EJB, Spring

Imre Gábor

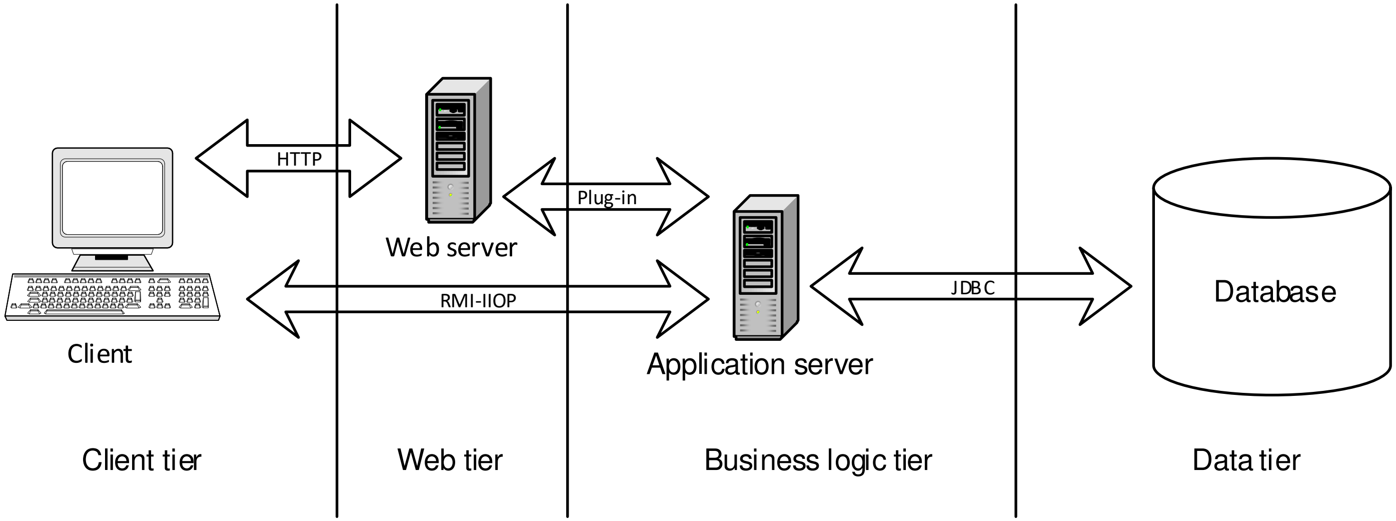
Q.B224

gabor@aut.bme.hu

Java EE

Java Enterprise Edition

- Java EE is an architecture for developing enterprise-scale applications with the Java language and internet technologies
- Open standard, with several implementations
- The software product implementing the Java EE specification: application server
- 3-layer architecture



Java EE services

- Multithreading
- Transactions
- Security
- Persistence
- Name service
- Handling the lifecycle of objects
- Remote method invocation
- Asynchronous messaging
- Scalability
- Load balancing

Java EE APIs

- The services can be accessed through different APIs, e.g.
  - Java Persistence API (JPA): object-relational mapping
  - Enterprise JavaBeans (EJB): distributed, business logic components
  - Java Transaction API (JTA): transaction handling
  - Java Authentication and Authorization Service (JAAS): security
  - Web technologies (Servlet, JSP, JSF)
  - Web services
    - Java API for XML-Based Web Services (JAX-WS): SOAP-based XML web services
    - Java API for RESTful Web Services (JAX-RS): RESTful web services
- Our appliction will be **portable** across Java EE-compliant application servers

Commonly used Java EE application servers

- Glassfish (reference implementation, open source)
- IBM WebSphere Application Server
- Oracle WebLogic Server
- JBoss (new versions: WildFly) (open source)
- Jetty (open source, only web container)
- Apache Tomcat (open source, only web container)
- TomEE (Tomcat+OpenEJB)

