Spring - Dag 1

Agenda

Andreas Evers & Ken Coenen

Spring Core

Steve De Zitter

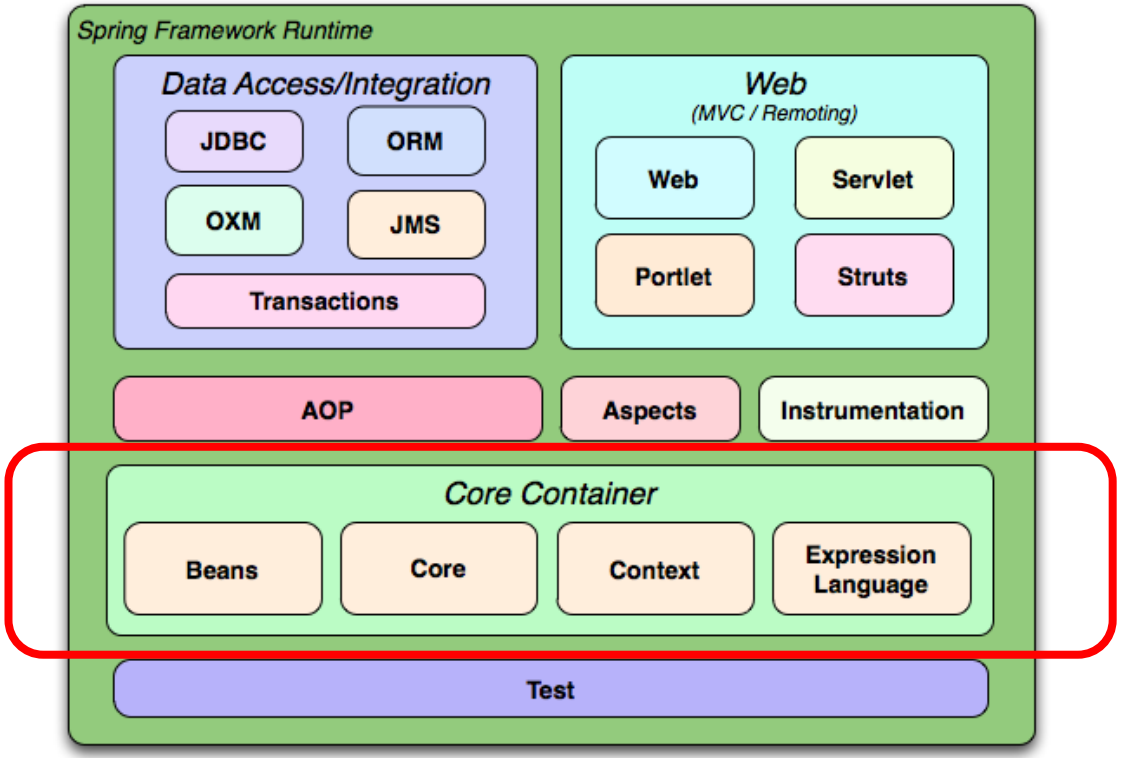
Servlets & Spring MVC

Core Spring

The Spring framework

- Alternative to JEE's EJB model
- Simplify Java enterprise application development
  - JDBC queries, LDAP connections, ...
- Reduce dependencies
  - No dependency cycles in their code

