Transactions

Spring Transactions

- We want to add Spring to our applications
  - To make **Spring and Hibernate applications**
  - EMF singleton, ThreadLocal and OpenEMinView are all easy to configure with Spring
  - Real value added is **Transaction Management**

We'll look at Transactions and Spring first.

BMT vs CMT

- Transaction management so far consisted of us writing .getTransaction().begin() and .commit()
  - When using a Bean this is called Bean Managed Transactions (**BMT**)
  - The container can also manage the transactions for you – Container Managed Transactions (**CMT**)

Transaction Requirement

- Many developers believe transactions are an optional part of database interactions
- In reality, **there is no such thing as a database interaction without a transaction**
- Most databases default to auto-commit mode
  - Wraps a transaction around each SQL statement
  - Effectively hiding the transaction from view

Auto Commit Mode

- Auto Commit is good for SQL console work
  - Console work is often ad-hoc (no tx needed)
  - Having to add begin / commit would be more work

- **Auto Commit is bad for applications**
  - More transactions means more overhead
  - Isolation is reduced without transaction boundaries

- Hibernate disables Auto Commit by default
  - Therefore you have to specify when to commit! (and begin)

No Transaction?

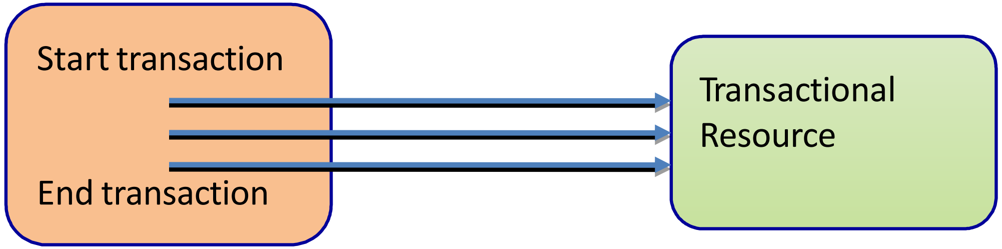
- If you don’t specify a transaction
  - A transaction will still be open at the JDBC level
  - Hibernate has turned off auto-commit
  - Hibernate will do nothing. If you flush, throw Exception

```
Exception in thread "main" javax.persistence.TransactionRequiredException: no transaction is in progress
	at edu.mum.cs.AppMain.persist(AppMain.java:23)
	at edu.mum.cs.AppMain.main(AppMain.java:173)
```

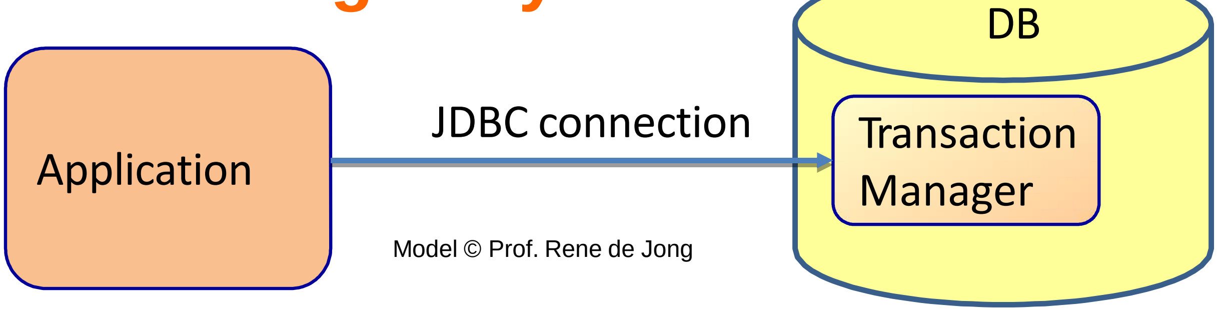
Spring Transactions: Global Transactions

Local Transactions

- So far we've only considered local transactions
  - Transactions that use a single transactional resource



- These transactions are **managed by the DB**
  - Simple and Fast

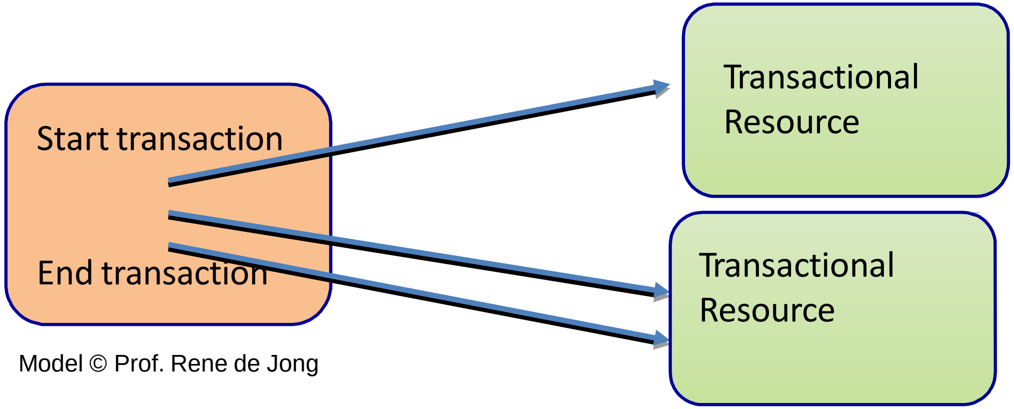


Model © Prof. Rene de Jong

Global Transactions

- Global Transactions are transactions that span **multiple transactional resources**
  - Such as databases or message
  - More common in enterprise applications
  - Also called XA transactions

