MUFFIN CONFERENCE | MusalaSoft 24.10.2014

Introduction to Spring Framework

Petyo Dimitrov

Agenda

- Purpose and history of the framework
- Spring modules
- Spring core
  1. beans
  2. lifecycle
  3. dependency injection

- Spring Data
- Spring MVC

What is Spring?

What is Spring (continued)

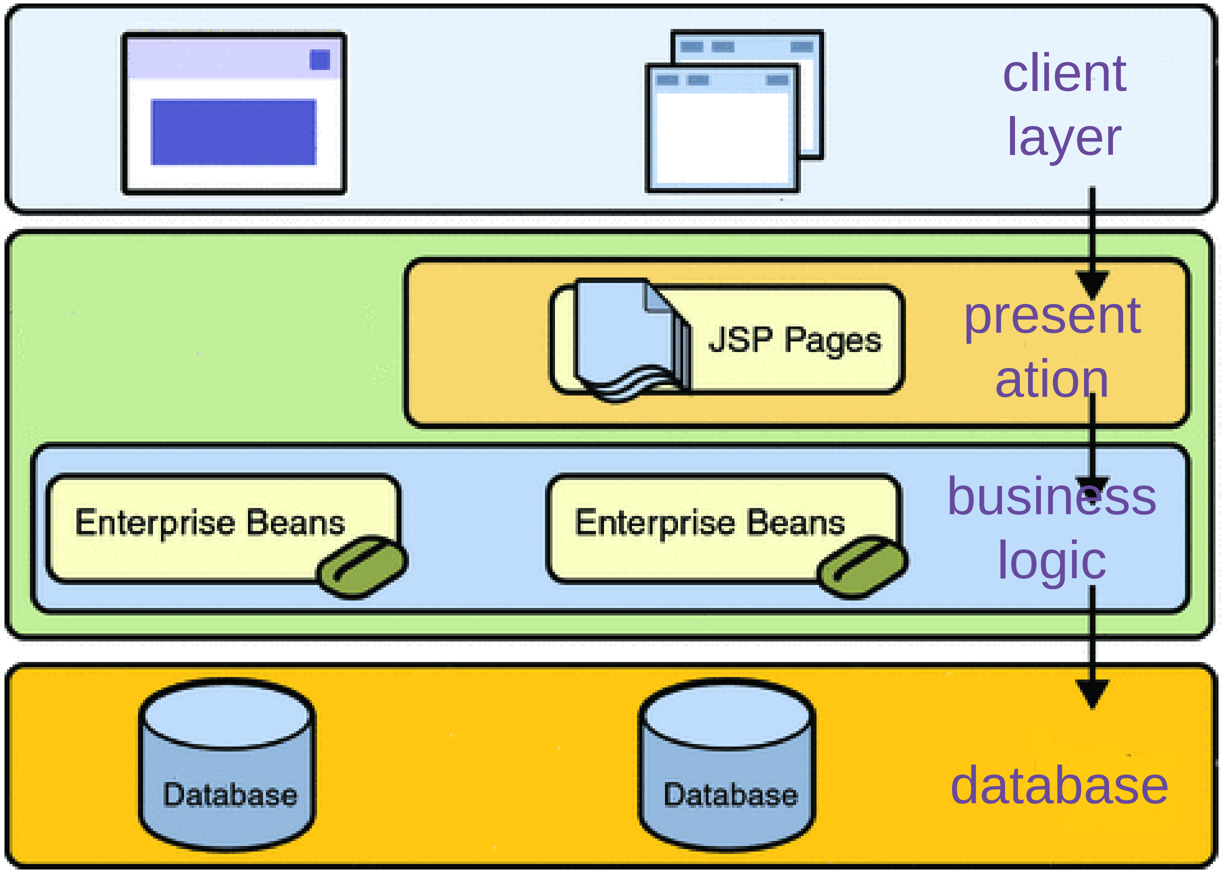
"Spring is amongst (if not the most) popular application development frameworks for enterprise Java™. Many developers use Spring to create high performing, easily testable, reusable code without any lock-in."

SpringSource

History

- first version was created by Rod Johnson and released in 2004
- version 2.0 – 2006
  - Java 1.3, AspectJ и JPA
- version 3.0 – 2009
  - Java5, annotations, SpEL, JavaConfig, REST
- version 4.0 – 2013
  - Java 8, Groovy 2, JavaEE7 support
- current version is 4.1.0 GA (by September 2014)

Origin – enterprise applications



- layers are composed of components
- every component contains part of the application's logic
- components aim for:
  1. high cohesion
  2. loose coupling

Origin – EJB (1)

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
public interface HelloWorld extends EJBObject {
    public String sayHello() throws RemoteException;
}
```

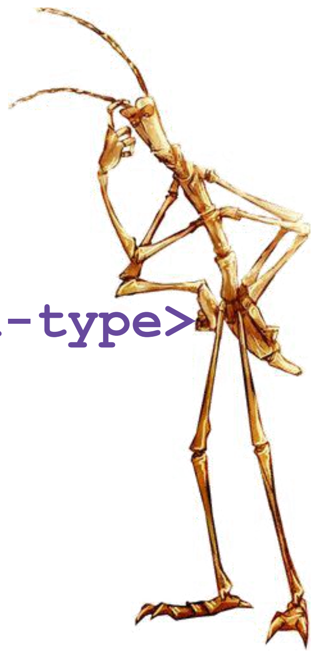
```
import java.rmi.RemoteException;
import java.ejb.CreateException;
import javax.ejb.EJBHome;
public interface HelloWorldHome extends EJBHome {
    public HelloWorld create()
        throws CreateException, RemoteException;
}
```

Origin – EJB (2)

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
public class HelloWorldBean implements SessionBean {
   protected SessionContext ctx;

   public String sayHello() {
      return "Hello, world !";
   }
   public void setSessionContext(SessionContext ctx) {
      this.ctx = ctx;
   }
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
}
```

Origin – EJB (3)

```
<ejb-jar>
  <description>HelloWorld deployment desc</description>
  <display-name>HelloWorld</display-name>
  <enterprise-beans>
    <session>
      <display-name>HelloWorld</display-name>
      <ejb-name>HelloWorld</ejb-name>
      <home>HelloWorldHome</home>
      <remote>HelloWorld</remote>
      <ejb-class>HelloWorldBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  ...
<ejb-jar>
```



Alternative implementation with Spring (1)

```
public interface HelloWorld {
   public String sayHello()
}

public class HelloWorldBean implements HelloWorld  {
   private String name;
   public void setName(String name) {this.name = name;}
   public String sayHello() {return "Hello, " + name;}
}

<beans ...>
   <bean id="hello" class="HelloWorldBean">
      <property name="name" value="colleagues"/>
   </bean>
</beans>
```

