*Web*开发技术

Web Application Development

第12课
WEB后端框架-SPRING JPA & IOC

Episode Twelve

Spring JPA & IoC

陈昊鹏

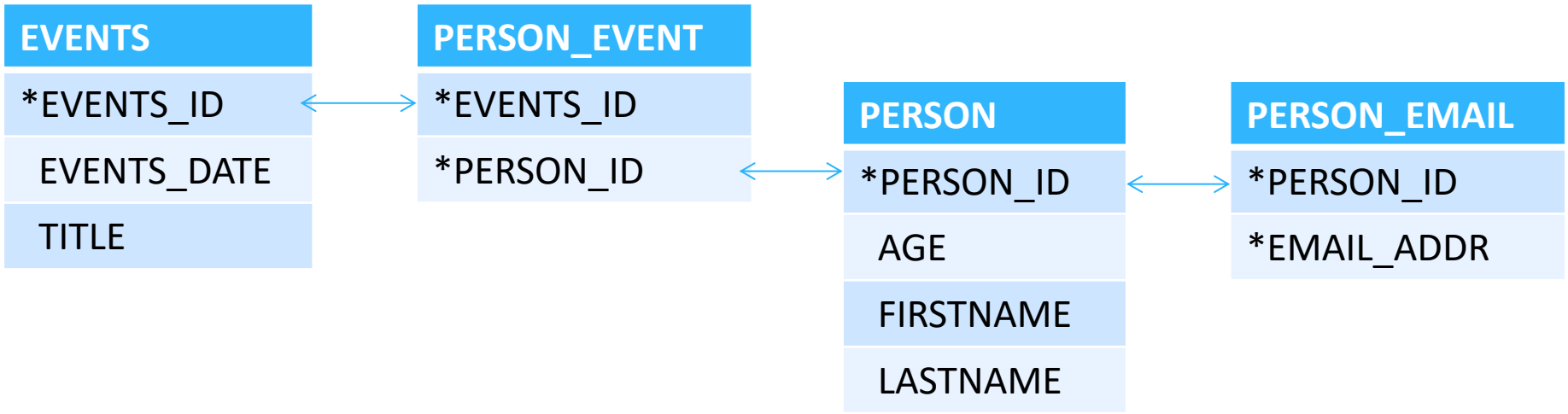
chen-hp@sjtu.edu.cn

Overview

- Spring Data JPA
  - Relationship Mapping
- Structure of web project

Database Schema

REliable, INtelligent & Scalable Systems



application.properties


```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=root
spring.datasource.password=reins2011
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Entities: Event


```
@Entity
@Table(name = "events", schema = "test", catalog = "")
@JsonIgnoreProperties(value = {"handler","hibernateLazyInitializer","fieldHandler"})
@JsonIdentityInfo(
      generator = ObjectIdGenerators.PropertyGenerator.class,
      property = "eventId")
public class Event {
   private int eventId;
   private String title;
   private Timestamp eventDate;

   @Id
   @Column(name = "EVENT_ID")
   @GeneratedValue(strategy = IDENTITY)
   public int getEventId() {       return eventId;    }
   public void setEventId(int eventId) {       this.eventId = eventId;    }

   @Basic
   @Column(name = "title")
   public String getTitle() {       return title;    }
   public void setTitle(String title) {       this.title = title;    }

   @Basic
   @Column(name = "EVENT_DATE")
   public Timestamp getEventDate() {       return eventDate;    }
   public void setEventDate(Timestamp eventDate) {       this.eventDate = eventDate;    }
```

Entities: Event


```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Event that = (Event) o;

        if (eventId != that.eventId) return false;
        if (title != null ? !title.equals(that.title) : that.title != null) return false;
        if (eventDate != null ? !eventDate.equals(that.eventDate) : that.eventDate != null) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = eventId;
        result = 31 * result + (title != null ? title.hashCode() : 0);
        result = 31 * result + (eventDate != null ? eventDate.hashCode() : 0);
        return result;
    }

    private List<Person> participants;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "PERSON_EVENT",joinColumns = @JoinColumn(name = "EVENT_ID"),
            inverseJoinColumns = @JoinColumn(name = "PERSON_ID"))
    public List<Person> getParticipants() {       return participants;    }
    public void setParticipants(List<Person> participants) {       this.participants = participants;    }
}
```

Entities: Person


