Spring Framework

Introduction

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- Spring handles the infrastructure so you can focus on your application.

Coupling

- Tight coupling
  - Tight coupling means the two classes often change together.
  - In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.
- Loose coupling
  - Classes are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled.

3

```
class Subject {
   Topic t = new Topic();
   public void startReading()
   {
      t.understand();
   }
}
class Topic {
   public void understand()
   {
      System.out.println("Tight coupling concept");
   }
}
```

4

IOC

- Heart of the Spring framework
- Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework.
- Giving control to the container to get an instance of the object is called Inversion of Control.
- Instead of you are creating an object using the new operator, let the container do that for you.

5

The Spring IoC Container

- In the Spring framework, the IoC container is represented by the interface ApplicationContext.
- The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle.
- Implementations
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - WebApplicationContext

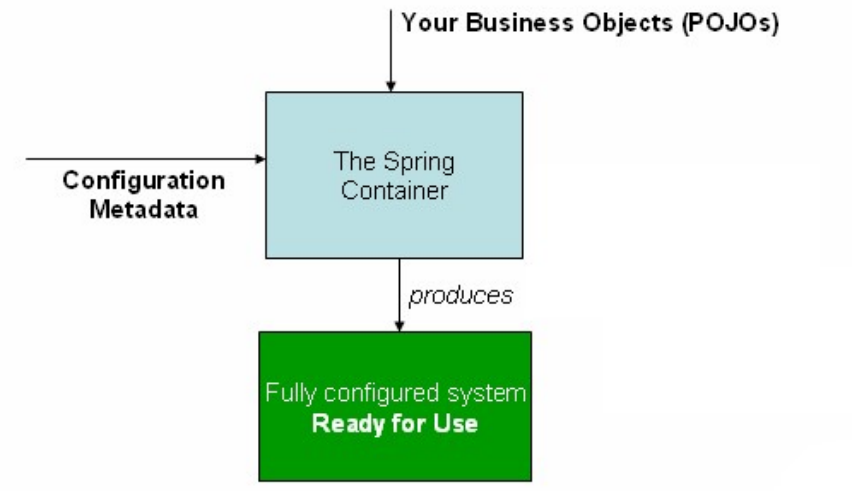
6

DI

- Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.
- Dependency injection (DI) is a process whereby objects define their dependencies.
  - Constructor Based
  - Setter Based

7

Container



8

Spring Beans

- The objects that form the backbone of your application and that are managed by the Spring IoC* container are called beans.
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML definitions.

9

Configuration Metadata


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

10

Using Container

```
ApplicationContext context =
            new ClassPathXmlApplicationContext("services.xml");

PetStoreService service = context.getBean("petStore",);

List<String> userList = service.getUsernameList();
```


HelloWorld.java

```
public class HelloWorld {
   private String message;
   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

beans.xml


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <bean id="helloWorld" class="com.HelloWorld">
     <property name="message" value="Hello World!"/>
  </bean>
</beans>
```

13

HelloWorldTest.java


```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class HelloWorldTest {
  public static void main(String[] args) {
     ApplicationContext context =
          new ClassPathXmlApplicationContext("Beans.xml");

     HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
     obj.getMessage();
  }
}
```

14

Dependency Injection - Constructor


```
public class Foo {
  public Foo(Bar bar, Baz baz) {
     // ...
  }
}
<beans>
  <bean id="foo" class="x.y.Foo">
     <constructor-arg ref="bar"/>
     <constructor-arg ref="baz"/>
  </bean>

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

</beans>
```


Dependency Injection - Setter


```
<bean id="exampleBean" class="examples.ExampleBean">
<property name="beanOne">
<ref bean="anotherExampleBean"/></property>
<property name="beanTwo" ref="yetAnotherBean"/>
<property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) {
     this.beanOne = beanOne;
  }
  public void setBeanTwo(YetAnotherBean beanTwo) {
     this.beanTwo = beanTwo;
  }
  public void setIntegerProperty(int i) {
     this.i = i;
  }
}
```

Collections

- Props
- List
- Set
- Map

Collection - Properties


```
<property name="adminEmails">
      <props>
         <prop key="administrator">administrator@example.org</prop>
         <prop key="support">support@example.org</prop>
         <prop key="development">development@example.org</prop>
      </props>
   </property>
```

19

Collection - List


```
 <property name = "addressList">
       <list>
         <value>INDIA</value>
         <value>USA</value>
        </list>
</property>
```

20

Collection - Map

```
<beans>
  <bean id="foo" class="x.y.Foo">
    <property name="accounts">
      <map>
        <entry key="one" value="9.99"/>
        <entry key="two" value="2.75"/>
        <entry key="six" value="3.99"/>
      </map>
    </property>
  </bean>
</beans>
```

```
public class Foo {
private Map<String, Float> accounts;
public void setAccounts(Map<String, Float> accounts) {
      this.accounts = accounts;
   }
}
```

21

Collection - Set

```
<property name = "addressSet">
      <set>
        <value>INDIA</value>
         <value>USA</value>
      </set>
 </property>
```

22

Bean Lifecycle

1. The framework factory loads the bean definitions and creates the bean.
2. The bean is then populated with the properties
3. If your bean implements any of Spring's interfaces, such as BeanNameAware or BeanFactoryAware, appropriate methods will be called.
4. The framework also invokes any BeanPostProcessor's associated with your bean for pre-initialization.
5. The init-method, if specified, is invoked on the bean.
6. The post-initialization will be performed if specified on the bean


```
public class SimpleCar implements  BeanNameAware, BeanFactoryAware {

public String describe() {
      return "Car is an empty car";
   }
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
      System.out.println("received the beanFactory " + beanFactory);

   }
public void setBeanName(String name) {
      System.out.println("the name of the bean is " + name);

   }
}
```

```
public class BeanPostProcessorTest implements BeanPostProcessor {
public Object postProcessBeforeInitialization(Object bean, String beanName)
                        throws BeansException {
            System.out.println("postProcessBeforeInitialization:"+beanName);
            return bean;
    }
public Object postProcessAfterInitialization(Object bean, String beanName)
                        throws BeansException {
            System.out.println("postProcessAfterInitialization:"+beanName);
            return bean;
    }
    }
```


XML Shortcut

- p-namespace
  - enables you to use the bean element's attributes, instead of nested <property/> elements
- c-namespace
  - allows usage of inlined attributes for configuring the constructor arguments rather then nested constructor-arg elements.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="myDataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="com.mysql.jdbc.Driver"
      p:url="jdbc:mysql://localhost:3306/mydb"
      p:username="root"
      p:password="helloworld"/>
</beans>
```

27

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:c="http://www.springframework.org/schema/c"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
 <bean id="bar" class="x.y.Bar"/>
 <bean id="baz" class="x.y.Baz"/>

 <-- 'c-namespace' declaration -->
 <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz"
   c:email="foo@bar.com">

</beans>
```

28

Lazy-initialized beans

- You can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized.
- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-
    init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

29

Autowiring collaborators

- No
  - No Auto Wiring
- byName
  - Autowiring by property name
- byType
  - Autowired if exactly one bean of the property type exists in the container
- constructor