Alternative implementation with Spring (2)

- standalone application example:

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

HelloWorld bean = context.getBean("hello",
HelloWorld.class);

bean.sayHello();
```

Purposes of Spring

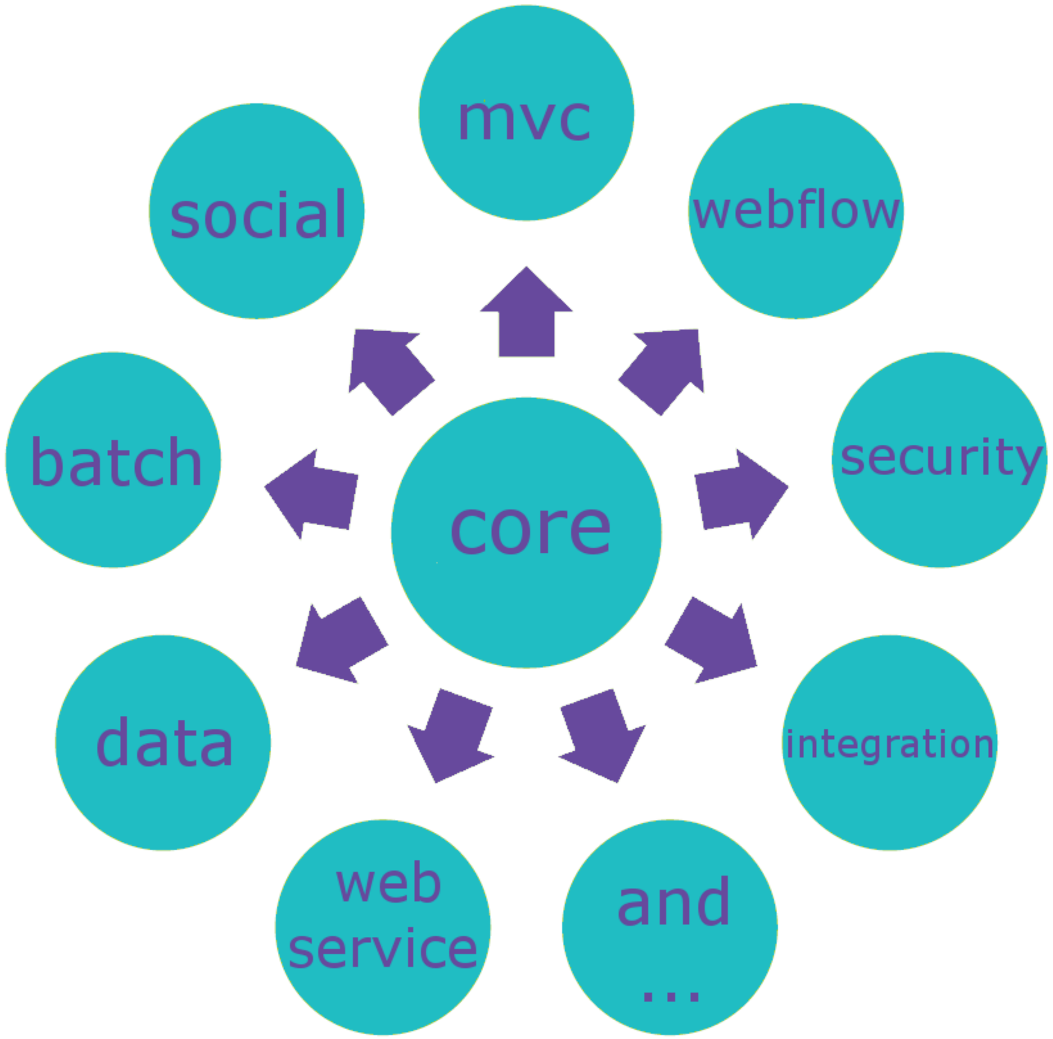
- simplify working with Java EE technologies
- encourage good practices for software development
- ease performing common tasks
- allow the developer to focus on the business problem at hand

Spring Framework

- open source framework
- unobtrusive (relies on **POJO**s)
- modular
- integrated with other frameworks

- de facto standard for developing enterprise Java applications

Spring modules



