Regel Fachkonzeptklassen in Form von Entities übergeben.
- Alternativ können auch anstelle von Entities sogenannte Data Transfer Objekte (DTO) verwendet werden.
- Dabei handelt es sich um Klassen, die genau auf die Daten des Clients angepasst wurden.
- Falls zum Beispiel der Client nur eine eingeschränkte Anzahl der Gesamtdaten visualisiert, kann auf diese Weise verhindert werden, dass alle Daten zum Client gelangen.
- Dies führt zu einer verbesserten Sicherheit und zum anderen zu einer erhöhten Performance.

ARCHITEKTUR

- Die DTOs werden beim Anlegen von Fachkonzeptobjekten auf dem Client oder in der Managed Bean generiert und an die Serverschicht gesendet.
- Die Anlegen-Methode des Data Access Objekts empfängt dann das Data Transfer Object und erzeugt daraus die Entity, die dann schließlich mit der Datenbank synchronisiert wird.
- Beim Auslesen vom Fachkonzeptobjekten erstellt die DAO-Methode das DTO und gibt dieses an die Managed Bean des Clients weiter.

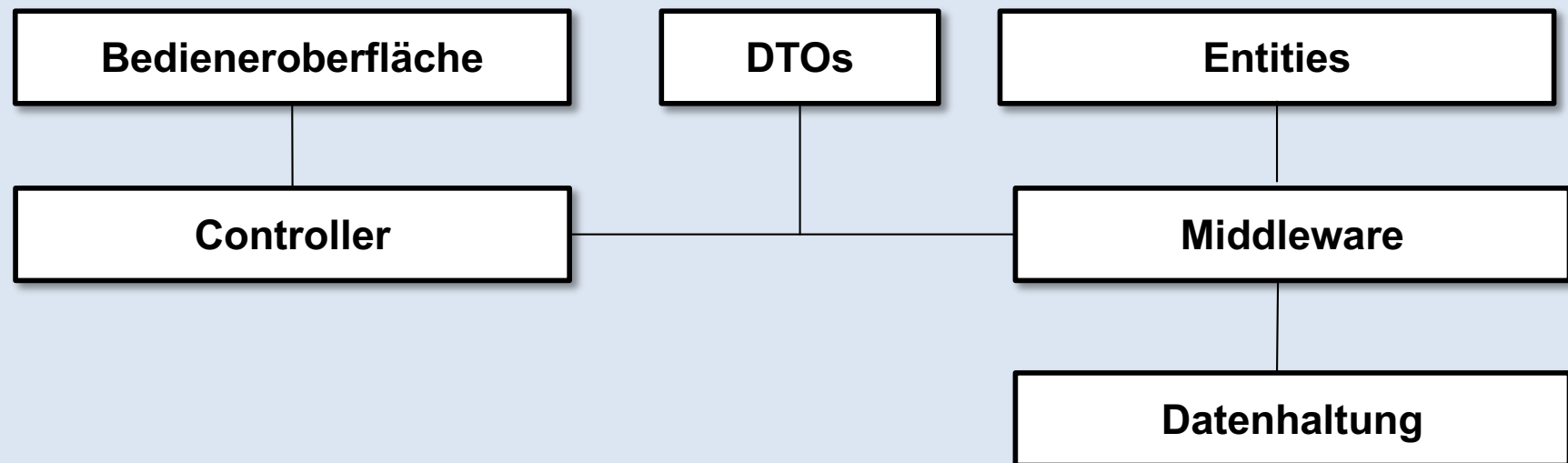
ARCHITEKTUR

- Architektur ohne DTOs



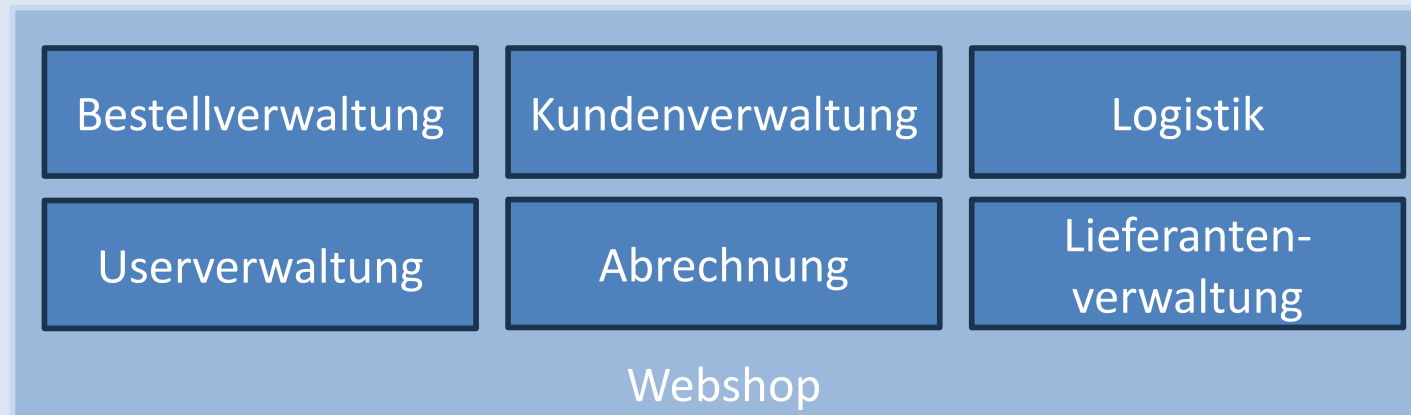
ARCHITEKTUR

- Architektur mit DTOs:



MICROSERVICES - MONOLITHISCHE APPLIKATIONEN

- Applikationen, wie sie in den vergangenen Kapiteln vorgestellt werden, nennt man monolithische Applikationen.
- Alle einzelnen Bestandteile befinden sich dabei in einer Applikation.
- Ein Webshop könnte beispielsweise aus folgenden Bestandteilen bestehen:



MICROSERVICES - MONOLITHISCHE APPLIKATIONEN

- Vorteile monolithischer Applikationen:
 - Monolithische Applikationen sind vergleichsweise einfach zu entwickeln. IDEs sind auf die Entwicklung von Monolithen vorbereitet.
 - End-to-end Tests können einfach über die gesamte Applikation hinweg ausgeführt werden.
 - Einfaches und schnelles Deployment, da die Applikation oft in nur einer oder wenigen jar/war/ear-Dateien vorliegt.
 - Mittels Load Balancer leicht zu skalieren.
 - Schnelle Entwicklungszeit und Time-to-Market.

MICROSERVICES - MONOLITHISCHE APPLIKATIONEN

- Nachteile monolithischer Applikationen:
 - Bei sehr großen Applikationen ist es für einzelne Entwickler schwer, die komplette Applikation zu kennen und zu verstehen.
 - Mehrere Personen ggf. Teams müssen sich absprechen.
 - Mehr Mergekonflikte, wenn viele Entwickler beteiligt sind.
 - Die Skalierung kann immer nur auf das Gesamtsystem ausgelegt werden. Wenn einzelne Teilsysteme unterschiedlich viel Speicher, z.B. aufgrund einer in-memory DB, oder unterschiedlich viel CPU-Zeit benötigen, muss ein Mittelweg gefunden werden.
 - Die Compilezeit ist bei monolithischen Applikationen mit der Zeit sehr hoch.