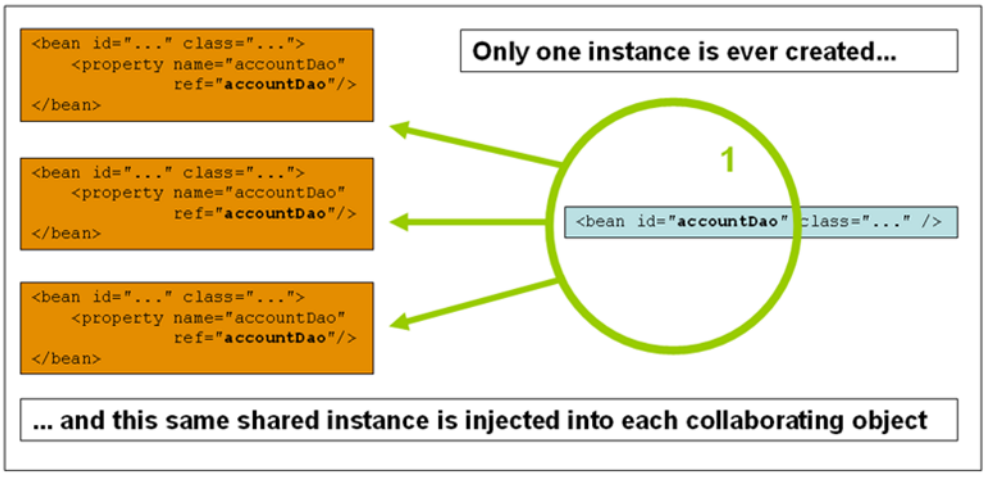
30

Bean Scopes

- singleton
- prototype
- request
  - New instance of the for every HTTP request
- session
  - New instance for lifetime of a single HTTP Session
- global
  - portlet-based web applications

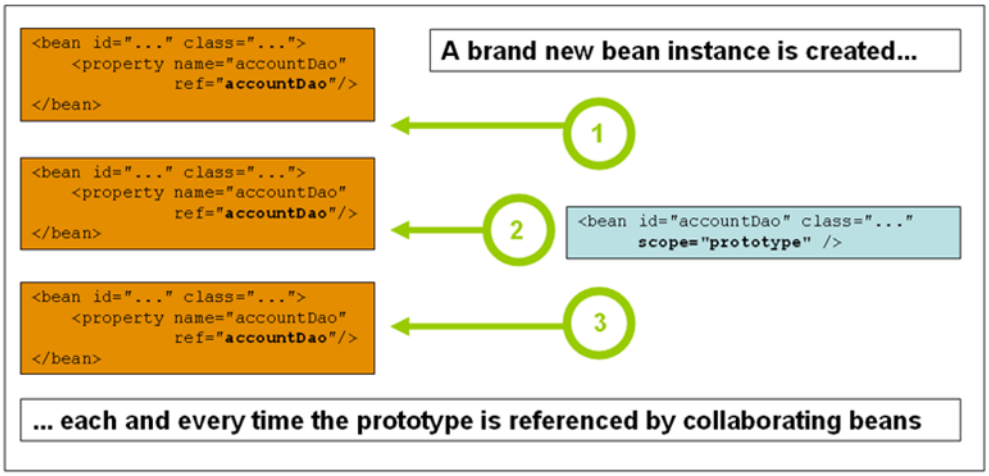
31

Singleton



32

Prototype



33

Annotation-based container configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example"/>
    </beans>
```

- Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.

34

Auto Detect Classes


```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
   ...
}

@Configuration
public class AppConfig {

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

35


```
<beans>
   <!-- picks up and registers AppConfig as a bean definition -->
   <context:component-scan base-package="com.acme"/>
   <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

   <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="url" value="${jdbc.url}"/>
      <property name="username" value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
   </bean>
</beans>
```

36

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}
```


```
@Configuration
public class ConfigA {
    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B b() {
        return new B();
    }
}
```

```
public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(ConfigB.class);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

AnnotationConfigApplicationContext


```
public static void main(String[] args) {
    ApplicationContext ctx = new
      AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

39

Annotation

- @Required
- @Autowired
- @Resource
- @PostConstruct
- @PreDestroy
- @Service
- @Component
- @Repository
- @Controller

40

Callbacks

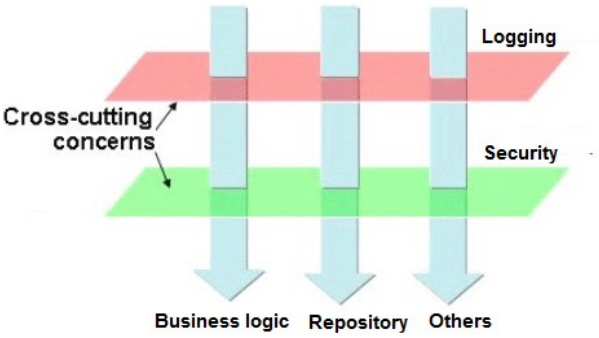
- Initialization callbacks
- Destruction callbacks

41

AOP

- In computing, aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.
- System services such as logging, transaction management, and security often find their way into components whose core responsibility is something else.
- These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.

42



43

AOP Terminology

- Aspect - Cross-cutting functionality
- Join point - A point during the execution of a program.
  - Execution of a method
  - Handling of an exception
  - Field Access
- Advice - Action taken by an aspect at a particular join point.
  - Around
  - Before
  - After

44

AOP Terminology

- Pointcut
  - A expression that matches join points
    - e.g - the execution of a method with a certain name
- Target object
  - object being advised by one or more aspects

- Weaving
  - linking aspects with other application types or objects to create an advised object

45

Advice

- Before advice
- After returning advice
- After throwing advice
- After (finally) advice

46

Declaring Aspect


```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
<!-- configure properties of aspect here as normal -->
 </bean>
```

```
package org.xyz;
 import org.aspectj.lang.annotation.Aspect;
@Aspect
public class NotVeryUsefulAspect { }
```

Note :@Aspect annotation is not sufficient for autodetection in the classpath: For that purpose, you need to add a separate @Component annotation

47

Declaring a pointcut

- Defines a pointcut named 'anyOldTransfer' that will match the execution of any methodnamed 'transfer'

```
@Pointcut("execution(* transfer(..))")
// the pointcut expression
private void anyOldTransfer() {
}
// the pointcut signature
```

48

Combining pointcut expressions


```
@Pointcut("execution(public * *(..))")

 private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..*)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()") private void
    tradingOperation() {}
```

49

Declaring advice

- Advice is associated with a pointcut expression
- Runs before, after, or around method executions matched by the pointcut.