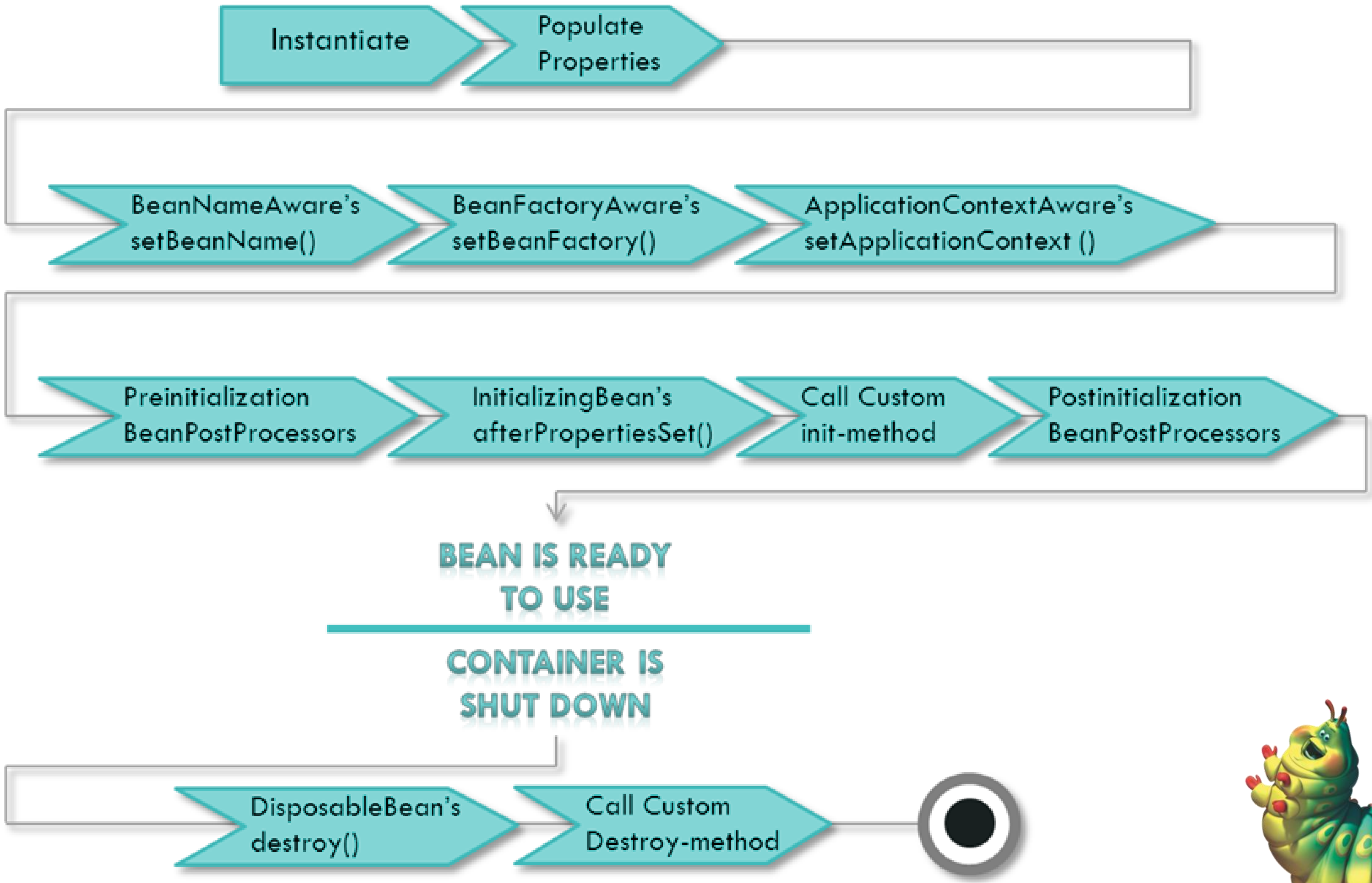
http://spring.io/projects

Spring Core

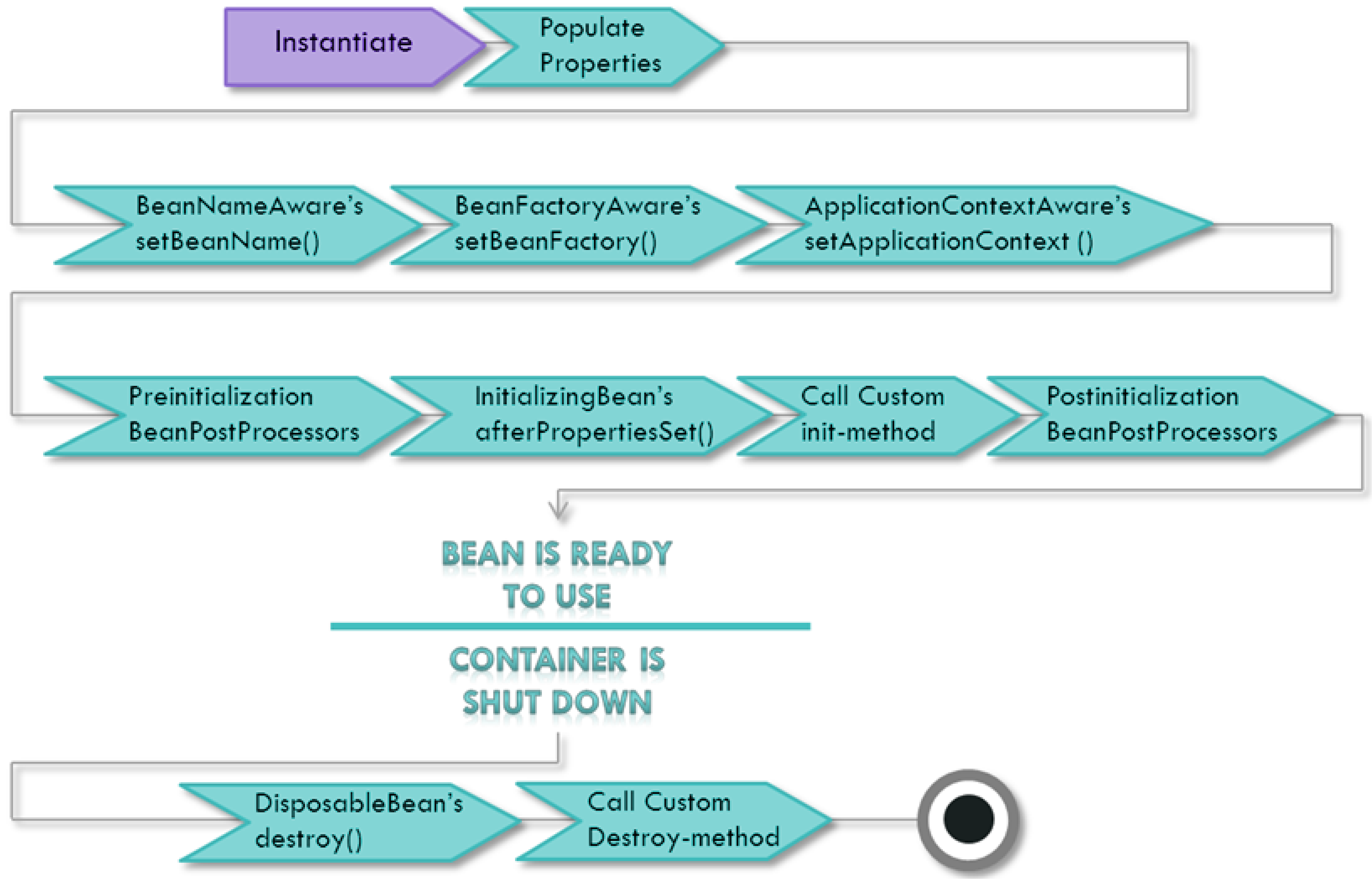
- **IoC container and beans:**
  1. backbone of the framework
  2. allows defining components (beans) with specific lifecycle
  3. allows using DI

- **Context** – mean for accessing beans (and other resources) in a unified manner (similar to JNDI)

Bean lifecycle


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

Bean instantiation


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

Bean instantiation (1)

- via constructor:

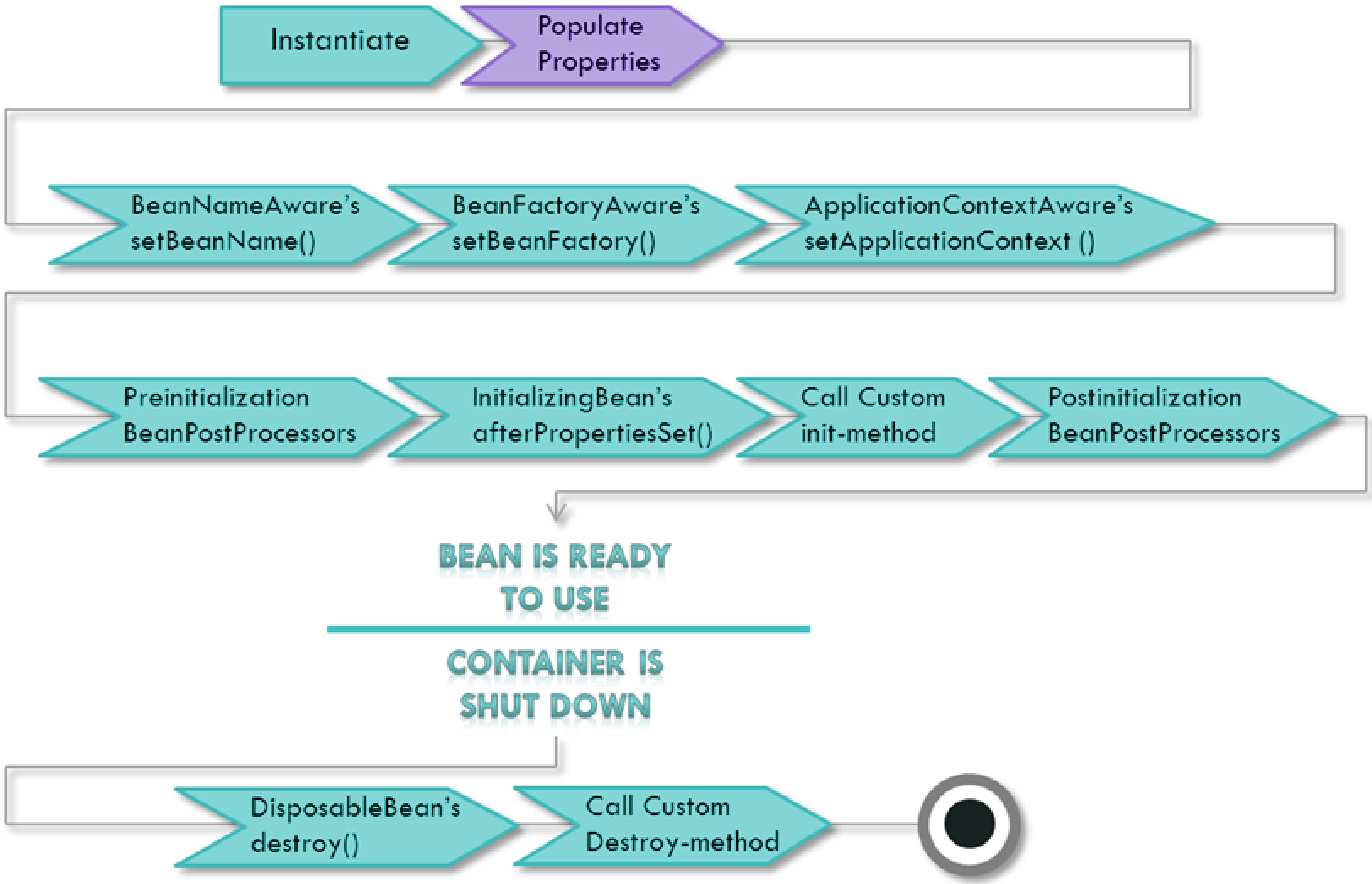
```
class MovieFinderImpl implements MovieFinder {
  public MovieFinderImpl() {}
}

