

Spring Framework

Session 1

Core, Container and Beans

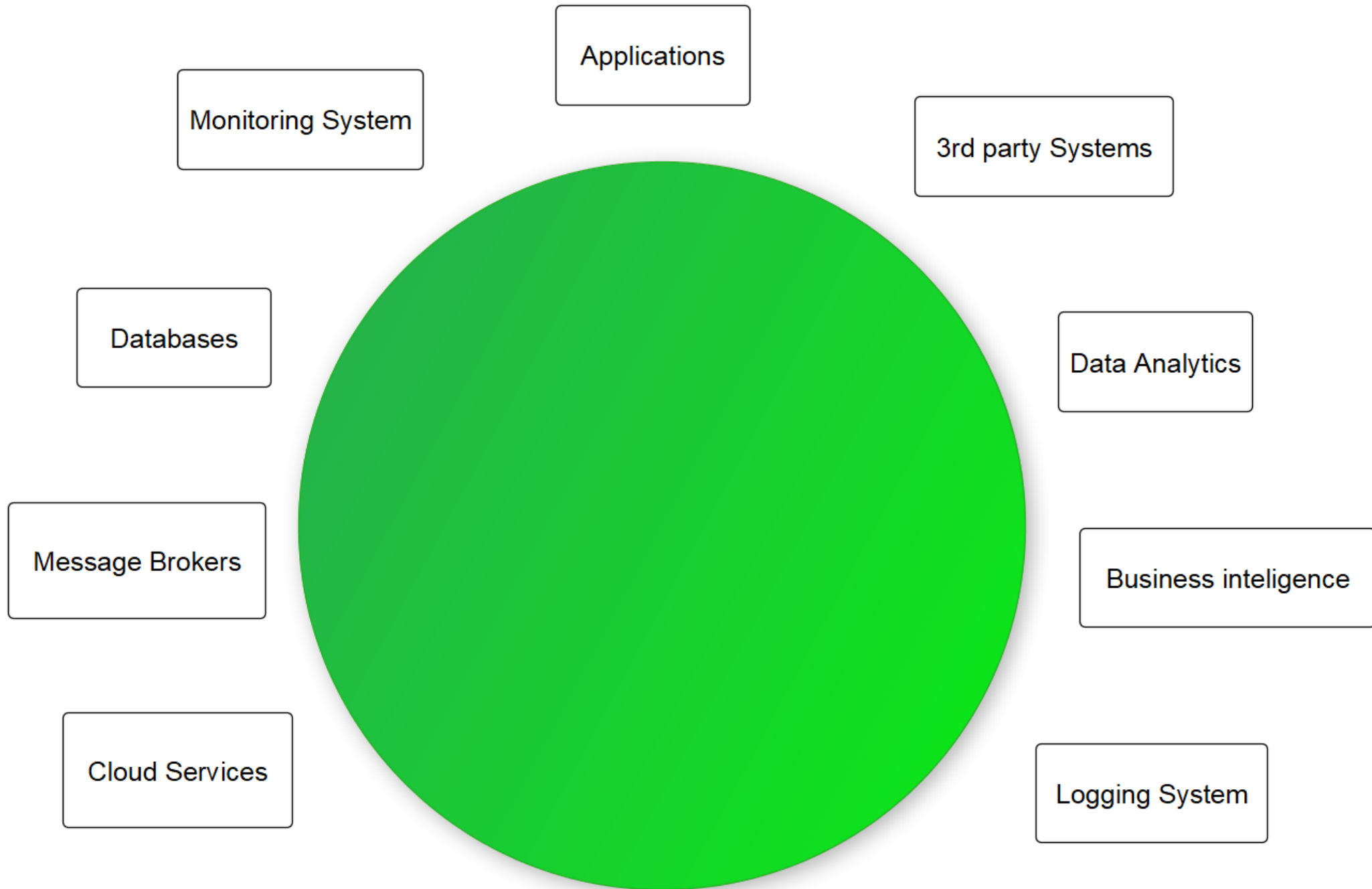
macalak@itexperts.sk

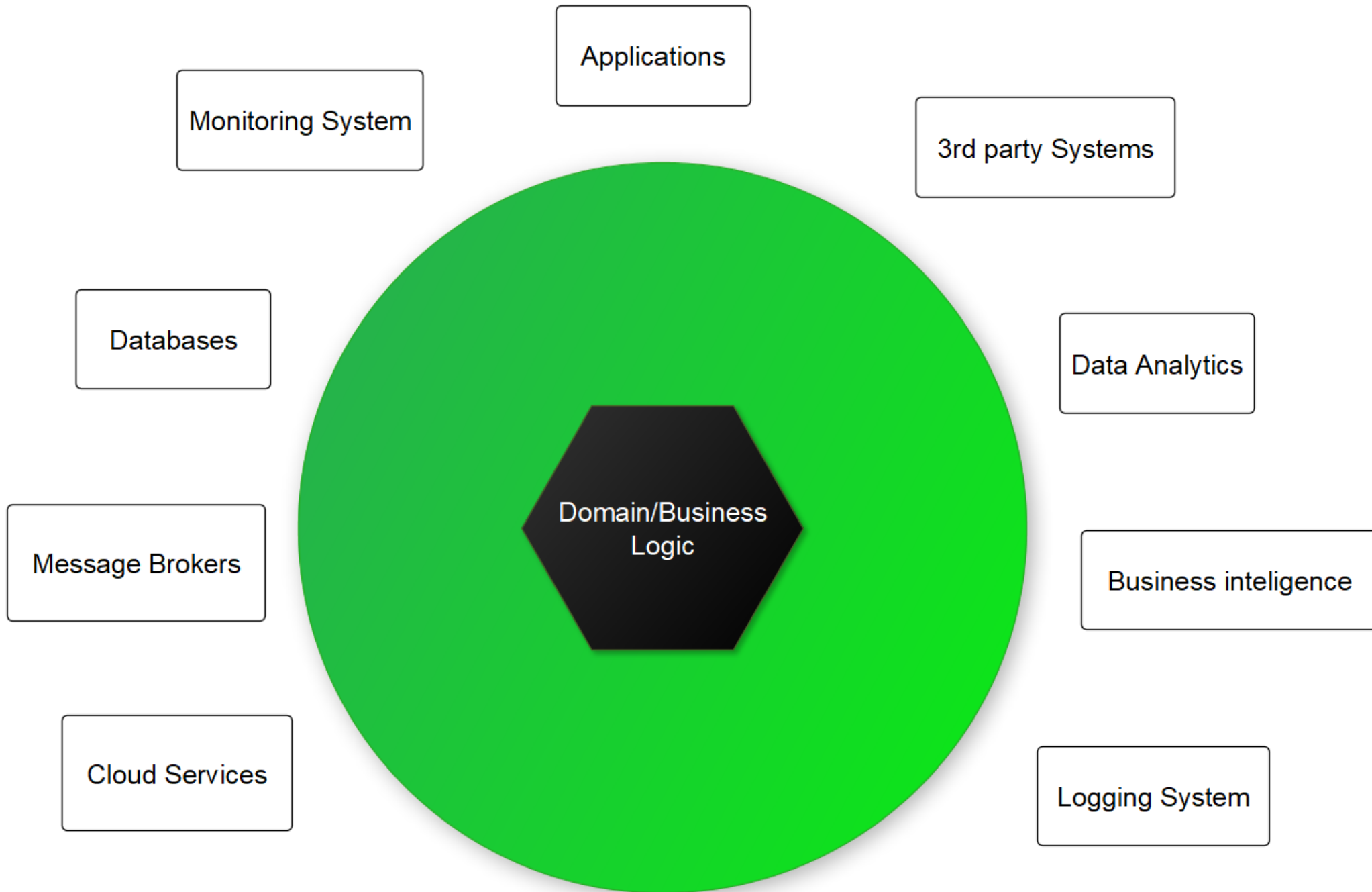


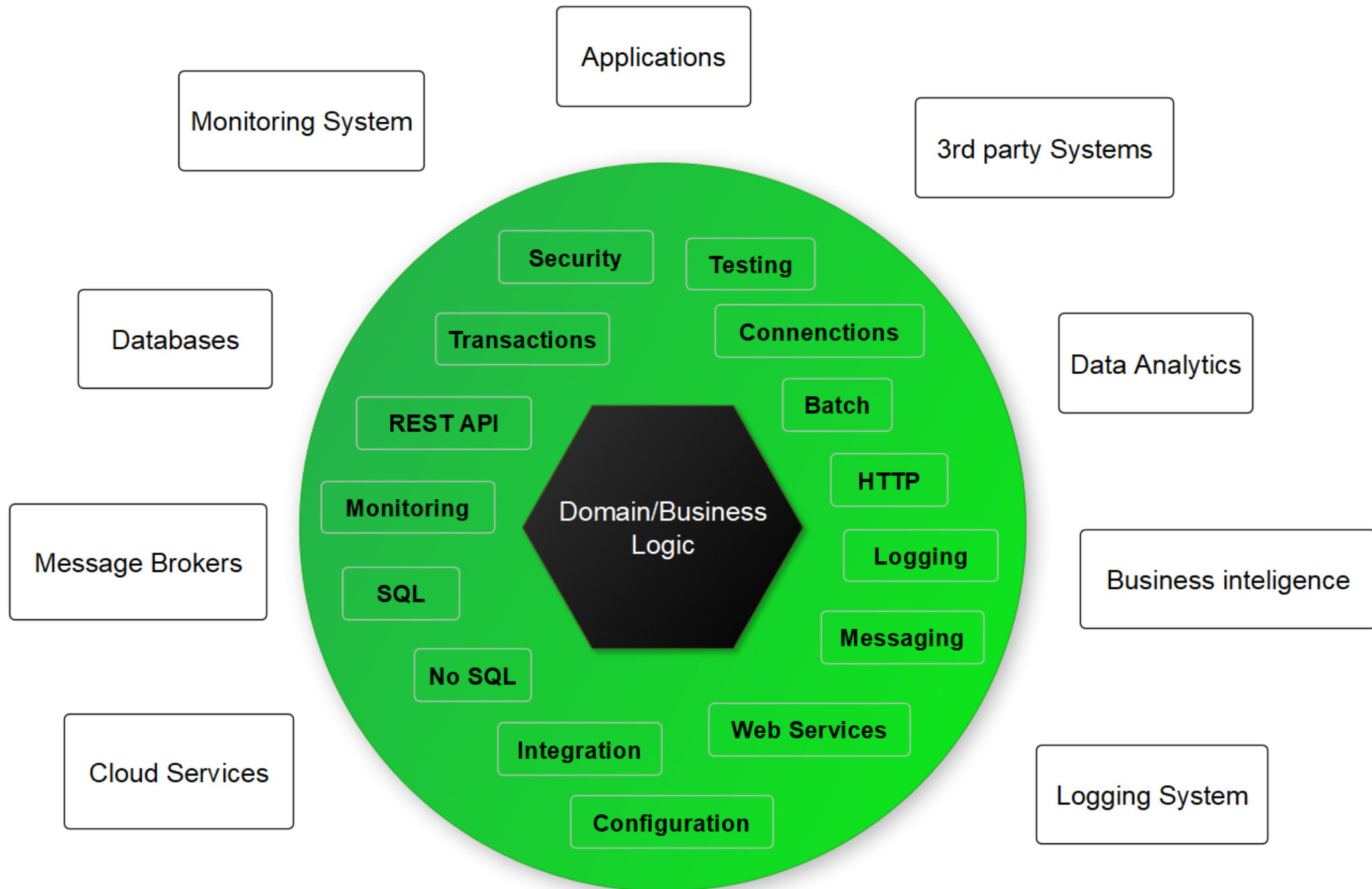
Session Agenda

- Introduction to Spring Framework
- Application Configuration and Container
- Components and Spring Beans
- Dependency Injection

Spring Framework Core Concept









Introduction (1)

- Open Source Lightweight Java platform that provides comprehensive infrastructure support for developing Java applications
- Spring handles the infrastructure
- You can focus on your application
- You can build any application in Java (not only server-side)
 - Stand-alone
 - Web
 - Java EE
 - Spring consists of features organized into modules
 - Utilizes Dependency Injection



Introduction (2)

- Enables application creation by loosely coupled building blocks
- You can add enterprise services to your POJO application
- Spring codifies formalized design patterns as first-class objects