```
@Aspect
public class BeforeExample {
@Before("com.xyz.myapp.dataAccessOperation()")
public void doAccessCheck() { // ... } }
```

50

AOP XML Elements

- <aop:before>
  - Defines an AOP before advice.
- <aop:config>
  - The top-level AOP element
- <aop:pointcut>
  - Defines a pointcut.

51

CarService


```
package com.surya;
public class CartService {
public void add() {
System.out.println("Adding to Cart....");
}
public void remove() {
System.out.println("Removing from Cart....");
}
public void update() {
System.out.println("Updating Cart....");
}
}
```

52

MyAspect.java

```
package com.surya;

public class MyAspect {
public void m1aspect() {
System.out.println("m1 methods of MyAspect fired.....");
}
public void m2aspect() {
System.out.println("m2 method of MyAspect fired.....");

}
public void m3aspect() {
System.out.println("m3 methods of MyAspect fired.....");

}
}
```

53

beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="cartService" class="com.surya.CartService" />
<bean id="myAspect" class="com.surya.MyAspect" />

<aop:config>
<aop:aspect id="aopmyAspect" ref="myAspect">
<aop:before method="m1aspect" pointcut="execution (* com.surya.CartService.add(..) )" />
<aop:after method="m2aspect" pointcut="execution (* com.surya.CartService.add(..) )" />
</aop:aspect>
</aop:config>
</beans>
```

54

TestAspect.java

```
package com.surya;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAspect {

	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

		CartService cartService = (CartService)context.getBean("cartService");
		cartService.add();

	}

}
```

55

Using Annotations – AuthenticationService.java

```
package com.ss.aop;
public class AuthenticationService {
public boolean  login(String username,String password)
{
return true;
}
}
```

56

LoggingAspect.java

```
@Aspect
public class LogginAspect {
    @Before("execution(* com.ss.aop.AuthenticationService.login(..))")
    private void before() {
    System.out.println("Before Demo.....");
    }
    @Before("execution(* com.ss.aop.*.*(..))")
    private void beforeall() {
    System.out.println("Before all Demo.....");
    }
            @AfterReturning("execution (* com.ss.aop.AuthenticationService.login(..))")
    private void afterreturning() {
    System.out.println("After Returning Demo.....");
    }
    @Around("execution (* com.ss.aop.AuthenticationService.login(..))")
    private void around()
    {
    System.out.println("Around Demo.....");
    }}
```

57

JDBC Access

- jdbcTemplate
  - Classic Spring JDBC approach and the most popular.
- NamedParameterJdbcTemplate
  - wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders

58

Data Source Configuration


```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
<property name="driverClassName" value="${jdbc.driverClassName}"/>
<property name="url" value="${jdbc.url}"/>
<property name="username" value="${jdbc.username}"/>
<property name="password" value="${jdbc.password}"/>
</bean>
```

59

Querying (SELECT)


```
int rCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor",
    Integer.class);

int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
"select count(*) from t_actor where first_name = ?",
Integer.class, "Joe");

String lastName = this.jdbcTemplate.queryForObject(
 "select last_name from t_actor where id = ?",
new Object[]{1212L}, S
tring.class);
```

60

Single Domain Object

```
Actor actor = this.jdbcTemplate.queryForObject(
"select first_name, last_name from t_actor where id = ?",
 new Object[]{1212L},
new RowMapper<Actor>() {
      public Actor mapRow(ResultSet rs, int rowNum) throws SQLException
         {
      Actor actor = new Actor();
      actor.setFirstName(rs.getString("first_name"));
         actor.setLastName(rs.getString("last_name"));
      return actor; }
       });
```

61

Multiple Domain Object

```
public List<Actor> findAllActors() {
return this.jdbcTemplate.query( "select first_name, last_name from t_actor",
new ActorMapper());
}

private static final class ActorMapper implements RowMapper<Actor> {
public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
Actor actor = new Actor();
actor.setFirstName(rs.getString("first_name"));
    actor.setLastName(rs.getString("last_name"));
return actor;
} }
```

62

INSERT/UPDATE/DELETE

- jdbcTemplate.update("insert into t_actor (first_name, last_name) values (?, ?)", "Leonor", "Watling");
- jdbcTemplate.update("update t_actor set last_name = ? where id = ?", "Banjo", 5276L);
- jdbcTemplate.update("delete from actor where id = ?", Long.valueOf(actorId));

63

NamedParameterJdbcTemplate

- Adds support for programming JDBC statements using named parameters, as opposed to only classic placeholder ('?') arguments.

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public int countOfActorsByFirstName(String firstName) {
String sql = "select count(*) from T_ACTOR where first_name = :first_name";
SqlParameterSource namedParameters = new
    MapSqlParameterSource("first_name", firstName);
return this.namedParameterJdbcTemplate.queryForObject(sql,
    namedParameters, Integer.class);
}
```

64

Retrieving auto-generated keys

- update() method supports the retrieval of primary keys generated by the database.

```
final String INSERT_SQL = "insert into my_test (name) values(?)";
final String name = "Rob";
KeyHolder keyhole = new GeneratedKeyHolder();

jdbcTemplate.update( new PreparedStatementCreator() {
public PreparedStatement createPreparedStatement(Connection
    connection) throws SQLException {
PreparedStatement ps = connection.prepareStatement(INSERT_SQL,
    new String[] {"id"}); ps.setString(1, name);
 return ps;
} }, keyHolder);
```

65

DBCP configuration


```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">
<property name="driverClassName" value="${jdbc.driverClassName}"/>
<property name="url" value="${jdbc.url}"/>
<property name="username" value="${jdbc.username}"/>
<property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

66

C3P0 configuration


```
<bean id="dataSource“ class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
   <property name="driverClass" value="${jdbc.driverClassName}"/>
   <property name="jdbcUrl" value="${jdbc.url}"/>
   <property name="user" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

67

Stored Procedure


```
public class JdbcActorDao implements ActorDao {
   private JdbcTemplate jdbcTemplate;
   private SimpleJdbcCall procReadActor;
   public void setDataSource(DataSource dataSource) {
      this.jdbcTemplate = new JdbcTemplate(dataSource);
      this.procReadActor = new SimpleJdbcCall(dataSource)
            .withProcedureName("read_actor");
   }