```
@Entity
@Table(name = "persons", schema = "test", catalog = "")
@JsonIgnoreProperties(value = {"handler","hibernateLazyInitializer","fieldHandler"})
@JsonIdentityInfo(
      generator = ObjectIdGenerators.PropertyGenerator.class,
      property = "personId")
public class Person {
   private int personId;
   private Integer age;
   private String firstname;
   private String lastname;

   @Id
   @Column(name = "PERSON_ID")
   public int getPersonId() {      return personId;   }
   public void setPersonId(int personId) {      this.personId = personId;   }

   @Basic
   @Column(name = "age")
   public Integer getAge() {      return age;   }
   public void setAge(Integer age) {      this.age = age;   }

   @Basic
   @Column(name = "firstname")
   public String getFirstname() {      return firstname;   }
   public void setFirstname(String firstname) {      this.firstname = firstname;   }

   @Basic
   @Column(name = "lastname")
   public String getLastname() {      return lastname;   }
   public void setLastname(String lastname) {      this.lastname = lastname;   }
```

Entities: Person


```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Person person = (Person) o;

    if (personId != person.personId) return false;
    if (age != null ? !age.equals(person.age) : person.age != null) return false;
    if (firstname != null ? !firstname.equals(person.firstname) : person.firstname != null) return false;
    if (lastname != null ? !lastname.equals(person.lastname) : person.lastname != null) return false;

    return true;
}

@Override
public int hashCode() {
    int result = personId;
    result = 31 * result + (age != null ? age.hashCode() : 0);
    result = 31 * result + (firstname != null ? firstname.hashCode() : 0);
    result = 31 * result + (lastname != null ? lastname.hashCode() : 0);
    return result;
}

private List<Event> activities;

@ManyToMany(fetch = FetchType.LAZY,mappedBy = "participants")
public List<Event> getActivities() {       return activities;    }
public void setActivities(List<Event> activities) {       this.activities = activities;    }

private List<String> emails = new ArrayList<String>();

@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable( name="PERSON_EMAIL",
      joinColumns = { @JoinColumn(name = "PersonId", referencedColumnName = "PERSON_ID")})
@Column(name="EMAIL_ADDRESS")
public List<String> getEmails() {       return emails;    }
public void setEmails(List<String> emails) {       this.emails = emails;    }

}
```

Repositories

public interface EventRepository **extends** JpaRepository<Event, Integer>{ }

public interface PersonRepository **extends** JpaRepository<Person, Integer>{ }

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	**deleteAllInBatch**() Deletes all entities in a batch call.
void	**deleteInBatch**(**Iterable**<**T**> entities) Deletes the given entities in a batch which means it will create a single **Query**.
List<**T**>	**findAll**()
<S extends **T**> **List**<S>	**findAll**(**Example**<S> example)
<S extends **T**> **List**<S>	**findAll**(**Example**<S> example, **Sort** sort)
List<**T**>	**findAll**(**Sort** sort)
List<**T**>	**findAllById**(**Iterable**<**ID**> ids)
void	**flush**() Flushes all pending changes to the database.
T	**getOne**(**ID** id) Returns a reference to the entity with the given identifier.
<S extends **T**> **List**<S>	**saveAll**(**Iterable**<S> entities)
<S extends **T**> S	**saveAndFlush**(S entity) Saves an entity and flushes changes instantly.

DAO & DAO Implementation


```
public interface EventDao {
   Event findOne(Integer id);
}

@Repository
public class EventDaoImpl implements EventDao {
      @Autowired
      private EventRepository eventRepository;

      @Override
      public Event findOne(Integer id) {
         return eventRepository.getOne(id);
      }
}

public interface PersonDao {
   Person findOne(Integer id);
}

@Repository
public class PersonDaoImpl implements PersonDao {
      @Autowired
      private PersonRepository personRepository;

