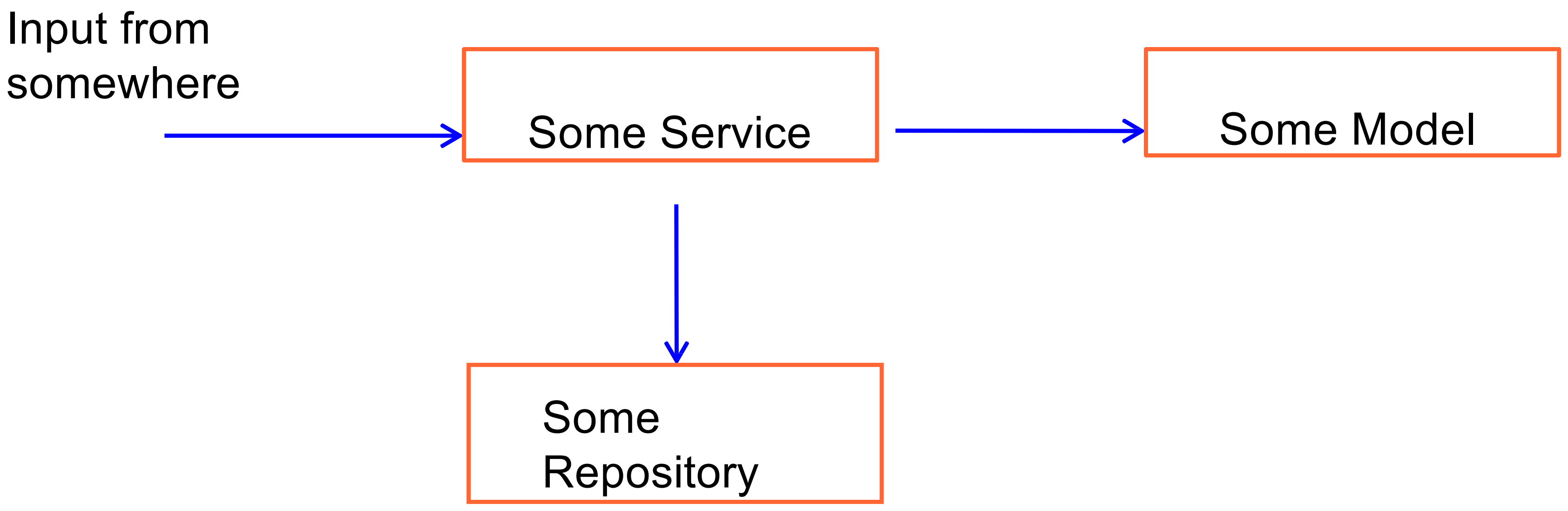
Instantiation and Configuration of Spring