Enterprise JavaBeans

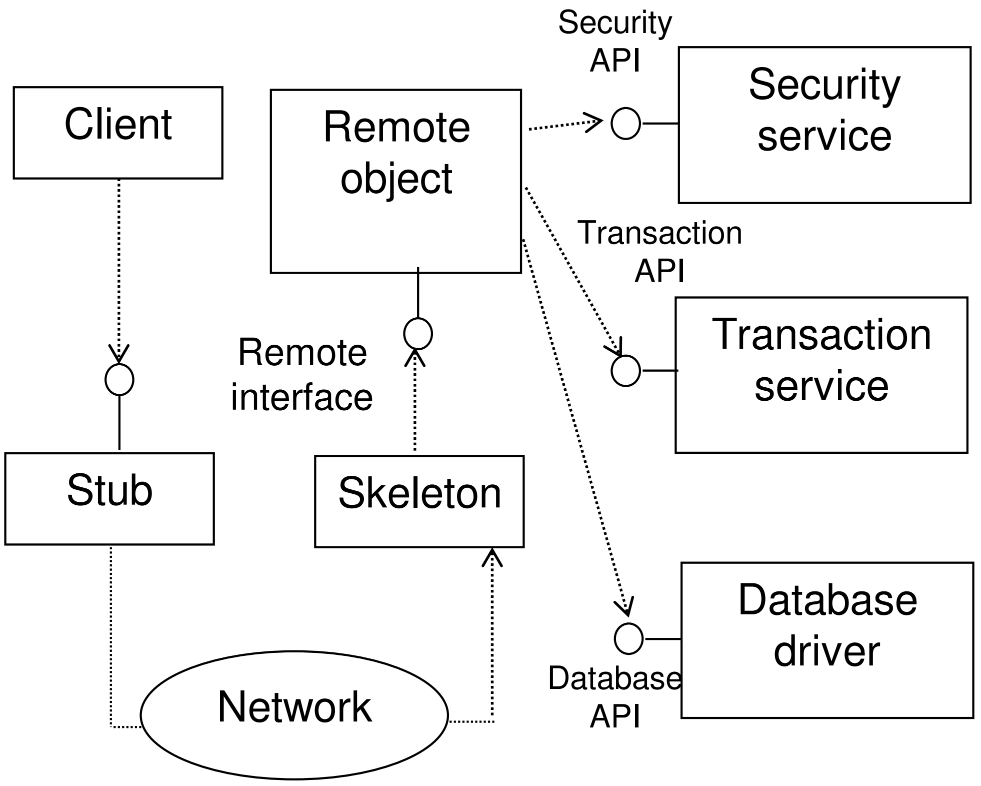
Concept of EJB

- Distributed, server-side components with standard interface containing the business logic of an application
- They run in an EJB-container, which
  - Hides the details of network communication (RMI-IIOP)
  - Hides multithreading
  - Hides transaction handling
  - Offers other services

Types of EJBs

- Session bean
  - Contains business logic functions
- (Entity bean)
  - Obsolete ORM solution, replaced by JPA
- Message-driven bean
  - Handles messages arriving into asyncronous messaging systems, integrated with the JMS API

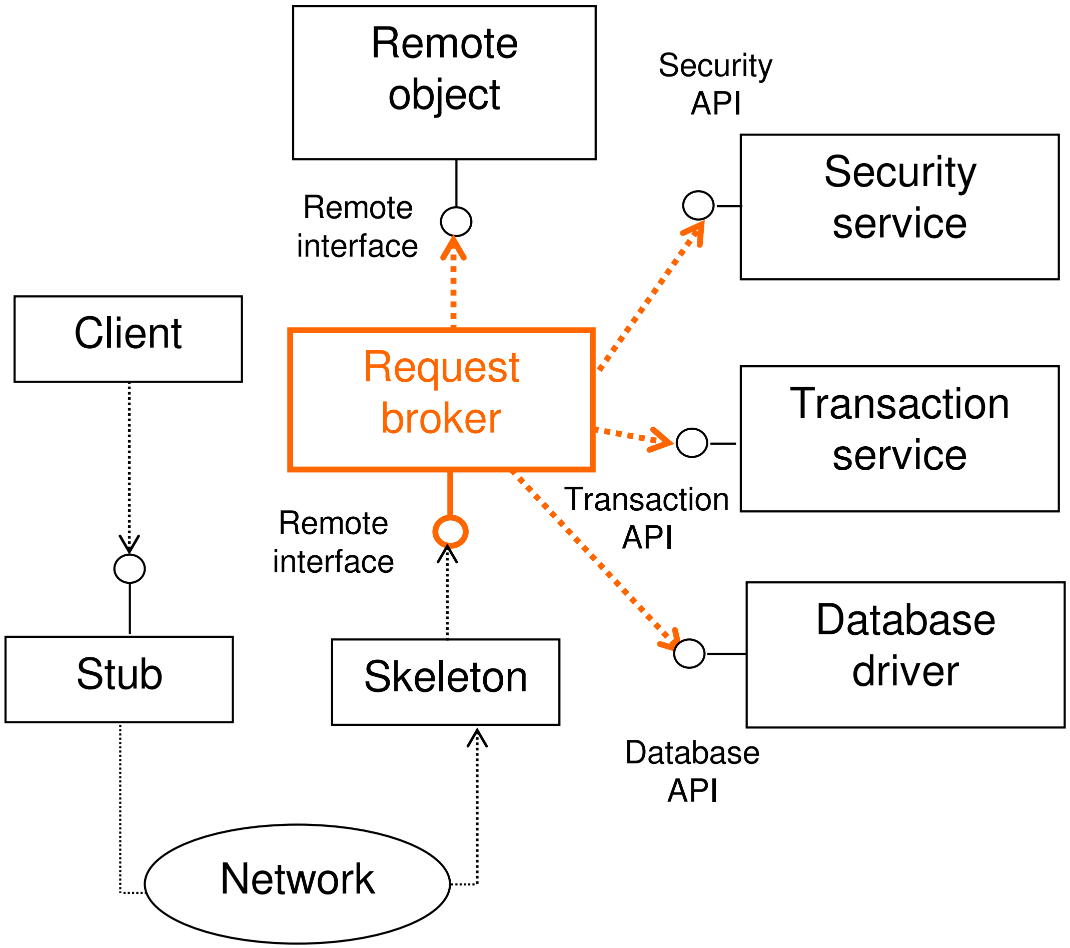
Explicit middleware



Drawbacks of explicit middleware

- Source code is bloated with middleware calls
- Not flexible (when selling the component, source code has to be given to customer if he wants different implementation e.g. in transaction handling)