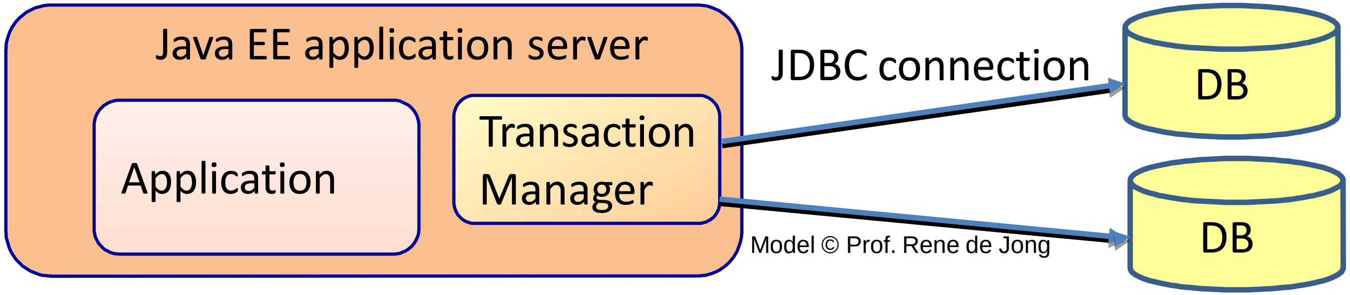
eXtended Architecture



Model © Prof. Rene de Jong

Transaction Manager

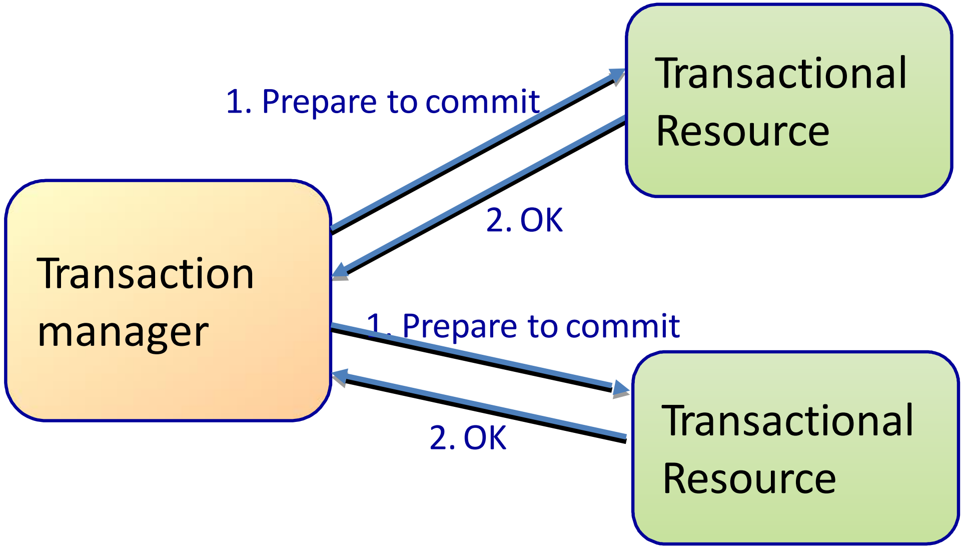
- Global Transactions have to be managed on the application side (to coordinate resources)
  - Generally done by a **Transaction Manager**
    - Standard Java Transaction API (JTA) interface
    - Required part of Java EE application servers
    - Stand Alone JTA implementations also exist