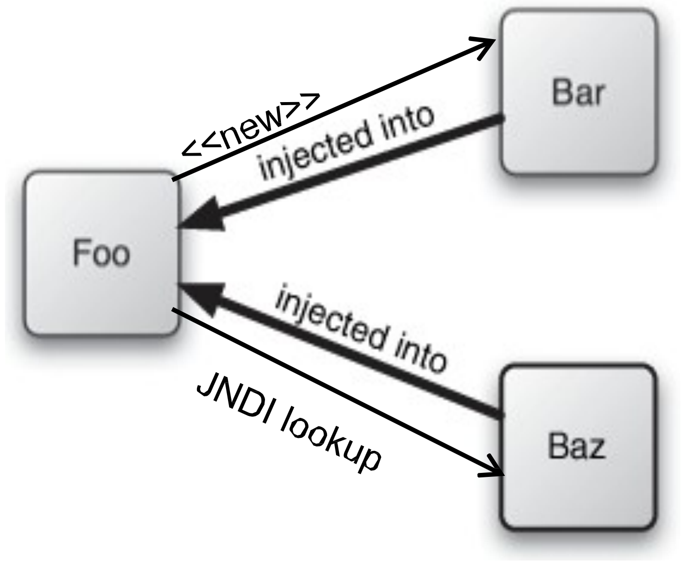
Typical Application Layout



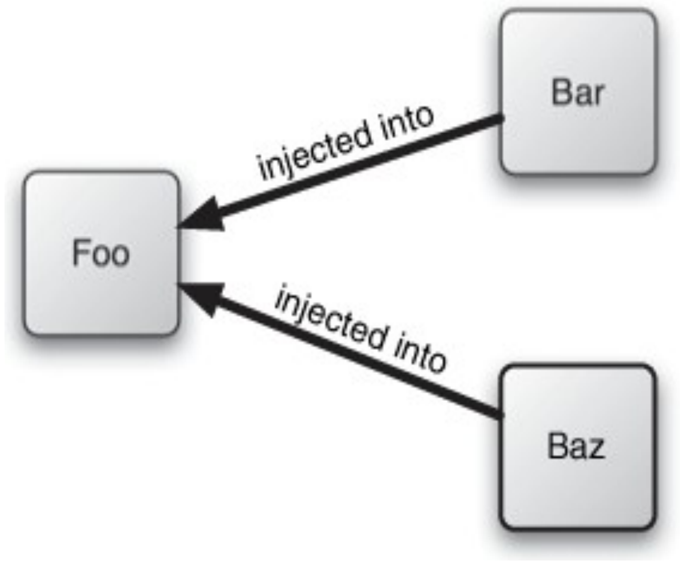
- What the repository, model, or service actually are is not important

Dependency Injection

- Design pattern to reduce coupling between components
- Objects are coupled to interfaces



Traditional Dependency Management



Dependency Injection

Inversion of Control (IoC)

- Design pattern closely related to dependency injection
- Object defines its dependencies and container handles injecting those dependencies when object is created
- Manages
  - Bean lifecycles
  - Object pooling
  - Bean dependencies via injection

Dependency Configuration

- Three mechanisms
  - Externalized in XML file
  - Internalized as annotations
  - Independent Java class
- There can be a mix-and-match of these
- Usually uses 3 files
  - Managed bean
  - Interface to the managed bean
  - Configuration file

Managed Bean

- POJO designed to be managed by Spring
- Does not necessarily follow rules of JavaBeans
- Should always have an interface

```
public class LibraryServiceImpl implements LibraryService{
  private BookRepository repository;

  public LibraryServiceImpl(BookRepository repository){
    this.repository = repository;
  }


  @Override // from interface

  public List<Book> getAllBooks() {
    return repository.getAllBooks();
  }
}
```

Coding Without Spring

```
class NoSpring {
  public static void main(String[] args) {
    Service myService = new ServiceImpl();
    Repository myRepository = new RepositoryImpl();
    service.setRepository(myRepository);
  }
}
```

- What is wrong with this?
  - Technically, nothing
  - But from a Spring standpoint?

Coding With Spring

- Service and repository are the same
  - Add interfaces if they don't exist
- `main` changes
- Configuration file is added
  - `application-config.xml`
  - Java configuration
  - Annotations (requires one of the above)

Main Class

```
public class MainApplication {
  public static void main(String[] args) {
    ApplicationContext appContext = new
      ClassPathXmlApplicationContext("resources/application-
      config.xml");

    LibraryService service = appContext.getBean("library",
      LibraryServiceImpl.class);
    Book book = (Book)service.getBook("An Artificial Night");
  }
}
```

Using `application-config.xml`

```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.xsd">

  <bean id="library" class="com.example.LibraryServiceImpl">
    <constructor-arg ref="bookSource"/>
  </bean>
  <bean id="bookSource" class="com.example.BookSourceImpl"/>

</beans>
```

Linking Beans Together

- Using Constructors and XML
  - `<constructor-arg ref="bookSource" />`
  - `public LibraryServiceImpl(BookSource source)`
- Using set methods and XML
  - `<property name="bookSource"/>`
  - `public void setBookSource(BookSource source)`

Using constructor-arg

```
<bean id="library" class="com.example.LibraryServiceImpl">
  <constructor-arg ref="bookSource"/>
</bean>
<bean id="bookSource" class="com.example.BookSourceImpl"/>
```

Would go with:

```
public LibraryServiceImpl(BookRepository repository){
  this.repository = repository;
}
```

Using `property`

```
<bean id="library" class="com.example.LibraryServiceImpl">
  <property name="bookSource" ref="bookSourceRef"/>
</bean>
<bean id="bookSourceRef" class="com.example.BookSourceImpl"/>
```

Would go with:

```
public LibraryServiceImpl(){ }

public void setBookSource(BookSource repository){
this.repository = repository;
}
```

Using Annotations for Configuration

```
@Component
public class LibraryServiceImpl implements LibraryService {
@Autowired
public LibraryServiceImpl(BookSource source){
    this.source = source;
}
}
```

```
<bean id="library" class="com.example.LibraryServiceImpl">
<constructor-arg ref="bookSource"/>
</bean>
```

Why Another Configuration Method?

