Web Application
with
Spring IO Platform

ivan.macalak@posam.sk

Workshop Agenda

- Introduction to Spring Framework
  - Application Configuration
  - Spring Beans
  - Dependency Injection
- Web Applications
  - Spring MVC
- Data Access with Spring
  - Transactions Support
- REST Web services

Module 1
Spring Framework Core Concept
IoC Container and Dependency Injection

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**
  - JEE

Introduction (2)

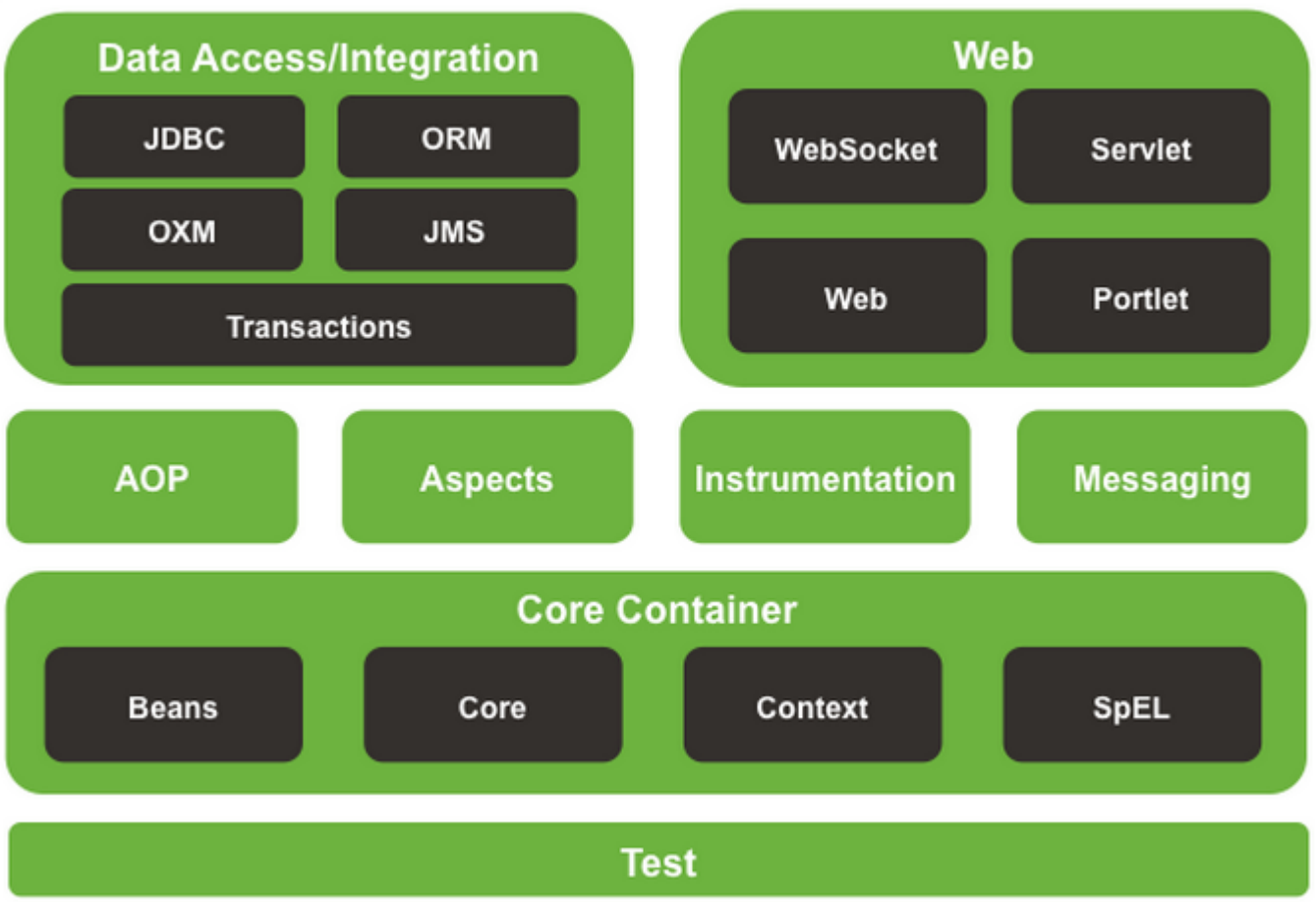
- Enables application creation by loosely coupled building blocks
- You can add enterprise services to your POJO application
- Spring codifies formalized design patterns as first-class objects
- You can integrate these objects into your own application
- Originally created to address the complexity of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