MICROSERVICES - MONOLITHISCHE APPLIKATIONEN

- Mit zunehmender Funktionalität steigt die Zeit bis die Applikation hochgefahren ist.
- Fehler wirken sich auf die gesamte Applikation aus. Fehler, die z.B. dazu führen, dass der Speicher voll läuft, crasht die gesamte Applikation
- Ein Austausch der Technologien ist oft schwierig, da oft historische Programmbibliotheken beteiligt sind, die ein Update schwer machen.
- Durch die Komplexität des Codes entstehen bei der Entwicklung mehr Fehler.

MICROSERVICES - MONOLITHISCHE APPLIKATIONEN

- Manuelle Tests dauern sehr lange, da die Applikation komplett getestet werden muss und dies aufgrund der Größe sehr lange dauert.
- Auch die Dauer der Ausführung von automatisierten Tests ist hoch, da immer die Tests für die gesamte Applikation ausgeführt werden müssen.
- Aufgrund der langen Tests kommt es auch oft zu einer langen Time-to-deployment.
- Dadurch ist auch oft kaum ein continuous integration (CI) / continuous deployment möglich.

MICROSERVICES - DEFINITION

- Einige der geschilderten Nachteile lassen sich durch Microservices vermeiden.
- Ein Microservice ist eine kleine Applikation, die eine enggefasste Funktionalität, wie E-Mail-Versand, User-Anmeldung, Bestellverwaltung, usw. implementiert.
- Ein solcher Service hat nach außen eine API, z.B. einen Web Service, durch die er angesprochen werden kann.
- Mehrere voneinander unabhängige, kleinere Services, die sich gegenseitig aufrufen, bilden so gemeinsam die Applikation.

MICROSERVICES - DEFINITION

- Pro Service wird ein eigener Server erstellt, es existiert eine eigene Datenbank, etc.
- Teams sind immer für einzelne Services zuständig und können daher den Code besser verstehen und damit besser warten und pflegen.
- Services melden sich an einer Service Registry an.
- Wenn ein Service ausfällt, werden die Requests gespeichert.

MICROSERVICES

- Vorteile von Microservices:
 - Da ein Microservice oftmals kleiner ist, als eine monolithische Applikation, sind die Quellcodes oft besser zu verstehen.
 - Microservices sind in sich abgeschlossen. Daher können Änderungen erfolgen, ohne dass in anderen Services Anpassungen erfolgen müssen.
 - Microservices können einzeln deployed und skaliert werden.
 - Teams können autonom arbeiten.
 - Services können untereinander nur über die API kommunizieren und daher besteht automatisch eine sehr lose Koppelung.

MICROSERVICES

- Unit-Tests beziehen sich nur noch auf den Code des Microservices und nicht mehr auf die gesamte Applikation. Dadurch wird die Dauer der Tests reduziert.
- Services können in unterschiedlichen Programmiersprachen erstellt werden und auf unterschiedlichen Servern betrieben werden.
- Es ist leichter neue Technologien auszuprobieren.
- Services können leichter ausgetauscht werden.

MICROSERVICES

- Nachteile von Microservices:
 - Es ist oft schwierig die richtige Trennung der Services zu finden. Eine ungeschickte Trennung führt oft zu Nachteilen beider Welten.
 - Durch die Netzwerkkommunikation kommt es zu zusätzlicher Komplexität. Zudem muss mit Netzwerkproblemen (Ausfällen / langsames Netz) umgegangen werden.
 - Es ist schwieriger Geschäftsprozesse zu realisieren, die über mehrere Microservices hinweg agieren, z.B. da die Datenbanken getrennt sind und dadurch übergreifende Transaktionen schwierig sind.
 - Zudem gibt es einen höheren Abspracheaufwand zwischen den Teams, da ggf. die Deployments aufeinander abgestimmt werden müssen.

MICROSERVICES

- IDEs sind oft nicht auf die Entwicklung verteilter Applikationen ausgelegt.
- Es ist schwieriger, Tests zu realisieren, die über mehrere Microservices hinweg agieren.
- Für die Administratoren kommt es zu einem höheren Aufwand, da mehrere Services mit ggf. mehreren Instanzen und Loadbalancern verwaltet werden müssen.
- Die Entwicklungszeit wird beeinflusst, da manche Aufgaben je Service implementiert werden müssen (Security, User Management, etc.), was insbesondere bei Startup-Unternehmen oft ein Problem ist.

Das Spring Boot Framework



ÜBERBLICK ÜBER SPRING

- Spring ist ein Entwicklungsframework für Enterprise Applikationen auf Java Basis.
- Somit ist Spring eine Alternative zu Jakarta EE.
- Die erste Version von Spring erschien 2004; seit 2009 erfolgt die Entwicklung durch VMware.
- Daher ist auch professioneller Support erhältlich.

HERKUNFT / ANSATZ

- Früher war die Konfiguration von Java EE recht komplex.
- Mit Spring Boot (ab 2014) existiert eine deutlich vereinfachte Konfiguration durch sinnvolle Vorkonfigurationen (Autokonfiguration)
- Projekte können beispielsweise komfortabel über die Spring-Website erzeugt werden (<https://start.spring.io>).
- Alternativ können Projekte natürlich ebenfalls über IDEs wie IntelliJ angelegt werden.
- Heutzutage ist die Konfiguration von Jakarta EE-Applikationen aber ebenfalls einfach zu bewerkstelligen. Von daher gilt dieser Vorteil nur noch eingeschränkt.

SPRING != SPRING BOOT

- Oftmals wird immer von Spring gesprochen, auch wenn oft Spring Boot gemeint wird.
- Es gibt jedoch wichtige Unterschiede:
 - Bei Spring muss der Container selbst konfiguriert werden.
 - Bei Spring Boot werden sinnvolle Voreinstellungen getroffen.
 - Der Focus liegt bei Spring Boot auf der einfachen Erstellung und Verwendung des Projekts.

UMFANG

- Spring setzt zum Teil die gleichen Enterprise Spezifikationen ein, wie Jakarta EE
 - z.B. Bean Validation, Mail-API, JPA, ...
- Spring Boot ist und enthält keinen Application Server auf dem eine Applikation deployt wird.
- Auch werden die Spezifikation eines Application Server nicht erfüllt.
- Stattdessen bringt Spring einen eigenen embedded Server (Apache Tomcat) mit, auf dem z.B. Webseiten betrieben werden.

SPRING GEHT ÜBER JAKARTA EE HINAUS

- In Jakarta EE wird zunächst ein Standard entwickelt (JPA, Servlets, ...) und danach werden Implementierungen für den Standard erstellt.
- In Spring werden neben dem eigenen Standard zusätzlich auch APIs unterstützt, die nicht dem Spring Standard entsprechen, wie z.B. die APIs des Jakarta EE Standards.
- Diese Technologien werden in Spring üblicherweise durch eine Abstraktionsschicht von Spring gekapselt.
- Zudem gibt es in Spring auch Lösungen, die es in Jakarta EE nicht gibt, wie eine API mit Cloud-Unterstützung, aspektorientierte Programmierung, Microservices, ...