- You can integrate these objects into your own application
- Originally created to address the complexity of enterprise application development



Spring History

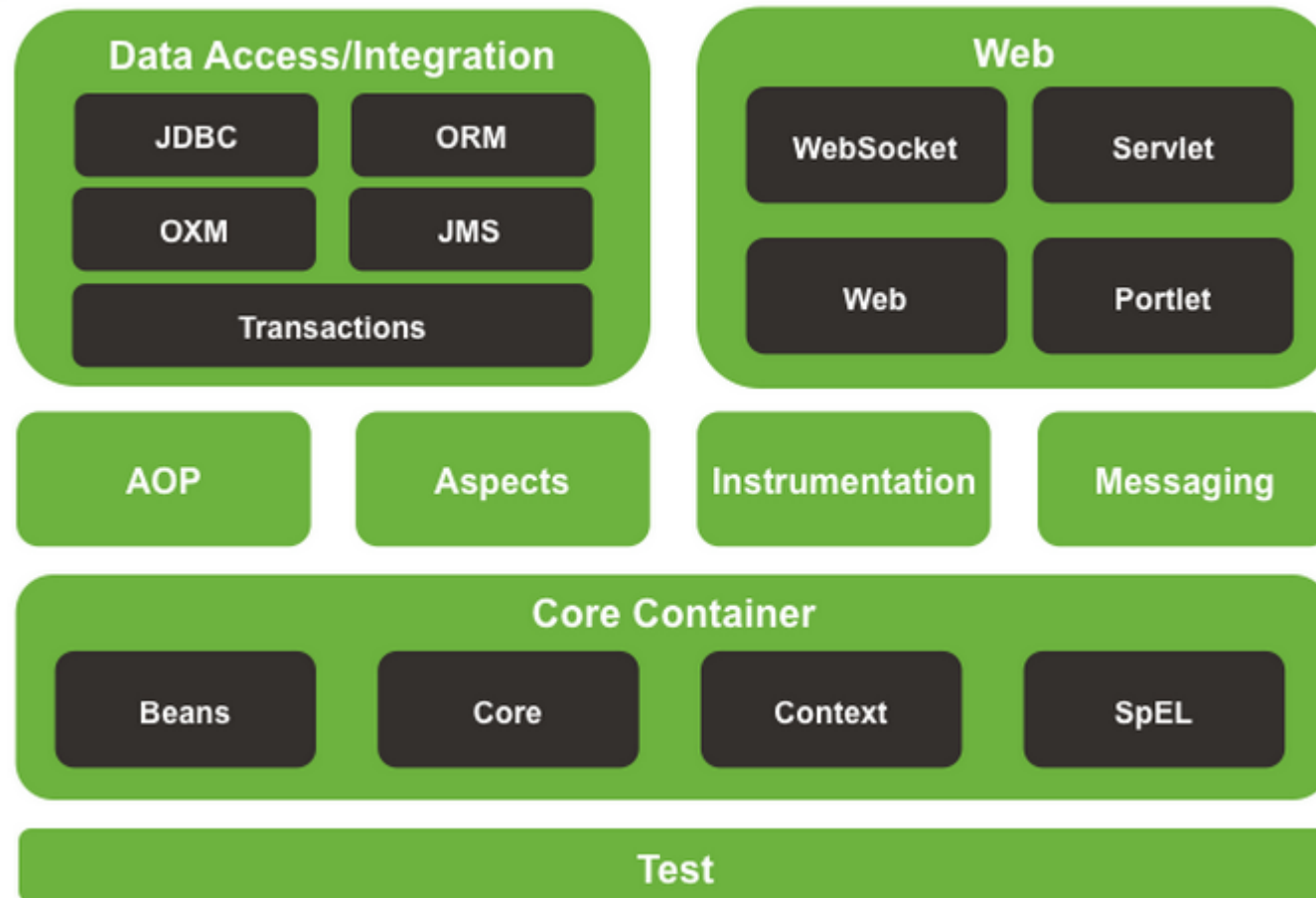
- 2002: First version written by Rod Johnson and released with the publication of his book *“Expert One-on-One J2EE Design and Development”*
- 2006: Spring 2.0 released – easier XML config, new Bean scopes, JPA support
- 2009: Spring 3.0 released – Java 5 annotations, modularization improvements, Spring Expression Language, REST support, embedded DB support
- 2013: Spring 4.0 released - Java SE8 support, Java EE7 API (JPA 2.1, Servlet 3.0+,...), WebSockets support (spring-websocket), Groovy Bean definition DSL – **End of Life 12/2020**
- 2017: Spring 5.0 released – Java SE9 support, Java EE8 API (Servlet 4.0), Reactive programming with Reactor 3.1, Kotlin support, Spring WebFlux (reactive web)
- 2019: Spring 5.2 (Boot 2.2) – Java 11, Reactive Tx management, RSocket support, and more ... **Spring 5.3 October 2020**



