SunilOS

Spring 5

www.sunilos.com

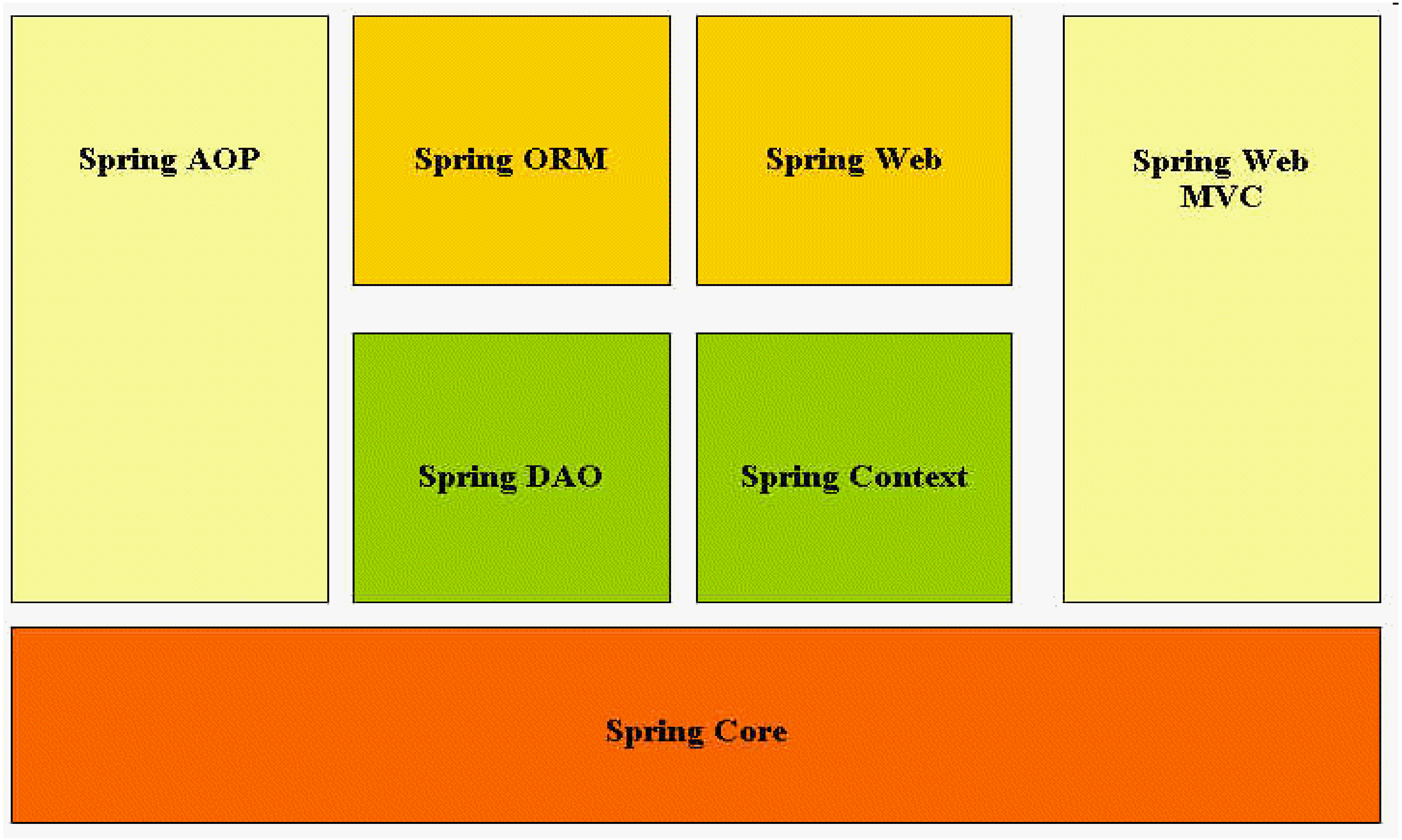
www.raystec.com

Think IT Think Us
RAYS

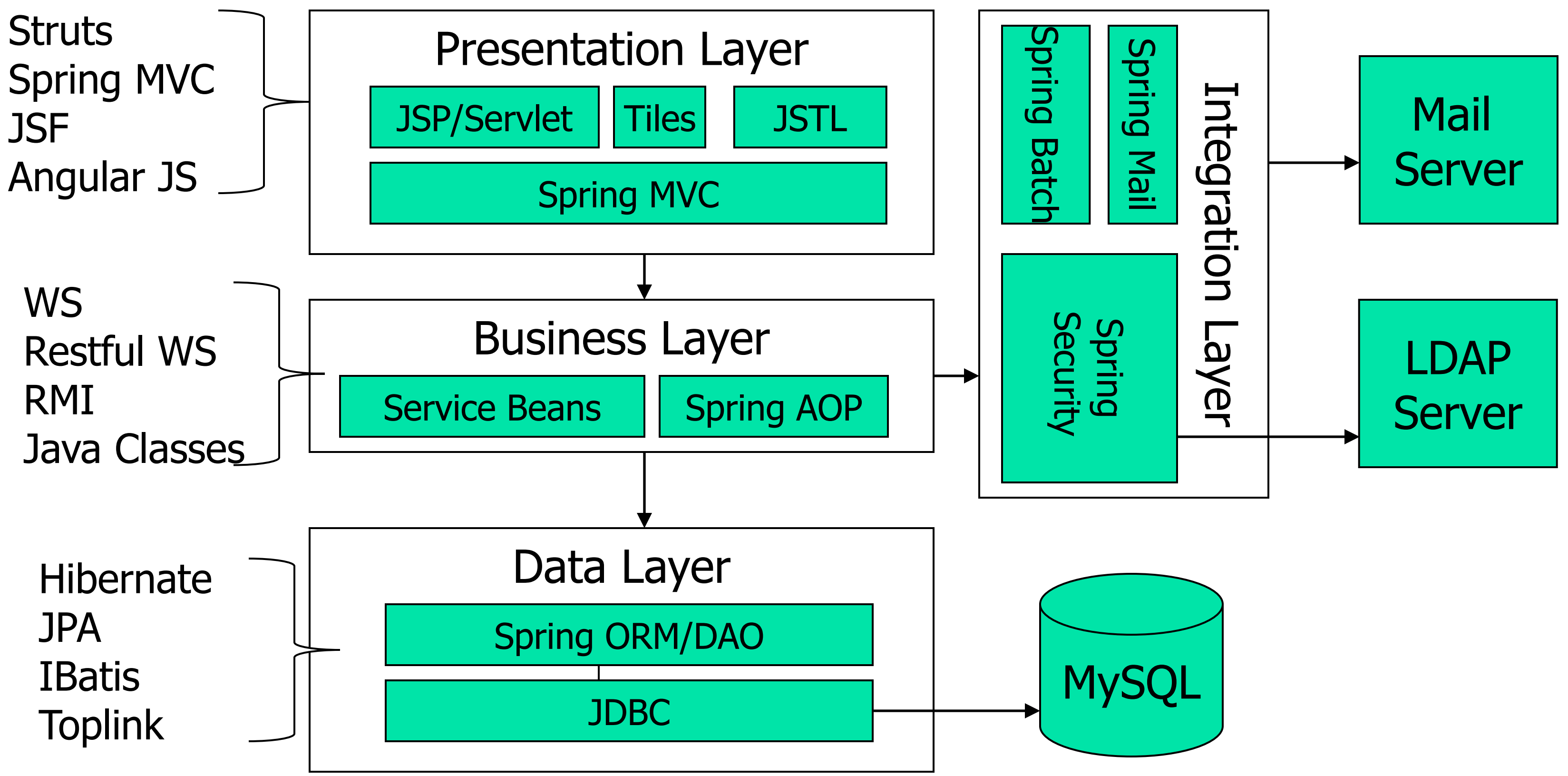
Agenda

- ❑Spring Container
- ❑Spring Core
- ❑Spring JDBC
- ❑Spring ORM
- ❑Spring AOP
- ❑Spring MVC
- ❑Spring Security
- ❑Spring Boot

Architecture



Application Architecture



More Application Layering Combinations

- Presentation/Business/Persistence
- Struts+Spring+Hibernate
- Struts + Spring + EJB
- Tapestry + Spring + EJB
- JavaServer Faces + Spring + iBATIS
- Spring + Spring + JDO
- Flex + Spring + Hibernate
- Struts + Spring + JDBC
- You decide…