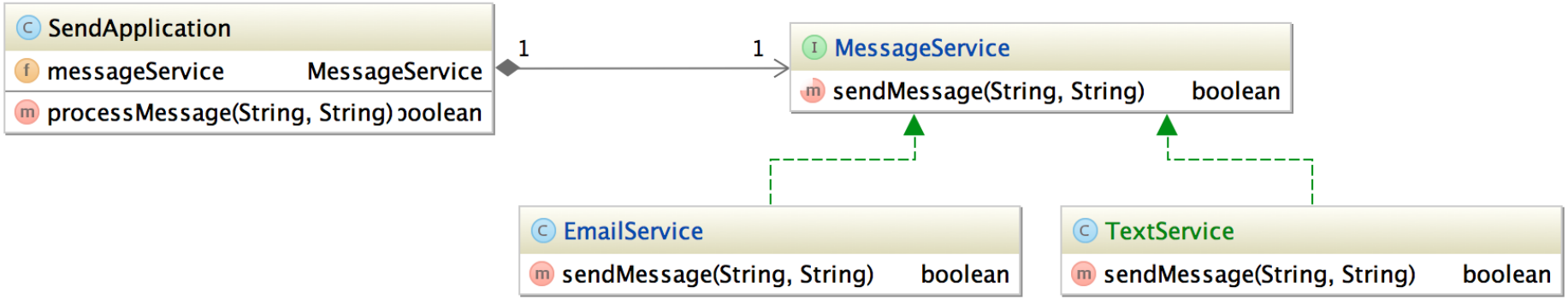
Spring Modules



Polymorphism

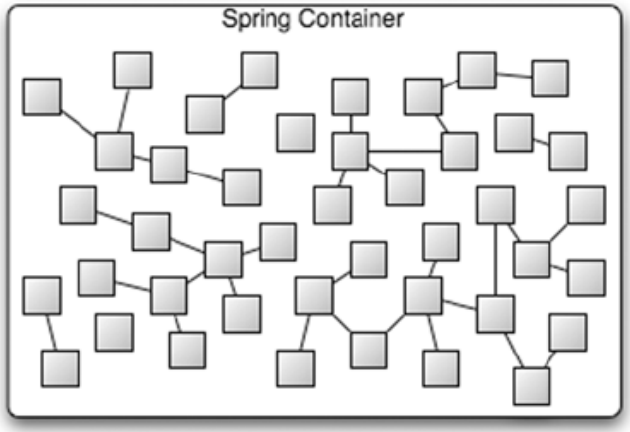
`Example1_Polymorphism.java`

- Design to interfaces
- `SendApplication` does not know the used implementation because it does not manage its own dependencies
- Loose coupling



Spring Container

- Container of **beans**
- Spring manages the **bean lifecycle**



Doing a `new Xyz()` yourself results in an object not managed by Spring

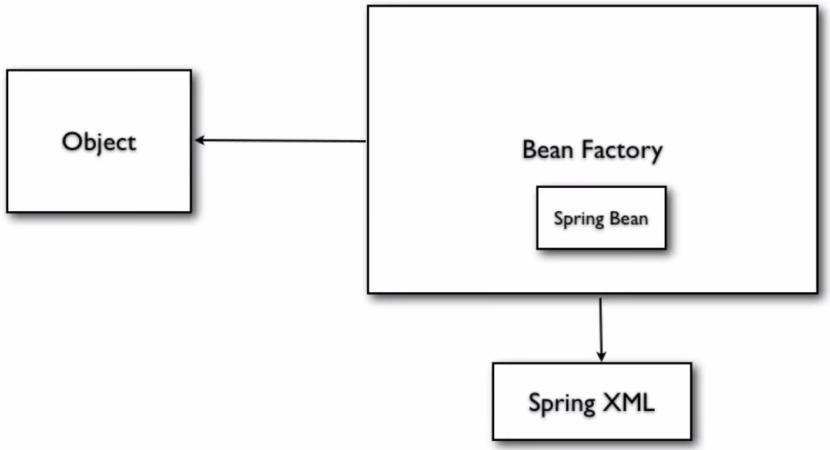
What is a bean?

- Loaded by the Application Context
- Any type of POJO
- No need to implement or extend Spring-specific classes
- Unique name
- Annotated with `@Component`, `@Service`, ...

Spring Bean Factory

`Example2_BeanFactory.java`

- Creates beans based on configuration



Bean initialization (XML)

```
<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=”messageService” class=”be.ordina.spring.service.EmailService” />

</beans>
```

context.xml

Bean Factory vs. Application Context

- Both load beans and wire them together
- Application Context offers much more
  - i18n, AOP, …
- Usually we use `ApplicationContext`

`Example3_ApplicationContext.java`

Bean example

EmailService.java

```
package be.ordina.spring.service;

public interface MessageService {
    boolean sendMessage(String msg, String receiver);
}
```

```
package be.ordina.spring.service;

@Service
public class EmailService implements MessageService {
    public boolean sendMessage(String msg, String rec) {
        System.out.println("Email Sent to "+rec+ " with Message="+msg);
        return true;
    }
}
```