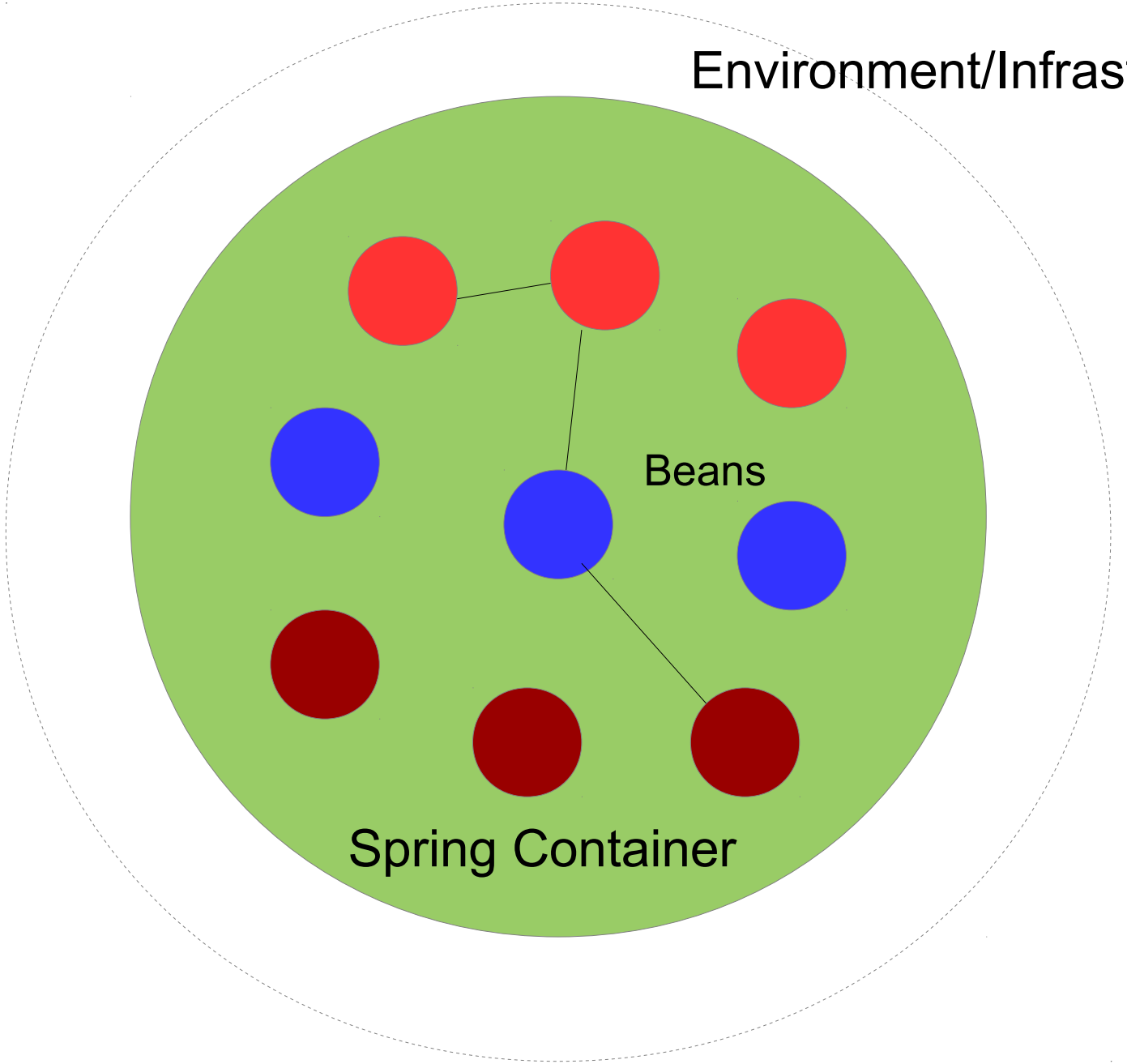
Spring History

- 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
- Today
  - Spring 4.2.2 is the current version released in October 2015
    - Java SE8 support, WebSockets support (spring-websocket), HTTP Streaming and Eventing, CORS support, HtmlUnit tests and more ...