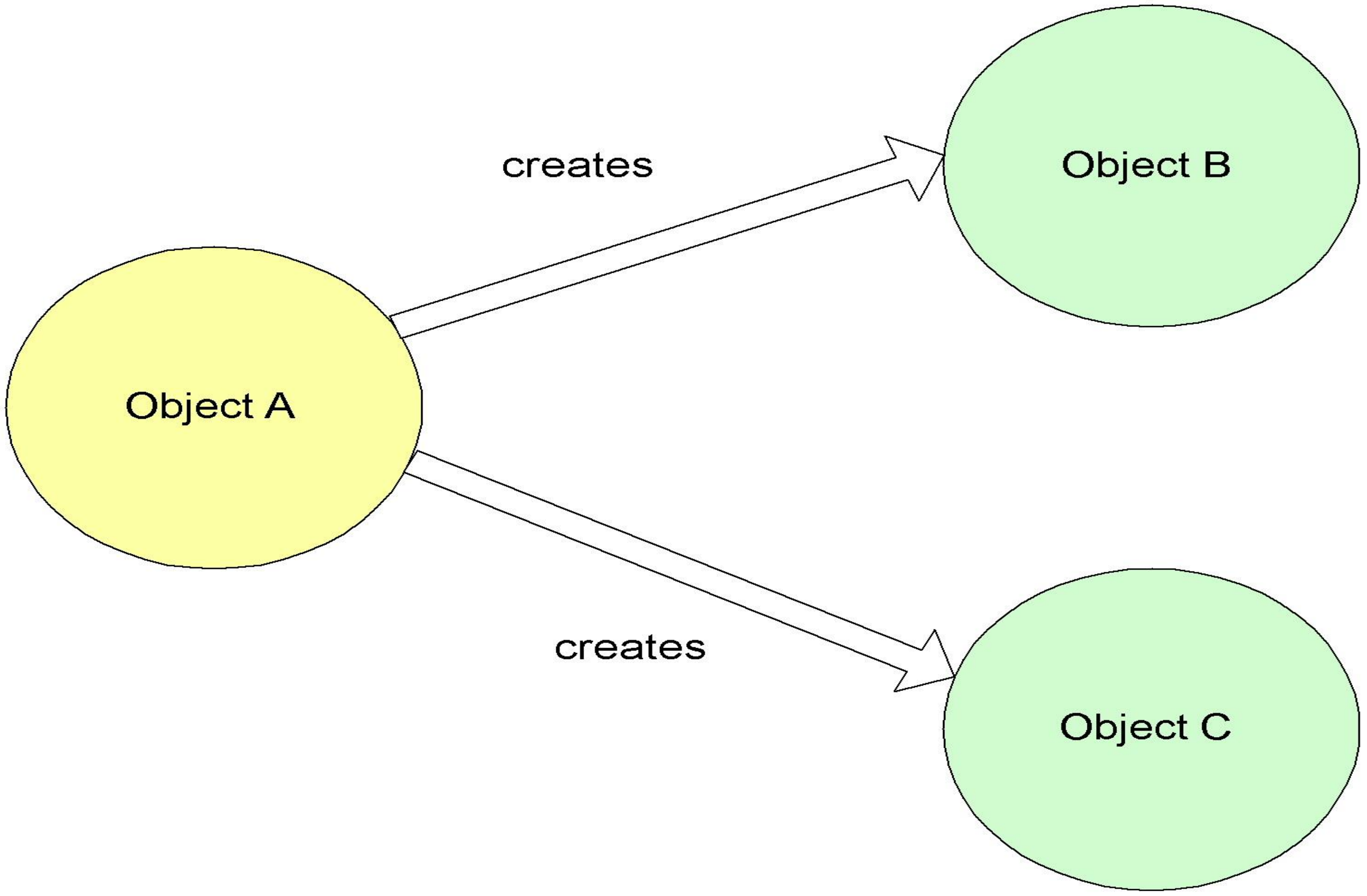
Introduction

- Light Weight Container
- Maintain the lifecycle of a bean
- Factory of all kinds of beans
- Inject dependencies

Inversion of Control (IOC)

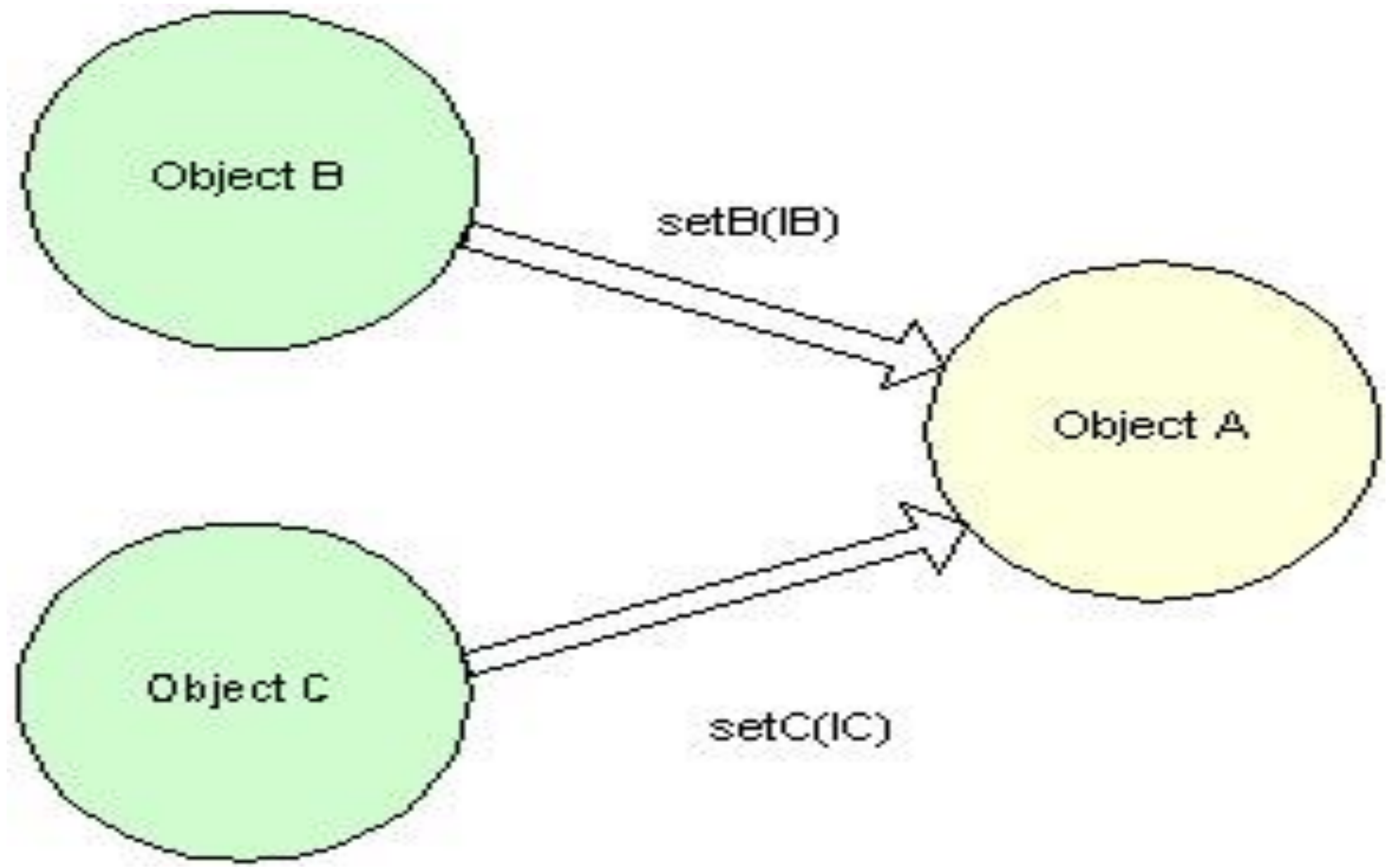
- ❑ Dependency injection
  - o Beans define their dependencies through constructor arguments or properties
  - o The container provides the injection at runtime
- ❑ Decouples object creators and locators from application logic
- ❑ Easy to maintain and reuse
- ❑ Testing is easier

Non-IoC / Dependency Injection



IoC / Dependency Injection

SunilOS



RMI
Email
Hibernate
DCP
POJO
Struts

Spring is a container

- Light Weight Container
- Maintain the lifecycle of a bean
- Factory of all kinds of beans

- **Shanta Claus of all beans**

SunilOS

Create the simplest Bean

```
public class User {
    private String name = null;
    private String login = null;
    private String password = null;

     public String getLogin() {
        return login;
    }
    public void setLogin(String login) {
        this.login = login;
    }
    //Other accessor methods
}
```

Configure Bean

- **<bean name="user" scope="prototype" class="com.sunilos.User">**
- <property name="login" value="sunilos" />
- <property name="password" value="pass" />
- </bean>
- It is configured in applicationContext.xml file.
- Here we configured bean and set its properties values.

- **@Component ("user")**
- **@scope("singleton")**
- public class User {

- <!--Scan @Component, @Repository, @Service,and @Controller spring beans -->
- <context:component-scan
- base-package="com.sunilos.ors" />

Get Bean instance

- // Create bean factory from appliationContext.xml
- BeanFactory factory = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));

- // get User bean
- User dto = (User) factory.getBean("user");
- //User dto = factory.getBean("user",User.class);

- //Print attributes
- System.out.println(dto.getLogin());
- System.out.println(dto.getPassword());

Bean Container Types

- There are two types of containers
- BeanFactory Container
  - It is light weight container.
- ApplicationContext Container
  - It adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.
  - Web application uses this container.
- Avoids the use of singletons and factories

BeanFactory Container

- This is light weight and basic container providing basic support for Dependency Injection (DI).
- It is implementation of interface org.springframework.beans.factory.BeanFactory.

- It is instantiated by
  - BeanFactory factory = new XmlBeanFactory (
  - new ClassPathResource("applicationContext.xml"));
  - User dto = (User) factory.getBean("user");
  - //User dto = factory.getBean("user",User.class);