      @Override
      public Person findOne(Integer id) {
         return personRepository.getOne(id);
      }
}
```

Services & Service Implementation

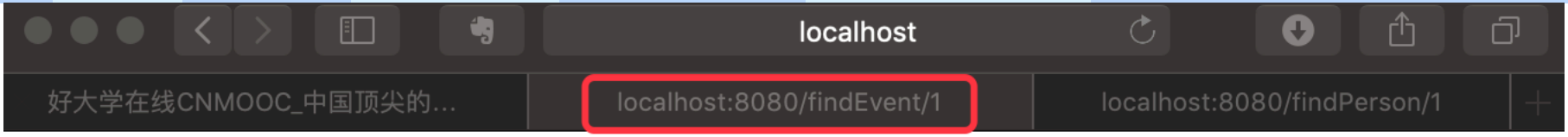

```
public interface EventService {
   Event findEventById(Integer id);
}

@Service
public class EventServiceImpl implements EventService {

   @Autowired
   private EventDao eventDao;

   @Override
   public Event findEventById(Integer id){
      return eventDao.findOne(id);
   }
}

public interface PersonService {
   Person findEventById(Integer id);
}

@Service
public class PersonServiceImpl implements PersonService {

   @Autowired
   private PersonDao personDao;

   @Override
   public Person findEventById(Integer id){
      return personDao.findOne(id);
   }
}
```

Controllers

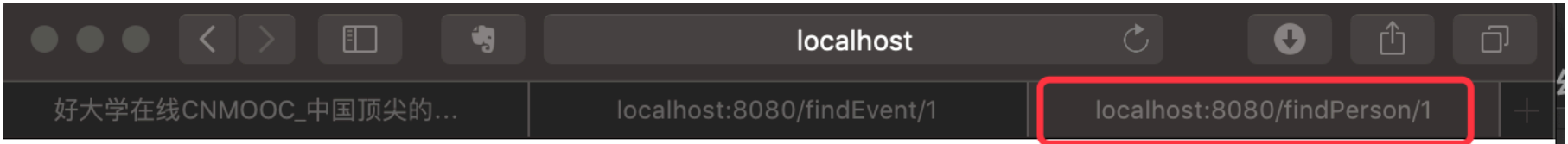

```
@RestController
public class EventController {

   @Autowired
   private EventService eventService;

   @GetMapping(value = "/findEvent/{id}")
   public Event findEvent(@PathVariable("id") Integer id) {
      System.out.println("Searching Event: " + id);
      return eventService.findEventById(id);
   }
}
```

```
@RestController
public class PersonController {

   @Autowired
   private PersonService personService;

   @GetMapping(value = "/findPerson/{id}")
   public Person findPerson(@PathVariable("id") Integer id) {
      System.out.println("Searching Person: " + id);
      return personService.findEventById(id);
   }
}
```

Results




```
{"eventId":1,"title":"party","eventDate":"2017-04-
20T05:00:00.000+0000","participants":
[{"personId":1,"age":47,"firstname":"Cao","lastname":"Cao","activities":[1,
{"eventId":5,"title":"class","eventDate":"2017-04-
19T05:00:00.000+0000","participants":[1]},
{"eventId":11,"title":"Avengers4","eventDate":"2019-04-
26T05:00:00.000+0000","participants":[1]}]},
{"personId":3,"age":26,"firstname":"Liang","lastname":"Zhuge","activities":[1]}]}
```