<https://spring.io/projects>



Spring Core Modules (1)





Spring Core Modules (2)

- Core and Beans
 - Fundamental parts of the framework
 - Dependency Injection
 - Manages life-cycle of Beans
 - `BeanFactory`
- Context
 - Extends Core and Beans
 - Internalization support
 - Event propagation
 - Resource loading
 - `ApplicationContext`



Spring Core Modules (3)

- Expression Language
 - Querying and manipulating an object graph at run-time
 - Getting property values
 - Property assignment
 - Method invocation
 - Extension of JSP 2.1 Expression Language
- JDBC
 - JDBC Abstraction layer
 - Error code and Exception handling
 - Tedious code elimination
 - `JdbcTemplate`



Spring Core Modules (4)

- ORM
 - Integration of popular object-relational mapping APIs
 - Hibernate and JPA support
 - `HibernateTemplate`
- OXM
 - Abstraction layer for Object/XML mapping
 - Supports JAXB, Castor, XMLBeans, JiBX and Xstream
- JMS
 - Producing and consuming messages
- Transaction
 - Programmatic and declarative transactions
 - JTA Support



Spring Core Modules (5)

- Web
 - Spring container initialization support
 - Web related Spring remoting
- Servlet
 - MVC (Model View Controller) implementation for web applications
- Portlet
 - MVC implementation in Portlet environment



Spring Core Modules (6)

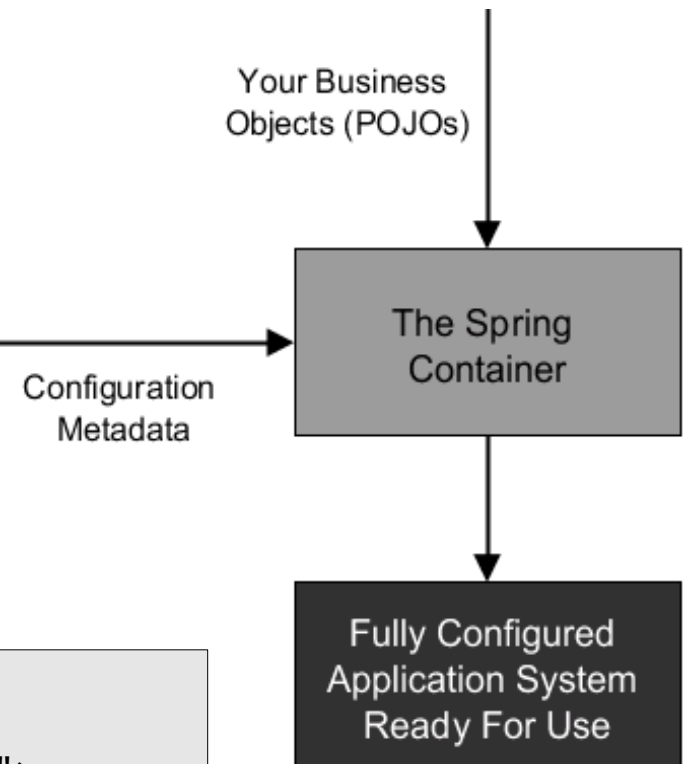
- AOP
 - Aspect Oriented Programming support
- Aspects
 - Integration with AspectJ
- Instrumentation
 - Class instrumentation support
- Test
 - Support of Spring application testing
 - Integration with JUnit and TestNG
 - Application Context loading
 - Provides mock objects



How Spring Works

Your Application Classes

```
public class LibraryServiceImpl implements LibraryService {
    private UserRepository userRepository;
    public LibraryServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    ...
}
public class InMemoryUserRepository implements UserRepository {
    private Map<Long,User> users = new TreeMap<Long,User>();
    public void initialize(){...}
    ...
}
```



Configuration Instructions

```
<beans>
  <bean id="libraryService"
        class="ite.librarymaster.service.LibraryServiceImpl">
    <constructor-arg ref="userRepository"/>
  </bean>
  <bean id="userRepository"
        class="ite.librarymaster.dao.InMemoryUserRepository"
        init-method="initialize"/>
</beans>
```

Container and Beans



Spring Container

- Represented by the `ApplicationContext` interface
- Responsible for
 - Instantiating beans
 - Configuring beans
 - Assembling beans using dependency injection
- Requires configuration meta-data
 - XML configuration files
 - Java annotations
 - Java code
- Several implementation of the `ApplicationContext`



Instantiating a Container

- Load configuration from XML file

```
ApplicationContext context =  
    new ClasspathXmlApplicationContext("application-config.xml");
```

- Look up application service

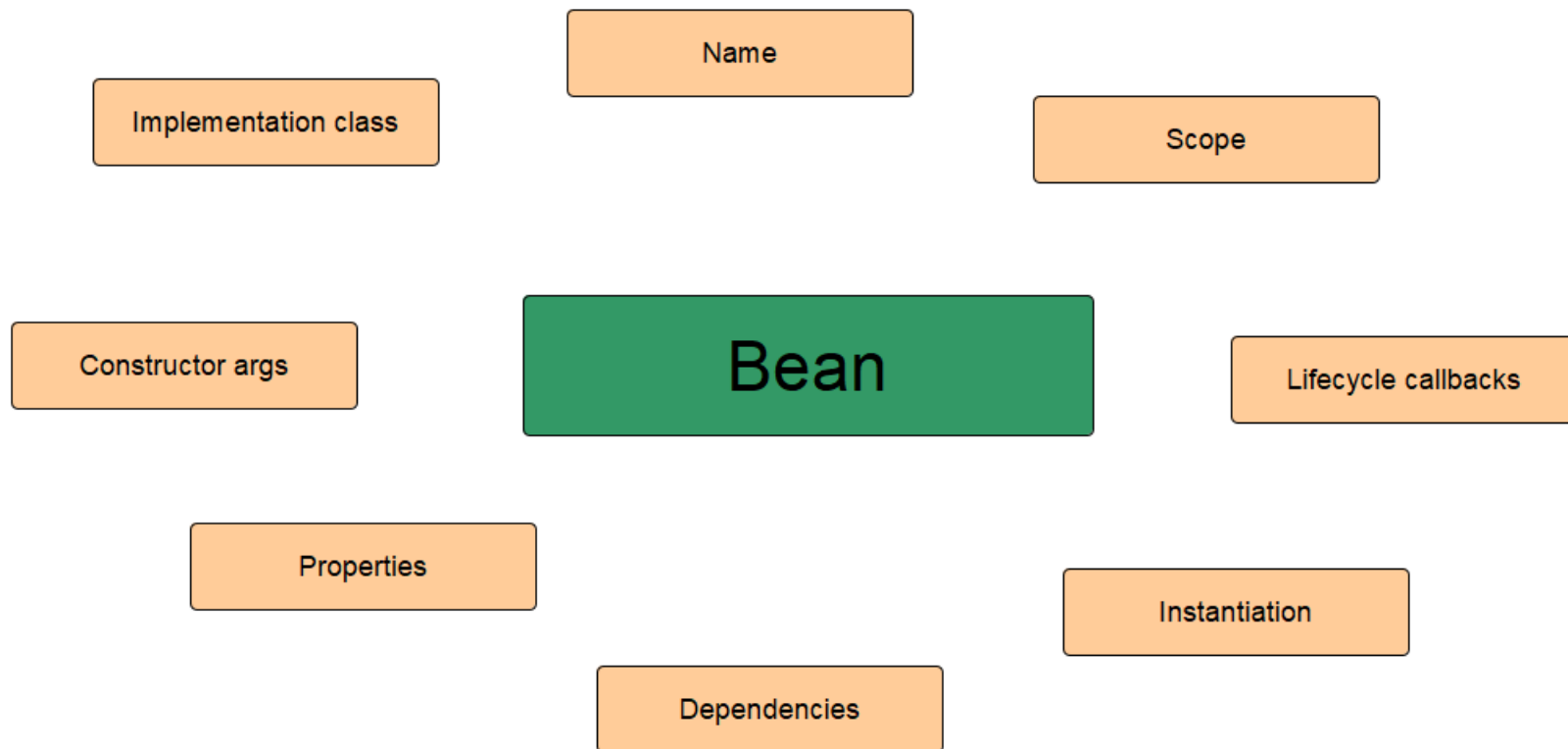
```
LibraryService libraryService = context.getBean(LibraryService.class);  
m2p01-container-lab
```