ApplicationContext Container

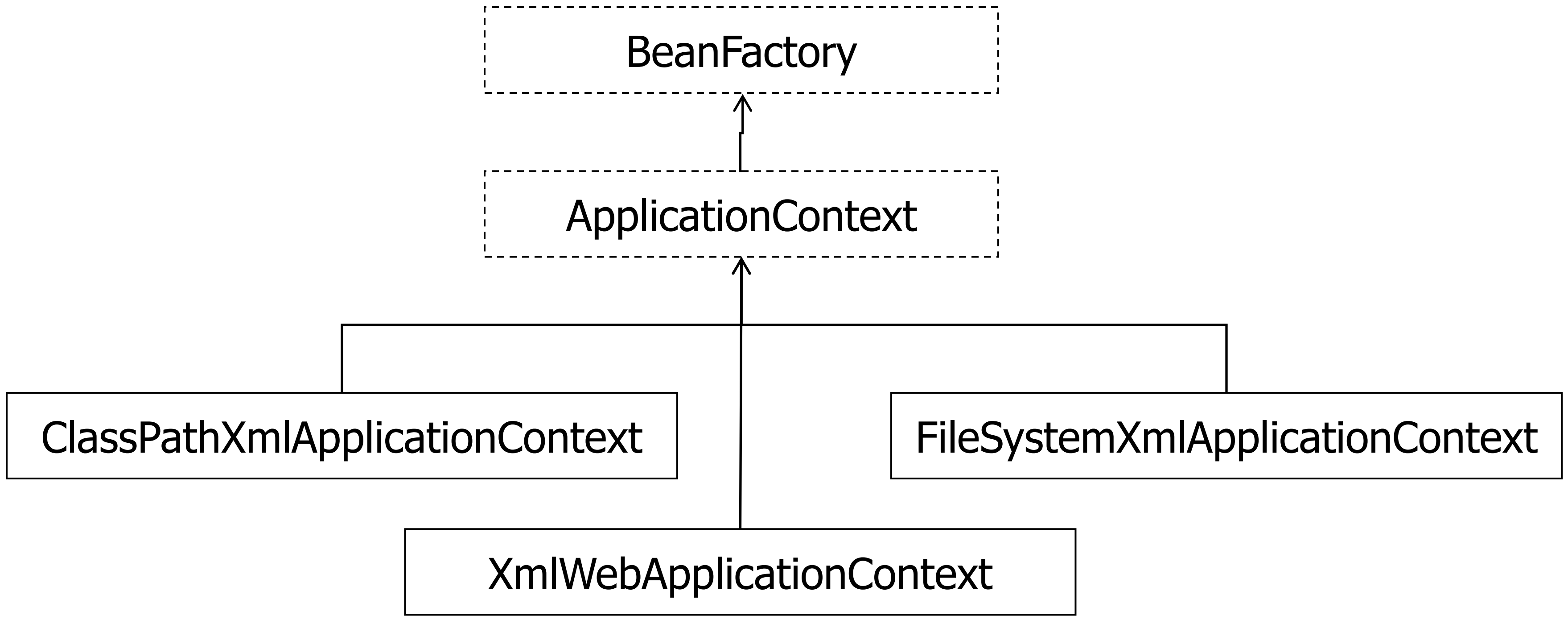
- It is child interface of `BeanFactory`. It is found in package `org.springframework.context.ApplicationContext`.
- The `ApplicationContext` container includes all functionality of the `BeanFactory` container, so it is generally recommended over the `BeanFactory`.
- It has easier integration with Spring AOP.
- It resolves textual messages from properties file.
- It has ability to publish application events to listed event listeners.
- It provides `WebApplicationContext` for web applications.

ApplicationContext Container

- Key implementation classes are
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext
- ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
- User dto = (User) context.getBean("user");

SunilOS

Context Hierarchy


BeanFactory
ApplicationContext
ClassPathXmlApplicationContext
FileSystemXmlApplicationContext
XmlWebApplicationContext

Load Multiple Files

- ApplicationContext context =
- new ClassPathXmlApplicationContext
- **(**
- **new String[] {“user-module.xml",“account-module.xml",“hr-module.xml"}**
- **);**

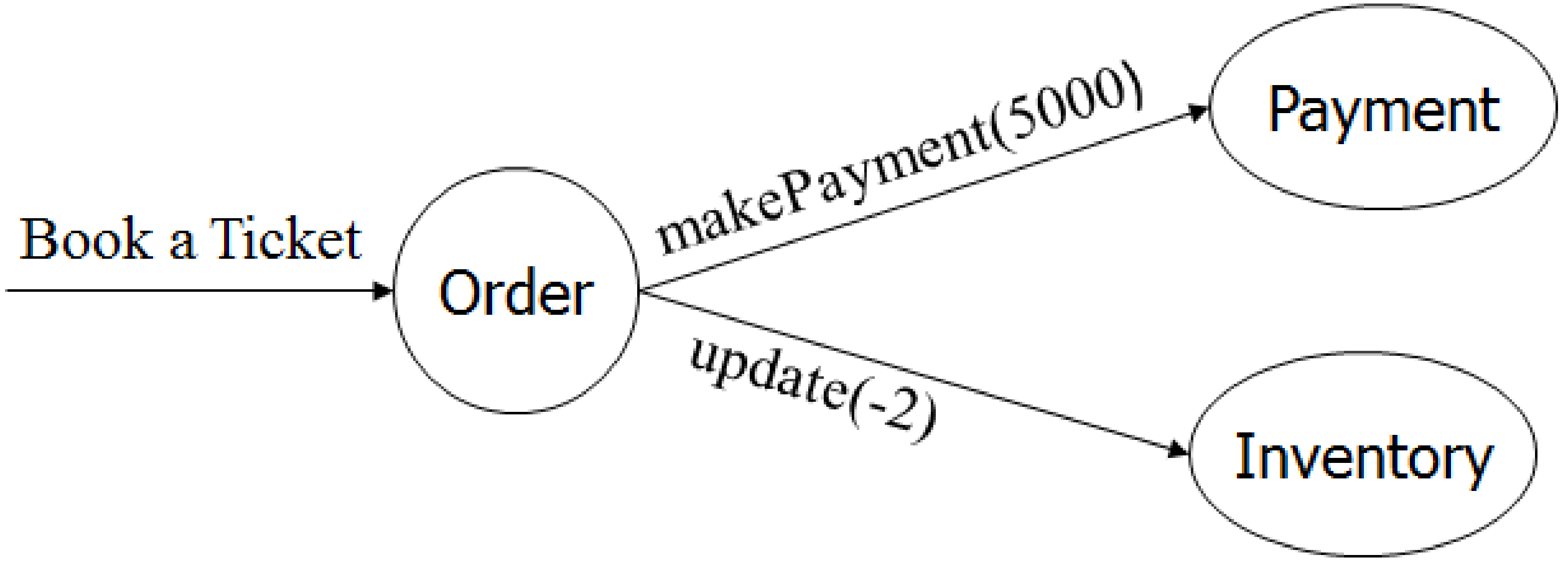
- **Or**
- <beans>
  - **<import resource=“user-module.xml"/>**
  - <import resource=“account-module.xml"/>
  - <import resource=“hr-module.xml"/>
- </beans>

Inversion of Control (IOC)

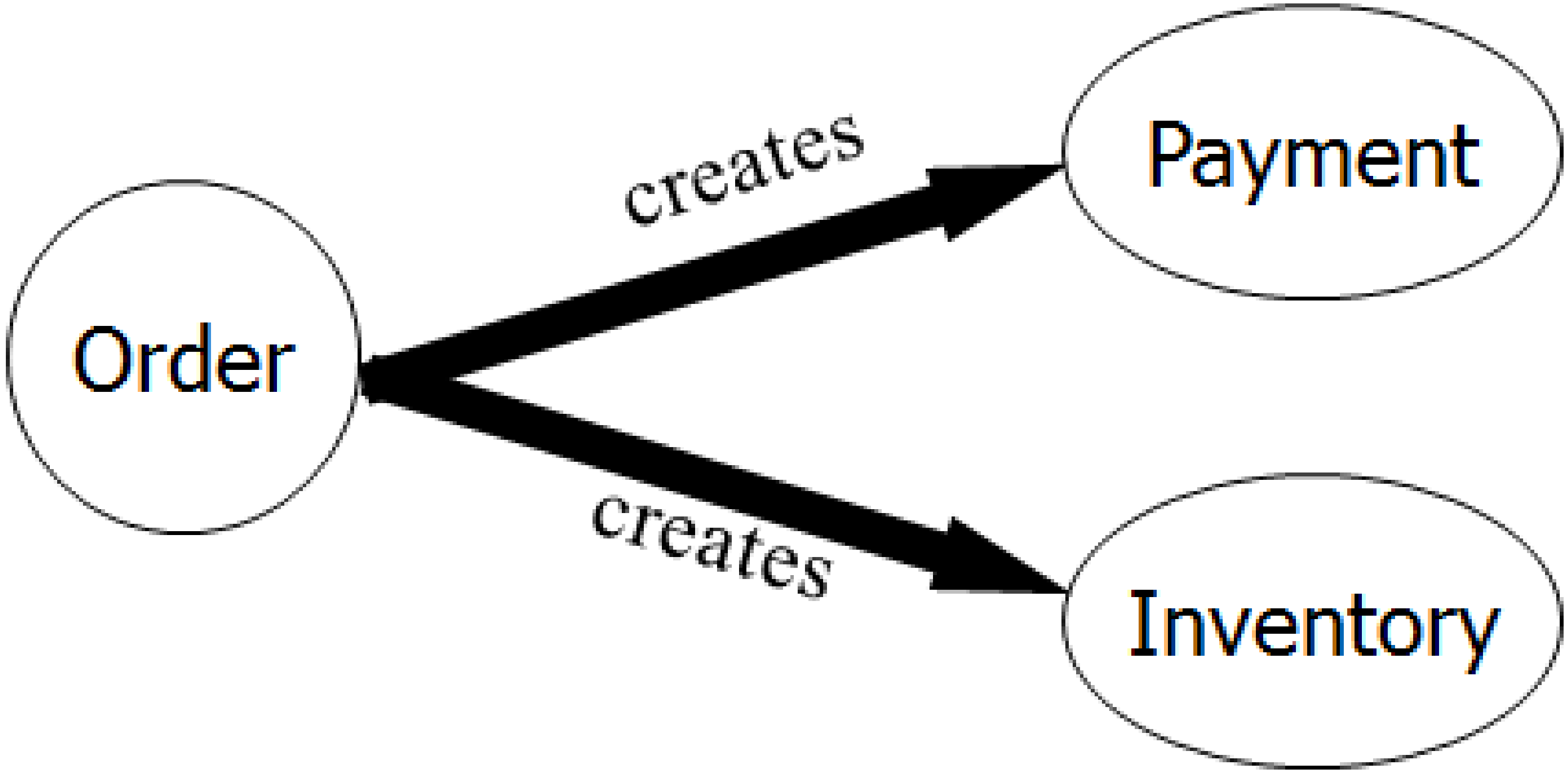
- An Application consists of multiple objects. Multiple objects collaborate to perform a business operation in an application thus one object is depend on other objects to perform a business operation, that is called dependency.
- IOC provides dependency injection (DI)
- Dependency Injection can be done by two ways:
  - Constructor, called constructor injection.
  - Setter method, called setter injection.
- The container provides the injection at runtime
- It decouples object creators and locators from application logic