   public Actor readActor(Long id) {
      SqlParameterSource in = new MapSqlParameterSource()
            .addValue("in_id", id);
      Map out = procReadActor.execute(in);
      Actor actor = new Actor();
      actor.setId(id);
      actor.setFirstName((String) out.get("out_first_name"));
      actor.setLastName((String) out.get("out_last_name"));
      actor.setBirthDate((Date) out.get("out_birth_date"));
      return actor;
   }
```

68

Transaction

- A transaction is a set of logically related operations.
- Atomic unit of work with boundary (begin and end)
- Money Transfer (Account A –> Account B)
  - 1. R(A)
  - 2. A = A - 10000
  - 3. W(A)
  - 4. R(B)
  - 5. B = B + 10000
  - 6. W(B)
- Transaction failure between 1 – 6 operations (in consitent state)

- Commit
  - If all the operations in a transaction are completed successfully then commit those changes to the database permanently.
- Rollback
  - If any of the operation fails then rollback all the changes done by previous operations

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

- Dirty Reads
  A transaction reads data that has not yet been committed.
- Nonrepeatable Reads
  - A transaction reads the same row twice but gets different data.
- Phantom
  - A transaction, new rows are added or removed by another transaction to the records being read.

Transaction Isolation Levels

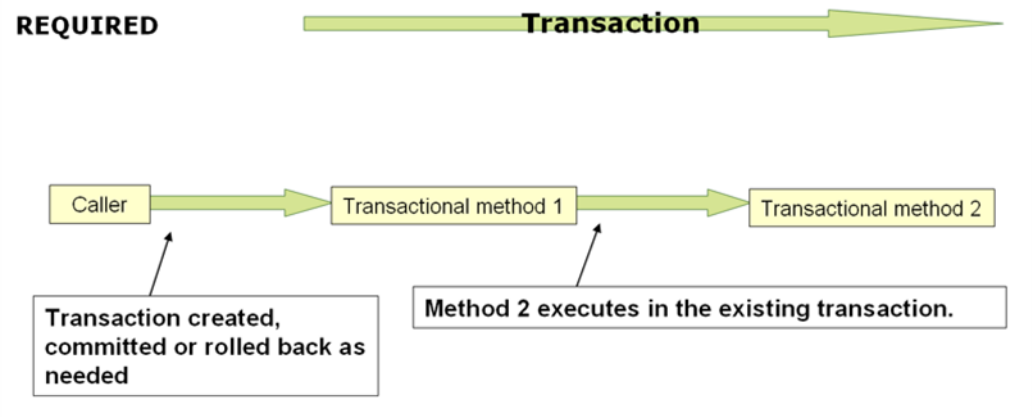
- The transaction isolation level is a state within databases that specifies the amount of data that is visible to a statement in a transaction.

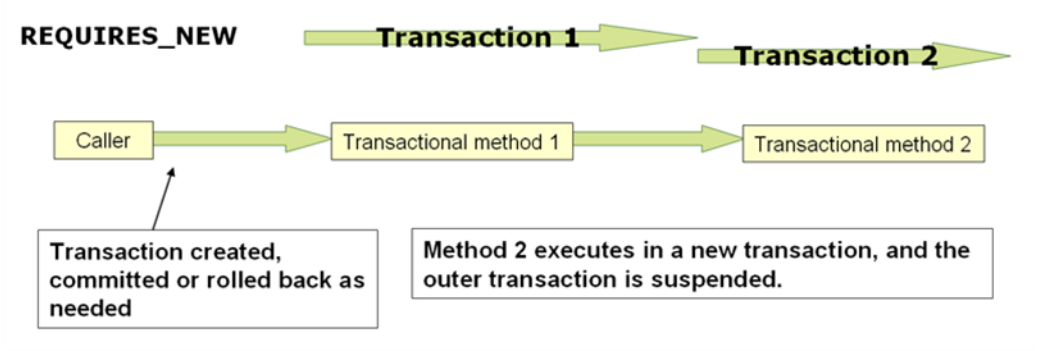
- Read uncommitted
- Read committed
- Repeatable read
- Serializable

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
Read uncommitted	X	X	X
Read committed	--	X	X
Repeatable read	--	--	X
Serializable	--	--	--

Spring Transaction Propagation

- REQUIRED
- REQUIRES_NEW
- MANDATORY
- NOT_SUPPORTED
- NEVER





XML Schema

```
<beans xmlns=http://www.springframework.org/schema/beans
    ....
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx=http://www.springframework.org/schema/tx
    ...
      http://www.springframework.org/schema/tx
      https://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/aop
      https://www.springframework.org/schema/aop/spring-aop.xsd">
/>
```

configuration

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

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
</bean>
<bean id="txManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
</bean>
```

Configuration

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save" propagation="REQUIRED“ isolation="DEFAULT"
    />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="saveOperation“
        expression="execution(* com.springone.jdbc.ProductDAO.save(..))" />
    <aop:advisor advice-ref="txAdvice“ pointcut-ref="saveOperation" />
</aop:config>
```

```
@Configuration
@ComponentScan(basePackages = { "com.contactmanager" })
@EnableTransactionManagement
@EnableAspectJAutoProxy
public class Config {
	@Bean("dataSource")
	public DriverManagerDataSource dataSource() {
		DriverManagerDataSource ds = new DriverManagerDataSource();
		ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
		ds.setUrl("jdbc:mysql://localhost/psldb");
		ds.setUsername("root");
		ds.setPassword("");
		return ds;
	}
	@Bean("jdbcTempate")
	public JdbcTemplate jdbcTemplate()
	{
		return new JdbcTemplate(dataSource());
	}
	@Bean
	public DataSourceTransactionManager dataSourceTransactionManager()
	{
		return new DataSourceTransactionManager(dataSource());
	}
}
```

```
@Transactional(rollbackFor=DataAccessException.class,
    propagation=Propagation.REQUIRED,isolation=Isolation.READ_COMMITTED)

public int create(Contact c) {
		contactDAO.create(c);
		contactDAO.create(c);
		c.setName("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
		contactDAO.create(c);
		contactDAO.create(c);
		return 0;
    }
```

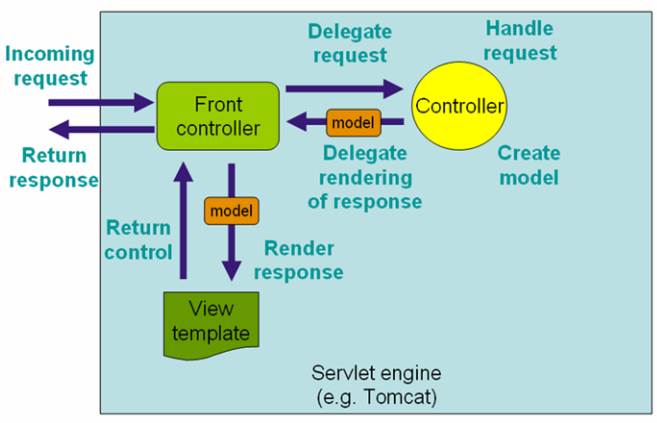
Spring Web MVC

- Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning.

83

The DispatcherServlet

- Servlet that dispatches requests to controllers.
- Integrated with the Spring IoC container.



DispatcherServlet declaration and mapping


```
<web-app>
   <servlet>
      <servlet-name>example</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>example</servlet-name>
      <url-pattern>/example/*</url-pattern>
   </servlet-mapping>

</web-app>
```

** upon initialization of a DispatcherServlet, Spring MVC looks for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

View resolvers

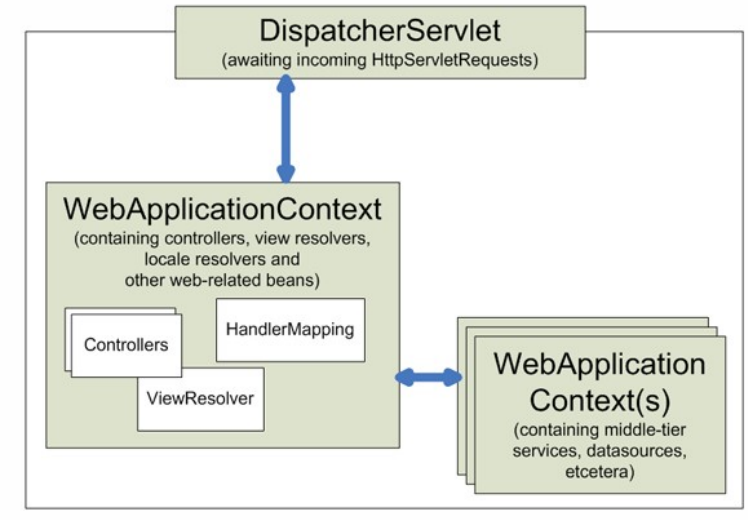
- View resolver resolve your views

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

Configuring the Servlet container programmatically


```
public class MyWebApplicationInitializer implements WebApplicationInitializer {
   @Override
   public void onStartup(ServletContext container) {
      ServletRegistration.Dynamic registration = container.addServlet("dispatcher",
     new DispatcherServlet());
      registration.setLoadOnStartup(1);
      registration.addMapping("/example/*");
   }
}
```

Context Hierarchy



Controllers

- Controllers provide access to the application behavior that you typically define through a service interface.

- Controllers interpret user input and transform it into a model that is represented to the user by the view.