Implicit middleware



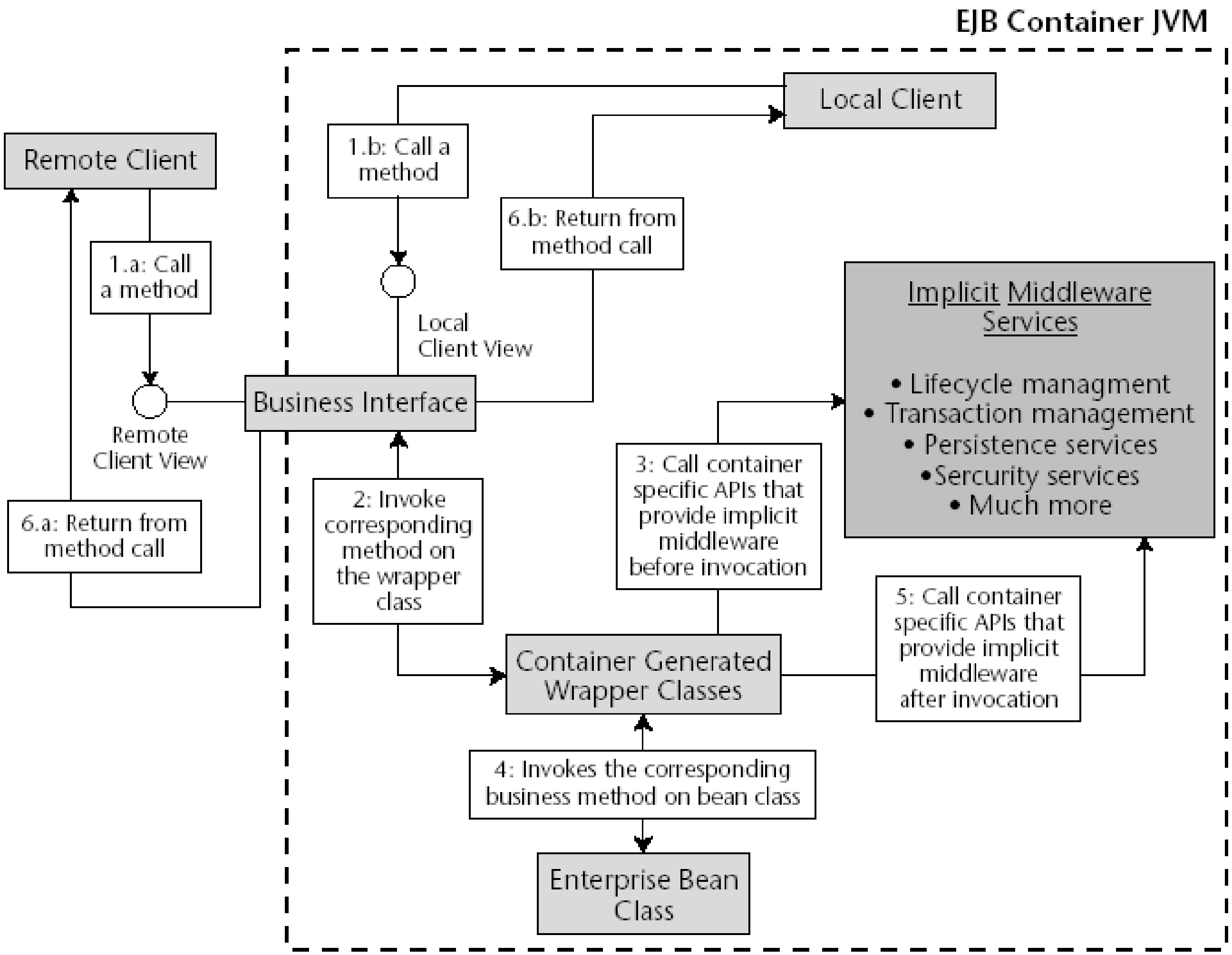
Implicit middleware

- A configuration file (descriptor) describes the way we would like to use middleware services
- The Request broker is generated based on this configuration file
- The source code contains only the business functionality
- The descriptor can be modifed by the customer, we do not have to publish the source code
- It can be implemented by separating interface and implementation

Structure of a session bean

- Enterprise bean class (implementation class, bean class)
- Business interface
  - Can be remote (acessible from different JVM) or local (accessible only in the same application server)
  - Can be omitted since EJB 3.1 (no-interface view)
- Container generated wrapper class
  - Implements the business interface or extends the bean class in case of no-interface view)
  - Calls middleware services and delegates the calls of the clients to the implementation class