- Use application service

```
List<Book> allBooks = libraryService.getAllBooks();
```



Bean





Spring Beans

- Objects instantiated, and managed by Spring container
- Beans are defined in container configuration meta-data
- Bean is represented by `BeanDefinition` within container
 - Bean implementation class
 - Bean behavioral configuration (scope, lifecycle callbacks, ...)
 - Bean collaborators and dependencies
 - Other properties
- Bean has unique identifier/name
 - id or name provided in configuration meta-data, used in references
 - If not specified container creates unique bean identifier for you



Instantiating Beans (1)

- Constructor instantiation
- Static factory method

```
<bean id="clientService"  
      class="examples.ClientService"  
      factory-method="createInstance"/>
```

```
public class ClientService {  
    private static ClientService clientService = new ClientService();  
    private ClientService() {}  
  
    public static ClientService createInstance() {  
        return clientService;  
    }  
}
```



Instantiating Beans (2)

- Instance factory method

```
<bean id="clientServiceFactory"
      class="example.ClientServiceFactory"/>
<bean id="clientService"
      factory-bean="clientServiceFactory"
      factory-method="getNewInstance"/>
```

```
public class ClientServiceFactory {

    public ClientService getNewInstance() {
        return new ClientService();
    }
    ...
}
```




Instantiating Beans (3)

- Spring `FactoryBean` interface
 - Factory beans (`FactoryBean` implementers) are auto-detected
 - Dependency injection using factory bean id causes `getObject()` to be invoked transparently

```
<bean id="clientService" class="example.ClientServiceFactoryBean"/>
<bean id="controller" class="example.MyController">
  <property name="service" ref="clientService" />
</bean>
```

```
public class ClientServiceFactoryBean implements FactoryBean<ClientService> {

    public ClientService getObject() throws Exception {
        return new ClientService();
    }

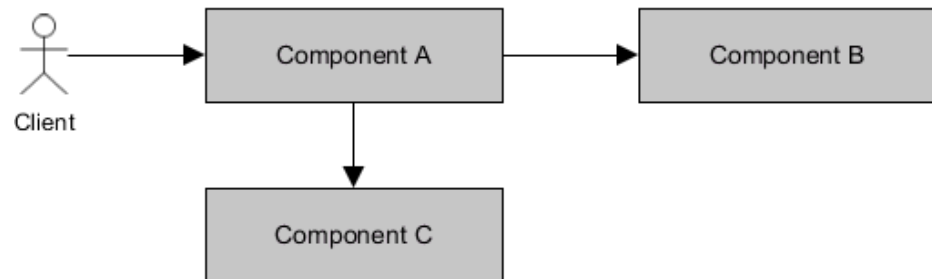
    public Class<?> getObjectType(){
        return ClientService.class;
    }

    ...
}
```



Dependency Injection

- A typical application consists of several parts working together to carry out a use case
- Instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked
 - Drives Application Design
 - Simplifies Application Configuration
 - Reduces Glue/Plumbing Code
 - Improves Testability





Dependency Injection Types (1)

- Constructor based dependency injection
 - Constructor argument's type dependency resolution
 - You can use argument names, types and indexes

```
public class LibraryServiceImpl implements LibraryService {  
    private UserRepository userRepository;  
    public LibraryServiceImpl(UserRepository userRepository) {  
        this.userRepository = userRepository;  
    }  
    ...  
}
```

- Setter based dependency injection
 - Class parameter setter is called after constructor

```
public class LibraryServiceImpl implements LibraryService {  
    private BookRepository bookRepository;  
    public void setBookRepository(BookRepository bookRepository) {  
        this.bookRepository = bookRepository;  
    }  
    ...  
}
```



Dependency Injection Types (2)

- Field based dependency injection
 - Private field injection support
 - Spring uses Java reflection
 - You do not need setter or constructor but,
 - don't work on final fields (immutability)
 - class coupled with DI container, can be harder to test
 - hidden dependencies

```
public class LibraryServiceImpl implements LibraryService {  
    @Autowired  
    private UserRepository userRepository;  
    ...  
}
```



Dependency Injection Types (3)

- You can mix DI styles
- Use constructor DI for mandatory dependencies
 - Promotes immutability (final fields)
- Use setter DI for optional dependencies
 - Inherited automatically
- Concrete type DI used depends on use case

<https://dzone.com/articles/spring-di-patterns-the-good-the-bad-and-the-ugly>



Phases of the Application Lifecycle

- The initialization phase
- The use phase
- The destruction phase





Initialization Phase

- Prepare for use
- Application services
 - Are created
 - Configured
 - May allocate system resources
- Application is not usable until this phase is complete



Use Phase

- Used by clients
- Application services
 - Process client requests
 - Carry out application behaviors
- Most of the time is spent in this phase



Destruction Phase

- Shuts down
- Application services
 - Release any system resources
 - Are eligible for garbage collection