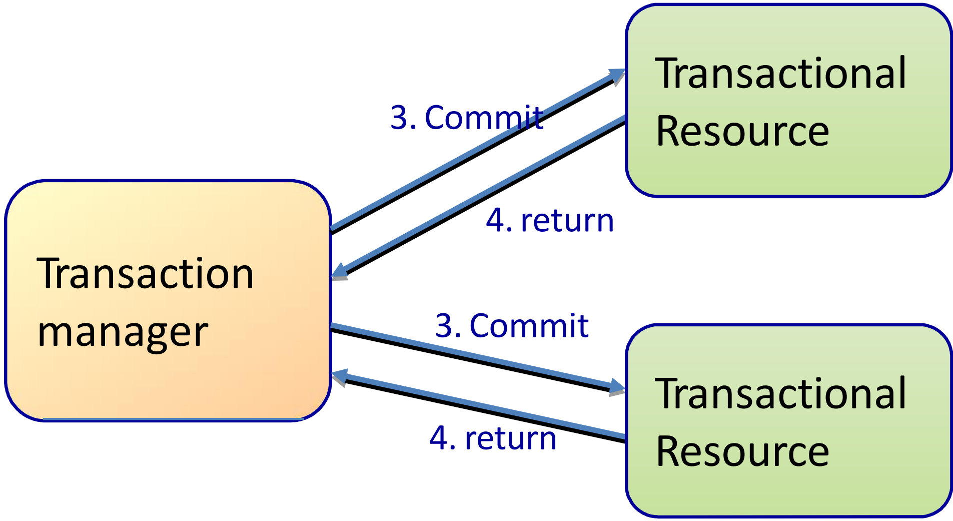
Model © Prof. Rene de Jong

2 Phase Commit (success)

- Phase 1

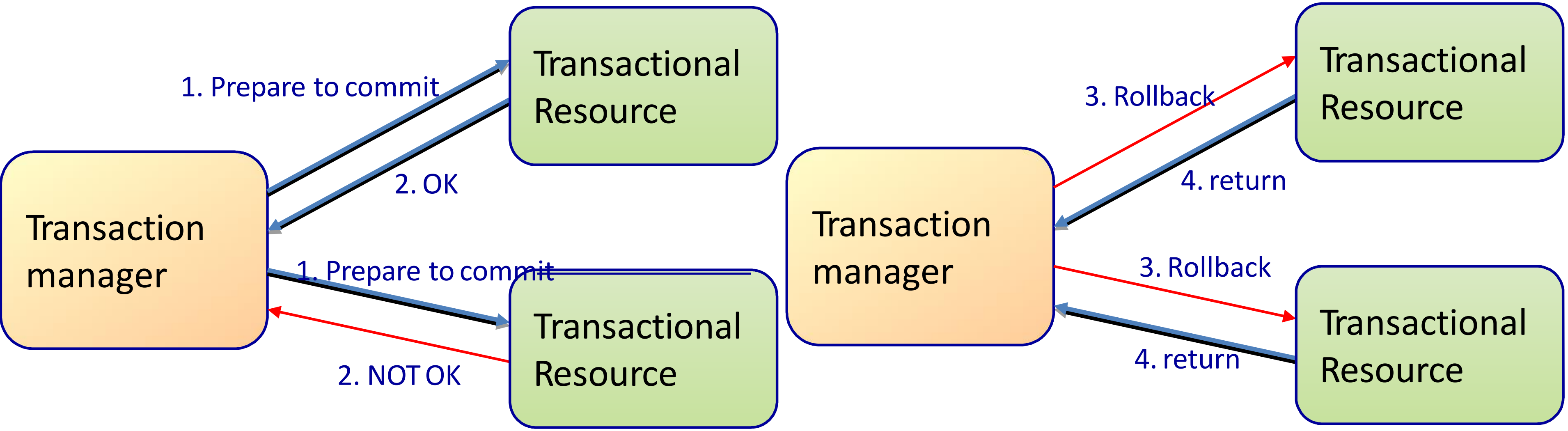


- Phase 2



2 Phase Commit (Failure)

- Phase 1
- Phase 2



Characteristics of XA TX

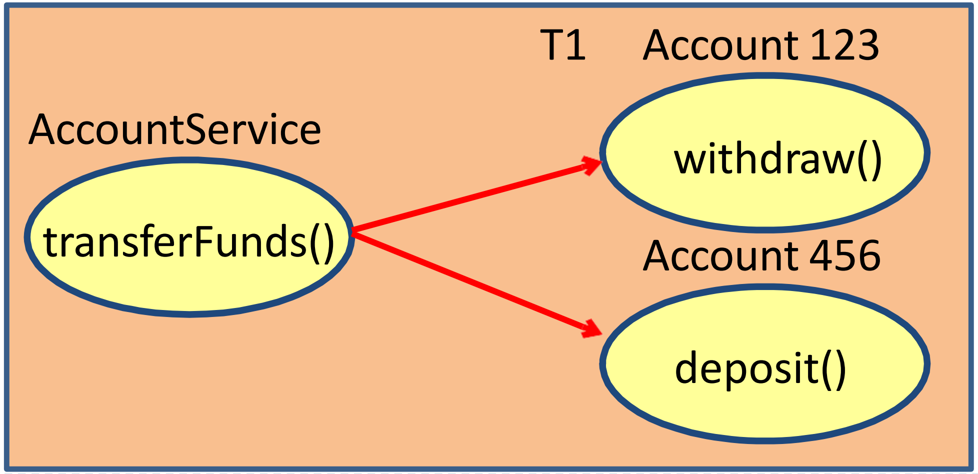
- 2 Phase Commit
  - Does not guarantee that nothing will go wrong
  - Is slow – multiple remote connections
- TX resources become dependent on each other
  - Need to keep locks until ALL resources finished
  - Again making things slower
- **The price you pay for coordinating!**

CS544 EA

Spring Transactions: Propagation

Transaction Propagation

- Transaction propagation defines the **interaction** between transactions and method calls
  - Normally any method called between begin() and commit() is part of the TX
  - A TX for transferFunds() will automatically propagates to withdraw() and deposit()



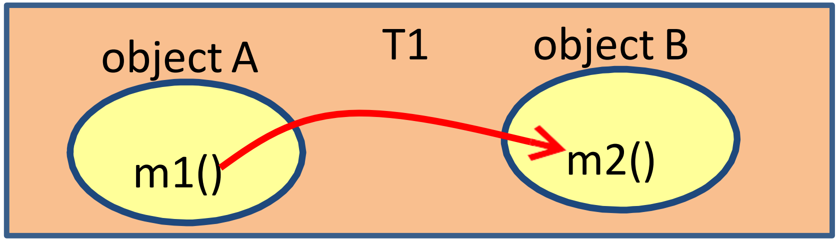
Propagation Options

- Spring provides **7 propagation options**:
  - REQUIRED
  - REQUIRES_NEW
  - MANDATORY
  - NESTED
  - SUPPORTS
  - NOT_SUPPORTED
  - NEVER

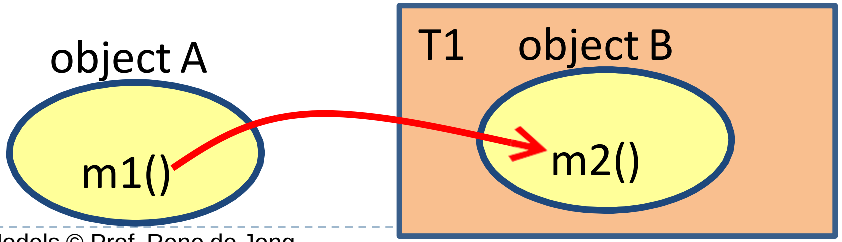
We'll also look at specifying isolation level, timeout, rollback and read-only

Propagation: REQUIRED

- If the calling method m1() runs in a transaction T1
  - Then method m2() joins the same transaction T1

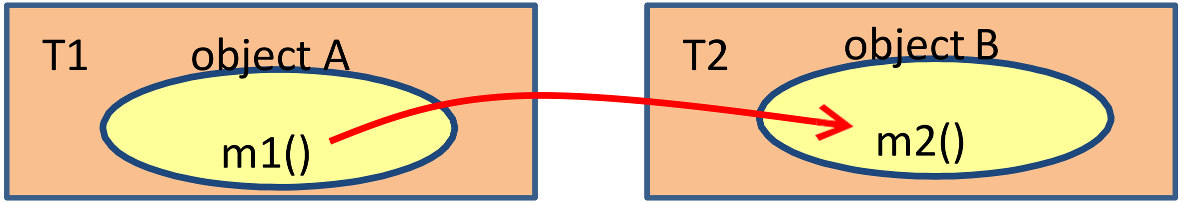


- If the calling method m1() does not run in a transaction
  - Then method m2() runs in a new transaction T1

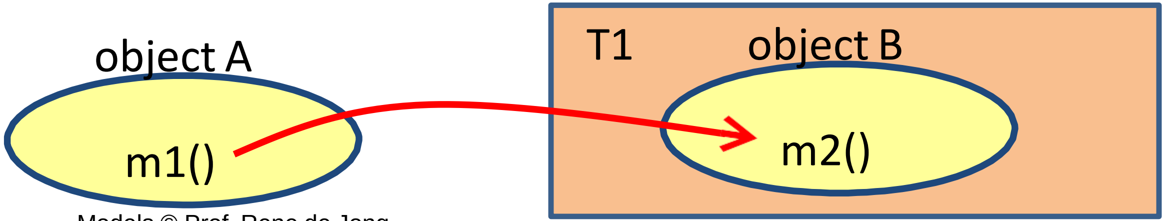


Propagation: REQUIRES_NEW

- If the calling method m1() runs in a transaction T1
  - Then method m2() runs in a new transaction T2