Implicit middleware in EJB 3: the process of calling an EJB


EJB Container JVM
Local Client
Remote Client
1.b: Call a method
6.b: Return from method call
1.a: Call a method
Local Client View
Implicit Middleware Services
• Lifecycle managment
• Transaction management
• Persistence services
•Sercurity services
• Much more
Business Interface
Remote Client View
6.a: Return from method call
2: Invoke corresponding method on the wrapper class
3: Call container specific APIs that provide implicit middleware before invocation
5: Call container specific APIs that provide implicit middleware after invocation
Container Generated Wrapper Classes
4: Invokes the corresponding business method on bean class
Enterprise Bean Class

Obtaining reference to an EJB

- The object the client sees is never an instance of the implementation class, but a client side stub generated by the container, because
  - Implicit middleware calls can be introduced this way
  - In case of remote clients, the network communication (serialization, deserialization..) can be implemented in the stub
- →cannot use the **new** keyword to create an instance
- Solution: name service, just like in the case of looking up a DataSource
  - Can be simplified with the **@EJB** annotation

Managing state in session beans

- Exactly one of the following annotations has to be added to the bean class: **@Stateless**, **@Stateful**, **@Singleton**
- Stateless sesison bean: calls of the same client can be routed to different server-side EJB instances, so no client-specific state can be stored in the member variables of the bean
- Stateful session bean: calls in the same client reference are routed to the same server-side EJB instance
- Singleton session bean: one server-side instance handles all client requests

Java EE annotations

- Java EE APIs define a lot of annotations, which are parsed by the application server when an application is deployed to it. The application server will provide runtime services based on these annotations.
- Early Java EE (J2EE) versions used XML deployment descriptors for this purpose, but they are harder to use for developers
- If a middleware aspect is defined both with annotations and XML, XML has higher priority (the idea of implicit middleware is preserved this way)

Enterprise JavaBeans

Using JPA in EJB environment

Managed persistence context

- Most important help from the EJB-container: the JPA persistence context can be injected, e.g.

```
@Stateless
class PersonService {
 @PersistenceContext
 EntityManager em;

 public void createEmployee{
  em.persist(new Employee(12345, "Gabor"));
 }
}
```

Managed persistence context

- The **EntityManagerFactory**
  - Is created once, at application startup
- Properties of the injected **EntityManager**:
  - It is created at the start of the transaction
  - It is closed at the end of the transaction (by default)
  - If other classes use an entity manager injected into them, they will access the same persistence context, as long as we are in the same transaction

Participants in a Java EE transaction

- Transactional object (component)
  - An application component taking part in a transaction (e.g. an EJB)
- Transactional resource
  - Can be read/written, e.g. database, message queue
- Resource manager
  - The component through which the transactional resource can be accessed, e.g. JDBC driver, JMS driver
- Transaction manager
  - It is needed at *distributed* transactions (where multiple transactional resources are involved)
  - It coordinates the distributed transaction via two-phase commit
  - In case of Java EE, it is part of the application server, and can be accessed via JTA (Java Transaction API)

JPA entities and transactions

- **Local** or **JTA** transaction management can be defined in the persistence.xml
- Local (for use without EJB-container): transactions are handled via the **EntityTransaction** interface (obtained from the EntityManager)
- JTA: transactions are handled by a transaction manager implementing JTA. The transaction manager calls the JDBC driver directly