Configure bean properties

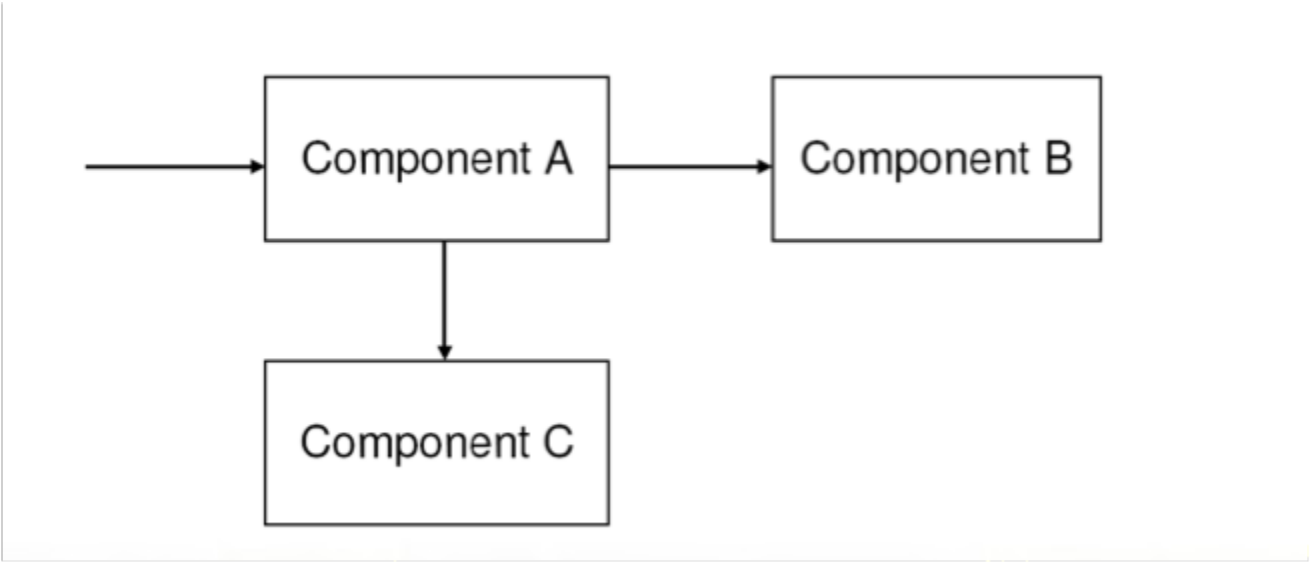
- Initialize a property through the Spring configuration file
  - `<property>`
  - `<constructor-arg>`

`Example4_PropertyInitialization.java`

`Example5_ConstructorInjection.java`

Using dependencies

- A typical application system consists of several parts working together to carry out a use case



Before Dependency Injection

- Explicit initialization
- Results in high coupling between components
- Difficult to test

```
@Component
public class SendApplication {
    private MessageService service;

    public SendApplication(){
        this.service = new EmailService();
    }

    public boolean processMessage(String msg, String rec){
        return this.service.sendMessage(msg, rec);
    }
}
```

Dependency Injection or “wiring”

- Also known as **Inversion of Control (IoC)**
- Classes need to know about each other
- Links beans together
- Results in cleaner code and simplified testing
  - Swap out the dependency with a different implementation
- Field-, setter- or constructor injection
- `@Autowired`

Dependency Injection example

```
<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="sendApplication" class="be.ordina.spring.components.SendApplication">
        <property name="messageService" ref="emailService"/>
    </bean>

    <bean id="emailService" class="be.ordina.spring.service.EmailService">
        <property name="footer" value="Best regards, Ken and Andreas"/>
    </bean>

</beans>
```

Example6_DependencyInjection.java

Injecting Collections

- `List`, `Set` and `Map` are supported

```
<constructor-arg>
    <list>
        <value>Ken</value>
        <value>Andreas</value>
        <value>Dieter</value>
    </list>
</constructor-arg>
```

Example7_Collections.java

Autowiring

Example8_Autowiring.java

- Name
- Type
- Constructor

Bean Scopes

- `scope` attribute on bean element

- **Singleton**
  - Once per Spring container
  - Default (!)
- **Prototype**
  - New object on each `ref` or `getBean`
  - Same as `new Xyz()`
- **Request, Session and Global Session (web scopes)**
  - New bean per request, session or global HTTP session