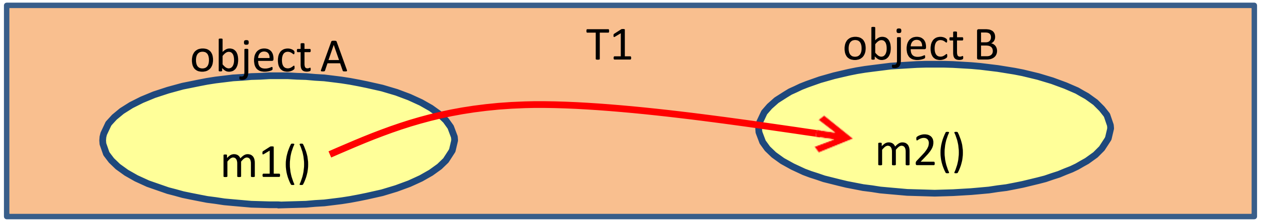
- If the calling method m1() does not run in a transaction
  - Then method m2() runs in a new transaction T1



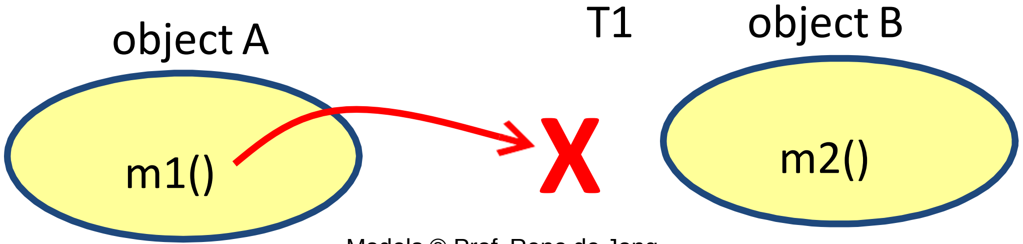
Models © Prof. Rene de Jong

Propagation: MANDATORY

- If the calling method m1() runs in a transaction T1
  - Then method m2() joins the same transaction T1



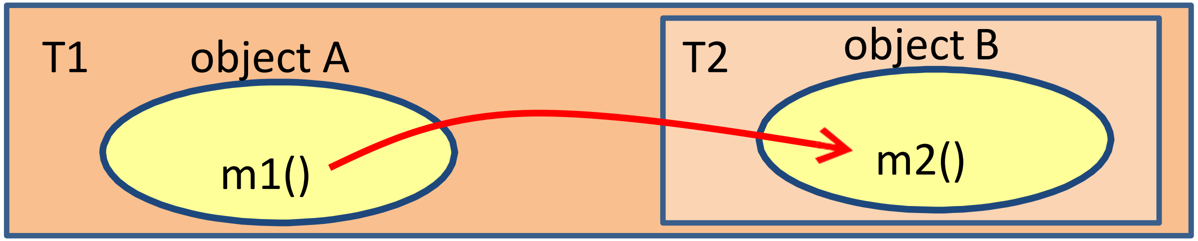
- If the calling method m1() does not run in a transaction
  - An exception is thrown



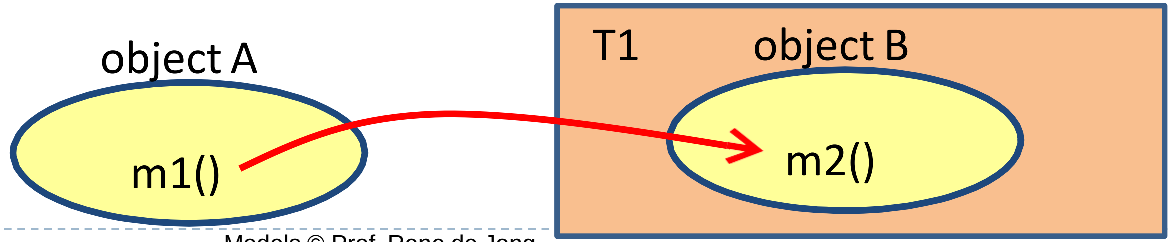
Models © Prof. Rene de Jong

Propagation: NESTED

- If the calling method m1() runs in a transaction T1
  - Then method m2() runs in a nested transaction T2

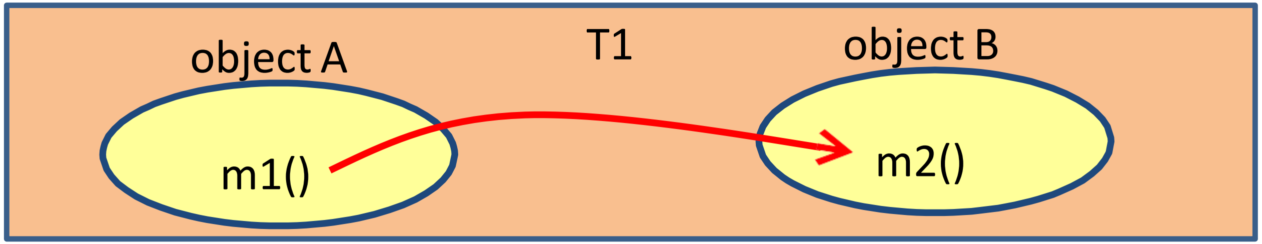


- If the calling method m1() does not run in a transaction
  - Then method m2() runs in a new transaction T1

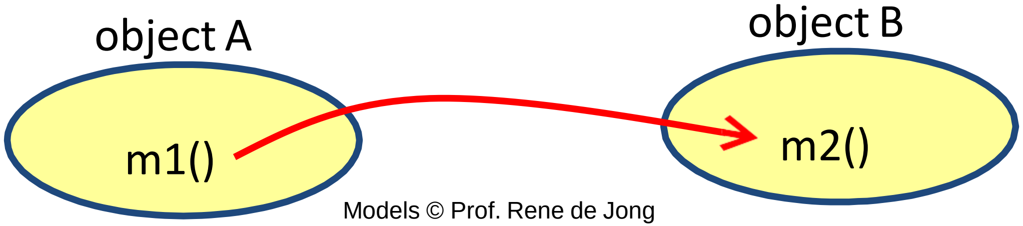


Propagation: SUPPORTS

- If the calling method m1() runs in a transaction T1
  - Then method m2() joins the transaction T1



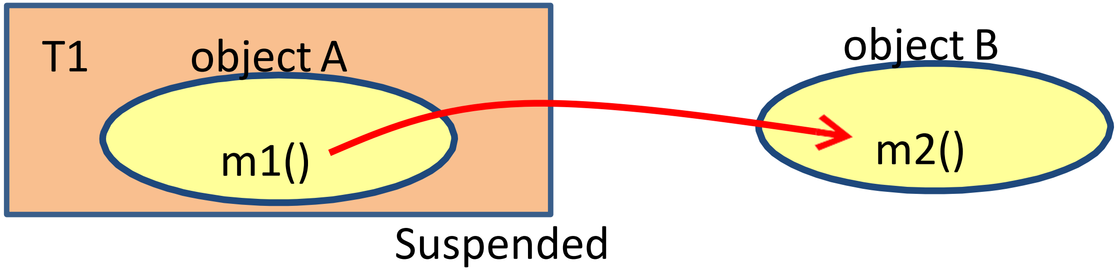
- If the calling method m1() does not run in a transaction
  - Then method m2() also does not run in a transaction