- Removes all XML requirements
  - Can still mix-and-match if you chose
- Removes typos
- Removes runtime time checks
- Normal Java class with same benefits of only using Java classes

@Configuration

- Annotates a Java class to be the mechanism of bootstrapping
  - In comparison to XML and Annotation methods we have seen
- Can have multiple files annotated, in which all would become configuration files

@Bean

- Defines the methods that will return our beans
  - These are the same classes that would have used the `<bean>` tag in XML configuration
- Can have multiple beans in a file

Using Java Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class JavaConfig {

  @Bean
  public LibraryService libraryService() {
    return new LibraryServiceImpl(bookSource());
  }

  @Bean
  public BookSource bookSource() {
    return new BookSourceImpl();
  }
}
```

ApplicationContext

- Usually use `org.springframework.context.ApplicationContext`
  - Child of `org.springframework.beans.factory.BeanFactory`
- Common implementations
  - `ClassPathXmlApplicationContext`
  - `AnnotationConfigApplicationContext`

Lab 1 – Hello Spring!

Lab 2 – Phone Book Service

Spring Configuration in XML

Objectives

When we are done, you should be able to:

- Describe what namespaces are
- Use property files
- Understand Spring bean scopes

Prefixes

- Files are located in different places
  - Can use different ApplicationContext objects
    - `ClassPathXmlApplicationContext`
    - `FileSystemXmlApplicationContext`
  - Can use prefixes
    - `classpath`
    - `file`
    - `http`
- Prefixes are used anywhere Spring deals with resources

Prefixes [cont.]

- Example:

```
ApplicationContext context = new ClassPathXmlApplicationContext
   ("file:\User\guest\constant.properties");
```

- Can use wildcards
  - `classpath*:conf/*-config.xml`
    - All classpath sources should be searched
  - `classpath:conf/*-config.xml`
    - Only search first classpath found

Scopes

- Technically only 3 scopes
  - **singleton** – One instance of bean per application
  - **prototype** – New instance every time bean is referenced
  - **custom** – Programmer defines the rules
    - This is where **session** and **request** scopes come in
- Default is `singleton`

Scopes [cont.]

In XML:

```
<bean id="library" class="com.example.LibraryServiceImpl"
  scope="singleton">
  <constructor-arg ref="bookSource"/>
</bean>
```

In Java:

```
@Bean
@Scope(value=ConfigurableBeanFactory.SCOPE_SINGLETON
public LibraryService libraryService() {
  return new LibraryServiceImpl(bookSource());
}
```

Namespaces

- XML-schemas that make life easier
- Add access to many classes
- Standard
  - aop – adds aspect oriented programming
  - context – helps with the building of application contexts
  - beans – the main namespace for Spring
  - util – extra utilities, such as collections

Context Namespace

- Primarily provides work to help with building context
- Compilation time checks instead of runtime checks
- `<context:property-placeholder>`
  - Brings in property sheets
- `<context:annotation-config>`
  - Turns on JSR 250 annotation usage
- `<context:component-scan>`
  - Turns on component scanning

Using Property Files

```
<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:property-placeholder location="library.properties"/>
<bean id="library" class="com.example.LibraryServicesImpl">
<property name="greeting" value="${library.greetings}"/>
</bean>
</beans>
```

`library.greetings` is the key in the properties file

Using Property Files [cont.]

```
@Configuration

@PropertySource(value = {"classpath:library.properties"})
public class JavaConfig {

  @Bean
  public static PropertySourcesPlaceholderConfigurer
    propertyPlaceholder() throws IOException {
    return new PropertySourcesPlaceholderConfigurer();
  }
}
// Different file
public class LibraryServiceImpl {
  @Value("${library.greeting}")
  private String greeting;
...
}
```

`library.greetings` is the key in the properties file

Component Scanning

- When using XML, just add
  - `@Component` to the bean class
  - `<context:component-scan>` to the XML file
- When using Java configuration
  - Beans annotated `@Component`
  - Beans not defined in JavaConfig file
  - JavaConfig adds `@ComponentScan`

```
@Configuration
@ComponentScan (basePackages= {"com.example.library.beans"})
  public class JavaConfig {

}
```

Collections

- In JavaConfig, ensure that variable type is that of the interface
- XML understands lists, sets and maps
  - Does not understand specific types
  - Not accessible from `ApplicationContext`
  - Only usable inside of `<property>` or `<constructor-arg>`

```
<bean id="bookSource" class="com.example.BookSourceImpl">
<property name="bookList">
    <list>
        <value>"An Artificial Night"</value>
        <value>"Rosemary and Rue"</value>
    </list>
</property>
</bean>
```

Lab 3 – Using Property Files