Bean lifecycle hooks

- **Creation**
  - `InitializingBean` interface and `afterPropertiesSet()`
  - `init-method` attribute in bean definition
  - `default-init-method` attribute in beans definition
  - `@PostConstruct`
- **Destruction**
  - `DisposableBean` interface and `destroy()`
  - `destroy-method` attribute in bean definition
  - `default-destroy-method` attribute in beans definition
  - `@PreDestroy`

InitializingBean and DisposableBean have precedence

BeanPostProcessor and BeanFactoryPostProcessor

- Plug in extra functionality
- Example is the `PropertyPlaceholderConfigurer`

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:email.properties"/>
</bean>
```

Example10_PropertyPlaceholderConfigurer.java

Application Context flavors

- `AnnotationConfigApplicationContext`
- `AnnotationConfigWebApplicationContext`
  - Requires `contextConfigLocation` parameter in `ContextLoader`
- `ClassPathXmlApplicationContext`
- `FileSystemXmlApplicationContext`
- `XmlWebApplicationContext`

Closing an Application Context

- Use `AbstractApplicationContext`
- Call `registerShutdownHook()` for desktop applications to tell Spring it can destory the beans

ApplicationContextAware

- Interface to retrieve the Application Context inside a bean

```
import org.springframework.context.ApplicationContext;

@Service
public class EmailService implements MessageService, ApplicationContextAware {
    private static final Logger logger = LoggerFactory.getLogger(EmailService.class);

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        logger.info("Application context is being set!");
    }
}
```

EmailService.java

XML vs. annotations

- Different ways to configure your Spring application

XML	Annotation

`<bean>`	`@Bean`
	`@Component`
	`@Service`
	`@Repository`

`id=”myComponent”`	`@Component(“myComponent”)`
`scope=”prototype”`	`@Scope("prototype")`
`<qualifier value="myService"/>`	`@Qualifier(“myService”)`

Required

- Used for required dependencies
- Exception at startup if no applicable bean is found
- Verified by a `RequiredAnnotationBeanPostProcessor`

`Example11_Required.java`

Autowired

- Wire dependencies automatically
- Equivalent of XML attribute `autowire="byName"`
- First by `@Qualifier` (if set), then by type, then by variable name
- Realised by a `AutowiredAnnotationBeanPostProcessor`

`Example12_Autowired.java`

Bean initialization (annotations)

- Different ways
  - Component scanning
  - Defining the bean explicitly in the Spring Configuration
  - Beans are loaded in order of dependency graph

```
package be.ordina.spring.config;

@Configuration
@ComponentScan(value={"be.ordina.spring"})
public class MyConfiguration {

    @Bean
    public MessageService messageService(){
        return new EmailService();
    }
}
```

When the context is loaded, the MessageService bean can be used by other components!

With Dependency Injection

```
@Component
public class SendApplication {
    //@Autowired
    private MessageService service;

//  @Autowired
//  public SendApplication(MessageService svc){
//      this.service=svc;
//  }

    @Autowired
    public void setService(MessageService svc){
        this.service=svc;
    }

    public boolean processMessage(String msg, String rec){
        return this.service.sendMessage(msg, rec);
    }
}
```

Field-based DI

Constructor-based DI

Setter-based DI

Running a Spring application (annotations)

```
package be.ordina.spring;

public class RunDependencyInjection {
     public static void main(String... args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(MyConfiguration.class);
        SendApplication app = context.getBean(SendApplication.class);

        app.processMessage("Hi Ordina!", "info@ordina.be");

        //close the context
        context.close();
    }
}
```

Example2_AnnotationConfig.java

Aspect-Oriented Programming (AOP)

- Eliminate repetitive boilerplate code
- Security, logging, transaction management

AOP advice types

- **Around**
  - Most common and powerful
  - Execute code before and after joinpoint
- **Before**
  - Executes before joinpoint, cannot stop execution
- **Throws**
  - Executes code if exception is thrown
- **After return**
  - Executes code after normal joinpoint execution

Resources

- https://github.com/kencoenen/spring-course
- http://javabrains.koushik.org/courses/spring_core