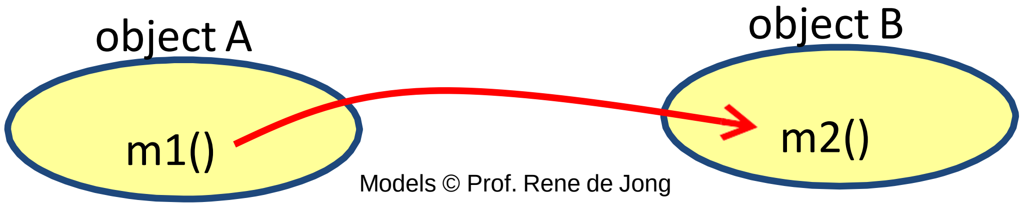
Models © Prof. Rene de Jong

Propagation: NOT_SUPPORTED

- If the calling method m1() runs in a transaction T1
  - Then method m2() does not run in a transaction



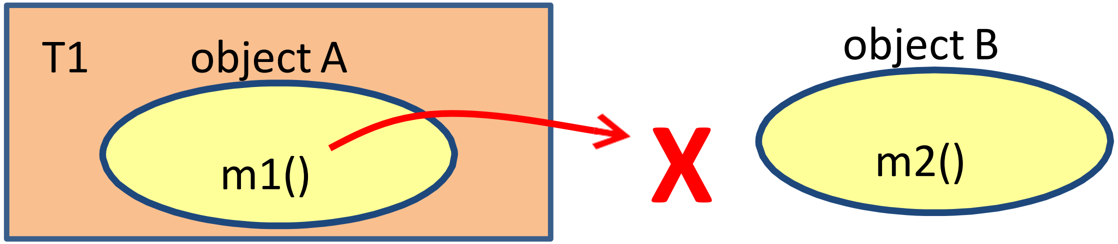
- If the calling method m1() does not run in a transaction
  - Then method m2() also does not run in a transaction



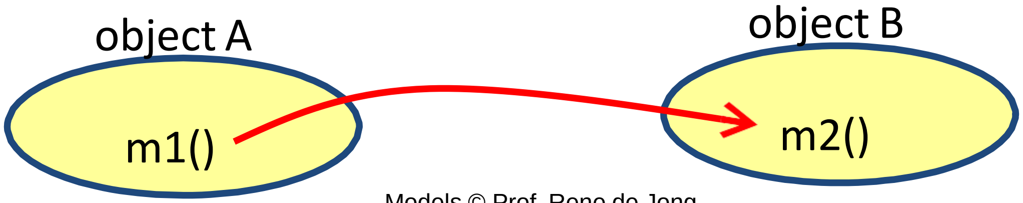
Models © Prof. Rene de Jong

Propagation: NEVER

- If the calling method m1() runs in a transaction T1
  - Then an exception is thrown



- If the calling method m1() does not run in a transaction
  - Then method m2() also does not run in a transaction



Models © Prof. Rene de Jong

Transaction Propagation

- Your propagation options are very dependent on your **transaction manager**
  - The default REQUIRED propagation is supported by every transaction manager (DB)
  - Propagation options that require transaction suspension or nesting are more problematic

Applications

CS544 EA

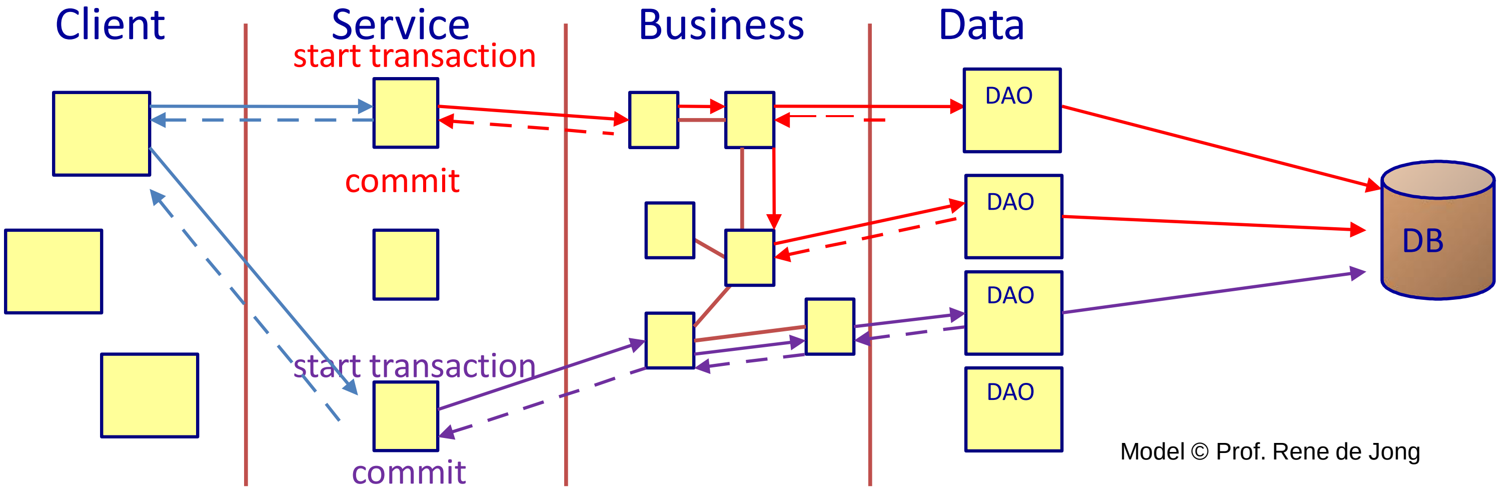
Spring Transactions

Spring Transaction Support

- Spring is not a transaction manager
  - We still need a transaction manager
    - JDBC transaction manager
    - Hibernate transaction manager
    - XA transaction manger (JTA)
- Spring provides an **abstraction for TX management**
  - You declare how transactions should be managed
  - Spring make it work with the underlying transaction manager