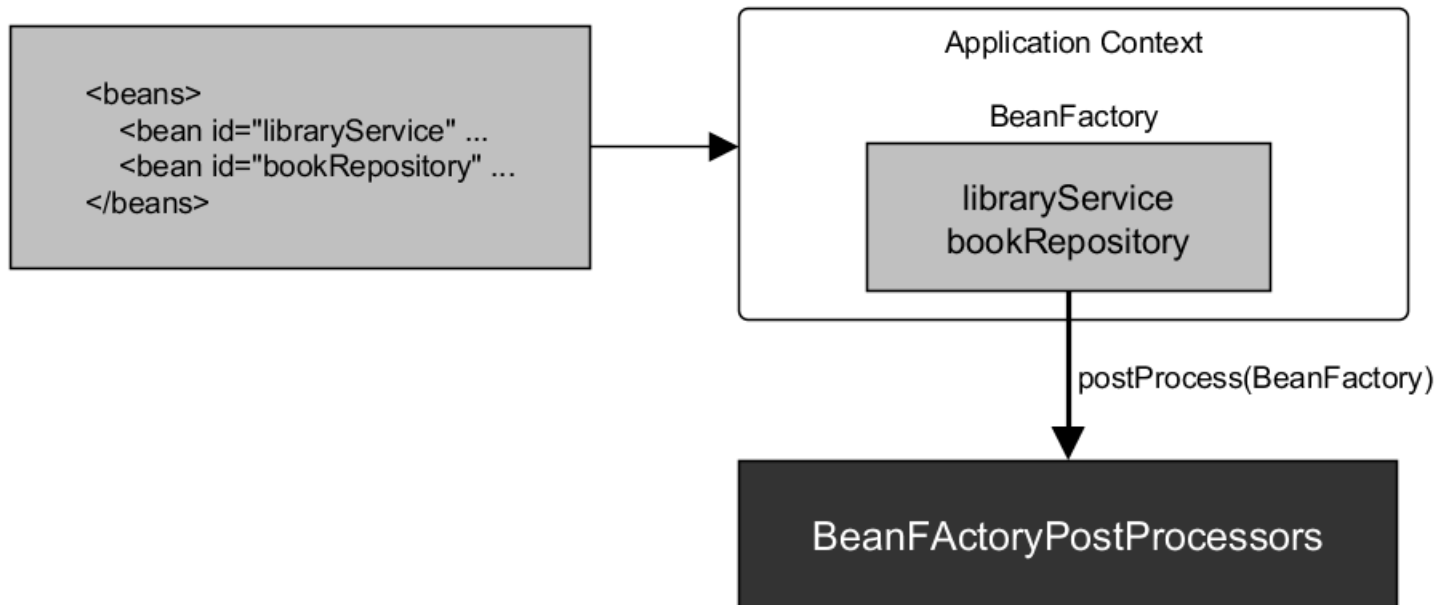
Initialization Phase Details

- Initialization phase completes when a context is created
- Spring loads the bean definitions
- Spring initializes bean instances



Load Bean Definitions

- The XML files are parsed (Class-path scanned for `@Configuration` classes)
- Bean definitions are loaded into the context's `BeanFactory`
- Special `BeanFactoryPostProcessor` beans are invoked
 - Can modify the **definition** of any bean before objects are created





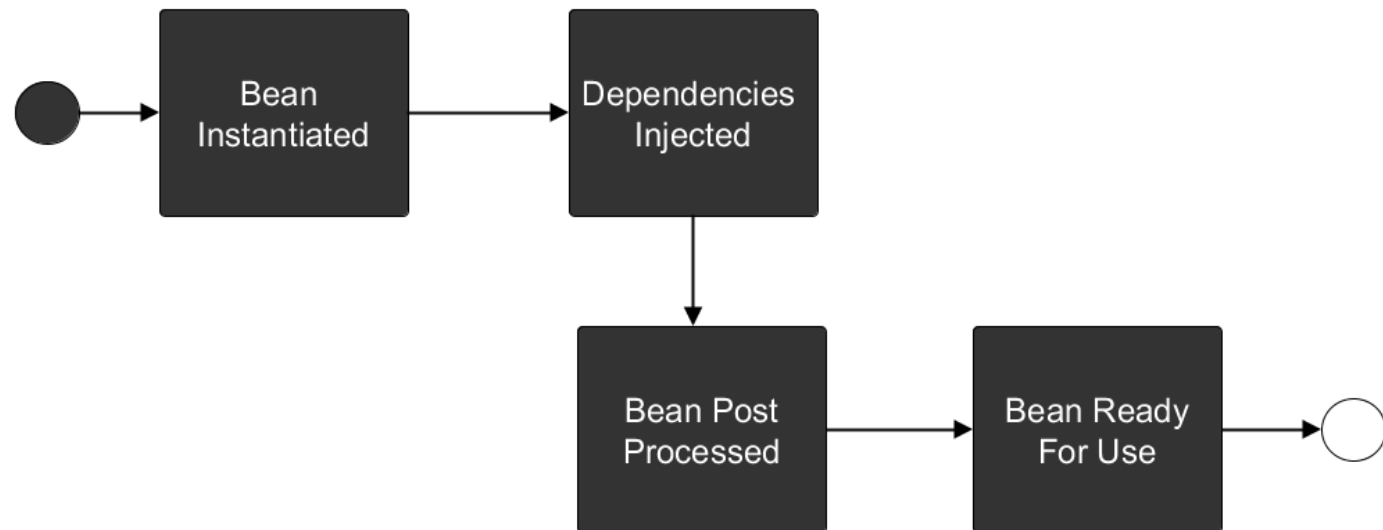
BeanFactoryPostProcessor

- Container extension point
- Useful for applying transformations to groups of bean definitions
 - Before objects are created
- Several useful implementation are provided by the framework
 - `PropertyPlaceholderConfigurer` substitutes `${variables}` in bean definitions with values from `.properties` files
- You can write your own
 - Implement `BeanFactoryPostProcessor` interface



Initializing Bean Instances

- Each bean is eagerly instantiated by default
 - Created in the right order with dependencies injected
- Post-processing phase is invoked
 - Further configuration and initialization
- After post-processing the bean is fully initialized and ready for use





Lazy Initialized Beans

- Singleton beans are eagerly created and configured as part of application context initialization process
 - Errors are discovered immediately
- What about beans which are expensive to create ?
- You can instruct container to lazy initialize bean
 - Bean is created when it is first requested
 - Speeds-up the start-up
 - When lazy-initialized bean is a dependency of not lazy-initialized singleton, the lazy-initialized bean is created at startup

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
```



Autowiring

- Container can auto-wire relationship between collaborating beans
- Use `autowire` attribute of `<bean/>` element in XML-based configuration
- Autowiring modes
 - no
 - byName
 - byType
 - constructor



Autowiring Modes

- **no** - (Default) No autowiring. Bean references must be defined via a ref element
- **byName** - Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired
- **byType** - Allows a property to be autowired if exactly one bean of the property type exists in the container
- **constructor** - Analogous to byType, but applies to constructor arguments



Bean Post Processing Steps

- Initialize the bean if instructed
 - `@PostConstruct` – preferred
 - `<bean/> init-method` attribute – if you can not control init method implementation
 - Implement `InitializingBean` Spring specific interface – ties your code to the framework
- Call special `BeanPostProcessors` to perform additional configuration



The BeanPostProcessor

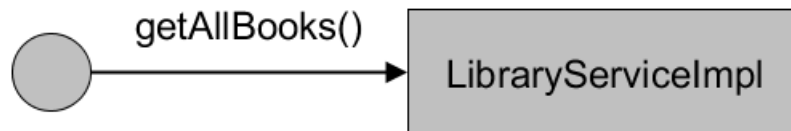
- Container extension point
- Can modify bean **instances**
- Several implementations are provided by the framework
- Not common to write your own

Use Phase Details

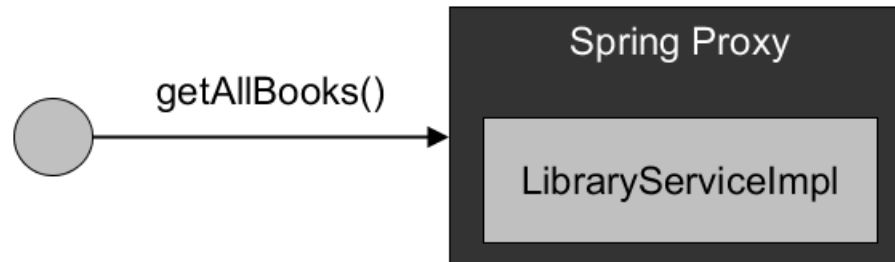
- You invoke a bean obtained from the context