<bean id="finder" class="MovieFinderImpl"/>
```

Bean instantiation (2)

- via static factory method

```
class MovieFinderImpl implements MovieFinder {
  private MovieFinderImpl() {}
  public static MovieFinder createInstance() {
    return new MovieFinderImpl();
  }
}

<bean id="finder" class="MovieFinderImpl"
factory-method="createInstance" />
```

Populate bean properties


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

Inversion of Control

- pattern for developing applications
- a.k.a. *"don’t call me, I’ll call you"*
- types:
  1. Factory pattern,
  2. Template Method pattern,
  3. Strategy pattern,
  4. **Dependency Injection,**
  5. etc.

Dependency Injection (1)

```
class MovieService {
 private MovieFinder finder;

 public MovieService() {
  finder = new MovieFinderImpl();
 }

}
```

Dependency Injection (2)

```
class MovieService {
 private MovieFinder finder;
 private Context ctx = …;
 public MovieService() {
  finder=(MovieFinder)ctx.lookup("id");
 }
}
...
ctx.rebind("id", new MovieFinderImpl())
```

Dependency Injection (3)

```
class MovieService {
 private MovieFinder finder;
 public MovieService(MovieFinder f){
  finder = f;
 }
 // or
 public setMovieFinder(MovieFinder f) {
  finder = f;
 }
}
```

Същност на Dependency Injection

- DI is a type of IoC
- DI is a pattern allowing components to define their dependencies, so that the container can inject the service in the dependent object (i.e. client)
- basic types of DI:
  1. via constructor (for mandatory dependencies)
  2. via setter method

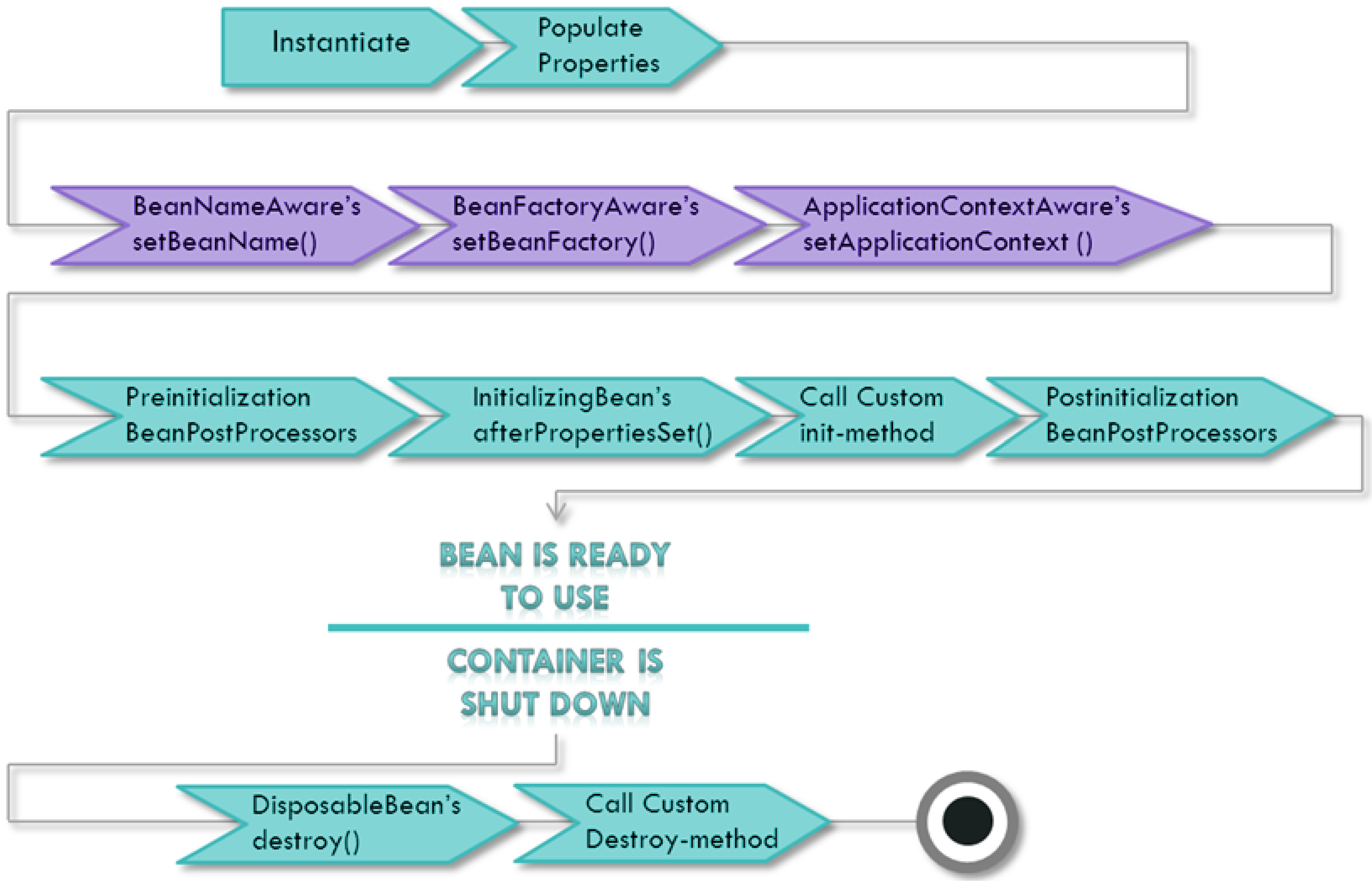
Dependency Injection – Spring example

```
<bean id="service" class="MovieService">
 <constructor-arg ref="finder"/>
</bean>

// or

<bean id="service" class="MovieService">
 <property name="finder" ref="finder"/>
</bean>
```

Dependency Injection advantages

- reduces the amount of code
- simplifies unit testing a component
- encourages writing logic complying with the interface
- ensures **loose coupling** between components
- supports **eager** and **lazy** loading
- provides control over the bean's lifecycle

Spring-aware interfaces

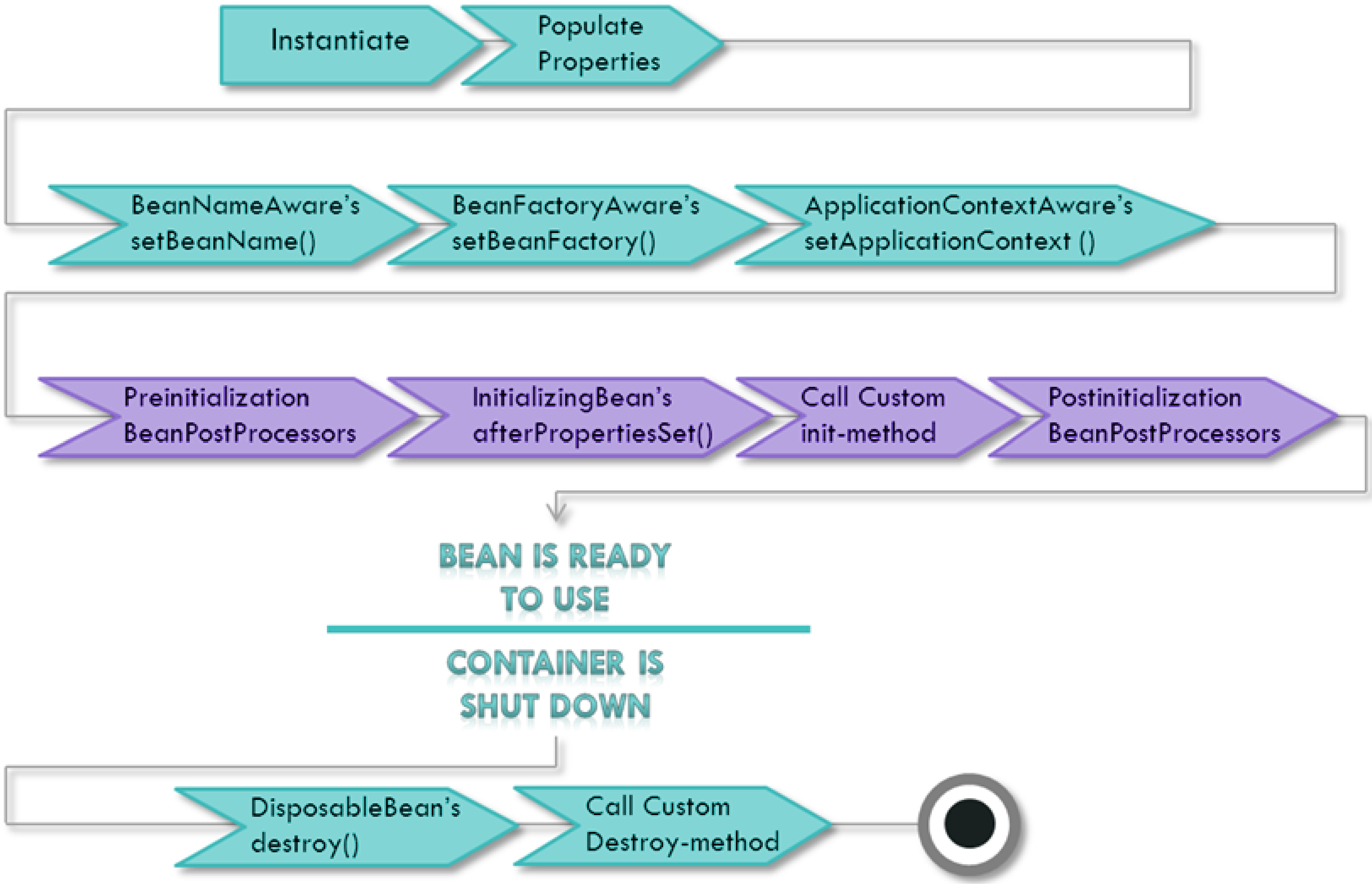

Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

Spring-aware interfaces – example

- BeanNameAware
- BeanFactoryAware
- ApplicationContextAware