Transaction Demarcation

- The transactional demarcation is the specification of the **transactional boundaries**
- This is typical at the service level
  - Multiple DAO’s can be involved in one transaction
  - Creating a transaction per unit of work



Model © Prof. Rene de Jong

BMT

```
public class CustomerService {
  private CustomerDAO customerDao = new CustomerDAO();
  private AddressDAO addressDao = new AddressDAO();
  private CreditCardDAO ccDao = new CreditCardDAO();
  private EntityManager em = EntityManagerHelper.getCurrent();

  public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
                           Address billAddr) {
    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    em.getTransaction().begin();
    addressDao.create(shipAddr);
    addressDao.create(billAddr);
    ccDao.create(cc);
    customerDao.create(cust);
    em.getTransaction().commit();
  }

  ...
```

Programmatically begins the transaction

Transaction is automatically propagated to enclosed methods

Programmatically ends the transaction

CMT

```
@Service
public class CustomerService {
 private CustomerDAO customerDao;
 private AddressDAO addressDao;
 private CreditCardDAO ccDao;

 @Transactional(propagation=Propagation.REQUIRED)
 public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
       Address billAddr) {

  cc.setAddress(billAddr);
  cust.setShipAddress(shipAddr);
  cust.setCreditCard(cc);

  addressDao.create(shipAddr);
  addressDao.create(billAddr);
  ccDao.create(cc);
  customerDao.create(cust);
 }

 ...
```

Simply declare that a transaction is needed

REQUIRED is the default and therefore optional

Spring takes care of opening and closing the TX

Transaction propagates to called methods as normal

Class Annotations

```
@Repository
@Transactional(propagation = Propagation.REQUIRED)
public class AddressDao {

      @PersistenceContext
      private EntityManager em;

      @Transactional(propagation = Propagation.MANDATORY)
      public void create(Address addr) {
            em.persist(addr);
      }

      public Address get(int id) {
            return em.find(Address.class, id);
      }

      public void update(Address addr) {
            em.merge(addr);
      }

      public void delete(Address addr) {
            em.remove(addr);
      }
}
```

Annotating a class specifies that
all its methods should be Transactional

You can add method level annotations
to specify exceptions

These are propagation REQUIRED

Additional Options

- You can also specify the **isolation** level

```
@Repository
@Transactional(propagation = Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED)
public class AddressDao {

        @PersistenceContext
        private EntityManager em;
```

- Or that a transaction should be **read only**

```
@Repository
@Transactional
public class AddressDao {

        @Transactional(readOnly=true)
        public Address get(int id) {
                return em.find(Address.class, id);
        }
```

Additional Options

- A **timeout** in seconds (needs TXManager support)

```
@Repository
@Transactional
public class AddressDao {

        @Transactional(timeout=10)
        public void update(Address addr) {
                em.merge(addr);
        }
```

By default rollback for unchecked exceptions
but not for checked exceptions

- What exceptions to **rollback** for

```
@Repository
@Transactional(
        rollbackFor={MyCheckedException.class},
        noRollbackFor={MyRuntimeException.class}
)
public class AddressDao {
```

Spring Transactions Summary

- All database interactions always use a TX
- Global (XA) transactions use multiple resources
- Spring gives 7 Propagation options
- @Transactional can be applied to a classes and methods and can specify:
  - Propagation, isolation, read-only, timeout, and what exceptions a transaction should rollback for

CS544 EA

Spring and Hibernate Web Apps

Spring and Hibernate Web Apps

- We want to create web applications that use Spring and Hibernate
  - We'll first integrate Spring in a **Web container**
  - Then look at integrating Spring and **Hibernate**
  - And finally add Spring **Transaction** demarcation

Web Container

- The web-container will be the main application
  - Starting the Spring container when it starts

Including Container Startup

- Web Containers can register listeners
  - Allowing you to listen to container events
  - Spring provides a ContextLoaderListener that we can register in the web container

Web.xml

- The **<context-param>** tag can store data visible to the whole web app (all servlets etc)
- The **<listener>** tag registers a listener

```
<web-app … version="3.0">
  …
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/springconfig.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  …
</web-app>
```

Param to specify where to find Spring config file

Will start Spring when the app starts

Without web.xml

```
package application03;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

public class MyWebAppInitializer implements WebApplicationInitializer {

		@Override
		public void onStartup(ServletContext container) throws ServletException {
			// Create the Spring 'root' application context
			AnnotationConfigWebApplicationContext rootContext =
						new AnnotationConfigWebApplicationContext();
			rootContext.register(Config.class);

			// Manage the lifecycle of the root application context
			container.addListener(new ContextLoaderListener(rootContext));

			ServletRegistration.Dynamic hello = container.addServlet("Hello", new Hello());
			hello.addMapping("/hello");
		}
}
```

Servlet 3.0 and later also allow you to configure the container with Java

The web container will automatically detect and run any class that implements WebApplicationInitializer

Servlet Registration can also be done with @WebServlet or in web.xml

Getting Spring Context in Servlet

```
public class ViewCustomer extends HttpServlet {
  private static final long serialVersionUID = 1L;

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException {

    int custId = Integer.parseInt(req.getParameter("custId"));

    // get customerService bean from spring
    ServletContext context = getServletContext();
    WebApplicationContext applicationContext =
       WebApplicationContextUtils.getWebApplicationContext(context);
    CustomerService custServ = applicationContext.getBean(
       "customerService", CustomerService.class);

    // make customer available in request, for view rendering
    Customer cust = custServ.getCust(custId);
    req.setAttribute("cust", cust);

    // forward to view customer page
    req.getRequestDispatcher("customer.jsp").forward(req, resp);
  }
}
```

Inside a Servlet or Filter get the Spring Context from Web Context

After which you can get Spring Beans from it

Spring and Hibernate-JPA

- Spring can **fully configure and start Hibernate**
  - Removing the need for persisntence.xml
  - Makes EntityManagerFactory Spring Bean (singleton)
  - Gives ThreadLocal functionality for EntityManager
  - Also provides OpenEntityManagerInView filter
    - Which integrates nicely with Spring TX management

Spring JPA Config XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">

        <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName"
                        value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost/cs544" />
                <property name="username" value="root" />
                <property name="password" value="root" />
        </bean>