JPA entities and transactions

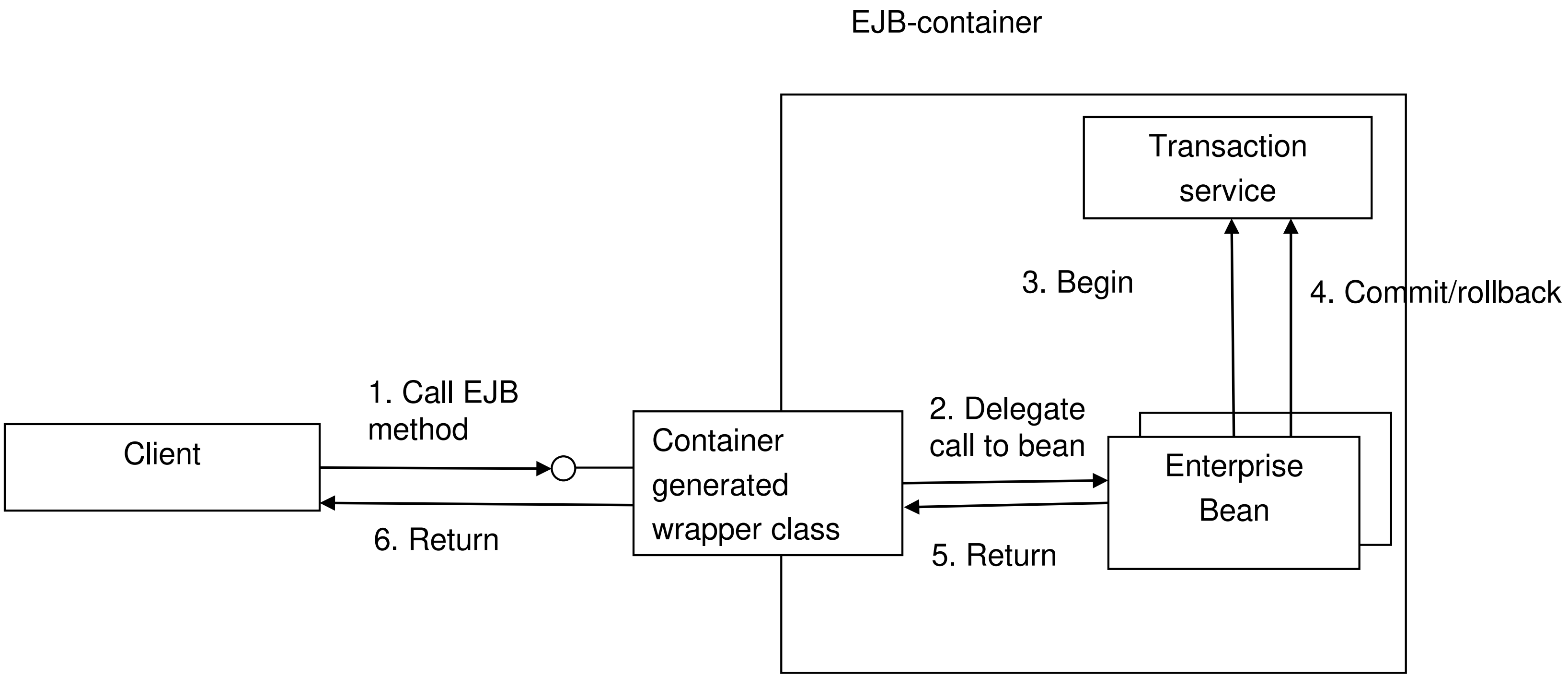
- The operations that modify entities (persist, merge, remove, refresh) can only be called inside of a transaction
- The entity instances found by a query are in managed state if they were loaded inside of a transaction
- If the query was run outside of a transaction, the found instances are in detached state by default

Transaction management of EJBs

- Two options:
  - bean managed (programmatic)
  - or container managed (declarative)
- Each EJB can use exactly one of these options:
- **@TransactionManagement(BEAN/CONTAINER)**
  - When omitting this annotation, container managed transaction handling is the default