```
{"personId":1,"age":47,"firstname":"Cao","lastname":"Cao","activities":
[{"eventId":1,"title":"party","eventDate":"2017-04-
20T05:00:00.000+0000","participants":[1,
{"personId":3,"age":26,"firstname":"Liang","lastname":"Zhuge","activities":
[1],"emails":[]}]},{"eventId":5,"title":"class","eventDate":"2017-04-
19T05:00:00.000+0000","participants":[1]},
{"eventId":11,"title":"Avengers4","eventDate":"2019-04-
26T05:00:00.000+0000","participants":[1]}],"emails":["new@new.com"]}
```

Architecture

- Layered Architecture
  - Separation of Interface and Implementation
  - Entity – Auto mapped from database schema
  - Repository – Extended from existing lib class
  - Dao – Your own access control logic
  - Service – Business logic
  - Controller – Dispatch requests
  - IoC/DI – Independent of implementation

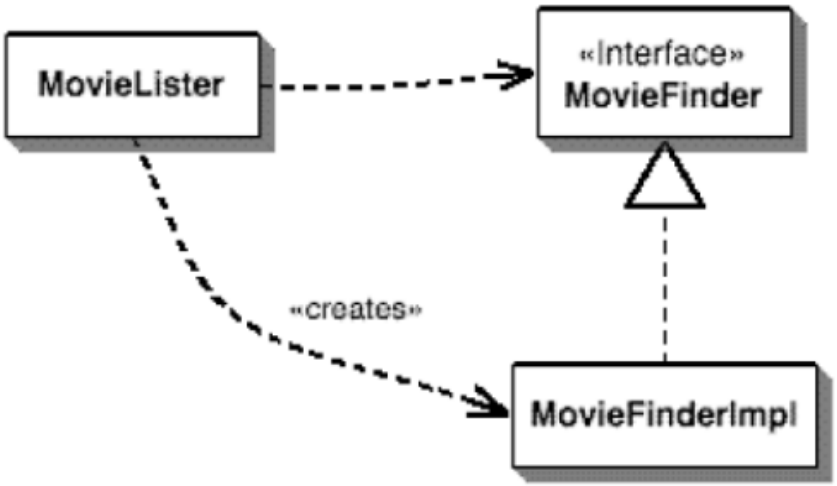
A Naive Example


```
class MovieLister...
  public Movie[] moviesDirectedBy(String arg) {
    List allMovies = finder.findAll();
    for (Iterator it = allMovies.iterator(); it.hasNext();) {
      Movie movie = (Movie) it.next();
      if (!movie.getDirector().equals(arg)) it.remove();
    }
    return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
}
```

- How we connect the `lister` object with a particular finder object?

A Naive Example