```
class MovieFinderImpl
    implements MovieFinder, BeanNameAware {
  public void setBeanName(String name) {
        …
  }
}
```

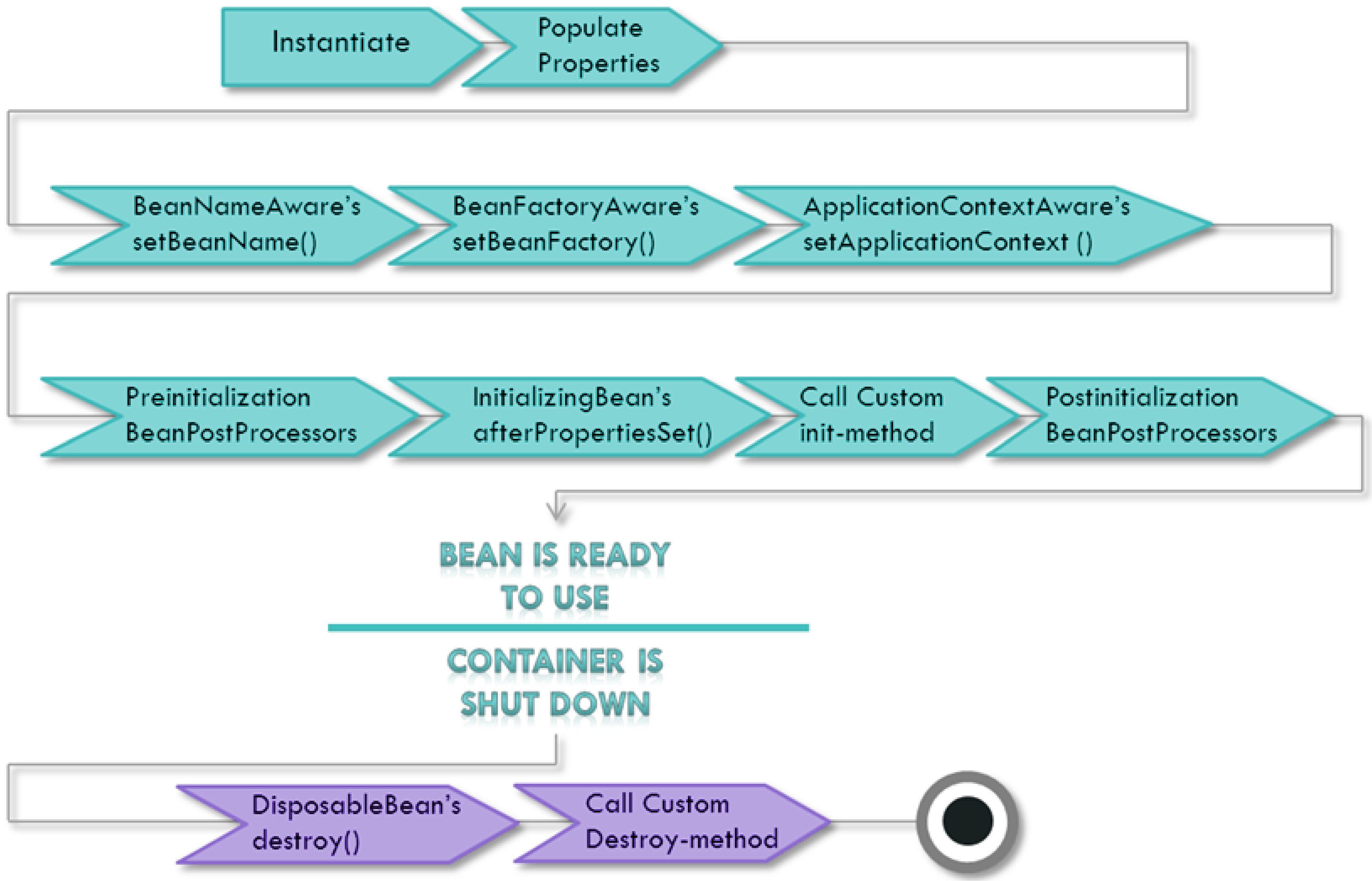
Bean initialization


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

Bean initialization - example

```
<bean id="b1" class="Bean1"/>
public class Bean1 {
  @PostConstruct
  public void initialize() {}
}
<bean id="b2" class="Bean2"/>
public class Bean2 implements InitializingBean {
  public void afterPropertiesSet() {}
}
<bean id="b3" class="Bean3" init-method="init"/>
public class Bean3 {
  public void init() {}
}
```

Bean destruction


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext ()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
BEAN IS READY TO USE
CONTAINER IS SHUT DOWN
DisposableBean's destroy()
Call Custom Destroy-method

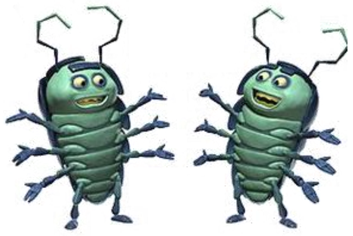
Bean destruction – example

```
<bean id="b1" class="Bean1"/>
public class Bean1 {
  @PreDestroy
  public void destroy() {}
}
<bean id="b2" class="Bean2"/>
public class Bean2 implements DisposableBean {
  public void destroy() {}
}
<bean id="b3" class="Bean3" destroy-method="clean"/
public class Bean3 {
  public void clean() {}
}
```

Bean scopes

- basic
  1. singleton
  2. prototype
  3. custom

- for web applications:
  1. request - new instance for every HTTP query
  2. session – new instances for every HTTP session
  3. global session – for portlets



Bean scope – singleton

```
<bean id="finderBean" class="MovieFinderImpl">

<bean id="app1" class="MovieService">
 <property name="finder" ref="finderBean"/>
</bean>

<bean id="app2" class="MovieService">
 <property name="finder" ref="finderBean"/>
</bean>
```

- single instances of **`finderBean`**
- must not keep state (unless synced)
- better performance

Bean scope – prototype

```
<bean id="finderBean" class="MovieFinderImpl"
      scope="prototype">
<bean id="app1" class="MovieService">
 <property name="finder" ref="finderBean"/>
</bean>

<bean id="app2" class="MovieService">
 <property name="finder" ref="finderBean"/>
</bean>
```

- two instances of **`finderBean`**
- may keep state
- worse performances

Annotations

- reduce the need of XML
- store configuration information in the code (+/-)
- used for:
  1. linking beans (and literals)
  2. defining beans(type, scope, etc.)
  3. registering in babies tune
  4. transaction demarcation
- and many more projects (e.g. MVC)

Example XML configuration