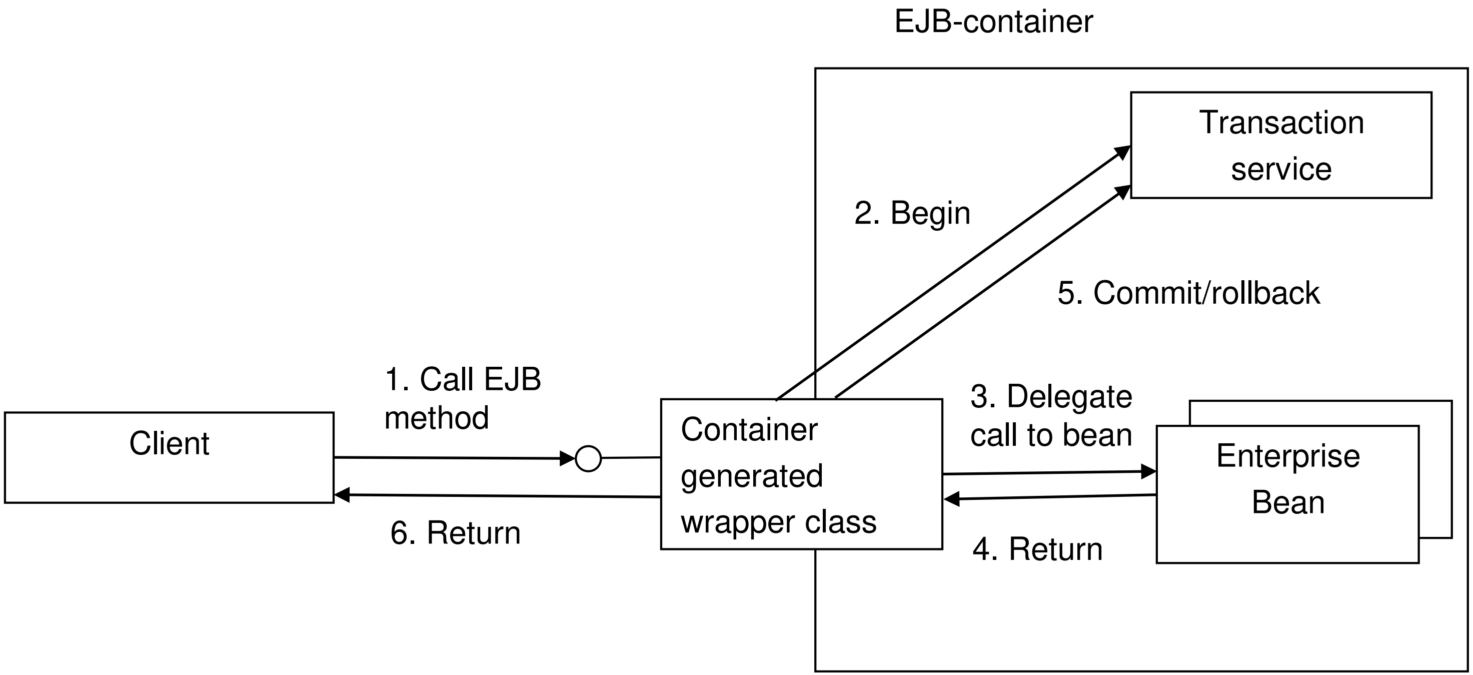
Bean managed transactions

- The source code of the implementation class contains the begin/commit/rollback of the transaction
- Transaction handling is fully controlled by the developer



Container managed transaction

- The container generated wrapper class contains the begin/commit/rollback, with these default rules:
  - Automatic rollback: if an exception was thrown that is of type RuntimeException or a subclass of it
  - Automatic commit: if no exception was thrown, or the thrown exception is not of type RuntimeException (nor a subclass of it)
- Overriding this default behavior:
  - Exception class annotated with **@ApplicationException(rollback=true/false)**
  - Call EJBContext.setRollbackOnly() in bean class → rollback will happen



Transaction attributes

- In case of container managed transactions, the beginning of transactions is defined at method level, via the **@TransactionAttribute** annotation
- **REQUIRED** is the default, if this annotation is omitted → each EJB method is transactional by default
- Note on **REQUIRES_NEW**: T1 is suspended while T2 runs → the success of T2 has no effect on the success of T1

Transaction attribute value	Tr. Existing at the call	Transaction of the called EJB
REQUIRED	None T1	T1 T1
REQUIRES_NEW	None T1	T1 T2, T1 suspended
SUPPORTS	None T1	None T1
MANDATORY	None T1	Exception T1
NOT_SUPPORTED	None T1	None None, T1 suspended
NEVER	None T1	None Exception