SunilOS

Online Payment



Non-IoC / Dependency Injection

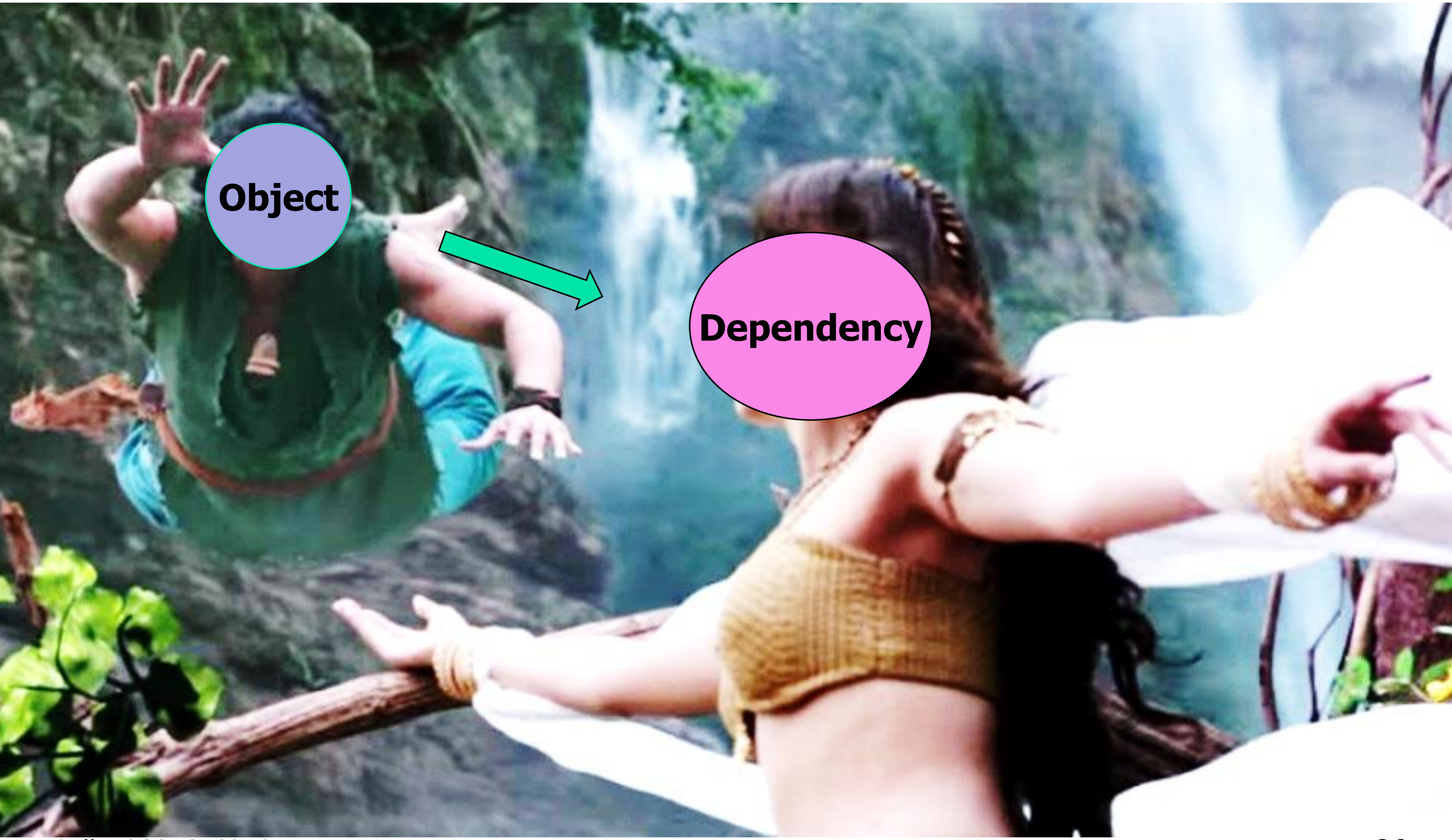


SunilOS

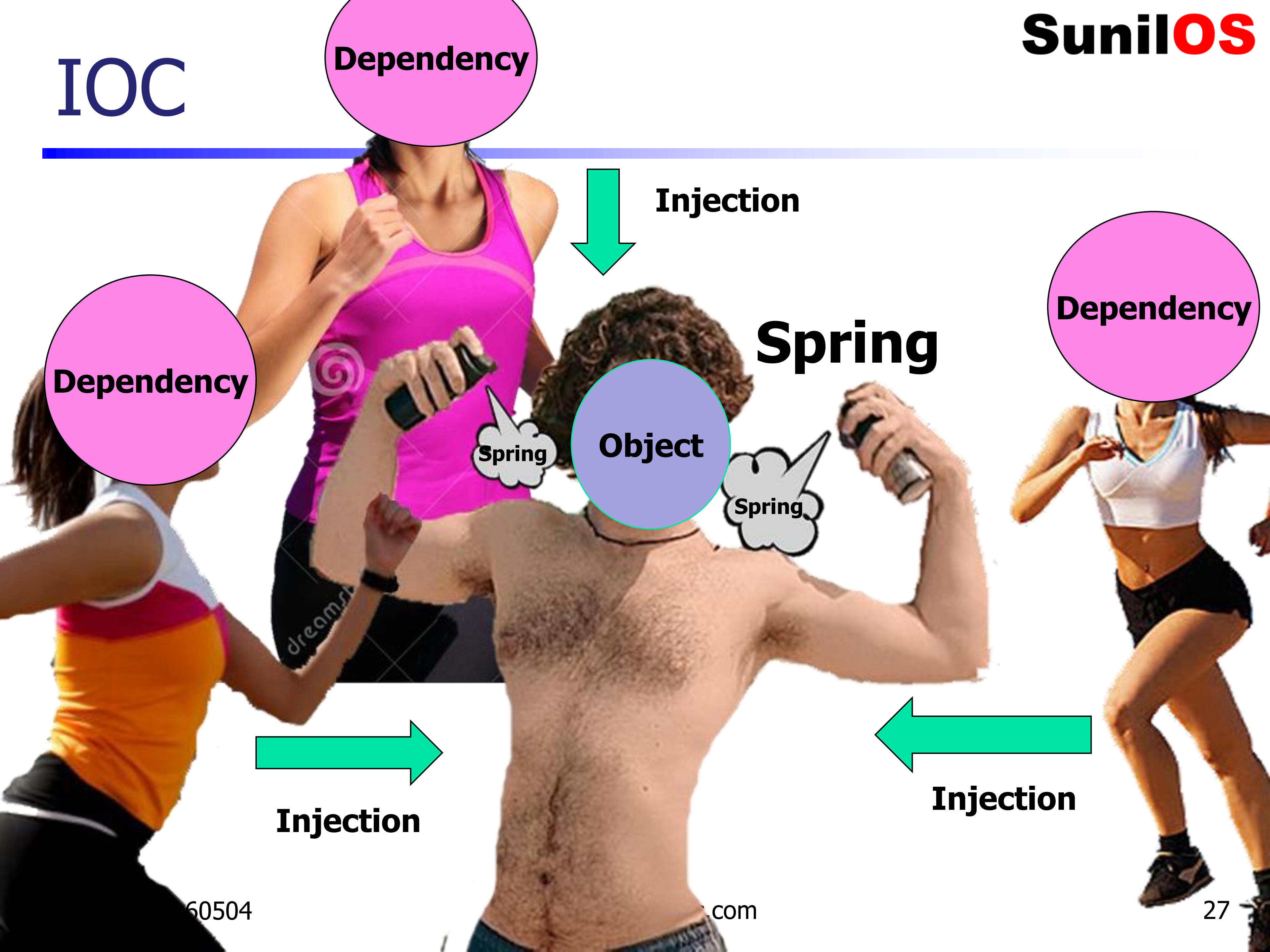
Non-IoC : Book A Ticket

```
public class Order {
  public void bookATicket(int item) {
    int price = 1000;

    //create dependent objects
    Payment p = new Payment();
    Inventory i = new Inventory();

    p.makePayment(items * price);
    i.update(item);
    System.out.println("Tickets are Booked");
  }
}
```