```
ApplicationContext context = new ClasspathXmlApplicationContext("application-config.xml");  
LibraryService libraryService = context.getBean(LibraryService.class);  
List<Book> allBooks = libraryService.getAllBooks();
```

- Objects is simply directly invoked (bean is raw object)



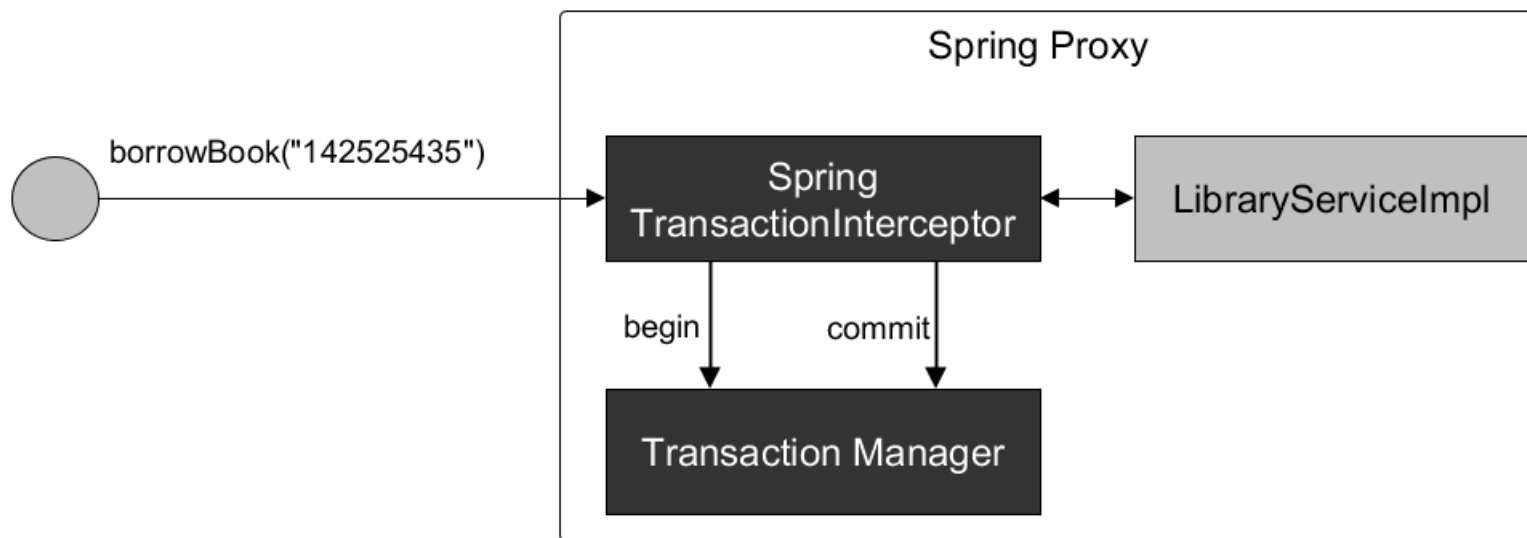
- Bean can be wrapped in a proxy





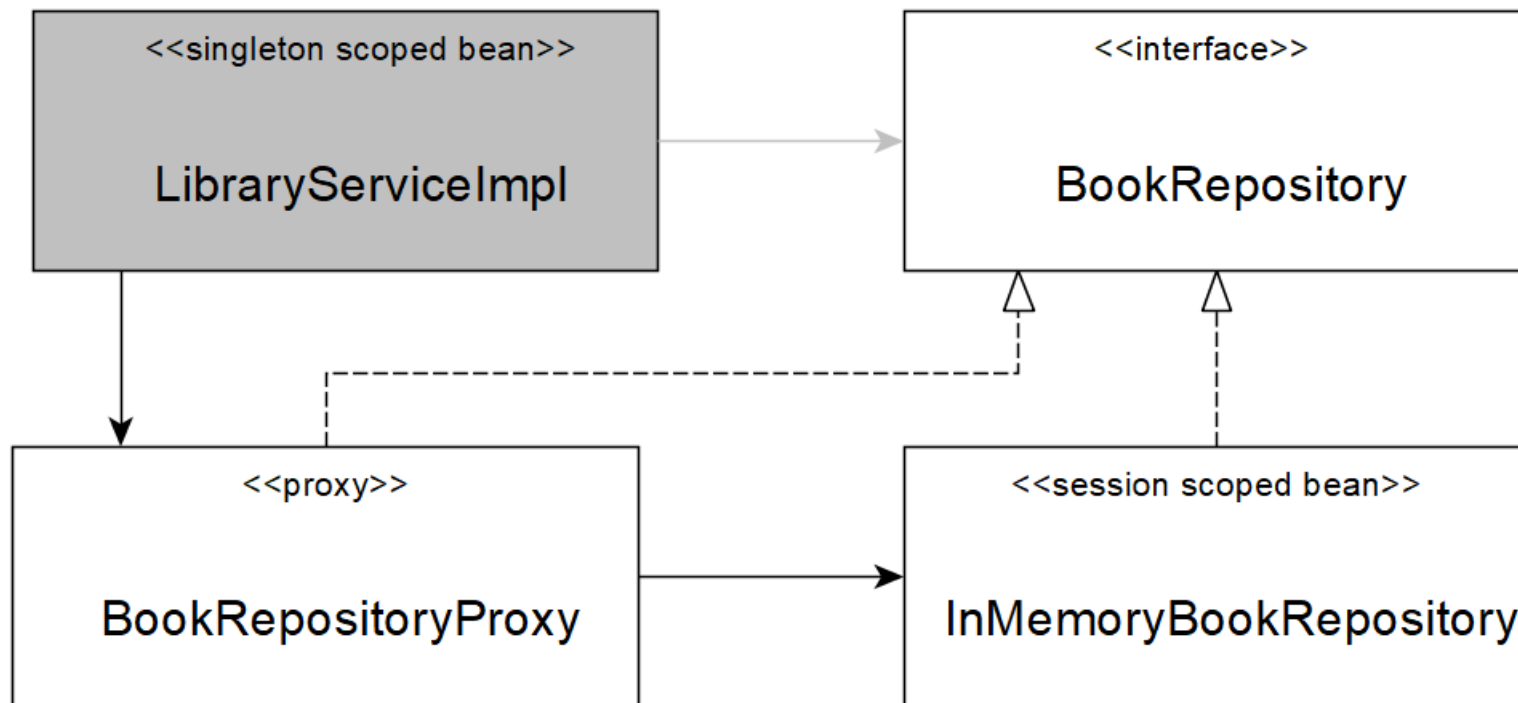
Spring Proxy

- A `BeanPostProcessor` may wrap your beans in a dynamic proxy and add behavior to your application logic transparently





Proxy Pattern





Destruction Phase Details

- Destruction phase completes when you close a context

```
ApplicationContext context = ...  
context.close();
```

- Spring destroys bean instances if instructed
- Spring context itself is destroyed
 - The context is not usable again



Destroy Bean Instances

- A bean can register for one or more destruction callbacks
 - When you need to release resources
- You can use
 - `@PreDestroy` Java annotation – preferred
 - `<bean/>` `destroy-method` attribute - if you can not control destroy method implementation
 - Implement Spring specific `DisposableBean` interface - ties your code to the framework



Bean Scope

- Spring puts each bean instance in a scope
- Singleton scope is the default
- Other scopes can be used by definition