TYPISCHE MODULE

- Spring stellt eine Reihe von Modulen bereit, die die Entwicklung einer Enterprise Applikation erleichtern.
- Beispiele:
 - Spring Data – Zugriff auf Datenbanken
 - Spring Web – Web Applikationen inklusive RESTful Web Services
 - Spring Security – Authentifizierung und Autorisierung
 - Spring Messaging – Implementierung von Messaging
 - Spring Cloud – Cloudbasierte Technologien, wie z.B. Amazon Web Services, Azure, etc.
 - ...

STARTER

- Um verschiedene Technologien, wie z.B. Datenbankzugriffe, Web Services, usw. zu integrieren, existiert das Konzept der Starter.
- Ein Starter ist eine Abhängigkeit, die innerhalb des verwendeten Build Tools (z.B. Maven) eingetragen wird.
- Diese sorgt dafür, dass alle für die entsprechende Technologie notwendigen Libraries geladen werden.
- Zudem erfolgt eine Autokonfiguration der verwendeten Klassen, so dass eine manuelle Konfiguration durch Entwickler auf das Notwendigste reduziert wird.
- Beim JDBC-Starter wird beispielsweise unter anderem eine Datasource vorkonfiguriert und ein Connection Pooling initialisiert.

KONZEPTE IN SPRING BOOT

Umsetzung der kennengelernten Konzepte in Spring

ANLEGEN EINES SPRING-BOOT PROJEKTS

- Der Start einer Spring Applikation erfolgt, wie bei Java SE Anwendungen, innerhalb der Methode `main`.
- Die Klasse, die die Methode `main` enthält, wird dabei mit `@SpringBootApplication` annotiert.
- In der Methode `main` wird die statische Methode `run` der Klasse `SpringApplication` aufgerufen und dabei eine Initialkonfiguration übergeben – hier der eigene Typ und das Parameter-Array aus der `main`-Methode.
- Beispiel:

```
@SpringBootApplication
public class SpringDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringDemoApplication.class, args);
    }
}
```

ANLEGEN EINES SPRING-BOOT PROJEKTS

- Die im vorhergehenden Beispiel gezeigte Klasse muss jedoch nicht selbst erstellt werden.
- Eine Spring Applikation kann über <https://start.spring.io> angelegt werden.

The screenshot shows the Spring Initializr web application in a browser window. The URL bar shows 'start.spring.io'. The page has a header with the 'spring initializr' logo and a hamburger menu on the left, and a settings icon on the right. The main content area is divided into three sections: 'Project', 'Language', and 'Dependencies'. The 'Project' section has radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for versions: '4.0.0 (SNAPSHOT)', '3.5.1 (SNAPSHOT)', '3.5.0' (selected), '3.4.7 (SNAPSHOT)', '3.4.6', '3.3.13 (SNAPSHOT)', and '3.3.12'. The 'Project Metadata' section contains input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). The 'Packaging' section has radio buttons for 'Jar' (selected) and 'War'. The 'Dependencies' section has a button 'ADD DEPENDENCIES... ⌘ + B' and the text 'No dependency selected'. At the bottom, there are three buttons: 'GENERATE ⌘ + ↵', 'EXPLORE CTRL + SPACE', and an ellipsis button.

ANLEGEN EINES SPRING-BOOT PROJEKTS

- Darüber hinaus kann eine Spring Applikation auch mittels IntelliJ angelegt werden.
- Neben unter anderem dem Namen, dem Build-System und der verwendeten Java-Version können auch Abhängigkeiten konfiguriert werden.
- Abhängig von den ausgewählten Dependencies werden dabei automatisch Starter konfiguriert.
- Zudem wird die Klasse, die mit `@SpringBootApplication` annotiert ist, erstellt.

ANLEGEN EINES SPRING-BOOT PROJEKTS

The screenshot shows the 'New Project' dialog box in an IDE. On the left, a sidebar lists various project generators. 'Spring Boot' is selected and highlighted in blue. Below it, other generators like JavaFX, Quarkus, Micronaut, Jakarta EE, Ktor, HTML, React, Express, Angular CLI, Vue.js, Vite, and Nuxt are listed. At the bottom of the sidebar is a link 'More via plugins...'. The main area of the dialog contains configuration fields for the new project. The 'Server URL' is set to 'start.spring.io'. The 'Name' is 'Componentware_Kapitel6_Spring_Demo'. The 'Location' is a file path: 'erlagen/Componentware/Kapitel 6 - Transaktionen/Vorlesung'. Below the location, it says 'Project will be created in: ~/Projekte/Lehr...riesung/Componentware_Kapitel6_Spring_Demo' and there is an unchecked checkbox for 'Create Git repository'. The 'Language' is set to 'Java'. The 'Type' is 'Gradle - Groovy'. The 'Group' is 'de.hadrys'. The 'Artifact' is 'Componentware_Kapitel6_Spring_Demo'. The 'Package name' is 'ys.componentware_kapitel6_spring_demo'. The 'JDK' is set to '25 Oracle OpenJDK 25'. The 'Java' version is '25'. The 'Packaging' is 'Jar'. The 'Configuration' is 'Properties'. At the bottom, there are buttons for '?', 'Cancel', and 'Next'.

New Project

Search:

New Project

- Java
- Kotlin
- Groovy
- Empty Project

Generators

- Maven Archetype
- Spring Boot**
- JavaFX
- Quarkus
- Micronaut
- Jakarta EE
- Ktor
- HTML
- React
- Express
- Angular CLI
- Vue.js
- Vite
- Nuxt

[More via plugins...](#)

Server URL: start.spring.io ⚙️

Name:

Location:

Project will be created in:
~/Projekte/Lehr...riesung/Componentware_Kapitel6_Spring_Demo

☐ Create Git repository

Language:

Type:

Group: ?

Artifact: ?

Package name:

JDK: ▼

Java: ▼

Packaging:

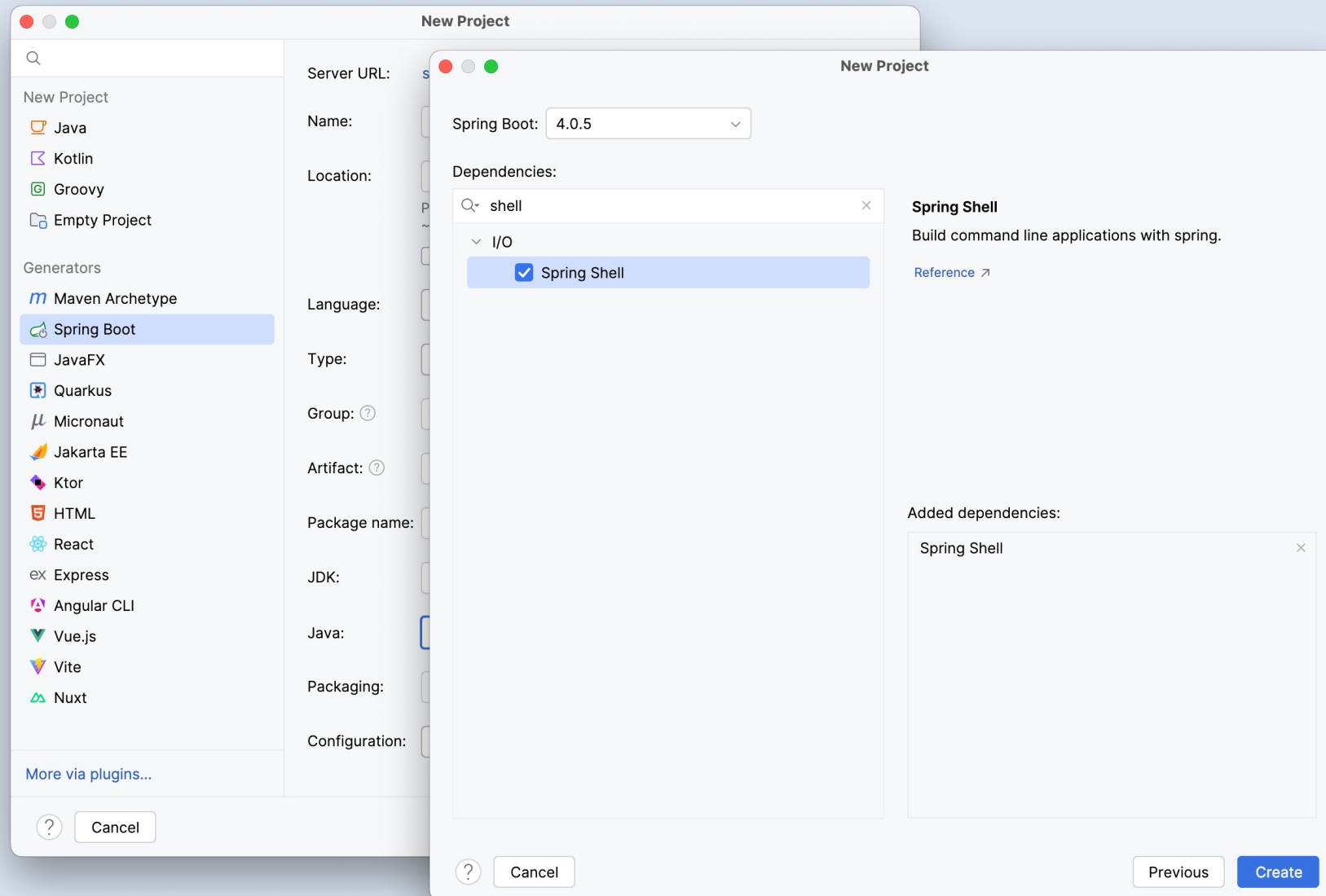
Configuration:

? Cancel **Next**

ANLEGEN EINES SPRING-BOOT PROJEKTS

- Auf der Folgeseite werden benötigte Abhängigkeiten, wie z.B. Datenbankframeworks, Web Services, etc. ausgewählt.
- Eine praktische Abhängigkeit stellt die Shell-Erweiterung dar.
- Shell-Applications ermöglichen einen einfachen Zugriff auf Spring Beans und eine komfortable Möglichkeit Methoden aufzurufen.

ANLEGEN EINES SPRING-BOOT PROJEKTS



INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Nach dem Abschluss des Dialogs mittels des Knopfs „Create“, wird ein Projekt inklusive der zugehörigen Starter erstellt.
- Falls Maven als Build-Tool verwendet wird, erzeugt IntelliJ in der Datei pom.xml dazu einen Starter mit der folgenden Dependency:

```
<dependency>  
  <groupId>org.springframework.shell</groupId>  
  <artifactId>spring-shell-starter</artifactId>  
</dependency>
```

INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Um die Shell-Methodik zu nutzen, wird eine Klasse erstellt, die mit der Annotation `@Component` versehen wird.
- Die darin enthaltenen Methoden werden wiederum mit `@Command` annotiert.
- Beispiel:

```
@Component
public class HelloWorld {
    @Command(description = "hello: Says Hello World!" )
    public String hello() {
        return "Hello World!";
    }
}
```

- Als Parameter der Annotation `@Command` kann ein Kommentar angegeben werden, der den Inhalt der Methode beschreibt.

INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Nach dem Start des Projekts wird ein Prompt angezeigt, durch den die in dem Projekt enthaltenen Shell Methoden anhand ihres Namens gestartet werden können.
- Beim Prompt wird der Name der Methode im Kebap-Case-Format mit ausschließlich Kleinbuchstaben eingegeben.
- Falls der Methodenname z. B. `testDB` lautet, wird als Prompt `test-db` eingegeben.

```
$>hello  
Hello World!
```

INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Über den Befehl `help` wird eine Liste aller Shell Befehle angezeigt.
- Dabei werden auch die verfügbaren Shell Methoden, inklusive der Kommentare in den Parametern der Methoden ausgegeben.
- Hinweis:
 - In neueren IntelliJ-Versionen wurde die Ausgabe in der Konsole geändert, so dass es nun nicht mehr um eine interaktive Konsole handelt.
 - Daher kommt kann beim Aufruf der Spring-Applikation nicht der in IntelliJ vorkonfigurierte Runner verwendet werden.
 - Stattdessen sollte der Aufruf über Maven erfolgen.

INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Um den Aufruf dennoch komfortabel zu gestalten, kann ein neuer Runner erstellt werden.
- Dazu:
 - Eine Datei (z.B. run.sh) im Hauptordner mit dem Inhalt `./mvnw spring-boot:run` als einzige Zeile erstellen.
 - Runner aufklappen > „Edit Configurations...” anklicken
 - Links oben „+“ klicken und Shell Script auswählen.
 - Oben links den Namen anpassen: Z.B. "run" eintragen
 - Oben rechts "Store as project file" anhängen.
 - Unter "Script path" das anfangs angelegte Script auswählen.
 - In Working Directory muss der Hauptordner des Projekts stehen, das sollte aber auch korrekt voreingestellt sein.

INTERAKTIVE ANWENDUNGEN MIT DER SPRING SHELL

- Falls Methoden mit Parametern aufgerufen werden sollen, müssen die Parameter einfach in der Parameterliste mittels `@Option` konfiguriert werden:

```
@Command(description="hello-to --name <name>:... " )  
public String helloTo(@Option(required = true) String name) {  
    return "Hello " + name;  
}
```

- Beim Aufruf wird der Parameter gesetzt, indem der Bezeichner des zu setzenden Parameters angegeben wird.

```
$>hello-to --name Petra  
Hello Petra
```

KOMPONENTEN

- Auch in Spring gibt es Managed Beans.
- Diese werden Spring-managed Beans bzw. Komponenten genannt.
- Managed Beans „leben“ im Spring-Container, der die Komponenten verwaltet, also anlegt, überwacht und Dependencies auflöst.
- Um beliebige POJOs zu einer Spring-managed Bean zu machen, werden Sie z. B. mit `@Component` annotiert.
- Diese Klassen müssen in einem Unterpaket desjenigen Pakets liegen, in dem sich die Klasse mit der main-Methode befindet.

KOMPONENTEN

- `@Autowired`
 - Diese Annotation wird verwendet um eine Dependency Injection durchzuführen. Die Annotation kann z.B. an einem Attribut angegeben werden, so dass der Wert dort injiziert wird.

- Beispiel:

```
@Autowired  
private SomeManagedBean bean;
```

- Die Annotation ähnelt `@Inject` aus dem Jakarta EE Standard, wobei die Spring Annotation mehr Attribute bietet (z. B. `required`).
- `@Inject` kann jedoch auch in Spring verwendet werden.
- `@Autowired` kann auch bei Methoden oder Konstruktoren verwendet werden.