```
@Controller
public class HelloWorldController {
@RequestMapping("/helloWorld")
public String helloWorld(Model model) {
 model.addAttribute("message", "Hello World!");
return "helloWorld";
}
}
```

** To enable autodetection of such annotated controllers, you add component scanning to your configuration.

```
<context:component-scan base-package="org.springframework.*"/>
```

@RequestMapping

- You use the @RequestMapping annotation to map URLs such as /appointments

```
@RequestMapping(method = RequestMethod.GET)
public Map<String, Appointment> get() {
    return appointmentBook.getAppointmentsForToday();
}
@RequestMapping(value="/{day}", method = RequestMethod.GET)
public Map<String, Appointment> getForDay(@PathVariable
   @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
    return appointmentBook.getAppointmentsForDay(day);
}
@RequestMapping(value="/new", method = RequestMethod.GET)
public AppointmentForm getNewForm() {
    return new AppointmentForm();
}
@RequestMapping(method = RequestMethod.POST)
public String add(@Valid AppointmentForm appointment, BindingResult result) {
    if (result.hasErrors()) {
        return "appointments/new";
    }
```

Path Patterns

@PathVariable - parameters for access to URI template variables.

@MatrixVariable - parameters for access to name-value pairs

@RequestParam - parameters for access to specific Servlet request parameters

@RequestBody - parameters for access to the HTTP request body

@RequestPart - parameters for access to the content of a "multipart/form-data"

@ModelAttribute –

@ResponseBody - indicates that the return type should be written to the HTTP response body

@CookieValue - allows a method parameter to be bound to the value of an HTTP cookie.

Spring's form tag library

```
<form:form action="addEmployee" modelAttribute="employee">
    <table><tr>
                    <td><form:label path="name">Name</form:label></td>
                    <td><form:input path="name" /></td>
            </tr><tr>
                    <td><form:label path="id">Id</form:label></td>
                    <td><form:input path="id" /></td>
            </tr><tr>
                    <td><form:label path="contactNumber">Contact
    Number</form:label></td>
                    <td><form:input path="contactNumber" /></td>
            </tr><tr>
                    <td><input type="submit" value="Submit" /></td>
            </tr></table>
</form:form>
```

Spring Boot

- Create stand-alone, production-grade Spring based Applications that you can "just run"
- Provides RAD – Rapid Application Development
- Getting-started experience for all Spring development.

Why do we need Spring Boot?

- Spring based applications have a lot of configuration.
- Any typical web application would use all these dependencies.
  - Spring - core, beans, context, aop
  - Web MVC - (Spring MVC)
  - Jackson - for JSON Binding
  - Validation - Hibernate Validator, Validation API
  - Embedded Servlet Container - Tomcat
  - Logging - logback, slf4j
- Spring Boot Starter Web comes pre packaged with these.

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

Starters

- Convenient dependency descriptors that you can include in your application.
- if you want to get started using Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency in your project, and you are good to go.

spring-boot-starter-parent

- Provides useful Maven defaults.
- Omit version tags dependencies.

Dependencies

- Spring - core, beans, context, aop
- Web MVC - (Spring MVC)
- Jackson - for JSON Binding
- Validation - Hibernate Validator, Validation API
- Embedded Servlet Container - Tomcat
- Logging - logback, slf4j

- Any typical web application would use all these dependencies.

- Spring Boot Starter Web comes pre packaged with these.

Spring Boot Starter Project Options

- Spring-boot-starter-web-services - SOAP Web Services
- spring-boot-starter-web - Web & RESTful applications
- spring-boot-starter-test - Unit testing and Integration Testing
- spring-boot-starter-jdbc - Traditional JDBC
- spring-boot-starter-hateoas - Add HATEOAS features to your services
- spring-boot-starter-security - Authentication and Authorization using Spring Security
- spring-boot-starter-data-jpa - Spring Data JPA with Hibernate
- spring-boot-starter-cache - Enabling Spring Framework's caching support
- spring-boot-starter-data-rest - Expose Simple REST Services using Spring Data REST
- spring-boot-starter-actuator - Monitoring & tracing to your application out of the box
- spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat - To pick your specific choice of Embedded Servlet Container
- spring-boot-starter-logging - For Logging using logback
- spring-boot-starter-log4j2 - Logging using Log4j2

Spring Boot CLI

- Command line tool to quickly develop a Spring application
- Installation
  - https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.0.4.RELEASE/spring-boot-cli-2.0.4.RELEASE-bin.zip
  - https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.0.4.RELEASE/spring-boot-cli-2.0.4.RELEASE-bin.tar.gz

Using CLI

- $ spring version
- $ spring help run

```
hello.groovy
@RestController
class WebApplication {
@RequestMapping("/")
String home() {
"Hello World!“
 }
 }
```

- $ spring run hello.groovy
- $ spring run hello.groovy --server.port=9000

```
$ spring init --dependencies=web,data-jpa my-project
$ spring init --list
```

Spring Boot Maven Plugin

- Offer a variety of features, including the packaging of executable jars.

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>

$ mvn package
$ mvn spring-boot:run
$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar
```

To build a war file that is both executable and deployable into an external container,

mark the embedded container dependencies as "provided".

```
<packaging>war</packaging>
  <dependencies>
    <dependency>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                    <scope>provided</scope>
    </dependency>
```

"provided" indicates JDK or a container to provide the dependency at runtime.

Auto Configuration

- Attempts to automatically configure your Spring application based on the jar dependencies
- @EnableAutoConfiguration / @SpringBootApplication

Disabling Specific Auto-configuration Classes


```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {

}
```

Developer Tools

- Applications that use spring-boot-devtools automatically restart whenever files on the classpath change.

```
<dependencies>
<dependency> <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
</dependency>
</dependencies>
```

```
spring.devtools.restart.exclude=static/**,public/**
```

Externalized Configuration

- Property files
- YAML - format for specifying hierarchical configuration data

```
@Component
public class MyBean {
   @Value("${name}")
}
```

```
application.properties
name=test property
```

```
$java -jar app.jar --name="Spring"
```

Application Property Files

- SpringApplication loads properties from application.properties
- You can also use YAML ('.yml') files as an alternative to '.properties'
- $ java -jar myproject.jar --spring.config.name=myproject

@Value

- @Value - Inject external configuration

```
@Value("${msg}")
private String msg;
```

application.properties

```
msg=Hello World!!!!!
```

```
environments:
    dev:
        url: http://dev.example.com
        name: Developer Setup
    prod:
        url: http://another.example.com
        name: My Cool App
```

```
environments.dev.url=http://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=http://another.example.com
environments.prod.name=My Cool App
```

@ConfigurationProperties

- Using the @Value("${property}") annotation to inject configuration properties can sometimes be cumbersome

```
@ConfigurationProperties("simple")
public class SimpleService {
private String msg;
public void setMsg(String msg) {
this.msg = msg;
}
application.properties
simple.msg=Hi There!!!!!
```

Profile Specific Properties

- Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments
  - spring.profiles.active=development
- application-{profile}.properties

```
@Configuration
@Profile("production")
public class ProductionConfiguration {
 // ...
}
```

Importing Additional Configuration Classes

- Put all your @Configuration into a single class.
- Use @Import to import additional configuration classes
  - *@Import({ MyConfig.class, MyAnotherConfig.class })*
- Use @ImportResourceannotation to load XML configuration files.