        <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="jpaVendorAdapter">
                        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                                <property name="generateDdl" value="true" />
                                <property name="database" value="MYSQL" />
                        </bean>
                </property>
                <property name="jpaProperties">
                        <props>
                                <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
                                <prop key="hibernate.show_sql">true</prop>
                                <prop key="hibernate.format_sql">true</prop>
                                <prop key="hibernate.id.new_generator_mappings">false</prop>
                                <prop key="javax.persistence.schema-generation.database.action">drop-and-create</prop>
                        </props>
                </property>
                <property name="packagesToScan" value="cs544" />
        </bean>
        ...
</beans>
```

Spring JPA Config Java

```
@Configuration
@ComponentScan("cs544")
public class Config {
        @Bean
        public DataSource dataSource() {
                DriverManagerDataSource dataSource = new DriverManagerDataSource();
                dataSource.setDriverClassName("com.mysql.jdbc.Driver");
                dataSource.setUsername("root");
                dataSource.setPassword("root");
                dataSource.setUrl("jdbc:mysql://localhost/cs544");
                return dataSource;
        }

        @Bean
        public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
                LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
                emf.setDataSource(dataSource());
                emf.setPackagesToScan("cs544");

                Properties properties = new Properties();
                properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
                properties.setProperty("hibernate.id.new_generator_mappings", "false");
                properties.setProperty("hibernate.show_sql", "true");
                properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");

                JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
                emf.setJpaVendorAdapter(vendorAdapter);
                emf.setJpaProperties(properties);
                return emf;
        }
}
```

Example from DB to Web

```
@Entity
public class Customer {
                @Id
                @GeneratedValue
                private Long id;
                private String name;

                public Long getId() {
                                return id;
                }

                public void setId(Long id) {
                                this.id = id;
                }

                public String getName() {
                                return name;
                }

                public void setName(String name) {
                                this.name = name;
                }
}
```

Using either the web.xml or WebApplicationInitializer shown earlier

Import.sql

```
INSERT INTO Customer VALUES(NULL, "James Reagon");
INSERT INTO Customer VALUES(NULL, "Lilly Johnson");
INSERT INTO Customer VALUES(NULL, "George Tall");
```

Example DAO

```
@Repository
public class CustomerDao {
		@PersistenceContext
		private EntityManager em;

		public List<Customer> getAll() {
				return em.createQuery("from Customer", Customer.class).getResultList();
		}
}
```

Example Service

```
@Service
public class CustomerService {
		@Resource
		private CustomerDao customerDao;

		public List<Customer> getCustomers() {
				return customerDao.getAll();
		}
}
```

Cannot do BMT, throws exception that you should use Spring TX (CMT).

We'll add these in the next section (for now Transaction Per Operation!)

Example Controller

```
@WebServlet(name = "Customers", urlPatterns = { "/customers" })
public class Customers extends HttpServlet {
            private static final long serialVersionUID = 1L;

            @Override
            protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                    throws ServletException, IOException {

              ServletContext context = getServletContext();
              WebApplicationContext applicationContext =
                WebApplicationContextUtils.getWebApplicationContext(context);
              CustomerService custServ = applicationContext.getBean(
                "customerService", CustomerService.class);


                      request.setAttribute("customers", custServ.getCustomers());
                      String jsp = "/Customers.jsp";
                      RequestDispatcher dispatcher = context.getRequestDispatcher(jsp);
                      dispatcher.forward(request, response);
            }
}
```

Example JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Customers</title>
</head>
<body>
        <h1>Customers:</h1>
        <ul>
                <c:forEach items="${customers}" var="customer">
                        <li>${customer.name}</li>
                </c:forEach>
        </ul>
</body>
</html>
```

CS544 EA

SH Web Apps: Transactions

Spring and Hibernate Transactions

- We'll add **@Transactional** annotations
  - Configure Spring to find them
  - Configure the Hibernate TX manager to use them

Both XML and Java Config Examples

Springconfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">

                ...

                <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
                                <property name="entityManagerFactory" ref="entityManagerFactory" />
                </bean>

                <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

Create a txManager bean using the EntityManagerFactory

Tell Spring to look for @Transactional annoations and use the txManager

Needs tx namespace

Config.java

```
@Configuration
@ComponentScan("cs544")
@EnableTransactionManagement
public class Config {
                @Bean
                public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {
                  JpaTransactionManager transactionManager = new JpaTransactionManager();
                  transactionManager.setEntityManagerFactory(emf);
                  return transactionManager;
                }

                ...
}
```

Tell Spring to look for @Transactional annotations

Needs a transactionManager bean in order to function

Minimal @Transactional

- Adding @Transactional to @Service classes will give **reasonable** transactional boundaries

```
@Service
@Transactional
public class CustomerService {
        @Resource
        private CustomerDao customerDao;

        public List<Customer> getCustomers() {
                return customerDao.getAll();
        }
}
```

More Serious

```
@Service
@Transactional(propagation = Propagation.REQUIRES_NEW)
public class CustomerService {
		@Resource
		private CustomerDao customerDao;

		public List<Customer> getCustomers() {
				return customerDao.getAll();
		}
}
```