```
class MovieService {
  private MovieFinder finder;
  public setMovieFinder(MovieFinder f) {
    finder = f;
  }
}

<bean id="finder" class="MovieFinderImpl"/>
<bean id="service" class="MovieService">
 <property name="finder" ref="finder"/>
</bean>
```

Automatic wiring

- via `@Autowired` (+ reflection)
- specifics:
  - + removes the need for configuration
  - + simplifies working with a bean
  - - not as precise as explicit searching (might need `@Qualifier`)
- implementation:
  1. via field name
  2. via field type
  3. via constructor

Automatic wiring – example

```
class MovieService {
   @Autowired
   private MovieFinder finder;
}

<context:annotation-config />
<bean id="finder" class="MovieFinderImpl"/>
<bean id="service" class=" MovieService" />
```

Automatic wiring – example

- via Spring типове:
  1. `@Component` – base stereotype
  2. `@Service` – for business logic
  3. `@Repository` – for accessing data bases (*)
  4. `@Controller` – for Spring MVC
  5. `@Configuration` – for Java configuration

- `@Scope` – 3 types of basic scopes
- simplifies marking the different layers of the application

Automatic wiring – example

```
@Service
class MovieService {
  @Autowired
  private MovieFinder finder;
}

<context:component-scan base-package=
"com.musala" />
<context:annotation-config />
```

Configuration

- options (via games):
  1. XML– for infrastructure beans and backward compatibility
  2. Annotations – for standard beans
  3. JavaConfig – for further reduce in XML config.

```
@Configuration
@ComponentScan("com.musala")
class TestConfiguration {
  @Bean public MovieFinder finder() {
      return new MovieFinderImpl();
  }
}
```

Spring Data

- JDBC support:
  1. DataSource – provides and manages connections to the database
  2. JDBCTemplate – helper class simplifying JDBC usage in Spring:
     `jdbcTemplate.queryForInt("select count(*) from movie")`

- ORM support:
  1. Hibernate – direct integration
  2. **JPA** – using a JPA persistence provider (e.g. Hibernate)

JPA integration (1)

```
@Bean
public EntityManagerFactory entityManagerFactory()
throws SQLException {
  HibernateJpaVendorAdapter vendorAdapter =
    new HibernateJpaVendorAdapter();
  vendorAdapter.setGenerateDdl(false);

  LocalContainerEntityManagerFactoryBean factory =
    new LocalContainerEntityManagerFactoryBean();
  factory.setDataSource(dataSource());
  factory.setPackagesToScan("com.musala.domain");
  factory.setJpaVendorAdapter(vendorAdapter);
  factory.afterPropertiesSet();
  return factory.getObject();
}
```

JPA integration (2)

```
@Bean
public DataSource dataSource() throws SQLException {
   EmbeddedDatabaseBuilder b =
      new EmbeddedDatabaseBuilder();
   b.addScript("sql/schema.ddl");
   return b.setType(EmbeddedDatabaseType.H2).build();
}

@Bean
public EntityManager entityManager(EntityManagerFactory
entityManagerFactory) {
  return entityManagerFactory.createEntityManager();
}
```

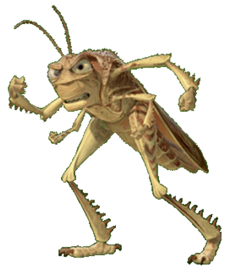
JPA integration (3)

```
@Repository
@Transactional
public class MovieServiceImpl implements MovieService {

  @PersistenceContext
  private EntityManager em;

  @Transactional(readOnly=true)
  public List<Movie> findAll() {
    List<Movie> m = em.createNamedQuery("findAllMovies",
       Movie.class).getResultList();
    return m;
  }
}
```

JPA Repository abstraction (1)

- wraps the EntityManager and provides a simple interface for database operations

```
public interface CrudRepository<T,ID>
    extends Repository<T, ID> {
  T save(T entity);
  T findOne(ID id);
  boolean exists(ID id);
  Iterable<T> findAll();
  long count();
  void delete(ID id);
  void delete(T entity);
  void delete(Iterable<? extends T> entities);
  void deleteAll();
}
```



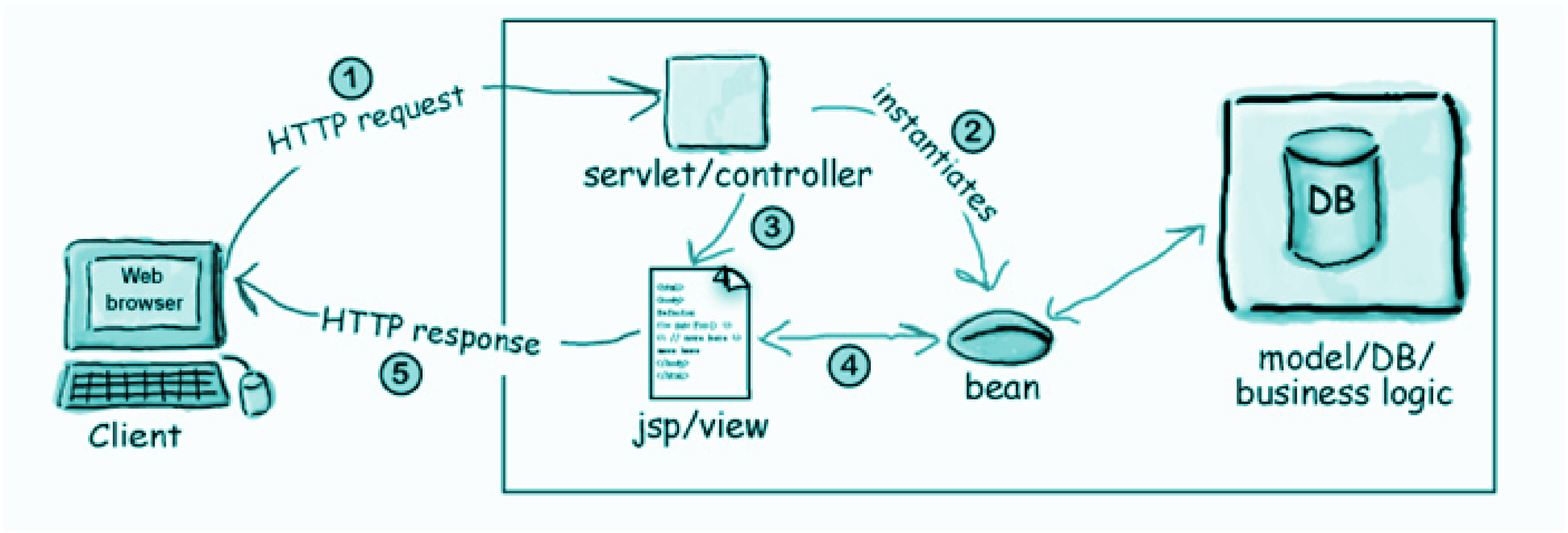
JPA Repository abstraction (2)

- no need to implement the basic CRUD methods
- custom methods following the naming convention, allow the framework to deduce their queries
- only methods with non-standard logic and non-compliant with the naming convention must be implemented