Static Content

- Spring Boot serves static content from :
  - /static
  - /public
  - /resources
  - /META-INF/resources

Developing Web Applications

- Use the spring-boot-starter-web module to get up and running quickly

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>
</parent>
```


```
@RestController
@EnableAutoConfiguration
public class Example {
    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }
}
//$ mvn package
//$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

Using the ApplicationRunner or CommandLineRunner

- Run specific code once the SpringApplication has started
- Implement
  the ApplicationRunner or CommandLineRunner interfaces.

```
@Component
public class MyBean implements CommandLineRunner {
    public void run(String... args) {
        // Do something...
    }
}
```

Template Engines

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

Data Source

- javax.sql.DataSource interface provides a standard method of working with database connections.

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Embedded Database Support

- In-memory databases do not provide persistent storage.
- Populate your database when your application starts
- Throw away data when your application ends.
- Spring Boot can auto-configure embedded H2, HSQL, and Derby databases.

Using H2's Web Console

- The H2 database provides a browser-based console that Spring Boot can auto-configure for you.
- spring.h2.console.enabled =true
- /h2-console

Using JdbcTemplate

- JdbcTemplate and NamedParameterJdbcTemplate classes are auto-configured
- @Autowire them directly into your own beans

```
@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public MyBean(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // ...

}
```

Initialize a Database

- Spring Boot can automatically create the schema (DDL scripts) of and initialize it (DML scripts)

- schema-${platform}.sql
- data-${platform}.sql
- platform is the value of spring.datasource.platform

What are microservices?

- An architectural style that structures an application as a collection of services
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable
  - Organized around business capabilities
  - Owned by a small team

127

Companies using microservices

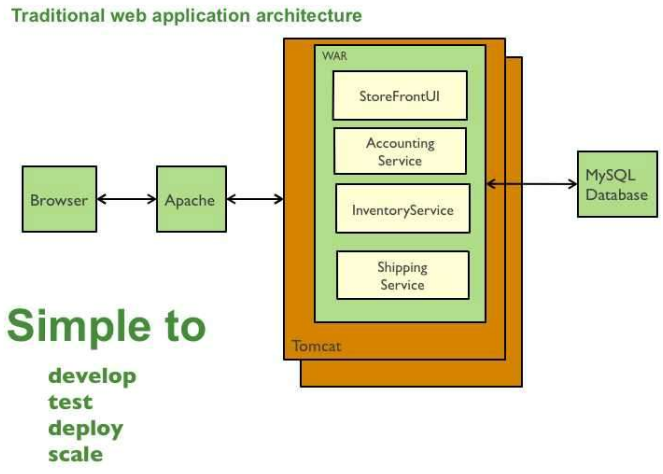
- Comcast Cable
- Uber
- Netflix
- Amazon
- Ebay
- Sound Cloud
- Groupon
- Hailo
- Gilt
- Zalando
- Lending Club
- AutoScout24

128

Monolith Architecture

- A single Java WAR file.
- A single directory hierarchy of Rails or NodeJS code

Monolithic Architecture



Benefits

- Simple to develop
- Simple to deploy
- Simple to scale

131

Drawbacks of Monolithic Architecture

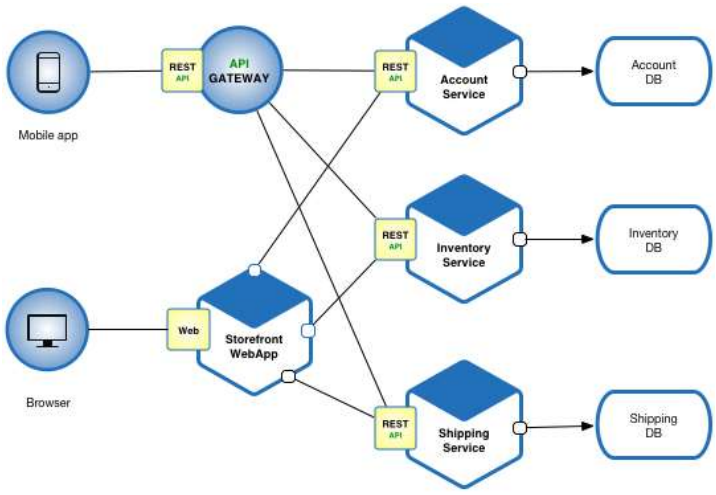
- Application is too large and complex
- The size of the application can slow down the start-up time.
- Difficult to scale when different modules have conflicting resource requirements.
- Redeploy the entire application on each update.
- Bug in any module (e.g. memory leak) can potentially bring down the entire process.
- Continuous deployment is difficult.
- Barrier to adopting new technologies.

132

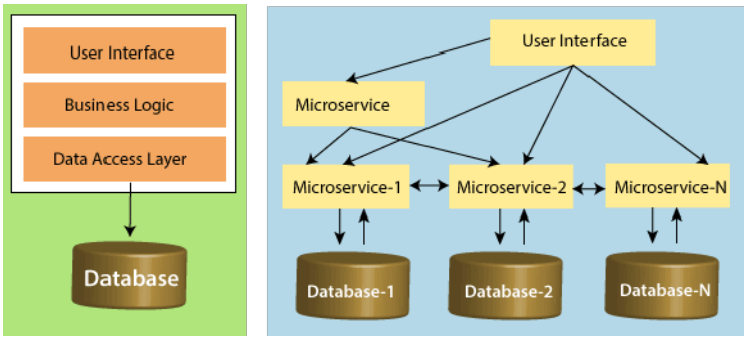
MicroServices

- Method of developing software applications as a suite of independently deployable, small, modular services.
- Each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.
- Each of these services can be deployed, tweaked, and then redeployed independently without compromising the integrity of an application. A

MicroServices Architecture



Micro vs Monolith



Monolithic vs Microservice Architecture

135

Benefits

- Independent development & deployments
- Small, focused teams
- Fault isolation
- Mixed technology stacks

136

Challenges

- Complexity (more moving parts)
- Development and test
- Lack of governance (hard to maintain due to different lang. and frameworks)
- Network congestion and latency (more interservice communication)
- Data integrity (persistence)
- Versioning (updates should not break the service)
- Skillset

137

Best Practices

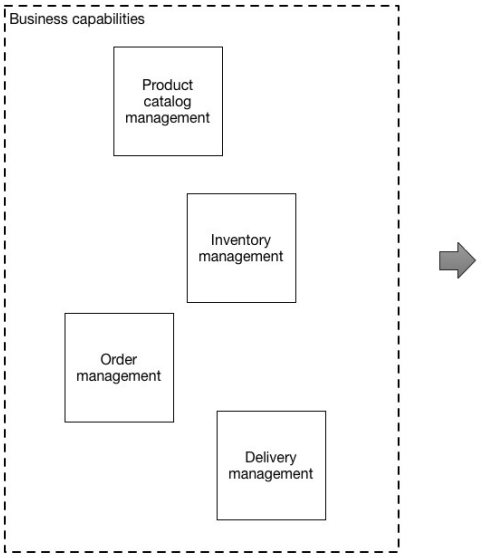
- Model services around the business domain.
- Decentralize everything.
- Individual teams are responsible for designing and building services.
- Avoid sharing code or data schemas.
- Services communicate through well-designed APIs.
- Avoid coupling between services.
  - Causes of coupling include shared database schemas and rigid communication protocols.