KOMPONENTEN

- Die Verwendung der Annotation an Attributen im Vergleich zur Verwendung bei Konstruktoren hat aber Nachteile:
 - Die Klasse kann nur im Spring-Kontext verwendet werden. Bei der Nutzung von Konstruktor-DI, kann die Klasse auch in anderen Kontexten verwendet werden indem der Parameter beim Aufruf des Konstruktors manuell übergeben wird.
 - Das „Mocken“ in Unit-Tests wird erleichtert, da der Parameter manuell übergeben werden kann.
 - Die Attribute können final sein und damit ist garantiert, dass das Attribut niemals den Wert null enthält.

KOMPONENTEN

- Beispiel:

```
@Component
public class HelloWorldComponent {
    public String sayHello() {
        return "Hello from component!";
    }
}
```

```
@ShellComponent
public class HelloWorld {

    private HelloWorldComponent helloWorldComponent;

    @Autowired
    public HelloWorld(HelloWorldComponent helloWorldComponent) {
        this.helloWorldComponent = helloWorldComponent;
    }

    @Command(description = "hello-from-component: Says ...")
    public String helloFromComponent() { ... }
}
```

KOMPONENTEN

- `@Component` wird verwendet, wenn eine nicht näher definierte Spring-managed Bean erstellt werden soll.
- Es gibt jedoch auch einige Alternativen dazu, die semantische Annotationen genannt werden.

KOMPONENTEN

- Service-Klassen werden mit `@Service` annotiert.
- Repositories (Data Access Objects) werden mit `@Repository` annotiert.
- Controller (Anfragen vom Frontend, z.B. Web Service) werden mit `@Controller` annotiert.
- Zusammenhang:
 - Ein Controller nimmt eine Anfrage entgegen und startet Methoden des Service.
 - Werden in den Servicemethoden Daten benötigt, ermittelt der Server diese Daten mittels eines Repositories.

CDI

- Durch das sogenannte Component Scanning sucht Spring nach Komponenten, also Klassen, die mit `@Component`, `@Repository`, `@Service` oder `@Controller` annotiert sind.
- Das Component Scanning ähnelt der Bean Discovery bei CDI.
- Aus Performancegründen wird standardmäßig nur in dem Paket, in dem sich die Hauptklasse (die mit `@SpringBootApplication` annotiert ist) und dessen Unterpaketen gesucht.
- Von allen gefunden Klassen wird ein Objekt, also eine Spring Managed Bean, erstellt.
- Solche Spring Managed Beans können an anderer Stelle injiziert werden.
- Das Component Scanning kann durch die Annotation `@ComponentScan` angepasst werden, da alle Klassen, die diese Annotation aufweisen, einen weiteren Suchpfad öffnen.

CDI

- `@Bean`
 - Mittels `@Bean` können Spring Werte injizierbar gemacht werden, die durch Methoden vorkonfiguriert werden. Das Verfahren entspricht in etwa den in CDI kennengelernten Producern.
 - Auch bei `@Bean` können die bereits in CDI kennengelernten Qualifier genutzt werden.
 - Die Erstellung von Qualifiern entspricht dem Vorgehen bei Jakarta EE.

CDI

- Beispiel:

```
@Bean  
public String name() {  
    return "Max Mustermann";  
}
```

```
class OtherClass {  
    private final String name;  
  
    @Autowired  
    public OtherClass(String name) {  
        this.name = name;  
    }  
}
```

- Hinweis: Der Name der Methode entspricht stets dem Namen des Attributs am Injection Point.

CDI

- Weiteres Beispiel:

```
@Bean
public List<String> names() {
    return List.of("Max Mustermann", "Maxi Musterfrau");
}
```

```
class OtherClass {
    private final List<String> names;

    @Autowired
    public OtherClass(List<String> names) {
        this.names = names;
    }
}
```

- Siehe Dateien im Paket beanAnnotation.
Aufruf in der Shell über: print-name

CDI

- Interceptoren:
 - Lifecycle-Interceptoren können wie in Jakarta EE verwendet werden (@PostConstruct, ...).
 - Alternativ können über AOP (aspect oriented programming) Querschnittsaufgaben wie Security, Logging oder transaktionales Verhalten definiert werden (wird hier aus Zeitgründen nicht vertieft).

ZUGRIFF AUF DATENBANKEN

- Spring realisiert Datenbankzugriffe auf verschiedenen Leveln:
 - Vereinfachte Zugriffe über JDBC mittels JDBC Templates.
 - Zugriff über JPA.
 - Automatische Erstellung von CRUD-Repositories (vergleichbar mit DTAs in Form von Session Beans).

SPRING DATA

- Jakarta Persistence bindet stets relationale Datenbank-Managementsysteme (RDBMS) an.
- Spring Data geht darüber hinaus und erlaubt die einheitliche Implementierung verschiedener Datenbank-Managementsysteme – relational genauso wie nicht relational.
- Hierfür stellt Spring jeweils spezielle Implementierungen, wie z.B. Spring Data JPA, LDAP, MongoDB, etc. bereit.

SPRING DATA - JPA

- Um auf Datenbanken zuzugreifen, wird ein weiterer Starter verwendet.
- Dazu wird die automatisch generierte Datei *pom.xml* im Abschnitt `<dependencies>` wie folgt erweitert.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- Zudem wird eine Abhängigkeit für den Datenbanktreiber eingetragen.

```
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```


SPRING DATA - JPA

- Darüber hinaus werden die folgenden Verbindungsparameter in die Datei *application.properties* eintragen:

```
spring.datasource.url= jdbc:mysql://localhost:3306/  
                        Componentware_Kapitel6_Demo?useTimezone=true  
spring.datasource.username=<username>  
spring.datasource.password=<password>  
spring.jpa.hibernate.ddl-auto=create-drop  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.jpa.properties.hibernate.dialect=  
                        org.hibernate.dialect.MySQLDialect
```

SPRING DATA - JPA

- Auch in Spring wird die Jakarta EE Klasse `EntityManager` mit den gleichen Methoden (`persist`, `find`, `createQuery`, etc.) benutzt, wie es schon im Kapitel über JPA gezeigt wurde.
- Ebenfalls werden alle anderen JPA-Annotationen wie z.B. die Annotationen zur Referenzierung von Objekten anderer Klassen (`@OneToOne`, `@ManyToOne`, ...) weiterhin verwendet.
- Das `EntityManager`-Objekt kann auch wieder per Injection ermittelt werden.
- Hier wird jedoch die für Spring typische Annotation `@Autowired` verwendet.
- Auf diese Weise können, wie für EJBs gezeigt, Data Access Objekte erstellt werden.

SPRING DATA - JPA

- Zu beachten ist, dass in Spring die Methoden nicht automatisch transaktional ablaufen, obwohl Transaktionsklammern stets verwendet werden müssen.
- Wenn daher Data Access Objects genauso erstellt werden, wie im Kapitel über JPA gezeigt, kommt es zu einer `TransactionRequiredException`.
- Um diese Exception zu verhindern, müssen alle Methoden des Data Access Objects mit der Annotation `@Transactional` versehen werden.
- Beispiel:
Siehe Dateien im Paket `springDataJpa`.
Aufruf in der Shell über: `test-db-using-jpa`

SPRING DATA - REPOSITORIES - SIMPLEJPARepository

- Eine Alternative für den Datenbankzugriff stellen Repositories dar.
- Repositories sind automatisch generierte Zugriffsklassen, die Daten aus der Datenbank lesen und Daten schreiben können.
- Spring enthält hierfür unter anderem die generisch typisierbare Klasse `SimpleJpaRepository<Type, Id>(Class-Objekt, EntityManager)`, durch die die Implementierung von Data Access Objects sehr erleichtert wird, da die meisten Datenbankzugriffsmethoden bereits implementiert sind.

SPRING DATA - REPOSITORIES - SIMPLEJPARepository

- Die Klasse bietet viele Methoden, wie z.B.
 - save: speichert ein Objekt
 - findAll: Ermittelt eine Liste aller Tabelleninhalte
 - findAllById(List<Long>): Liefert eine Liste aller Objekte deren Ids in der Liste im Parameter angegeben wurden.
 - ...

SPRING DATA - REPOSITORIES - SIMPLEJPARepository

- Die Klasse wird bei der Deklaration generisch typisiert.
- Als erster Parameter wird der fachliche Datentyp angegeben. Hierbei muss es sich um eine Entity handeln.
- Bei dem zweiten generischen Parameter handelt es sich um den Datentyp des Id-Attributs.
- Beim Erzeugen wird dem Konstruktoraufbau ein Class-Objekt der fachlichen Klasse und eine Instanz des EntityManagers übergeben.
- Die Instanz des EntityManager-Objekts kann per Dependency Injection ermittelt werden.

SPRING DATA - REPOSITORIES - SIMPLEJPARepository

- Beispiel:

```
private EntityManager em;

@Autowired
public ClassName(EntityManager em) { this.em = em; }

public List<Book> findAllBooks() {
    SimpleJpaRepository<Book, Long> repository =
        new SimpleJpaRepository<>(Book.class, em);
    List<Book> books = repository.findAll();
    return books;
}
```

- Siehe
BookSimpleJpaRepository.java und
BookSimpleJpaRepositoryShell.java
im Paket springDataRepositories.
Aufruf in der Shell über: test-db-using-simple-jpa-repository

SPRING DATA - REPOSITORIES - CRUD REPOSITORIES

- Eine weitere Möglichkeit der automatischen Erstellung von Repositories stellt das Interface `CrudRepository<T, Id>` dar.
- Auch hier handelt es sich bei dem ersten Parameter um den fachlichen Datentyp und bei bei dem zweiten generischen Parameter um den Datentyp des Id-Attributs.
- Das Interface stellt wiederum diverse Methoden für den Datenbankzugriff bereit, wie z.B.:
 - `<S extends T> S save(S entity)`
 - `<S extends T> Iterable<S> saveAll(Iterable<S> entities)`
 - `Iterable<T> findAll()`
 - `Iterable<T> findById(Iterable<ID> ids)`
 - `void delete(T entity)`
 - `void deleteById(ID id)`
 - `void deleteAll()`

SPRING DATA - REPOSITORIES - CRUD REPOSITORIES

- Um das Interface zu nutzen, wird ein eigenes Interface definiert, in dem die benötigten Methoden angegeben werden.
- Dieses eigene Interface erweitert das Interface `CrudRepository`.
- Das Interface kann dann per Constructor Injection in eine Klasse injiziert und verwendet werden.

SPRING DATA - REPOSITORIES - CRUD REPOSITORIES

- Eine Erweiterung stellt zudem das `ListCrudRepository` dar, das selbst von `CrudRepository` erbt.
- Das Interface stellt schon selbst die folgenden zusätzlichen Methoden bereit, die zudem den Typ `List` anstelle von `Iterable`-Objekten verwenden:
 - `<S extends T> List<S> saveAll(Iterable<S> entities)`
 - `List<T> findAll()`
 - `List<T> findById(Iterable<ID> ids)`
- In diesem Fall muss im Interface keine Methode mehr angegeben werden.

SPRING DATA - REPOSITORIES - CRUD REPOSITORIES

- Darüber hinaus gibt es noch spezielle Interfaces, wie z.B. das Interface `JpaRepository`, das z.B. Sortierbarkeit und Pagination bietet.

```
public interface BookJpaRepository extends JpaRepository<Book, Long> {  
}
```

```
public class BookJpaRepository {  
    private BookJpaRepository bookJpaRepository;  
  
    public BookJpaRepository(BookJpaRepository bookJpaRepository) {  
        this.bookJpaRepository = bookJpaRepository;  
    }  
  
    @Transactional  
    public String testDbUsingJpaRepository() {  
        List<Book> allBooks = bookJpaRepository.findAll();  
        ...  
    }  
}
```

Hinweis: Es ist keine Injection z. B. mittels `@Autowired` notwendig.

SPRING DATA - REPOSITORIES - CRUD REPOSITORIES

- Siehe
BookJpaRepository.java und BookJpaRepositoryShell.java
im Paket springDataRepositories.
Aufruf in der Shell über: test-db-using-jpa-repository

MESSAGING

- Spring liefert für Messaging eine Implementierung für ActiveMQ mit.
- Um Messaging einzusetzen, muss der Datei *pom.xml* der ActiveMQ-Starter hinzugefügt werden, der alle erforderlichen Abhängigkeiten für die Integration von JMS und ActiveMQ in Spring bereitstellt.

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>
```

MESSAGING

- Während in älteren Versionen von Spring Boot der Messaging Server automatisch gestartet wurde, ist dies ab der Spring Version 4 nicht mehr der Fall, so dass der Start nun manuell erfolgen muss.
- Am einfachsten ist es, den Server als Docker Container zu starten.
- Dazu kann folgender Befehl in der Eingabeaufforderung / Terminal verwendet werden:

```
docker run -d --name activemq -p 61616:61616 -p 8161:8161 rmohr/activemq
```

- Zu beachten ist, dass dazu die Docker Engine laufen muss.
- Falls auf dem System kein Docker installiert ist, kann es unter <https://www.docker.com/> heruntergeladen werden.

MESSAGING

- Vorgehen:
 - Für den Empfang von Nachrichten wird zunächst eine Konfiguration erstellt.
 - Danach kann eine Methode erstellt werden, die den Empfang vornimmt.
 - Zum Testen wird abschließend ein Sender erstellt, der eine Nachricht versendet.

MESSAGING - KONFIGURATION

- Spring stellt `MessageConverter` bereit, die Basistypen, wie `String`, `Map` oder `Serializable` in Messages umwandeln können.
- Wenn anstelle solcher Typen, Objekte von selbsterstellten Klassen, die nicht `Serializable` implementieren, übertragen werden sollen, wird ein anderer Mechanismus benötigt.
- In diesem Fall kann die Bibliothek Jackson verwendet werden, um solche Objekte in ein Textformat zu konvertieren.
- Die Übertragung der Objekte erfolgt dabei per JSON.
- Um zu kennzeichnen, welcher Java-Typ im JSON übertragen wird, fügt Jackson ein `Typattribut` zum JSON Objekt hinzu, in dem der Java-Klassenname als String angegeben wird.