Spring IO Platform

Spring Framework

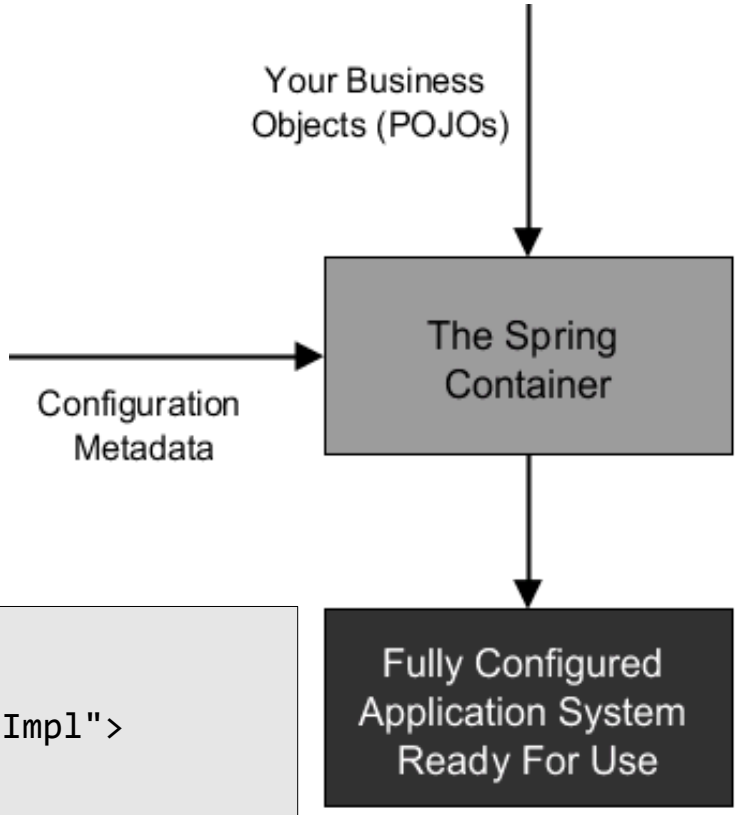



Environment/Infrastructure
Beans
Spring Container

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>
```

Spring Container

- Represented by `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 `ApplicationContext`

Spring Beans

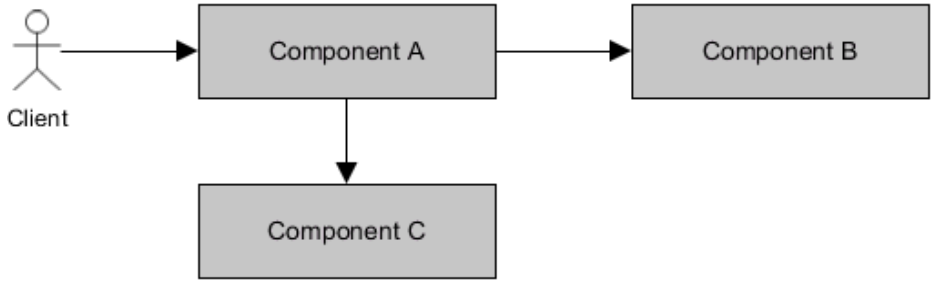
- Objects instantiated, and managed by Spring container
- Beans are defined in container configuration meta-data
- Bean is represented by `BeanDefinition` within container
  - Bean implementation class
  - Bean behavioral configuration (scope, lifecycle callbacks, ...)
  - Bean collaborators and dependencies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

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

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

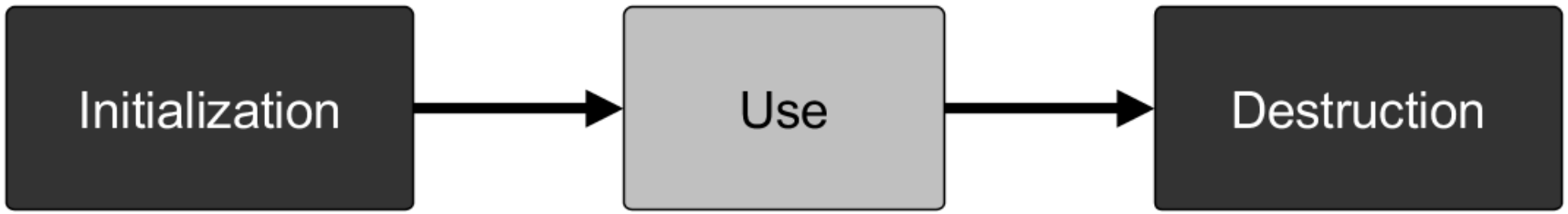
```
public class LibraryServiceImpl implements LibraryService {
    private UserRepository userRepository;
    public LibraryServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
…
}
```

- 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;
    }
…
}
```