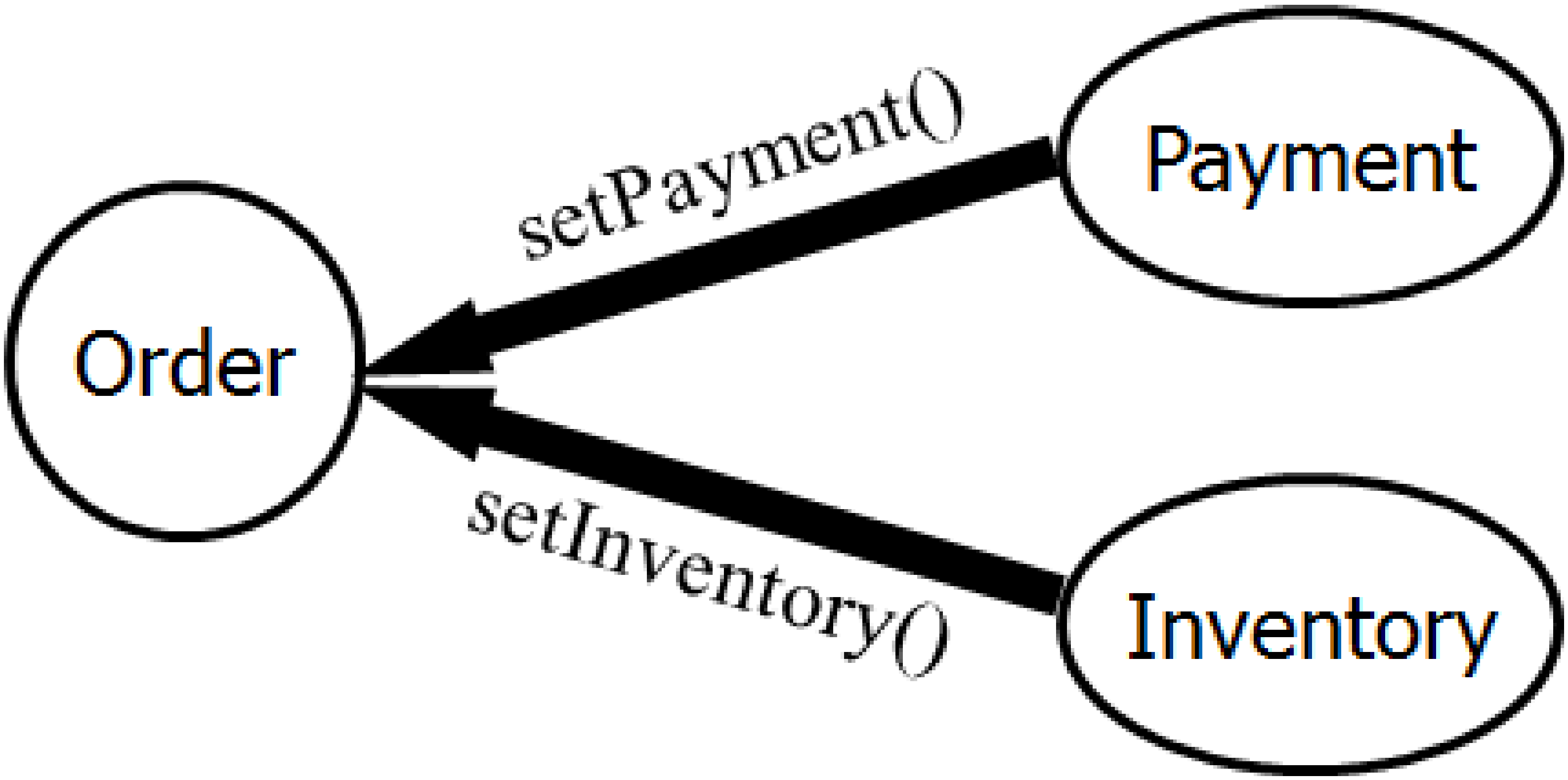
Non-IOC



IOC


Dependency
Injection
Dependency
Spring
Dependency
Spring
Object
Spring
Injection
Injection

IoC / Dependency Injection



SunilOS

IoC : Book A Tickets

- public class Order {
-   private Payment payment = null;
-   private Inventory inventory = null;
-   //Setter Injection
-   **public void setPayment(Payment p) { payment = p; }**
-   **public void setInventory(Inventory i) { inventory = i; }**

-   public void bookATicket(int items) {
-     int price = 1000;
-     **payment.makePayment(items * price);**
-     **inventory.update(items);**
-     System.out.println("Tickets are Booked");
-   }
- }

SunilOS

IoC : Book A Tickets (Cont.)

```
<bean name="payment" scope="prototype" class="Payment" />
<bean name="inventory" scope="prototype" class="Inventory" />
<bean name="order" scope="prototype" class="Order">
  <property name="payment" ref="payment" />
  <property name="inventory" ref="inventory" />
</bean>
```

```
@Component ("order")
public class Order {
   @Autowired (required=false)
   @Qulifier("payment")
  private Payment payment = null;
```

```
ApplicationContext context = …;
Order o = (Order) context.getBean("order");
o.bookATicket(2);
```

Constructor Injection

- public class Order {
- private Payment payment = null; private Inventory inventory = null;
- **public Order (Payment p, Inventory i){**
- **payment = p;**
- **inventory = i;**
- **}**
- //..
- }

- <bean name="payment" scope="prototype" class="Payment" />
- <bean name="inventory" scope="prototype" class="Inventory" />
- **<bean name="order" scope="prototype" class="Order">**
- **<constructor-arg ref=" payment " />**
- **<constructor-arg ref=" inventory" />**
- **</bean>**

SunilOS

Bean Life Cycle

Lifecycle methods

```
<bean id="order"  class="com.sunilos.Order"
init-method="init" destroy-method="cleanup"/>
public class Order{
    public void init() {
        // do some initialization work
    }
    public void cleanup() {
        // do some destruction work (like closing connection)
    }
}
```

- If you are using annotation **@Component** instead of **<bean>** tag then lifecycle methods can be defined by **@PostConstruct** and **@PreDestroy** annotations.

Static Method for instantiation

- public class ServiceLocator{ ...
- private ServiceLocator(...) { ... }
- public static ServiceLocator getInstance() {
  - ..

- <bean id="locator" class="com.sunilos.ServiceLocator"
- factory-method="getInstance" />

SunilOS

Set Collection data

```
<property name="someList">
  <list>
    <value>One</value>
    <value>Two</value>
   </list>
 </property>

 <property name="someMap">
  <map>
    <entry>
      <key><value>yup an entry</value></key>
      <value>just some string</value>
    </entry>
    <entry>
      <key><value>yup a ref</value></key>
      <ref bean="myDataSource"/>
    </entry>
  </map>
 </property>
```

SunilOS

Dependent beans

- <bean id="beanOne" class="ExampleBean" depends-on="manager">
- <property name="manager"><ref local="manager"/></property>
- </bean>

- <bean id="manager" class="ManagerBean"/>

SunilOS

Abstract and child bean definitions

- <bean id="inheritedTestBean" abstract="true"
-     class="org.springframework.beans.TestBean">
-   <property name="name" value="parent"/>
-   <property name="age" value="1"/>
- </bean>

- <bean id="chieldClass" class="org.springframework.beans.DerivedTestBean"
-     parent="inheritedTestBean" init-method="initialize">
-   <property name="name" value="override"/>
- </bean>

SunilOS

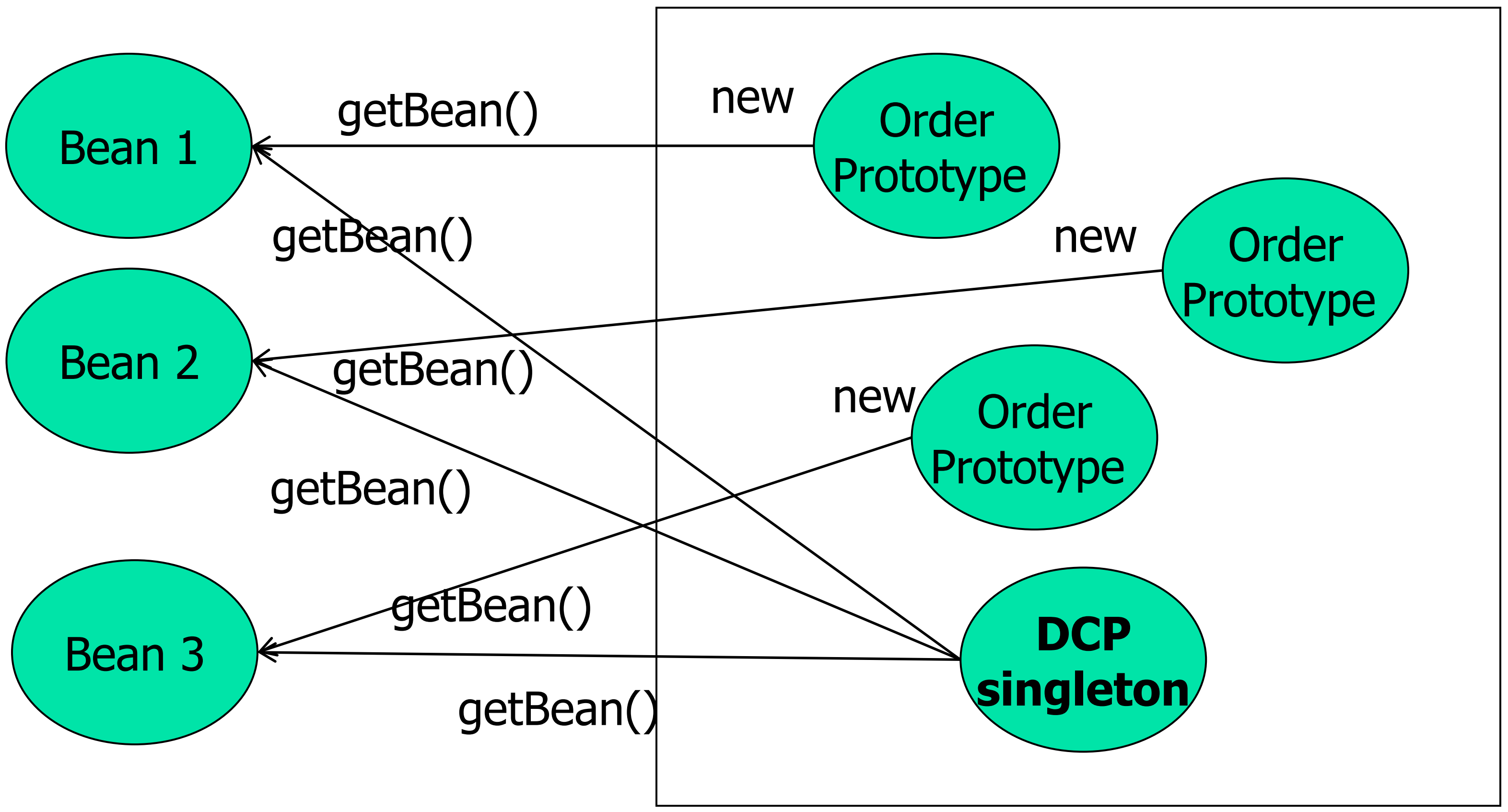
Scope of Beans

- There are five scope for a bean. Default is singleton

singleton	single object instance will be created for a Spring IoC container.
prototype	When you call getBean() a new bean instance is created.
request	Bean instance is created and kept in HttpRequest object of web container. This scope is valid when Spring is integrated with Web Application.
session	Bean instance is created and kept in HttpSession object of web container. This scope is valid when Spring is integrated with Web Application.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

SunilOS

Singlton Vs Prototype



Bean Auto wiring

- It enables implicit bean dependency injection.
- Bean reference injections will not be defined explicitly in XML file.
- Spring Auto Wiring can be done by two ways:
  - XML Configuration
    - <bean id="order" class=".." **autowire="default"** >
  - Annotations
    - **@Autowired (required = false)**
      public UserListCtl (UserService service) {}
  - For byName you can use @Qualifier ("payment") annotaton.