MESSAGING - KONFIGURATION

- Die folgende Klasse definiert eine Spring Bean, die für die Umwandlung zwischen JMS-Nachrichten und Java-Objekten zuständig ist.

```
@Bean
public MessageConverter jacksonJmsMessageConverter() {
    // Jackson Konverter zwischen JSON und dem Fachobjekt.
    JacksonJsonMessageConverter converter =
        new JacksonJsonMessageConverter();

    // Festlegung, wie das Typ Attribut heißen soll. Der
    // Name _type ist dabei eine gebräuchliche Konvention.
    converter.setTypeIdPropertyName("_type");

    // Konfiguration von Mappings, in diesem Fall für Buch.
    // => Wenn _type="buch", dann verwende die Klasse Buch.
    Map<String, Class<?>> typeIdMappings=new HashMap<>();
    typeIdMappings.put("buch", Buch.class);
    converter.setTypeIdMappings(typeIdMappings);

    return converter;
}
```

MESSAGING - KONFIGURATION

- Spring Boot erkennt daraufhin das Vorliegen eines solchen Converters und verbindet ihn mit Objekten vom Typ `JmsTemplate` (siehe später) und automatisch erstellten Container-Factories.

MESSAGING - KONFIGURATION

- Die folgenden Konfigurationspunkte werden der Datei *application.properties* hinzugefügt:

```
# Zugangsdaten zu ActiveMQ
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin

# Queue Name
app.queue.name=buch.queue

# Verwendetes Verfahren:
# false: Queue / Point-to-Point
# true: Topic / Publish-Subscribe
# optional, denn false ist default.
spring.jms.pub-sub-domain=false
```

MESSAGING - KONFIGURATION

- Die `SpringBootApplication`-Class (die Klasse, die mit `@SpringBootApplication` annotiert ist) wird um die Annotation `@EnableJms` **erweitert**.

```
@SpringBootApplication
@EnableJms
public class ComponentwareKapitel6SpringDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            ComponentwareKapitel6SpringDemoApplication.class, args)
    }
}
```

MESSAGING - ERSTELLUNG EINES RECEIVERS

- Ein Receiver empfängt eingehende Nachrichten.
- Receiver entsprechen Message Driven Beans im Jakarta Beans Standard.
- Die Implementierung erfolgt, indem in einer Spring Managed Bean (z.B. einer Klasse, die mit `@Component` annotiert ist) eine Methode erstellt wird, die mit `@JmsListener` annotiert wird.
- Die Annotation wird dabei mindestens mit dem Parameter `destination` versehen, in dem der Name der zu nutzenden Queue angegeben wird.
- Diese Queue wird von Spring automatisch erzeugt.
- Optional kann zusätzlich der Parameter `containerFactory` gesetzt werden, falls anstelle der automatisch generierten Factory eine eigene Implementierung verwendet werden soll.

MESSAGING - ERSTELLUNG EINES RECEIVERS

- Die Methode kann einen beliebigen Namen haben und liefert kein Ergebnis (`void`) zurück.
- Die Methode erhält zwei Parameter:
 - Der erste ist der zu empfangende Typ (z.B. `Person`, `Buch`, etc.).
 - Bei dem zweiten Parameter handelt es sich um den Typ `Message`.
 - Beide Werte werden von Spring durch Dependency Injection automatisch gefüllt.
- Sobald diese Methode beim Deployment erkannt wird, erstellt und startet Spring einen Listener-Container, der automatisch beim Eingang einer Nachricht in dieser Queue aufgerufen wird.

MESSAGING - ERSTELLUNG EINES RECEIVERS

- Beispiel:

```
@Component
public class MessageReceiver {

    private final Logger logger =
        LoggerFactory.getLogger(MessageReceiver.class);

    @JmsListener(destination = "springQueue")
    public void receiveMessage(Book receivedBook, Message message) {
        logger.info("Message:" + message);
        logger.info("Buch: " + receivedBook);
    }
}
```

MESSAGING - ERSTELLUNG EINES SENDERS

- Für die Erstellung eines Senders wird zunächst ein Objekt der Klasse `ConfigurableApplicationContext` benötigt.

- Dieses Objekt kann mittels Dependency Injection erhalten werden:

```
@Autowired  
private ConfigurableApplicationContext context;
```

- Um eine Nachricht zu senden, stellt Spring die Klasse `JmsTemplate` bereit.
- Ein Objekt dieser Klasse wird über das oben injizierte Objekt der Klasse `ConfigurableApplicationContext` ermittelt.

```
JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);
```


MESSAGING - ERSTELLUNG EINES SENDERS

- Nun kann eine Nachricht versendet werden, indem auf dem `JmsTemplate`-Objekt die Methode `convertAndSend` aufgerufen wird, der sowohl der Name der Queue, als auch das zu versendende Objekt übergeben wird.

```
jmsTemplate.convertAndSend("queueName", new Book("Buch", 42.99));
```

- Gesamtes Beispiel im Paket `springMessaging`:
`ComponentwareKapitel6SpringDemoApplication.java`,
`JmsConfiguration.java`,
`MessageReceiver.java` und
`MessagingShell.java`

Aufruf in der Shell über: `send-message`

SPRING WEB MVC

- Mittels Spring MVC stellt Spring eine Unterstützung für die Web-Entwicklung bereit.
- Um die entsprechenden Klassen zu nutzen, muss der Starter `spring-boot-starter-web` in die Datei *pom.xml* eingetragen werden.
- Spring Web MVC unterstützt unter anderem Jakarta Servlets und damit alle darauf aufbauenden Technologien.
- Durch den embedded Tomcat Webserver, besteht keine Notwendigkeit sich um das Bereitstellen und das Aufsetzen eines Servers zu kümmern.
- Darüber hinaus ist das RESTful Paradigma vollständig integriert, so dass Web Services erstellt werden können.

SPRING WEB MVC - RESTFUL WEB SERVICES

- `HttpRequests` werden in Spring von einem Controller behandelt.
- Um einen Web Service zu erstellen wird somit zunächst ein Controller benötigt.
- Dazu stellt Spring die Annotation `@RestController` bereit.
- Eine mit `@RestController` annotierte Klasse ist eine Spring-Managed Bean. Daher können darin andere Klassen injiziert werden.
- Durch die Annotation `@RequestMapping("<url>")`, an einer Methode oder der Klasse wird die Methode oder die Klasse mit einem Pfad verknüpft (wie in Jakarta EE bei `@Path`).
- Falls die Annotation jeweils an der Klasse und an einer Methode erfolgt, dann werden die URLs sequentiell verbunden.

SPRING WEB MVC - RESTFUL WEB SERVICES

- Die Annotation hat zudem weitere Attribute, so dass nicht nur Pfad sondern auch die Request Methode angegeben werden kann.

```
@RequestMapping( path    = "<url>",  
                  method = { RequestMethod.GET|POST|PUT|... } )  
public .. method() {  
    ...  
}
```

- Alternativ existiert für die einzelnen Request Methoden eine *****Mapping-Methode**:

```
@GetMapping("<url>")  
public .. method() {  
    ...  
}
```

- Zudem existieren auch die Methoden @PostMapping, @PutMapping, etc.