Phases of the Application Lifecycle

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



Application Initialization Phase

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

Application Use Phase

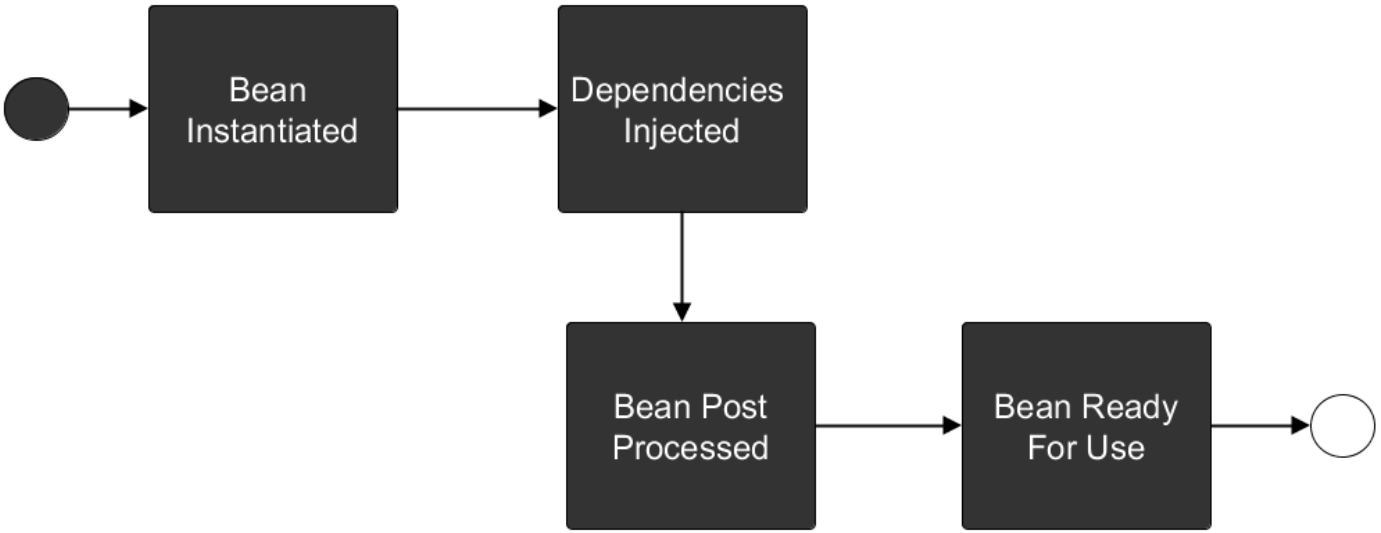
- Used by clients
- Application services
  - Process client requests
  - Carry out application behaviors
- Most of the time is spent in this phase

Application Destruction Phase

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

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



Autowiring

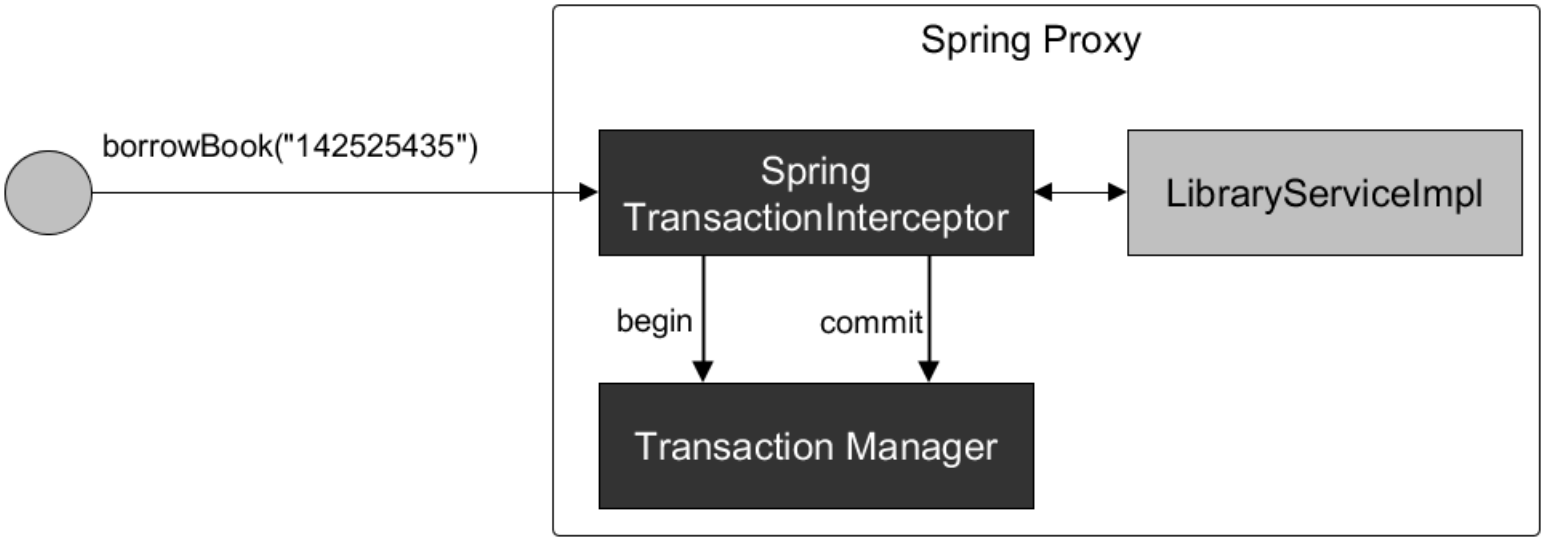
- Container can auto-wire relationship between collaborating beans
- Use `autowire` attribute of <bean/> element in XML-based configuration
- Autowiring modes
  - no
  - byName
  - byType
  - constructor

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

Spring Proxy

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



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

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>
```