SunilOS

Auto wiring Modes

- There are 4 modes of auto wiring:
- autowire=”no”
  - auto-writing is disabled
- autowire=”byName”
  - Auto-wiring is resolved by property name. Setter method is used for DI.
- autowire=”byType”
  - Auto-wiring is resolved by property type.
- autowire=”constructor”
  - byType auto wiring is applied on constructor.

Excluding a bean from Auto wiring

- You can exclude a bean from being autowired.
- It can be done by setting autowire-candidate attribute of <bean> tag to false.

- <bean id="userService" scope="prototype" class="in.co.sunrays.service.UserService" **autowire-candidate="false"** />

SunilOS

Primary bean

- When we define two or more beans of the same type, and these beans are get auto wired by `byType` in some other bean then you can define primary bean.
- Primary bean will be candidate of auto wiring.
- We can create a primary bean by setting the primary attribute of <bean> tag to true.

- <bean id=“.." class=“.." **primary="true"** />

Auto wiring by Annotation

- You can use annotation `@Autowired` to auto wire a bean.
- Annotation can be applied on
  - attributes,
  - setter methods
  - and constructors.
  - By default it resolves dependency by Type.

- `@Autowired(required = false)`
- `@Qualifier("userService")`
- `private UserService service = null;`

SunilOS

Auto wiring annotations

- Spring 3.x provides three annotations for autowiring:
  - o Spring's own `@Autowired` annotation.
  - o The `@Inject` annotation (JSR-330).
  - o The `@Resource` annotation (JSR-250).
- Annotation based configuration is enabled by `<context:annotation-config />` tag in `applicationContext.xml` file.

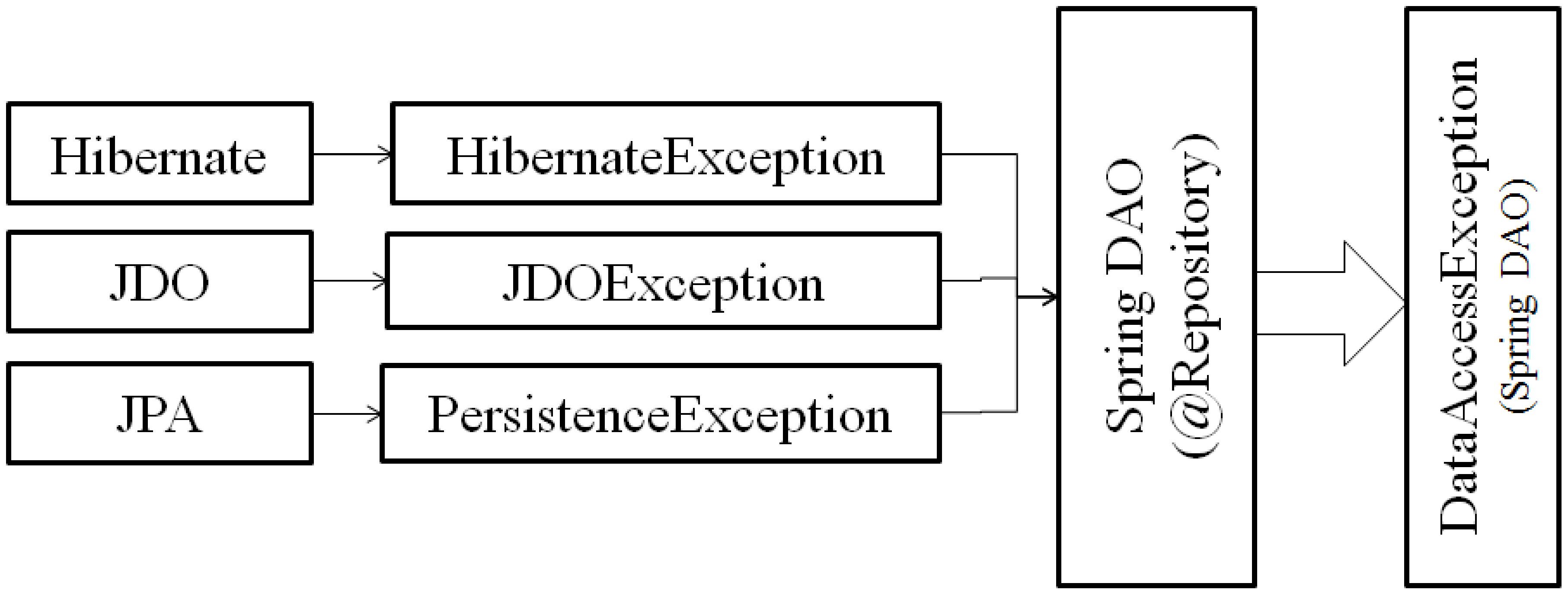
Spring DAO (Data Access Object)

- It provides a consistent way to use data access frameworks like JDBC, Hibernate, JPA, JDO etc.
- It provides consistent exception hierarchy. This makes your application independent from underlying data access framework.
- DAOs are defined with help of `@Repository` annotation.
- A DAO requires access to a Persistent Resource (Data Connection Pool) to get connection to the Database

Consistent Exception Hierarchy

- JDBC raises SQLException, Hibernate raises HibernateException, and JDO raises JDOException.
- Spring-DAO translates ORM specific exceptions into DataAccessException or into its subclasses
- Spring enables transparent exception translation using @Repository annotation.
- It makes you free from handling different exceptions when you are working on different persistent frameworks

ORM Exception translation



- DataAccessException is unchecked exception class.

@Repository Annotation

- Any POJO can be configured to Spring DAO using @Repository annotation.

- @Repository
- public class CollegeDAOJDBCImpl implements CollegeDAOInt {
-     private JdbcTemplate jt;
-     ..
-     @Autowired
-     public void setDataSource(DataSource dataSource) {
-       this.jdbcTemplate = new JdbcTemplate(dataSource);
-   }
- Annotated classes are scanned by following tag:
- <context:component-scan base-package="com.sunilos.dao" />

SunilOS

Data Connection Pool

- A DAO requires DCP to get connection from Database. Different ORMs need different Persistent Resources like JDBC need `DataSource`, JPA need `EntityManager`, and Hibernate need `SessionFactory`.
- @Repository
- public class CollegeDAOHibImpl implements CollegeDAOInt {
-     **@Autowired**
-     **private SessionFactory sf;**

- @Repository
- public class JPACollegeDAOImpl implements CollegeDAOInt {
-     **@PersistenceContext**
-     **private EntityManager entityManager;**
- }

SunilOS

Data Source

- <bean id="**dataSource**“
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/ST_JAVA" />
    <property name="username" value="root" />
    <property name="password" value="" />
  </bean>

Session Factory

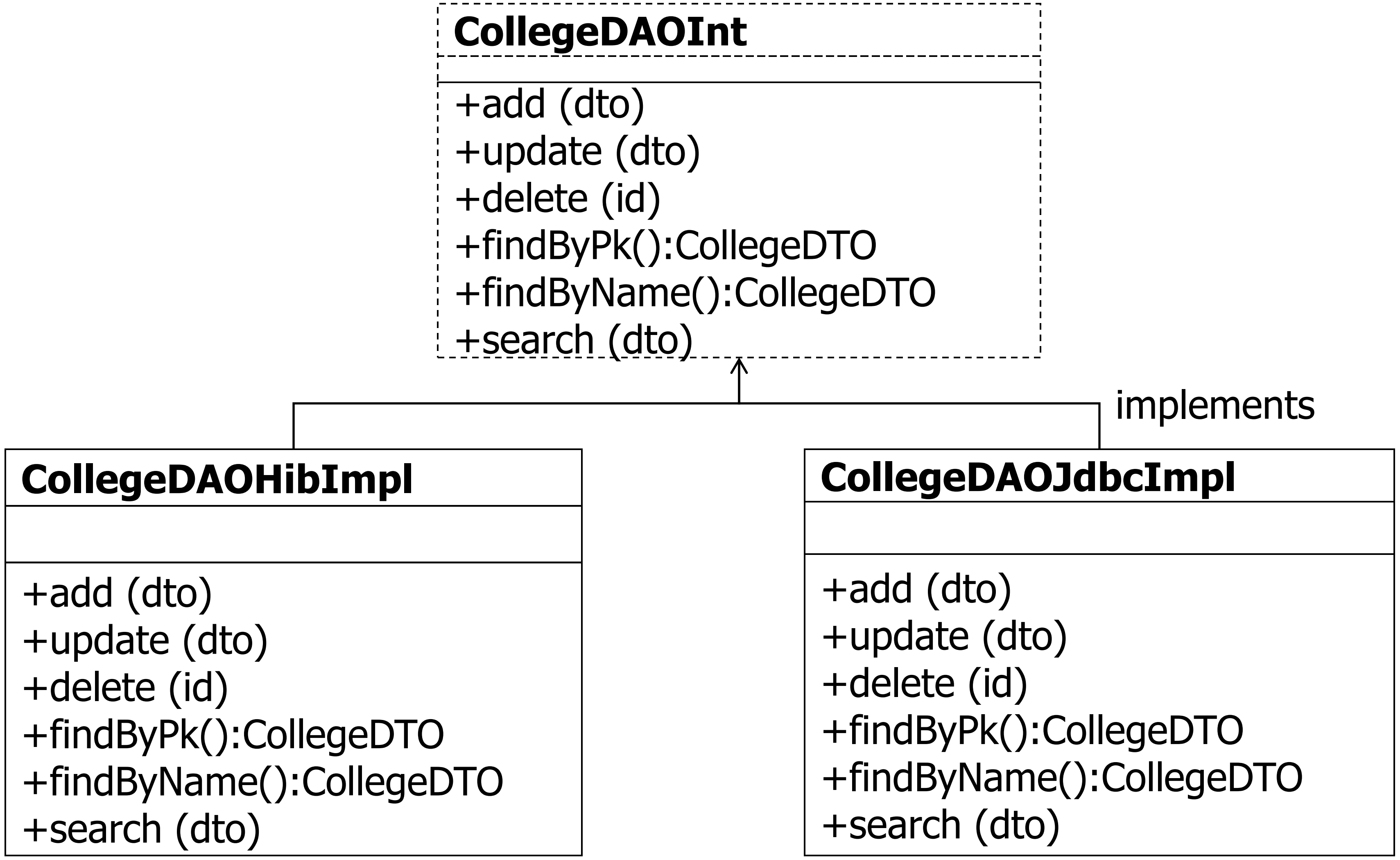
- <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
- **<property name="dataSource" ref="dataSource" />**
- <property name="hibernateProperties">
- <props>
- <prop key="hibernate.dialect">
- org.hibernate.dialect.MySQLDialect
- </prop>
- </props>
- </property>
- <property name="mappingResources">
- <list>
- <value>com/sunilos/dto/Account.hbm.xml</value>
- </list>
- </property>
- </bean>