```
public interface MovieRepository
    extends CrudRepository<Movie, Long> {
  public List<Movie> findByTitle(String title);
  public List<Movie> findByTitleAndReleaseDate(String title,
    Date releaseDate);
}
```

Model-View-Controller basics

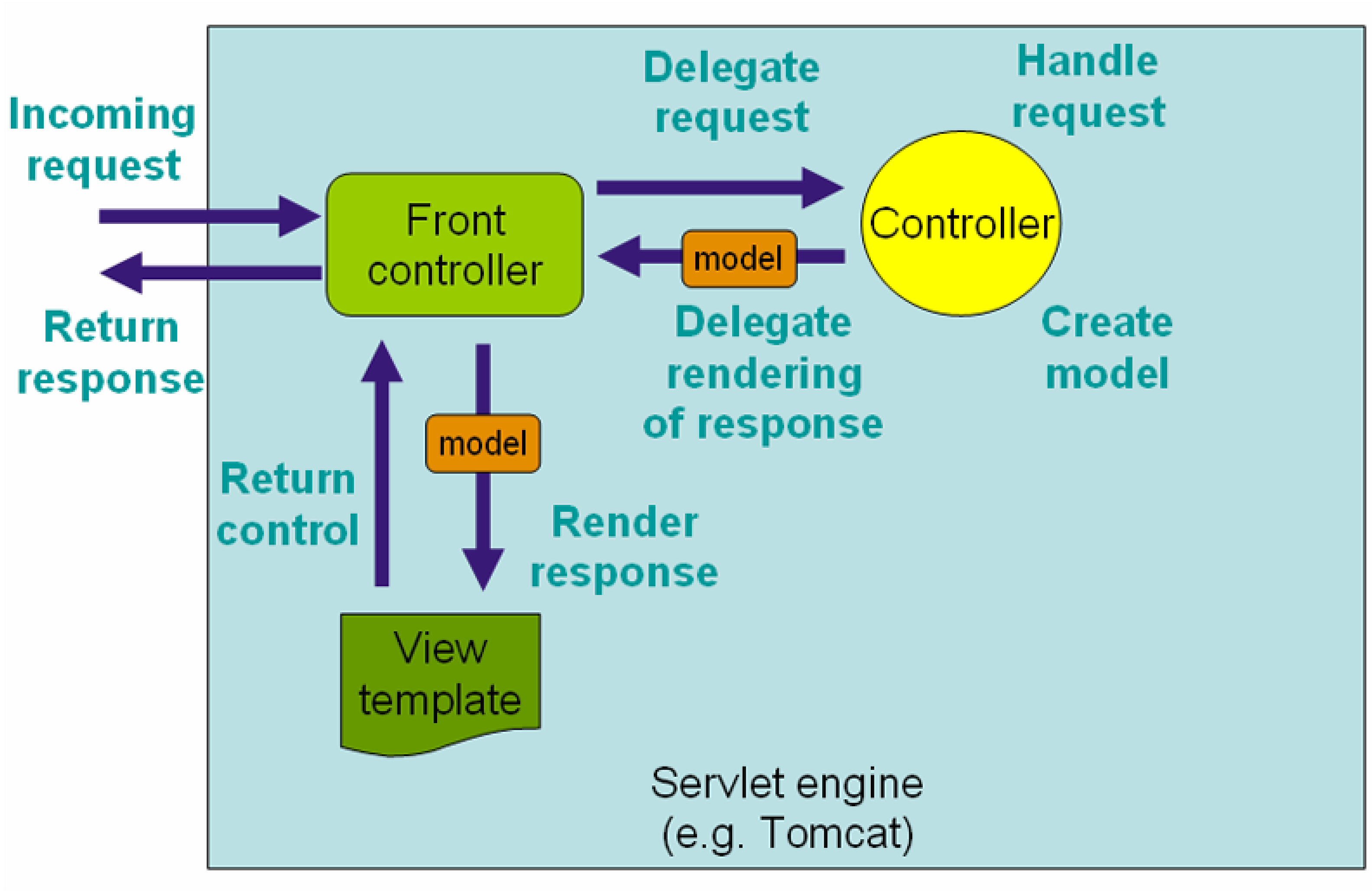
1 user sends a request to a specific URL
2 controller calls the model and receives data
3 controller associates the model and view and passes the control to the view
4 view uses the model and generates a representation
5 users receives the representation sent as response



Spring MVC

- model-view-controller framework
- uses `DispatcherServlet` in order to direct queries to their handler
- handlers can be tweaked via `@Controller` and `@RequestMapping`
- dynamic view selection, changing locale and visual theme
- allows creation of RESTful web services

Spring MVC workflow



Controller – example

```
@Controller
@RequestMapping("/movies")
public class MovieController {
  @Autowired
  private MovieService movieService;

  @Autowired
  private ConversionService conversionService;
  ...
  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  @ResponseBody public MovieView getById(@PathVariable String id) {
    return conversionService.convert(movieService.getById(id), MovieView.class);
  }

  @RequestMapping(value = { "/", "" }, method = RequestMethod.POST)
  @ResponseBody public MovieView create(@Valid @RequestBody MovieCreateForm f){
    return conversionService.convert(movieService.create(f), MovieView.class);
  }

}
```

Converter – example

- allows converting fields (bean fields) and complete beans

```
@Component
public class MovieSearchViewConverter
    implements Converter<Movie, MovieSearchView> {

  @Override
  public MovieSearchView convert(Movie movie) {
    return new MovieSearchView(movie.getId(),
      movie.getTitle(), movie.getReleaseDate());
  }
}
```

Validation – example

- declarative validation of input data
- standard JSR-303 annotations or custom implementation (hibernate)

```
public class MovieCreateForm {
  @Size(min = 3, max = 100)
  @NotEmpty
  private String title;

  @NotNull
  private Date releaseDate;
}
```

+ @Valid

Demo

Q&A

petyo.dimitrov
@musala.com

MUFFIN
CONFERENCE
MusalaSoft