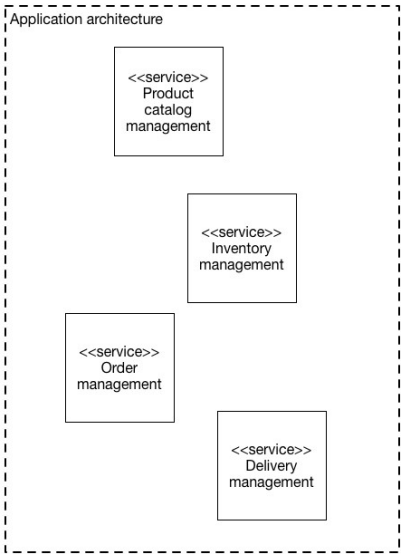
138

How to decompose an application into services?

- Business capabilities are often organized into a multi-level hierarchy.
- Define services corresponding to business capabilities.
- A business capability often corresponds to a business
  - Order Management
  - Customer Management
  - Catalog Management
  - Shipping Management

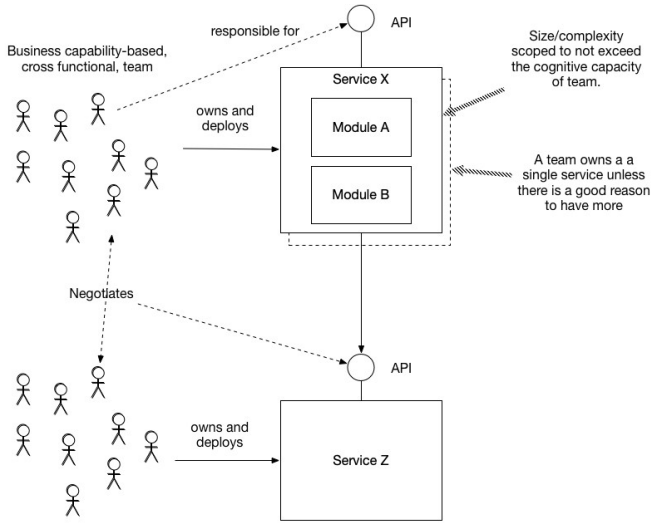
139



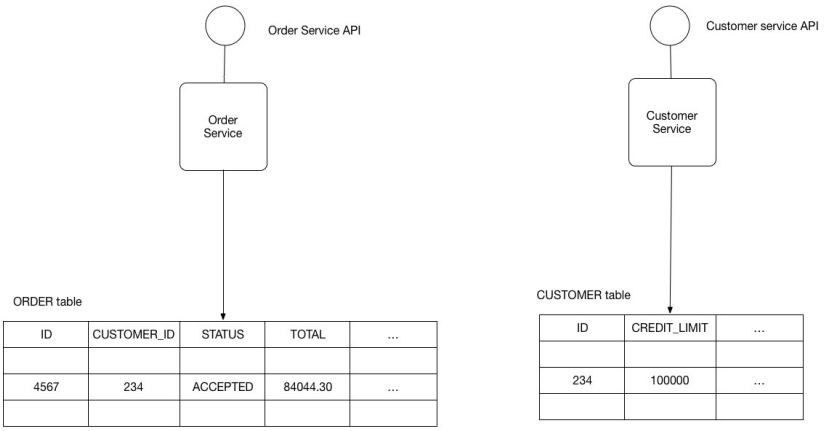


140

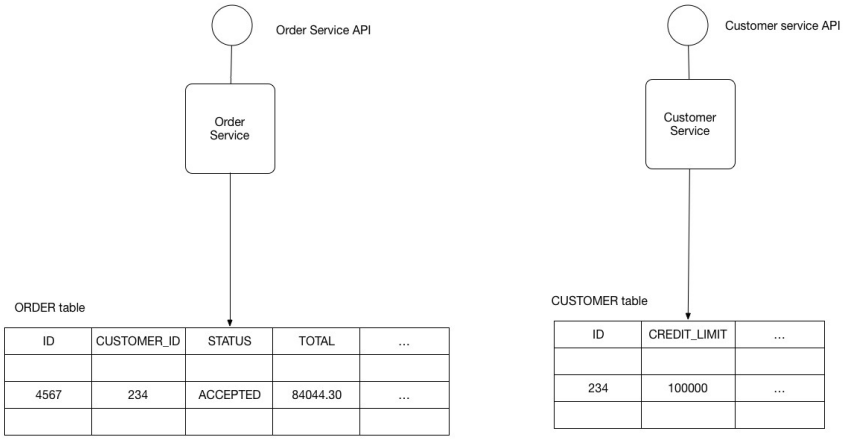
Service Per Team



Database per service



Database per service



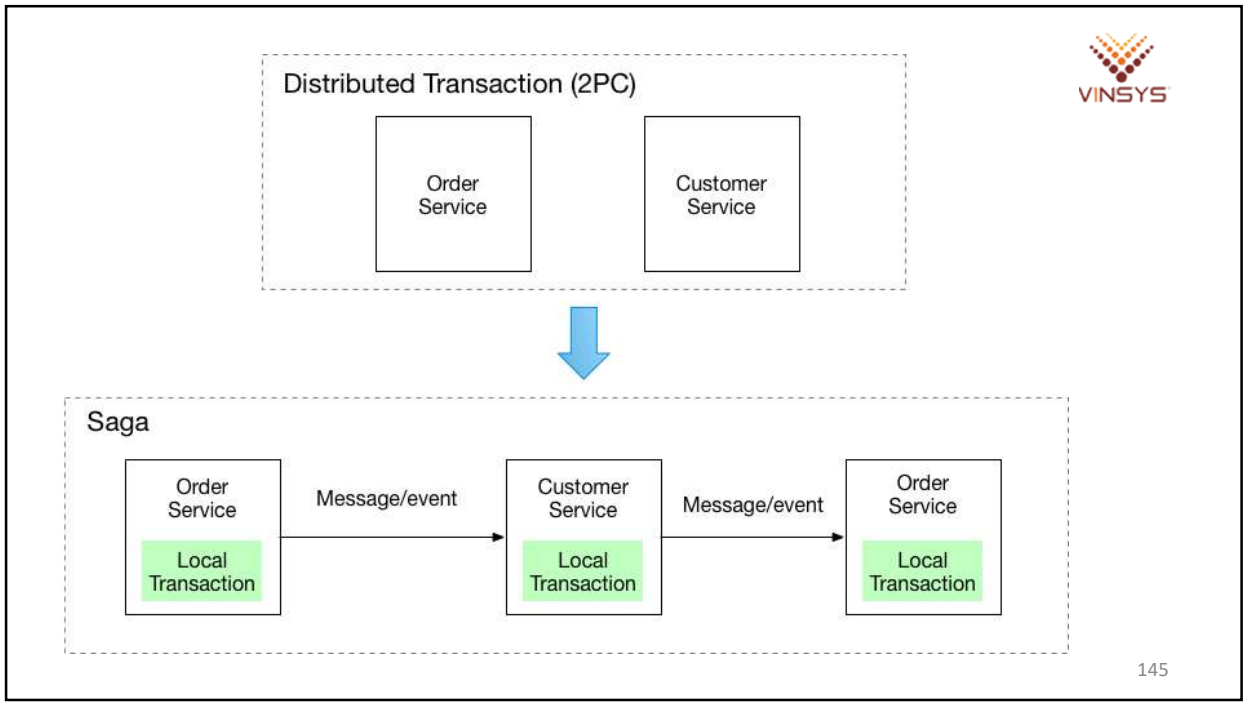
143

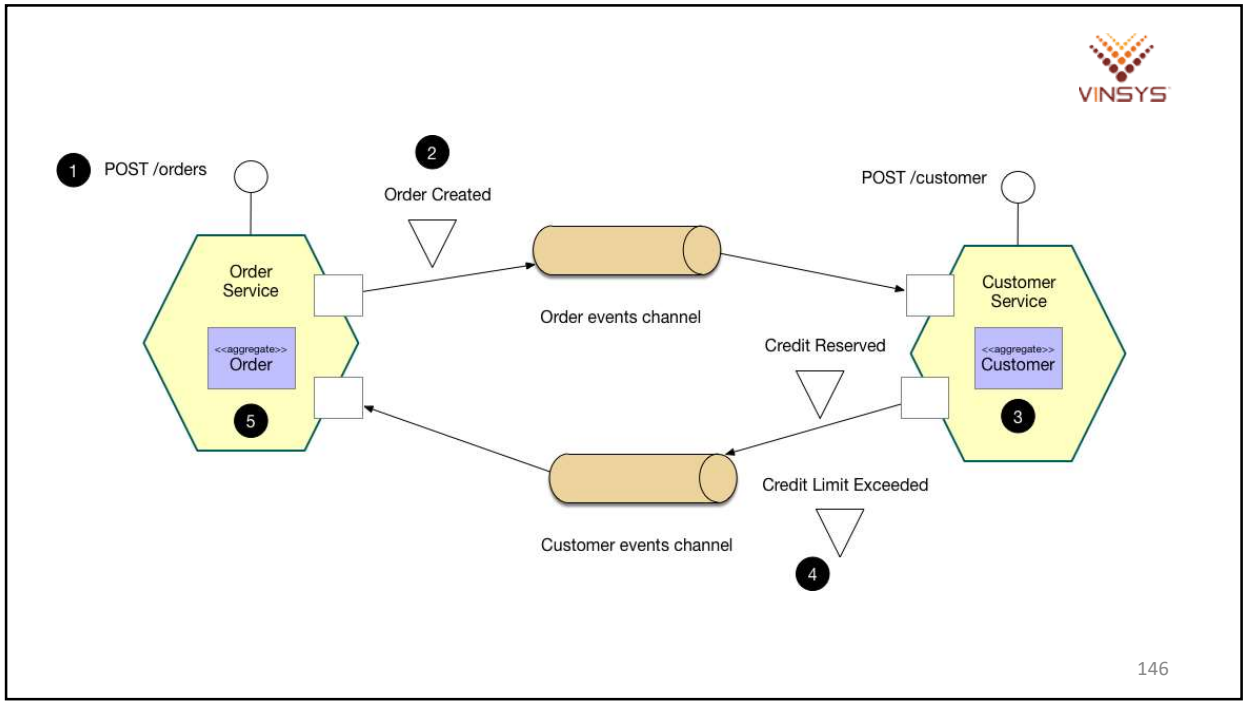
Saga - Pattern

- Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.

- A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

144


Distributed Transaction (2PC)
Order Service
Customer Service
Saga
Order Service
Local Transaction
Message/event
Customer Service
Local Transaction
Message/event
Order Service
Local Transaction
145


1
POST /orders
2
Order Created
POST /customer
Order Service
<<aggregate>>
Order
5
Order events channel
Credit Reserved
Customer Service
<<aggregate>>
Customer
3
Credit Limit Exceeded
Customer events channel
4
146

12 Factor

- The twelve-factor app is a methodology for building software-as-a-service apps
- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project
- Suitable for deployment on modern cloud platforms

- I. Codebase
  - One codebase tracked in revision control
- II. Dependencies
  - Explicitly declare and isolate dependencies
- III. Config
  - Store config in the environment
- IV. Backing services
  - Treat backing services as attached resources
- V. Build, release, run
  - Strictly separate build and run stages
- VI. Processes
  - Execute the app as one or more stateless processes

- VII. Port binding
  - Export services via port binding
- VIII. Concurrency
  - Scale out via the process model
- IX. Disposability
  - Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity
  - Keep development, staging, and production as similar as possible
- XI. Logs
  - Treat logs as event streams
- XII. Admin processes
  - Run admin/management tasks as one-off processes

I. Codebase

- One codebase tracked in revision control

- A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion.

II. Dependencies

- Explicitly declare and isolate dependencies
- Declare all dependencies, completely and exactly, via a dependency declaration manifest.