Spring JDBC

- Spring JDBC provides an abstract way to communicate with database using plan old JDBC objects.
- It takes care of all the low-level details to open the connection, prepare and execute the SQL statements, process exceptions, handle transactions and finally close the connection.
- It provides several approaches and correspondingly classes to communicate with database.
- The most popular approach is to use **JdbcTemplate** class to communicate with database.
- JdbcTemplate is the central framework class that manages all the database communication and exception handling.

SunilOS

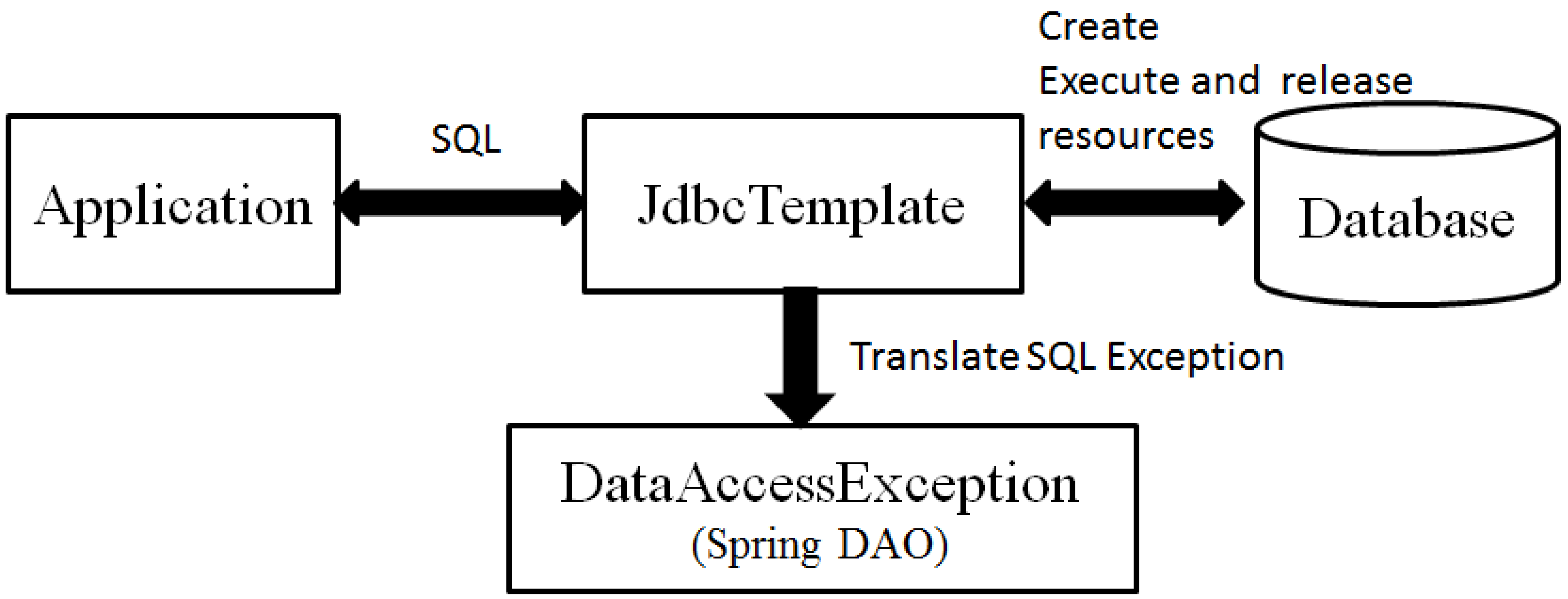
College DAO



SunilOS

JDBC Template

- It is thread-safe.
- It handles the creation and release of resources.
- It executes SQL statements
- It translates SQLException into DataAccessException



SunilOS

CollegeDAOJdbcImpl

- @Repository
- public class CollegeDAOJDBCImpl implements CollegeDAOInt {
-
- private JdbcTemplate jdbcTemplate;
-
- **@Autowired**
- public void setDataSource(DataSource dataSource) {
- **this.jdbcTemplate = new JdbcTemplate(dataSource);**
- }

SunilOS

Insert/Update/Delete

- String sql = "INSERT INTO ST_COLLEGE VALUES(?,?,?)";
- jdbcTemplate.**update**(sql, 1, "IPS", "Rau");

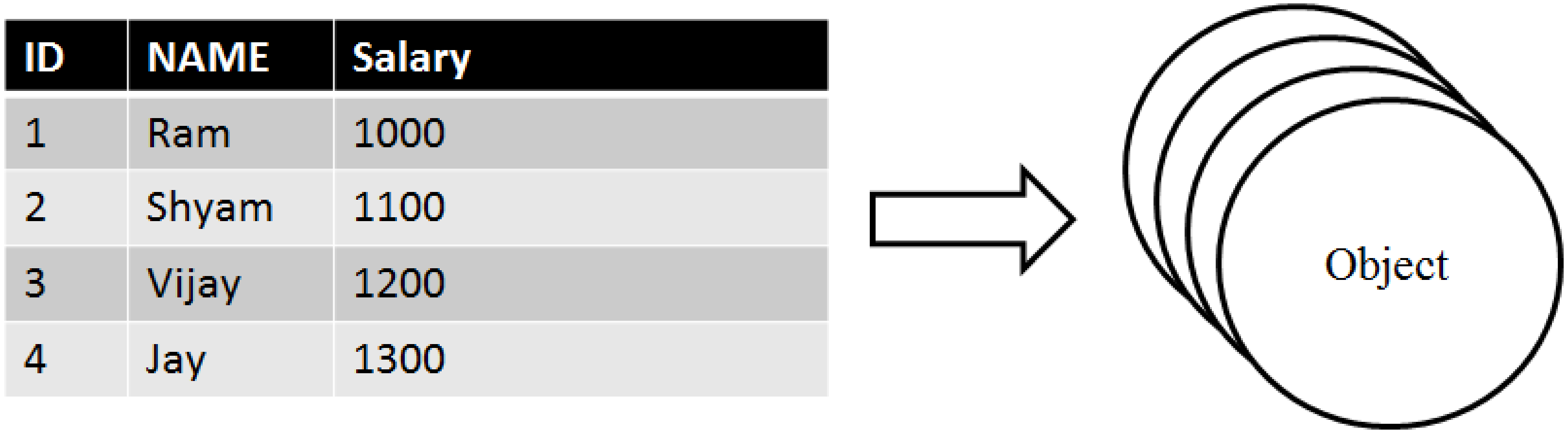
- String sql = "UPDATE ST_COLLEGE SET NAME=?,ADDRESS=? WHERE ID=?";
- jdbcTemplate.**update**(sql, "Medicaps", "Rau",1);

- String sql = "delete from ST_COLLEGE where id = ?";
- int i = jdbcTemplate.**update**(sql, 1);

- //Stored Procedure
- jdbcTemplate.update("call USER_COUNT(?)", 25L);

Row Mapper class

- It is an interface.
- JdbcTemplate uses this to map a row of ResultSet to an Object.
- It is stateless thus reusable.
- It can be used with OUT parameters of Stored Procedure and Function.



ID	NAME	Salary
1	Ram	1000
2	Shyam	1100
3	Vijay	1200
4	Jay	1300

Map ResultSet rows to Objects

CollegeMapper class

SunilOS

- public class CollegeMapper implements **RowMapper<College>** {
-     public College mapRow(ResultSet rs, int args) throws SQLException {
-         College dto = new College();
-         dto.setId(rs.getLong("id"));
-         dto.setName(rs.getString("name"));
-         return dto;
-     }
- }
- public College findByPK(long pk) {
-     String sql = "SELECT * FROM ST_COLLEGE WHERE ID=?";
-     **College dto = jdbcTemplate.queryForObject (sql,**
-         **new Object[] { pk }, new CollegeMapper());**
-     return dto;
- }

Execute count queries

- int rowCount = jdbcTemplate.queryForObject(
- "SELECT MAX(ID) FROM ST_COLLEGE", **Integer.class**);

- int rowCount = jdbcTemplate.queryForObject(
- "select count(*) from PART where UNIT_ID = **?**", **Integer.class**, 2);

Spring ORM

- It integrates ORM frameworks Hibernate, JPA, and JDO for data access object, transaction and resource management implementation.
- It converts ORM exceptions to DataAccessException.
- It provides declarative transactions with help of Spring AOP.