SPRING WEB MVC - RESTFUL WEB SERVICES

- Die Annotationen `@RequestMapping` und `@GetMapping` können darüber hinaus verknüpft werden, ohne dass bei `@GetMapping` eine URL angegeben wird:

```
@RequestMapping("/hello")
public class HelloWebService {
    @GetMapping // hier ist kein Pfad angegeben
    public ... method() {
    }
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Beispiel:

```
@RestController
@RequestMapping("/hello")
public class HelloWebService {
    @GetMapping
    public String sayHello() {
        return "Hello from Web Service";
    }
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Zudem können in den @***Mapping Annotationen auch MediaTypes angegeben werden:

```
@GetMapping(path = "<url>",  
            produces = MediaType.<typ>))  
public .. method() {  
    ...  
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Parameter können z. B. als Request-Parameter in der URL übergeben werden:

```
@RestController
@RequestMapping("/hello")
public class HelloWebService {

    @GetMapping("/to")
    public String sayHelloTo(
        @RequestParam(value = "name", defaultValue = "World")
        String name) {
        return "Hello to " + name;
    }
}
```

Aufruf: `http://localhost:8080/hello/to?name=Peter`

SPRING WEB MVC - RESTFUL WEB SERVICES

- Oder auch als Path-Parameter:

```
@RestController
@RequestMapping("/hello")
public class HelloWebService {

    @GetMapping("/to/{name}")
    public String sayHelloToParam(@PathVariable String name) {
        return "Hello to " + name;
    }
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Rückgabe von Objekttypen:
 - Um Objekte zurück zu liefern, wird die Klasse `ResponseEntity` verwendet.
 - Die Klasse verfügt über diverse Methoden um den Status bei der Rückgabe zu setzen, wie z.B. `ok(Object o)`.
 - Das zu sendende Objekt wird dieser Methode somit als Parameter übergeben.
 - Die Übertragung erfolgt dabei automatisch via JSON.
- Beispiel:

```
@GetMapping("/book/{id}")
public ResponseEntity<?> getBookById(@PathVariable long id){
    Book book = bookJpaRepository.getReferenceById(id);
    return ResponseEntity.ok( book );
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Empfang von Objekten:
 - Um Objekte zu empfangen, wird das Objekt als Parameter in der Web Service Methode eingetragen.
 - Für den Parameter wird zudem die Annotation `@RequestBody` verwendet.
- Beispiel:

```
@PostMapping(path = "/book")
public ResponseEntity<?> saveBook(@RequestBody Book book) {
    System.out.println(book);
    Book b = bookJpaRepository.save( book );
    return ResponseEntity.ok( b );
}
```

SPRING WEB MVC - RESTFUL WEB SERVICES

- Hinweis: Um eine List<> korrekt zu übertragen, muss, je nach Spring Version, in der Datei *application.properties* folgender Eintrag gesetzt werden:

```
spring.jackson.serialization.FAIL_ON_EMPTY_BEANS=false
```

- Beispiele:
HelloWebService.java und BookWebService.java im Paket springWeb
im Projekt Componentware_Kapitel6_Spring_Demo.

Aufruf über separate Test Clients für den BookWebService, die sich im
Projekt Componentware_Kapitel6_Spring_Demo_TestClients
befinden.

TRANSAKTIONEN

- In Spring gibt es zwei grundlegende Arten, Transaktionen zu steuern:
 - Deklarativ (mit `@Transactional`)
 - Programmgesteuert (manuell durch den Entwickler)

TRANSAKTIONEN

- Wenn Methoden automatische Transaktionen verwenden sollen, können sie für einen Ablauf in einem Transaktionskontext konfiguriert werden, indem sie mit der Methode `@Transactional` annotiert werden.
- Die Annotation kann auch an der Klasse angegeben werden, wodurch alle Methoden der Klasse in einem Transaktionskontext ablaufen.
- Wenn es in der Methode zu einer `RuntimeException` kommt, wird die Transaktion zurückgerollt und ansonsten committed.
- Falls die Methode einen neuen Thread startet, wird diesem der Transaktionskontext nicht propagiert.

TRANSAKTIONEN

- Über eine Parametrisierung der Annotation `@Transactional` kann auf das Transaktionsverhalten Einfluss genommen werden, wie z.B.
 - `readOnly = true` (default `false`) wodurch ein Performancevorteil erreicht wird,
 - es kann das Isolationsverhalten (`REQUIRED`, `REQUIRES_NEW`) angepasst werden oder
 - es kann konfiguriert werden welche Exceptions zum Rollback führen.
- Siehe auch
 - <https://docs.spring.io/spring-framework/reference/data-access/transaction/declarative/annotations.html>
 - <https://docs.spring.io/spring-framework/reference/data-access/transaction/declarative/tx-propagation.html>

TRANSAKTIONEN

- Wenn Entwickler selbst die volle Kontrolle über das Transaktionsgeschehen benötigen, kann das *programmatic transaction management* verwendet werden.
- Bei programmatischer Steuerung entscheiden Entwickler selbst:
 - wann eine Transaktion startet
 - wann sie committed wird oder
 - wann ein Rollback passiert

TRANSAKTIONEN

- In Spring bilden die Klassen `PlatformTransactionManager`, `TransactionDefinition` und `TransactionStatus` den Kern der programmgesteuerten Transaktionsverwaltung.
- Die Klasse `PlatformTransactionManager` führt die Transaktion technisch aus.
- Beispielsweise startet Sie Transaktionen, committed Transaktionen und führt ggf. Rollbacks aus.

TRANSAKTIONEN

- Das Objekt vom `PlatformTransactionManager` wird typischerweise als Attribut deklariert und per `Dependency Injection` gesetzt.
- Das Erzeugen der Transaktion erfolgt über die Methode `transactionManagerObject.getTransaction(transactionDefinitionObject)` die ein Objekt vom Typ `TransactionStatus` zurückliefert.
- Das Beenden der Transaktion geschieht mit den Methoden `void commit(TransactionStatusObject)` und `void rollback(TransactionStatusObject)`.

TRANSAKTIONEN

- Für die Erzeugung wird somit ein Objekt der Klasse `TransactionDefinition` benötigt.
- Diese Klasse beschreibt die Eigenschaften der Transaktion.
- Über Methoden können beispielsweise die folgenden Einstellungen vorgenommen werden:
 - Propagation
 - Timeout
 - Read-Only
 - Namen der Transaktion
- Das Objekt kann einfach über einen Konstruktor ermittelt werden.

```
TransactionDefinition def = new DefaultTransactionDefinition();
```

TRANSAKTIONEN

- Die Methoden `commit` und `rollback` benötigen jeweils ein Objekt vom Typ `TransactionStatus`, welches beim Erzeugen der Transition von der Methode `getTransaction(...)` zurückgeliefert wird.
- Ein Objekt vom Typ `TransactionStatus` repräsentiert die aktuell laufende Transaktion.
- Über das Objekt kann beispielsweise abgefragt werden, ob die Transaktion aktuell läuft oder ob Rollback-only gesetzt ist.

TRANSAKTIONEN

- Zusammenfassendes Beispiel:

```
@Service
public class DoSomethingService {

    @Autowired
    private PlatformTransactionManager transactionManager;

    public void doSomething() {
        TransactionDefinition definition =
            new DefaultTransactionDefinition();
        TransactionStatus status =
            transactionManager.getTransaction(definition);

        try {
            ...
            transactionManager.commit(status);
        } catch (Exception e) {
            transactionManager.rollback(status);
        }
    }
}
```

TRANSAKTIONEN

- Beispiele:
Siehe Klassen im Paket transaction im Projekt
Componentware_Kapitel6_Spring_Demo.