```
<beans>  
  <bean id="someBean" class="example.SomeBeanImpl" scope="...">  
  
    </bean>  
</beans>
```

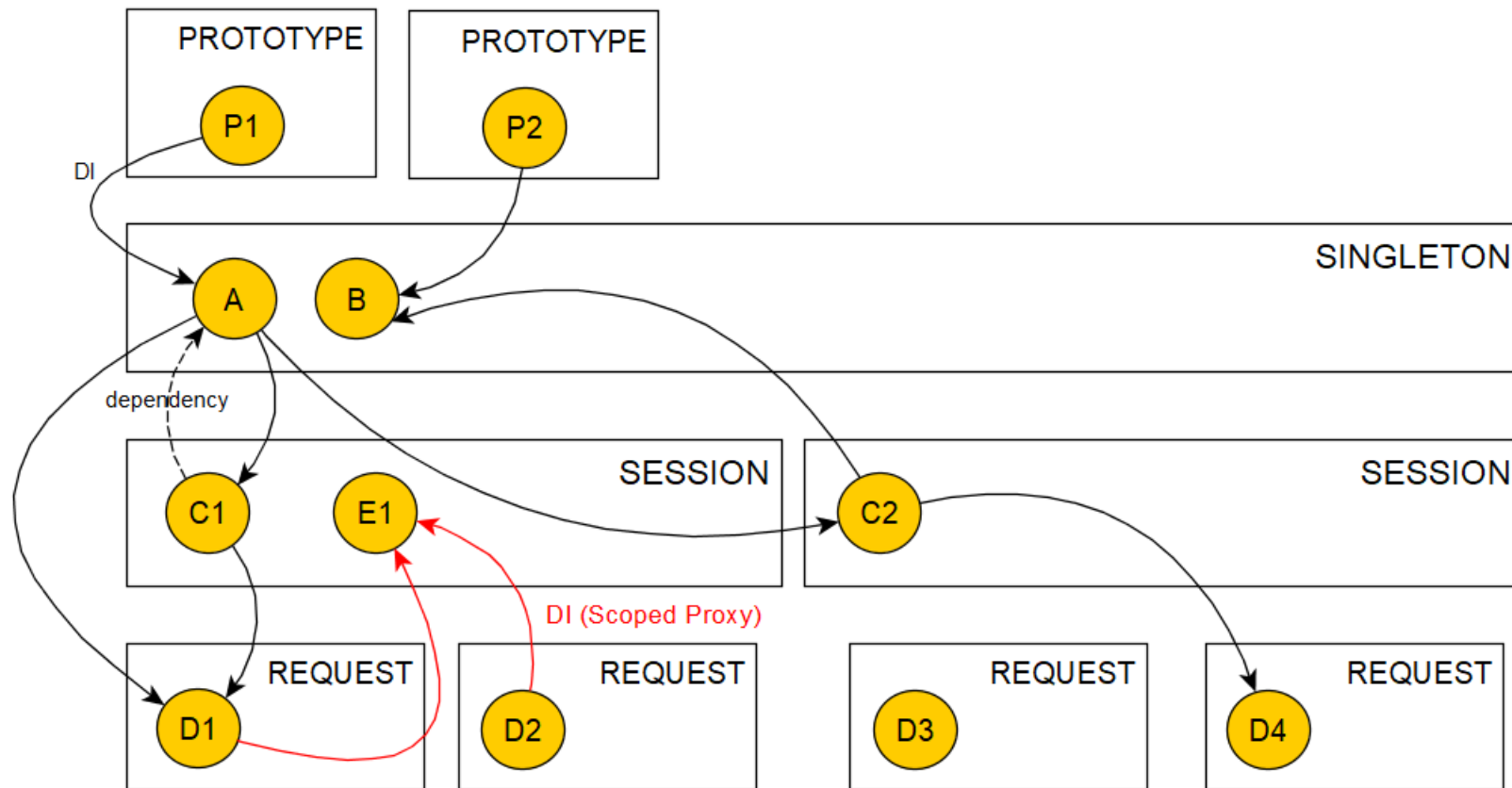



Available Bean Scopes

- **singleton** – Only one shared bean instance is created
- **prototype** – A new instance is created each time the bean is referenced
- **request** - A new instance is created once per request
- **session** - A new instance is created once per user session (HTTP Session of web application)
- **global session** - A new instance is created once per global session (global HTTP Session of portlet-based application)
- **custom** – You can define your own rules



Bean Scopes and DI





One-time Injection

- Request, Session or custom scoped beans often acts as dependency of singleton

```
<bean id="libraryService" class="LibraryServiceImpl">  
  <property name="borrowings" ref="borrowings"/>  
</bean>  
<bean id="borrowings" class="Borrowings" scope="session"/>
```

- Injection occurs only once during start-up
- Service would use the same Borrowings every time
 - You will get error, if there is no HTTP Session
- Solution is to use proxy scoped dependency



Scoped Proxy

- Proxy delegates to correct instance of current request, session, or custom context
- Same proxy can be used by singleton for its entire lifecycle
- Built-in feature of Spring using aop namespace

```
<bean id="libraryService" class="LibraryServiceImpl">  
    <property name="borrowings" ref="borrowings"/>  
</bean>  
<bean id="borrowings" class="Borrowings" scope="session">  
    <aop:scoped-proxy/>  
</bean>
```



Annotation Based Configuration

XML based dependency injection

```
<bean id="libraryService"
      class="ite.librarymaster.service.LibraryServiceImpl">
  <constructor-arg ref="userRepository"/>
</bean>
```

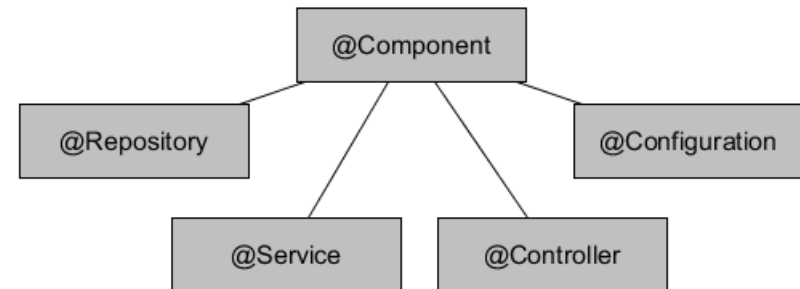
Annotation based dependency injection

```
@Component
class LibraryServiceImpl implements LibraryService {
  public private UserRepository userRepository;
  @Autowired
  public LibraryServiceImpl(UserRepository userRepository) {
    this.userRepository = userRepository;
  }
  ...
}
```



Component Scanning

- Base package with sub-packages are scanned at start-up
- All classes annotated with Spring stereotype annotations are detected and loaded as Spring beans
 - @Component
 - @Resource
 - @Service
 - @Controller, @RestController
- You can name component using annotation parameter
 - Definition @Service("libraryService")
 - Usage @Qualifier("libraryService")





@Autowired

- @Autowired indicates that Spring should inject dependency
 - Based on type (default)
 - Based on dependency name (@Qualifier annotation)

```
@Autowired  
public LibraryServiceImpl(UserRepository userRepository) {  
    this.userRepository = userRepository;  
}
```

```
@Autowired  
public LibraryServiceImpl(@Qualifier("userRepository")  
                        UserRepository userRepository) {  
    this.userRepository = userRepository;  
}
```

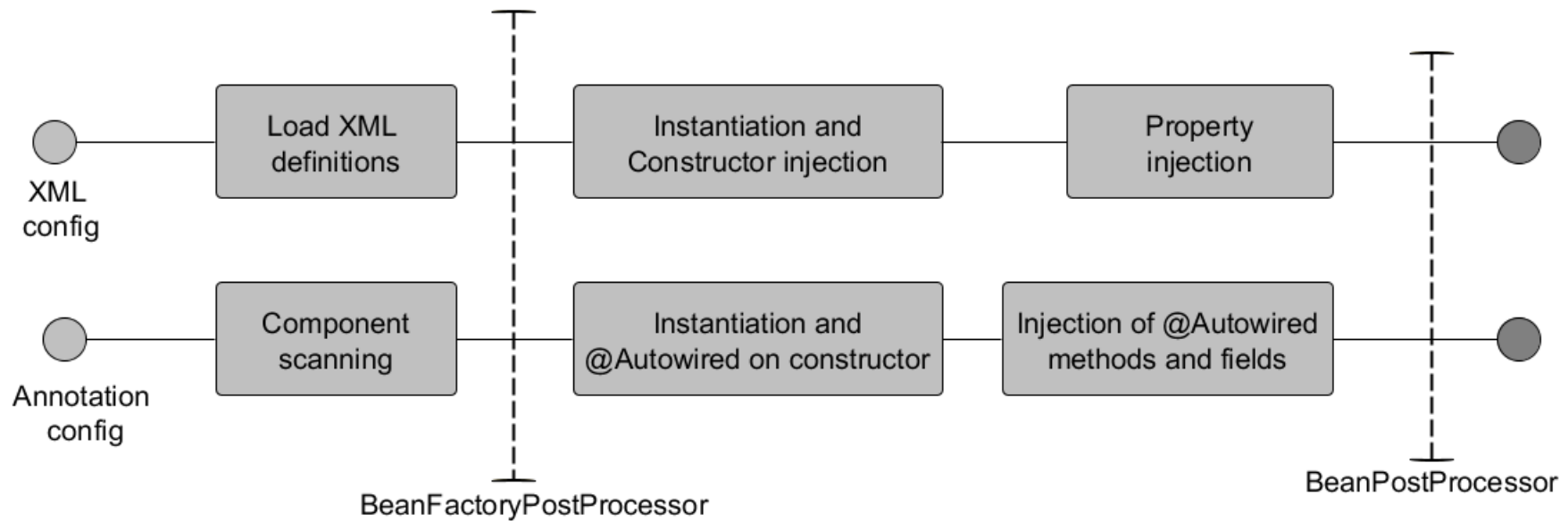


Usage of @Autowired

- You can annotate
 - Constructor for constructor type injection
 - Method for method type injection
 - Field for field type injection
- Use @Qualifier annotation to inject unique Spring bean in case of ambiguity



Initialization Sequence

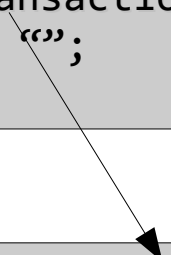




Meta Annotations

- Annotations which annotate other annotations
- You can create your own stereotypes
 - Example: All services should be transactional

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Service
@Transactional(timeout=60)
public @interface TransactionalService{
    String value default "";
```



```
@TransactionalService
public class LibraryServiceImpl implements LibraryService{ ... }
```