Hibernate Integration

- Hibernate objects can be defined in Spring container as a bean and injected in DAO objects.
- **@Repository**
- public class CollegeDAOHibImpl implements CollegeDAOInt {
- **@Autowired**
- private SessionFactory sessionFactory;
- public long add(College dto) {
- **Session session = sessionFactory.getCurrentSession();**
- session.save(dto);
- return dto.getId();
- }

Hibernate 3 API

- Hibernate 3 has a feature called contextual sessions.
- Hibernate itself manages one current Session per transaction in contextual sessions.
- Contextual Session is used by DAO for database manipulation.
- This approach is recommended to develop DAO classes in Hibernate.

SunilOS

Hibernate 3 API approch

- **@Repository**
- public class CollegeDAOHibImpl implements CollegeDAOInt {
- **@Autowired**
- private SessionFactory sessionFactory;
- public long add(College dto) {
- **Session session = sessionFactory.getCurrentSession();**
- session.save(dto);
- return dto.getId();
- }
- Always get current session to manipulate database.

Service class

- It contains business logic.
- Defined by @Service annotation.
- It does transaction handling.
- Transactions are handled by Spring AOP.
- Transaction handling is called declarative transaction handling.
- Two ways to apply transactions in Service classes:
  - XML Configuration
  - @Transactional annotation

SunilOS

Define Service class

- **@Service(value = "collegeService")**
- public class CollegeServiceImpl implements CollegeServiceInt {

- **@Autowired**
- private CollegeDAOInt dao = null;

- **@Transactional(readOnly = true)**
- public College get(long id) {
-     return dao.findByPK(id);
- }

- **@Transactional(propagation = Propagation.REQUIRED)**
- public long add(College dto) {
-     return dao.add(dto);
- }

SunilOS

Transaction Attribute

Attribute	@Transactional OrderService .bookATicket()	@Transactional PaymentService .makePayment()
Required	OrderTx	OrderTx
	NULL	PaymentTx
Required New	OrderTx	PaymentTx
	NULL	PaymentTx
Supported	OrderTx	OrderTx
	NULL	NULL
Not Supported	OrderTx	NULL
	NULL	NULL
Mandatory	OrderTx	OrderTx
	NULL	Exception
Never	NULL	NULL
	OrderTx	Exception

SunilOS

Spring AOP

- Aspects enable modularization such as transaction management that works across multiple types and objects

AOP key elements

- **Aspect:**
- A modularization operation that cuts across multiple objects.
- Transaction management is a good example of a crosscutting concern in J2EE applications.
- In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (@AspectJ style).

AOP key elements (Contd.)

- **Join point**
  - A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.
- **Advice**
  - It is the action taken by an aspect at a particular join point.
  - different types of advices include "around," "before" and "after" advice.
  - An advice as an interceptor, maintaining a chain of interceptors "around" the join point.
- **Pointcut**
  - An expression that identifies join points.
  - Advice is associated with a pointcut expression, that runs on matching joining points.

AOP

Advice

Join Points

SunilOS

AOP Xml Configuration

- <bean id="**transactionManager**"
-     class="org.springframework.orm.hibernate3.HibernateTransactionManager">
-     <property name="sessionFactory" ref="sessionFactory" />
- </bean>

- <tx:advice id="**txAdvice**" transaction-manager="**transactionManager**">
-     <tx:attributes>
-         **<tx:method name="find*" read-only="true" />**
-         **<tx:method name="search*" read-only="true" />**
-         **<tx:method name="*" propagation="REQUIRED" />**
-     </tx:attributes>
- </tx:advice>

AOP Xml Configuration (Cont.)

- <aop:config>
-    <aop:pointcut id="serviceOperations"
-       expression="execution(* **com.sunilos.service.*ServiceImpl.*(..))**" />

-    <aop:advisor advice-ref="txAdvice"
-          pointcut-ref="serviceOperations" />
- </aop:config>

- Enable Annotation
- <tx:annotation-driven transaction-manager=*"transactionManager"* />

SunilOS

Email Bean

- <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">

-     <property name="host" value="smtp.gmail.com" />
-     <property name="port" value="587" />
-     <property name="protocol" value="smtp" />
-     <property name="username" value="yourmail@gmail.com" />
-     <property name="password" value="pass1234" />

-     <property name="javaMailProperties">
-         <props>
-             <prop key="mail.smtp.auth">true</prop>
-             <prop key="mail.smtp.starttls.enable">true</prop>
-             <prop key="mail.smtp.debug">false</prop>
-         </props>
-     </property>
- </bean>

Inject Email bean

- @Service(value = "userService")
- public class UserServiceImpl implements UserServiceInt {
-     @Autowired
-     **private JavaMailSenderImpl mailSender;**
-
-     //..

SunilOS

Send Email

- MimeMessage msg = mailSender.createMimeMessage();

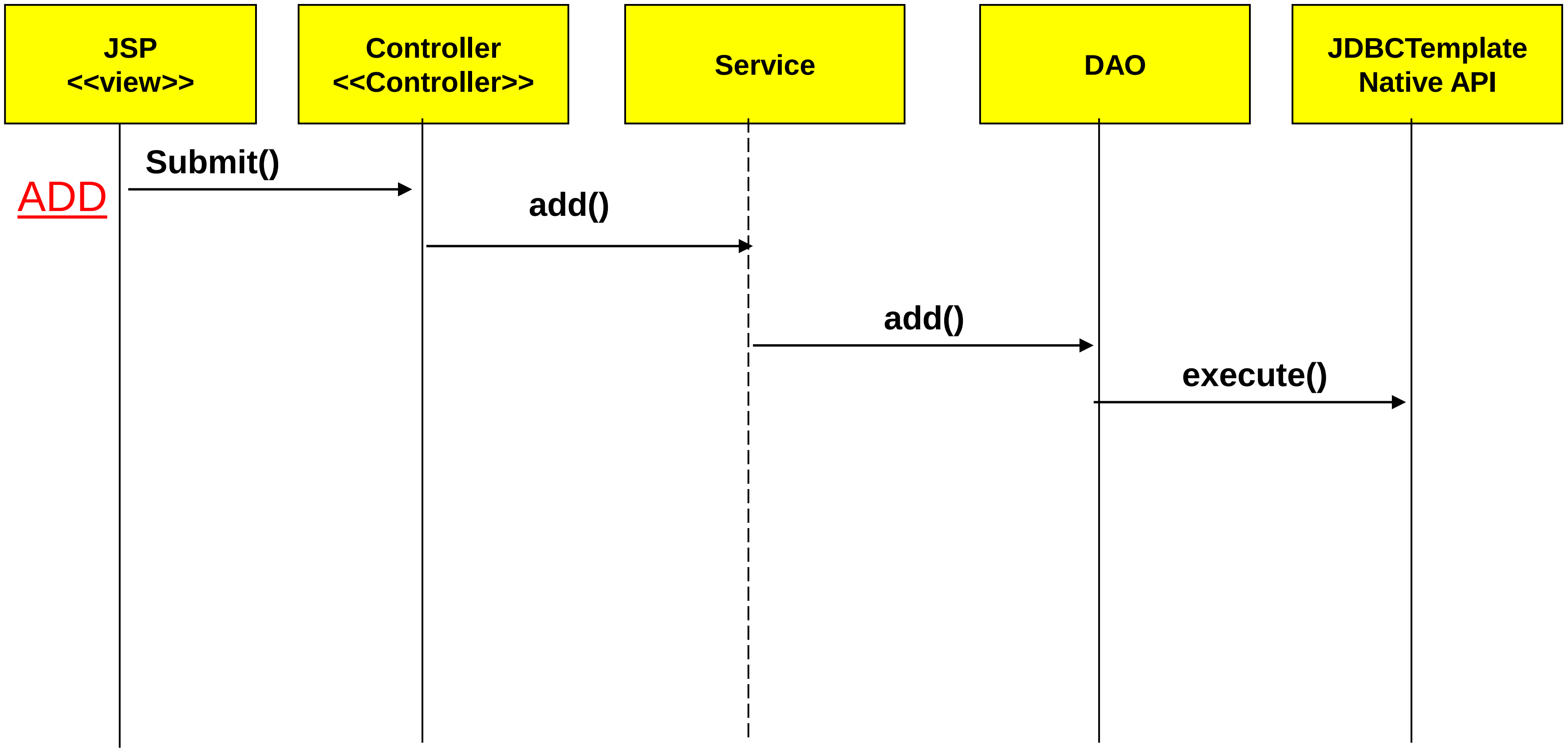
- // use the true flag to indicate you need a multipart message
- MimeMessageHelper helper;

- try {
-     helper = new MimeMessageHelper(msg, true);
-     helper.setTo("sender@gmail.com"));
-     helper.setSubject("Test Subject");
-     // use the true flag to indicate the text included is HTML
-     helper.setText("Test Msg", true);
- } catch (MessagingException e) {
-     e.printStackTrace();
- }
- mailSender.send(msg);

More Application Layering Combinations

- Presentation/Business/Persistence
- Struts+Spring+Hibernate
- Struts + Spring + EJB
- Tapestry + Spring + EJB
- JavaServer Faces + Spring + iBATIS
- Spring + Spring + JDO
- Flex + Spring + Hibernate
- Struts + Spring + JDBC
- You decide…

Low Level Sequence Diagram



SunilOS

Other View Framework Integration

- Velocity and FreeMarker
  - Template Framework
- JSTL -JavaServer Pages Standard Tag Library
- Tiles
- XSLT - is a transformation language for XML
- iText - Generate PDF
- POI – Generate Excel
- Jasper reports - iReport Editor/ Can convert report in HTML/PDF/Excel/DOc

Scheduling and Thread Pooling

- Timer part of the JDK since 1.3, and the Quartz Scheduler
- Quartz uses Trigger, Job and JobDetail
  - Trigger contains time
  - Job contains operations
  - JobDetails contains parameters