Each service level method should have own TX

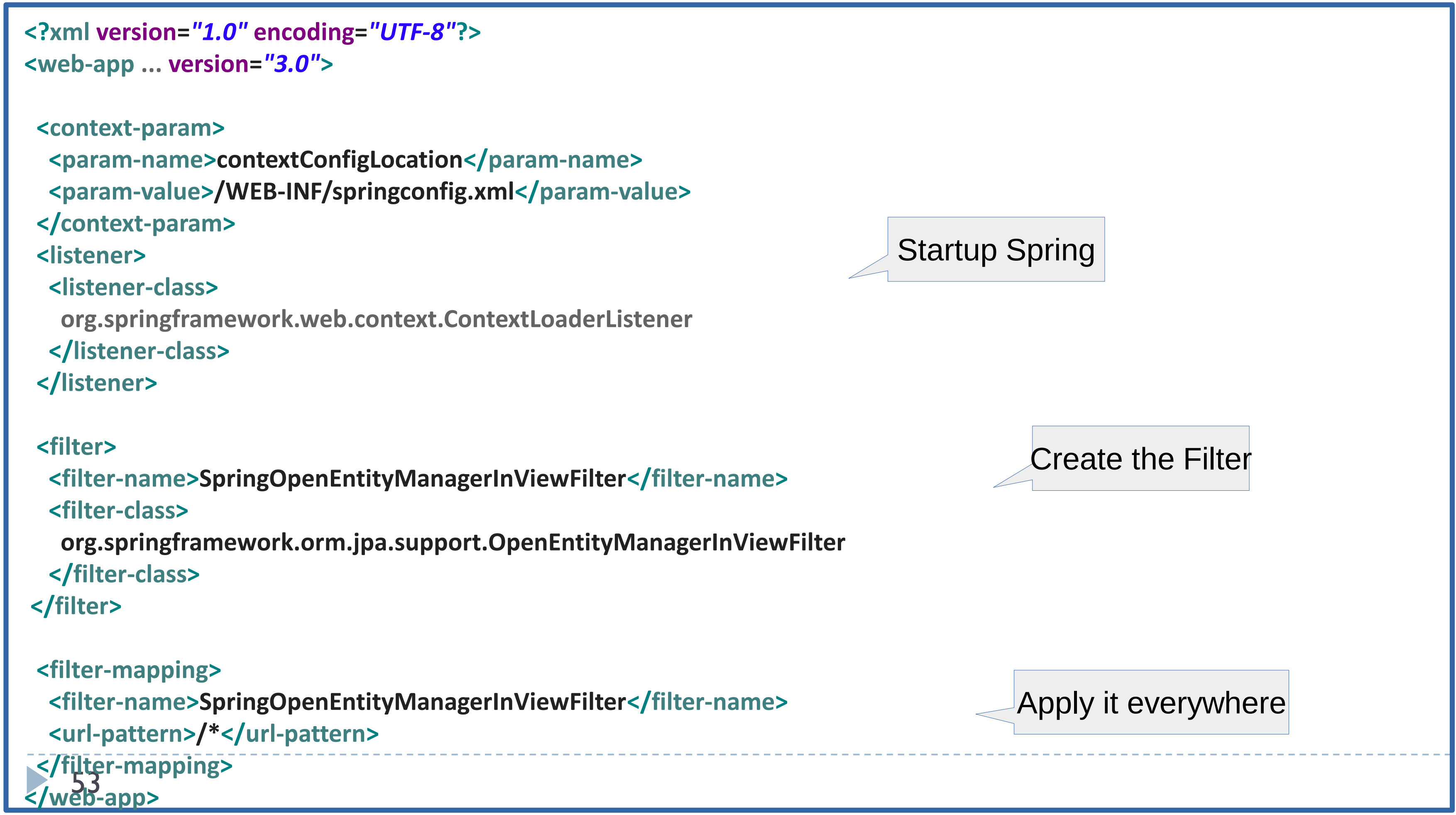
```
@Repository
@Transactional(propagation = Propagation.MANDATORY)
public class CustomerDao {
		@PersistenceContext
		private EntityManager em;

		public List<Customer> getAll() {
				return em.createQuery("from Customer", Customer.class).getResultList();
		}
}
```

DAO methods should never be called without a TX

CS544 EA

SH Web Apps: OpenEntityManagerInView

Web.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... version="3.0">

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/springconfig.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <filter>
    <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
    <filter-class>
      org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
    </filter-class>
 </filter>

  <filter-mapping>
    <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Startup Spring

Create the Filter

Apply it everywhere

WebApplicationInitializer

```java
package cs544.application05;

import javax.servlet.FilterRegistration;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;

import org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

public class MyWebAppInitializer implements WebApplicationInitializer {

            @Override
            public void onStartup(ServletContext container) throws ServletException {
                        AnnotationConfigWebApplicationContext rootContext =
                                                new AnnotationConfigWebApplicationContext(
                        rootContext.register(Config.class);
                        container.addListener(new ContextLoaderListener(rootContext));

                        FilterRegistration.Dynamic openInView =
                                                container.addFilter("OpenInView", new OpenEntityManagerInViewFilter());
                        openInView.addMappingForUrlPatterns(null, true, "/*");
            }
}
```

Or if you use a WebApplicationInitializer instead of web.xml you can register the filter like this

From DB to Web (with Filter)

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String name;
        @OneToOne(fetch = FetchType.LAZY)
        private Address address;

        …
```

Added a LAZY association to demonstrate OpenEntityManagerInView working correctly

```
@Entity
public class Address {
        @Id
        @GeneratedValue
        private Long id;
        private String place;

        …
```

Import.sql

```
INSERT INTO Customer VALUES(NULL, "James Reagon", 1);
INSERT INTO Customer VALUES(NULL, "Lilly Johnson", 2);
INSERT INTO Customer VALUES(NULL, "George Tall", 3);
INSERT INTO Address VALUES(NULL, "New York");
INSERT INTO Address VALUES(NULL, "Los Angeles");
INSERT INTO Address VALUES(NULL, "Chicago");
```

DAO and Service

Using the more serious @Transactional

```
@Repository
@Transactional(propagation = Propagation.MANDATORY)
public class CustomerDao {
		@PersistenceContext
		private EntityManager em;

		public List<Customer> getAll() {
				return em.createQuery("from Customer", Customer.class).getResultList();
		}
}
```

```
@Service
@Transactional(propagation = Propagation.REQUIRES_NEW)
public class CustomerService {
		@Resource
		private CustomerDao customerDao;

		public List<Customer> getCustomers() {
				return customerDao.getAll();
		}
}
```

Controller

```
@WebServlet(name = "Customers", urlPatterns = { "/customers" })
public class Customers extends HttpServlet {
        private static final long serialVersionUID = 1L;

        @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException {

          ServletContext context = getServletContext();
          WebApplicationContext applicationContext =
            WebApplicationContextUtils.getWebApplicationContext(context);
          CustomerService custServ = applicationContext.getBean(
            "customerService", CustomerService.class);


                request.setAttribute("customers", custServ.getCustomers());
                String jsp = "/Customers.jsp";
                RequestDispatcher dispatcher = context.getRequestDispatcher(jsp);
                dispatcher.forward(request, response);
        }
}
```

Same as before

JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Customers</title>
</head>
<body>
        <h1>Customers:</h1>
        <ul>
                <c:forEach items="${customers}" var="customer">
                        <li>${customer.name}: ${customer.address.place}</li>
                </c:forEach>
        </ul>
</body>
</html>
```

Lazy Loads Address

Summary

- Spring can integrate with a web container
  - By registering it as a listener
- Hibernate configuration can be done in Spring
  - Spring starts and configures Hibernate
- Spring Transactional Demarcation
  - Uses the hibernate transactionManager
- Spring provides an EntityManagerInViewFilter