Spring

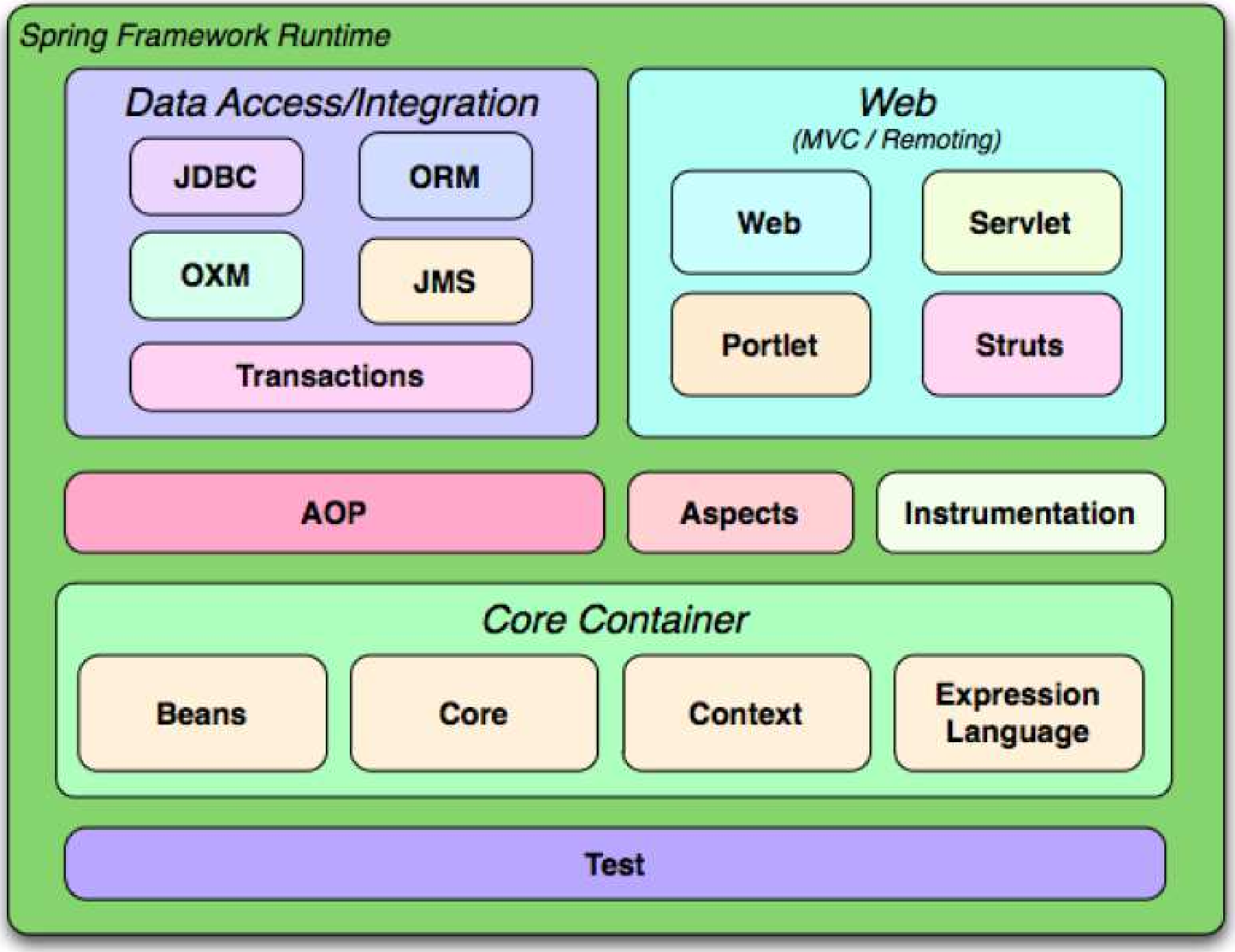
Principles of Spring

- Goal: solutions for enterprise applications simpler than in the early versions of J2EE (mostly EJB 1.x and 2.x were complicated)
- Java EE 5, 6 took over many things from here later
- Usage of interfaces
- OO design
- Modular structure
- Components can be tested in isolation
- Exceptions are RuntimeExceptions
- Java SE or simple web container is enough, no application server is needed

What does Spring offer?

- A lightweight, non-invasive container that takes care of the configuring, "wiring" of our application objects
  - Dependency Injection (DI) or Inversion of Control (IoC)
- A unified abstraction for transaction management, with pluggable transaction manager (local/JTA)
- Helper classes to write simpler JDBC code
- Facilities to use ORM technologies (**JPA**, JDO, iBatis, …) more efficiently
- Full AOP support (e.g. declarative transactions)
- A web framework (Spring MVC) following MVC pattern, supporting several view technologies
- …

Spring modules



Spring

Dependency injection

Advantages of dependency injection (DI)

- Objects do not hardwire the implementation classes of the member variables they use
- The so called *injector* performs the creation of complex object graphs
- It can eliminate complex initialization code (e. g. JNDI lookup)
- Environment-dependent behavior of the application is easily achievable by simply reconfiguring the injector
- We can replace the dependencies of an object with mock objects when writing unit tests

Dependency injection in Spring

- Supports member-, constructor- and setter injection
- Bean: a class handled by Spring, that can be injected to other beans, and can inject other beans as well
- Every module in Spring is built on the DI feature → the behavior of these modules (e.g. Spring Security) can be customized via built-in or own beans injected into other beans offered by the modules
- Beans have a scope: singleton by default (other options: prototype, request, session)
- The evolution of Spring configuration:
  1. XML file
  2. Annotations + Java classes (JavaConfig)
  3. Spring Boot: automatic default configuration based on classpath
     - Customizable via .properties or .yaml files
     - Fully overridable with JavaConfig

Spring DI example

```
public class CommandService {
    private SettingsService settingsService;

    public void setSettingsService(SettingsService setServ) {
            this.settingsService = setServ;
    }
    public void sendCommand(String command){
            DateFormat df = settingsService.getDateFormat() ;
            …
    }
}

public class SettingsService {
    public SimpleDateFormat getDateFormat() { …}
}
```

- This code does not use Spring API at all! (→ non-invasive)

Spring DI example: beans.xml

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

  <bean id="settingsService"     class="hu.bme.aait.SettingsService"/>

  <bean id="commandService"  class="hu.bme.aait.CommandService">
          <property name="settingsService" ref="settingsService"/>
  </bean>
...
</beans>
```

Spring DI example: getting a bean

```
ApplicationContext ctx =
 new ClassPathXmlApplicationContext(
     new String[] {"beans.xml"});
CommandService service =
 (CommandService)ctx.getBean("commandService");
service.sendCommand("Command1");
```

- The above code is rarely needed, because we can configure Spring to automatically create the ApplicationContext at the startup of the application

Spring DI with annotations

- The injection points are annotated with `@Autowired` (or with `@Inject` from JSR-330)
- One line is enough in the beans.xml:

```
<context:component-scan base-package="hu.bme.aait"/>
```

- We should put Spring beans into this package, and annotate them (`@Service` is one possible annotation, but other options canbe used as well, e.g. `@Component`):

```
@Service
public class CommandService {
  @Autowired
  private SettingsService settingsService;
...
}
@Service
public class SettingsService {...}
```

Spring DI JavaConfig

```
@Configuration
public class AppConfig {
  @Bean
  public MyService myService() {
        return new MyServiceImpl();
  }
}
```

- Instead of this:

```
<beans>
  <bean id="myService"
        class="com.acme.services.MyServiceImpl"/>
</beans>
```

Spring

Support for data access

DAO support

- Support for the simpler use of JDBC, JPA, JDO, Hibernate, iBatis:
  - Injectable **EntityManager**, **SessionFactory** (Hibernate), **JDBCTemplate**
  - Unified model for transacton handling (local/global, declarative)

JPA with Spring

- EntityManager can be injected into Spring beans

```
@PersistenceContext
EntityManager em;
```

- Necessary configuration in XML:

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="myPU" />
</bean>
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
```

- Or with JavaConfig

```
@Configuration
public class JPAConfigOne {
      @Bean
      public LocalEntityManagerFactoryBean entityManagerFactory () {
            LocalEntityManagerFactoryBean lemfb = new
                                 LocalEntityManagerFactoryBean();
            lemfb.setPersistenceUnitName("myPU");
            return lemfb;
      }

}
```

JPA with Spring

- Previous solutions assumed a persistence.xml conatining the details of the database (JNDI name), and the persistence unit name in the XML (**myPU**) is referenced by the configuration
- Spring supports configuration of JPA without persistence.xml:

```
@Configuration
public class JPAConfigTwo {

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean lcemfb = new LocalContainerEntityManagerFactoryBean();
    lcemfb.setDataSource(dataSource());
    lcemfb.setPersistenceUnitName("myPU");
    lcemfb.setPackagesToScan(new String[]{"com.xy.entities"});
    return lcemfb;
  }

  @Bean
  public DataSource dataSource() {
    JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
    return dataSourceLookup.getDataSource("java:comp/env/jdbc/mydb");
  }
}
```

- With Spring Boot JPA can be used almost without configuration:
  - Dependency **spring-boot-starter-data-jpa** in the classpath
  - 1 row in application.properties: **spring.datasource.jndi-name=jdbc/mydb**

Spring Data

- Separate module for advanced support of data access
- With the use of Repository interfaces, a siginficant part of the data access code is not to be written by the developer

```java
public interface CrudRepository<T, ID extends Serializable> extends
Repository<T, ID> {

        <S extends T> S save(S entity);

        Optional<T> findById(ID primaryKey);

        Iterable<T> findAll();

        Long count();

        void delete(T entity);

        boolean exists(ID primaryKey);
...
}
```

Spring Data

- Use of repository interfaces
  - Write a specific Repository for your own entity, e.g.

```
interface PersonRepository extends JpaRepository<Person, Long> { }
```

  - Configuration:
    - `@EnableJpaRepositories`,
    - or `<jpa:repositories base-package="com.acme.repositories"/>`
    - or spring-boot-starter-data-jpa
  - The repository can be injected into other Spring beans. Spring Data will generate the implementation for the interface at application startup:

```
@Autowired private PersonRepository repository;
```

Spring Data

- More features:
  - Supports paging, sorting via `PagingAndSortingRepository`
  - Queries are generated based on the names of the methods added to the repository interface, e.g.

```
interface PersonRepository extends JpaRepository<Person, Long> {
   List<Person> findByLastname(String lastname);
   List<Person> findDistinctPeopleByLastnameOrFirstname(String
        lastname, String firstname);
}
```

  - Besides JPA, other technologies are supported as well:
    - JDBC, MongoDB, Neo4j, Redis, Couchbase, Solr, Elasticsearch
  - The repository methods can be published directly through RESTful interface (spring-data-rest)

Spring

Transaction management

Transactions in Spring

- Unified API for transaction management, a concrete transaction manager implementation can be configured to customize the behavior, e.g.
  - JDBC connection level
  - Use the EntityTransaction of JPA
  - Call a distributed transaction manager through JTA
  - Use the JDO API (an older alternative of JPA)

Configuration of the transaction manager

- Assuming a configured entityManagerFactory

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">

   <property name="entityManagerFactory"
ref="entityManagerFactory" />

</bean>
```

- Or

```
@Bean
public PlatformTransactionManager transactionManager(EntityManagerFactory emf){
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(emf);

    return transactionManager;
}
```

Types of transaction handling

- Programmatic:
  - Our code starts/ends the transaction through the API provided by Spring
  - Very rarely used
- Declarative:
  - Transaction start/end is controlled at method level, via annotations or XML config
  - We cannot manage code units smaller than a method
  - Has to be enabled in config:
    - <tx:annotation-driven>
    - Or @EnableTransactionManagement

Programmatic transaction management

```
@Autowired
PlatformTransactionManager txManager;
...
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Declarative transaction management

```
@Service
public class LogService {
        @PersistenceContext
        EntityManager em;

        @Transactional
        public void create(LogItem logItem) {
                em.persist(logItem);
        }
        …
}
```

- **@Transactional** can have parameters
- No transactions by default (unlike with EJB)

Parameters of @Transactional

- rollbackFor: which Exceptions should cause rollback (default: RuntimeException and its children)
  - Other options: noRollbackFor, rollbackForClassName, noRollbackForClassName
- timeout
- value: bean id of the transaction manager (needed when accessing multiple databases)
- propagation: what should happen when calling a transactional method from another transactional method (similar to EJB transaction attribute values)