@Value

- You can use @Value for SpEL expressions in Java code
 - Usage on a field, method, or parameter of autowired constructor or method

```
@Value("#{systemProperties['user.region']}")
private String defaultLocale;

@Value("#{systemProperties['user.region']}")
public void setDefaultLocale(String defaultLocale){...}

public void configure(BookRepository bookRepository,
@Value("#{systemProperties['user.region']}") String defaultLocale){...}
```



Java Based Configuration

- You can use Java-based configuration to configure container
 - @Configuration annotated classes are the key artifacts
 - @Bean methods instead of <bean/> elements
 - @Bean indicates that a method instantiates, configures and initializes a new object to be managed by the Spring container
 - More control over bean instantiation and configuration
 - More type-safety



@Configuration Example

- Configuration class must have default constructor
 - Multiple @Configuration classes may exist
 - @Bean methods may not be private

```
@Configuration
public class LibraryConfig {

    @Bean
    public LibraryService libraryService() {
        LibraryServiceImpl service = new LibraryServiceImpl();
        service.setBookRepository(bookRepository());
        return service;
    }

    @Bean
    public BookRepository bookRepository() { ... }
    ...
}
```

Method produces a bean definition

method name taken as bean id

Dependency injection



Enabling @Configuration

1. Use component scanning (processes @Configuration)

```
<context:component-scan base-package="ite.librarymaster"/>
```

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");
```

2. Use AnnotationConfigApplicationContext

```
ApplicationContext context =  
    new AnnotationConfigApplicationContext(LibraryConfig.class);
```

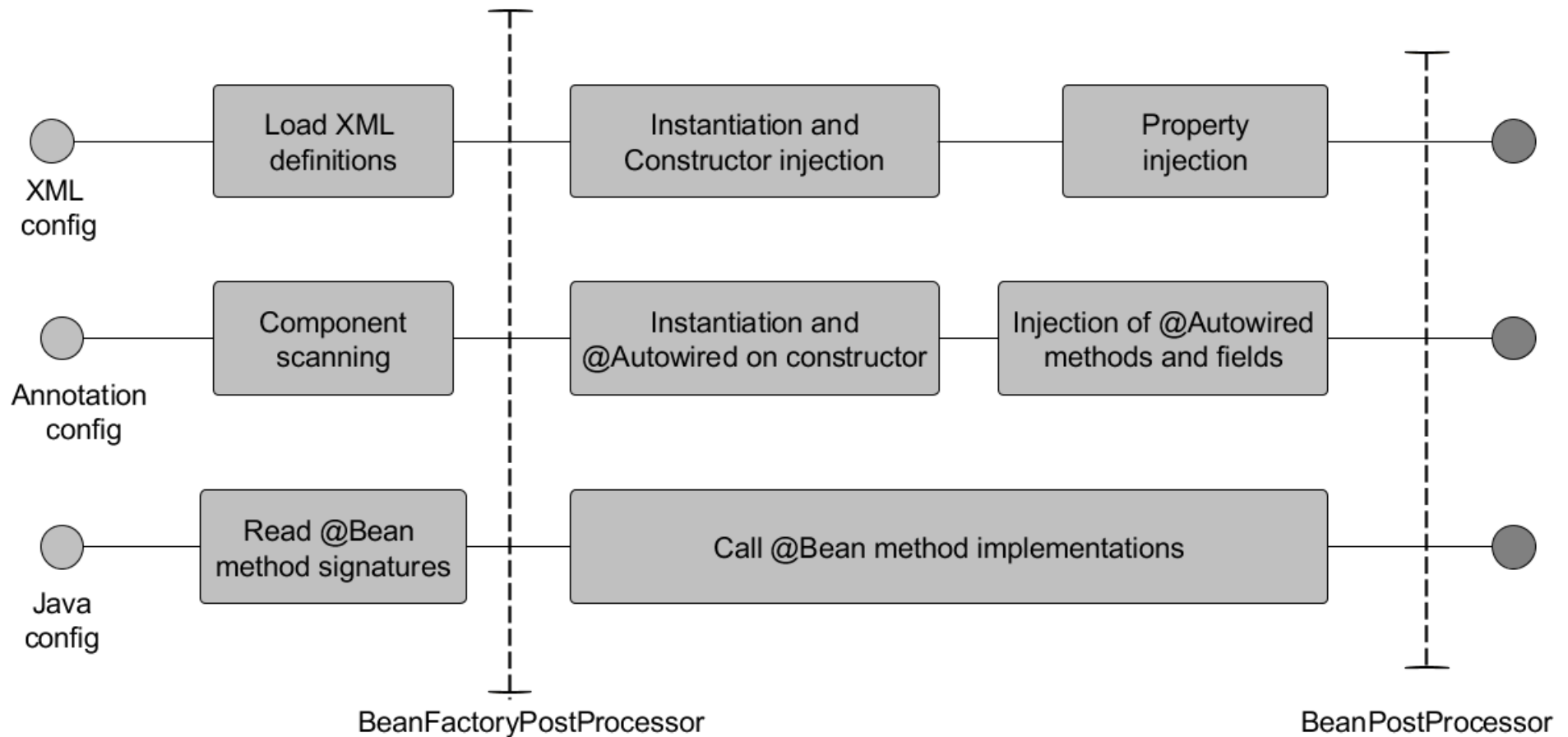


@Configuration Details

- Spring creates singletons out of *@Bean* annotated methods
- *@Scope* annotation can change scope:
 - *@Bean @Scope("prototype")*
- Singletons are ensured by configuration class Proxy, which stores created beans in *ApplicationContext*



Initialization Sequence





Mixing Configuration Styles

- Application can mix injection styles
- XML
 - No major technical limitations but more verbose
- Annotation
 - You can only annotate the classes you write
- Java
 - Does not have namespaces



XML Configuration Usage Pros/Cons

- For infrastructure and more 'static' beans
- Pros:
 - Is centralized in one or a few places
 - Can be used for all classes (3rd party classes)
 - Most familiar configuration format for most people
- Cons:
 - Limited configurations options
 - Can lead to XML configuration hell
 - Some people just don't like XML



Annotations Configuration Usage Pros/Cons

- For frequently changing beans
- Pros:
 - Single place to edit and maintain (Java class)
 - Allows rapid development
- Cons:
 - Configuration spread across your code
 - Only works for your own code



Java Configuration Usage Pros/Cons

- For full control over instantiation and configuration
 - Alternative to implementing FactoryBean
- Pros:
 - Configuration still external to code
 - Integrates well with legacy code
- Cons:
 - Not visible in development tool (STS)
 - No equivalent to the XML namespaces



References

- Spring IO
 - <https://spring.io/>
- Spring tool Suite development tool
 - <https://spring.io/tools/sts>
- Spring Initializer
 - <https://start.spring.io/>
- Spring Documentation
 - <https://spring.io/docs>