  - npm, yarn, maven, gradle

III. Config

- Store config in the environment
- An app's config is everything that is likely to vary between deploys (staging, production, developer environments).
  - Resource handles to the database and other backing services
  - Credentials to external services such as Amazon S3 or Twitter
- Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

IV. Backing services

- Treat backing services as attached resources
- A backing service is any service the app consumes over the network as part of its normal operation. Eg. MySQL , RabbitMQ
- Accessed via a URL or other locator/credentials stored in the config.
- A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code.

V. Build, release, run

- Strictly separate build and run stages
- Build stage is a transform which converts a code repo into an executable bundle known as a build.
- The release stage takes the build produced by the build stage and combines it with the deploy's current config.
- The run stage (also known as "runtime") runs the app in the execution environment.

VI. Processes

- Execute the app as one or more stateless processes
- The app is executed in the execution environment as one or more processes.
- Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

VII. Port binding

- Export services via port binding
- Make sure that your service is visible to others via port binding
  - http://localhost:5000/

VIII. Concurrency

- Scale out via the process model
- This is all about scalability.
- Small, defined apps allow scaling out as needed to handle the varying loads.
- Each process should be individually scaled

IX. Disposability

- Maximize robustness with fast startup and graceful shutdown
- The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice.
- This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

X. Dev/prod parity

- Keep development, staging, and production as similar as possible

XI. Logs

- Treat logs as event streams
- Application shouldn't concern itself with the storage of this information.
- Logs should be treated as a continuous stream that is captured and stored by a separate service.

XII. Admin processes

- Run admin/management tasks as one-off processes
- Administrative or management tasks should be executed as separate short-lived processes.
- The technology ideally supports command execution in a shell that operates on the running environment.