```
public interface MovieFinder { List findAll(); }
class MovieLister...
  private MovieFinder finder;
  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
  }
```



- The `MovieLister` class is dependent on both the `MovieFinder` interface and upon the implementation!

Inversion of Control

- Problem
  - Implemented class of `MovieFinder` needn't connect to program during compiling the program.
    - Hope to plug-in concrete implemented class during run-time.
  - How to make `MovieLister` class to cooperate with other instances while they don't know the details of the implemented class?

- Solution
  - Inversion of Control

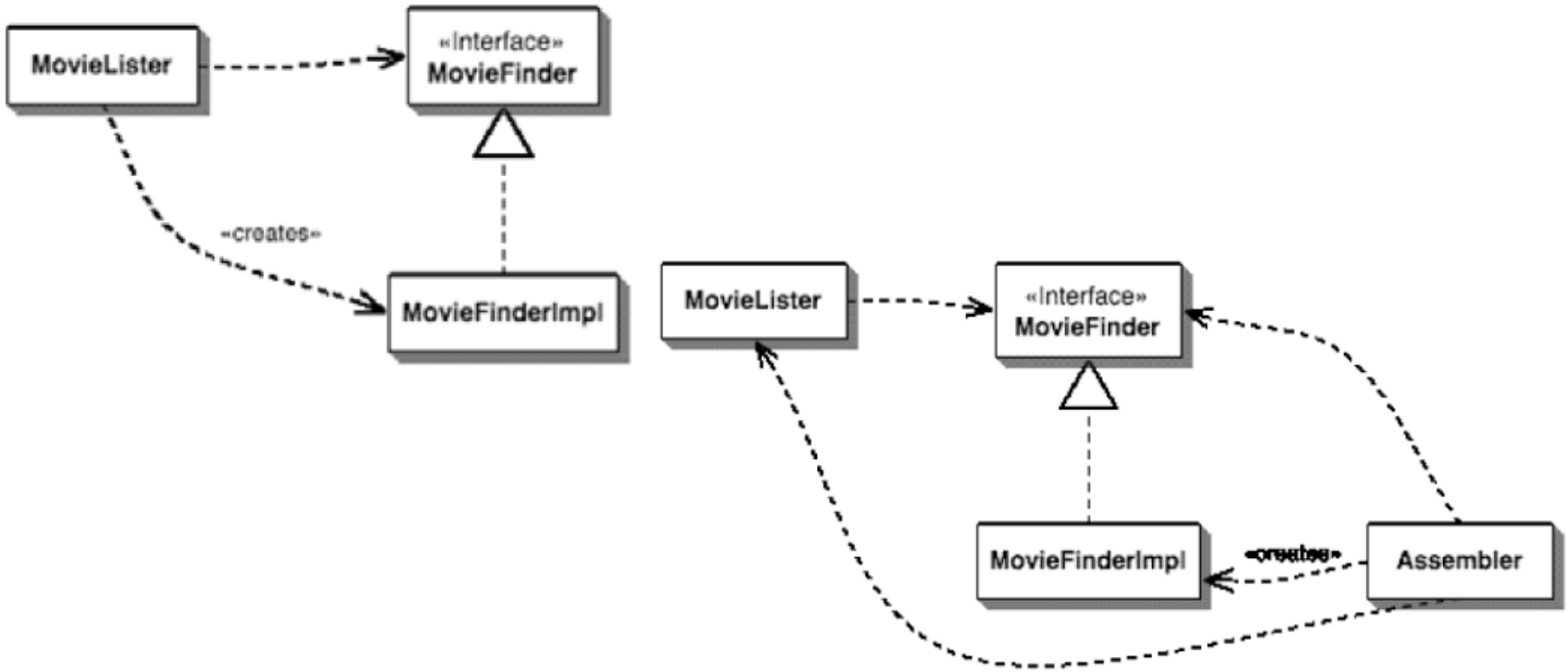
Inversion of Control

- What aspect of control are they inverting?
  - In naive example the `lister` looked up the finder implementation by directly instantiating it
    - This stops the finder from being a plugin.
  - The inversion is about how they lookup a plugin implementation.
    - Any user of a plugin follows some convention that allows a separate assembler module to inject the implementation into the `lister`.

- Dependency Injection

Dependency Injection

- The basic idea of the Dependency Injection
  - Have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface



Forms of Dependency Injection

- Two main styles of dependency injection.
  - Constructor Injection
  - Setter Injection

Constructor Injection - Example

- Uses a constructor to decide how to inject a `finder` implementation into the `lister` class.

```
class MovieLister...
  public MovieLister(MovieFinder finder) {
    this.finder = finder;
  }

class ColonMovieFinder...
  public ColonMovieFinder(String filename) {
    this.filename = filename;
  }
```

Setter Injection -Example

- To get my movie `lister` to accept the injection, I define a setting method for that service

```
class MovieLister...
  private MovieFinder finder;
  public void setFinder(MovieFinder finder) {
    this.finder = finder;
  }
class ColonMovieFinder...
  public void setFilename(String filename) {
    this.filename = filename;
  }
```

Dependency Injection - Example

- The third step is to set up the configuration for the files.

```
<beans>
    <bean id="MovieLister" class="spring.MovieLister">
        <property name="finder">
            <ref local="MovieFinder"/>
        </property>
    </bean>
    <bean id="MovieFinder" class="spring.ColonMovieFinder">
        <property name="filename">
            <value>movies1.txt</value>
        </property>
    </bean>
</beans>
```

Dependency Injection - Example

- Test code:

```
public void testWithSpring() throws Exception {
  ApplicationContext ctx = new
        FileSystemXmlApplicationContext("spring.xml");
  MovieLister lister = (MovieLister)
        ctx.getBean("MovieLister");
  Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
  assertEquals("Once Upon a Time in the West",
        movies[0].getTitle());
}
```

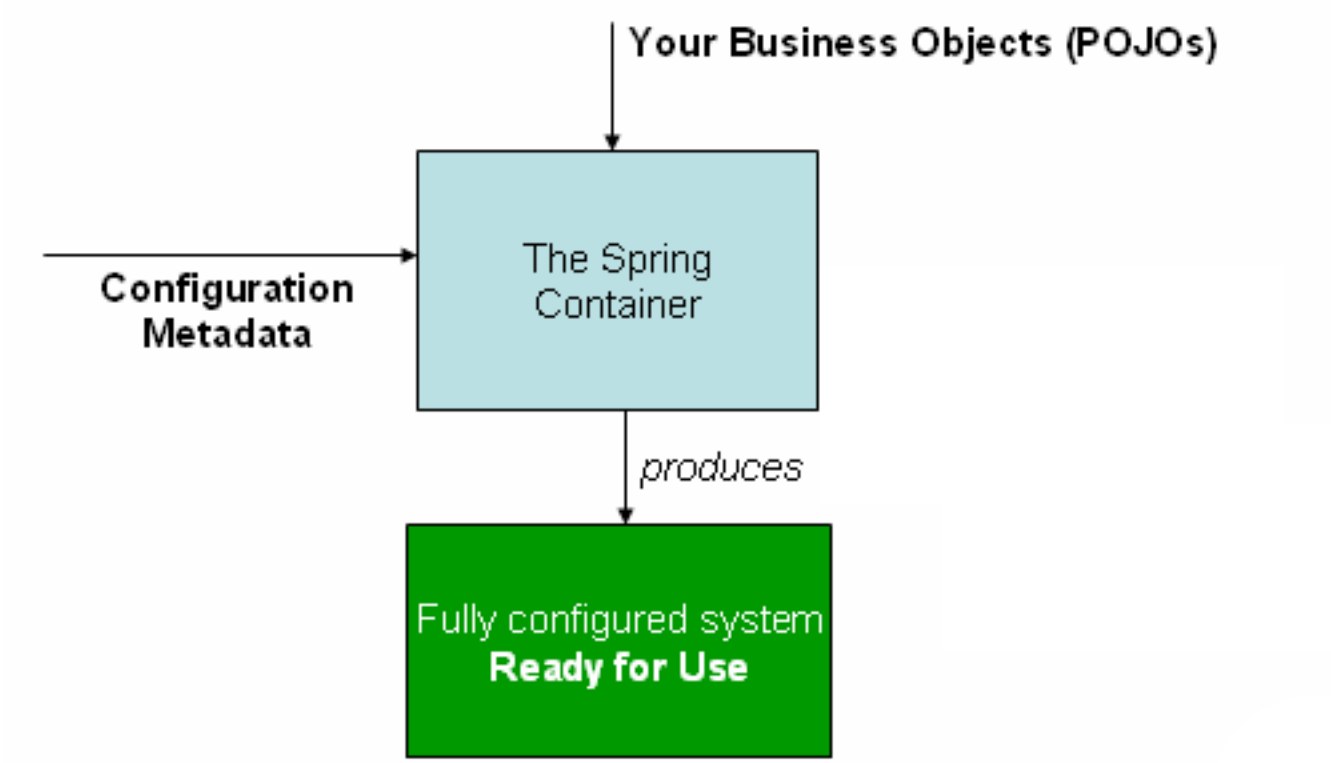
Spring

- Spring Framework is
  - a Java platform that provides comprehensive infrastructure support for developing Java applications.
  - Spring handles the infrastructure so you can focus on your application.

- Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs.
  - This capability applies to the Java SE programming model and to full and partial Java EE.

- Examples of how you, as an application developer, can use the Spring platform advantage:
  - Make a Java method execute in a database transaction without having to deal with transaction APIs.
  - Make a local Java method a remote procedure without having to deal with remote APIs.
  - Make a local Java method a management operation without having to deal with JMX APIs.
  - Make a local Java method a message handler without having to deal with JMS APIs.

Core Technologies - IoC container



An Example

- **hello/MessageService.java**

```
package hello;

public interface MessageService {
    String getMessage();
}
```

An Example

- **hello/MessagePrinter.java**

```
package hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MessagePrinter {

 @Autowired
 private MessageService service;

 public void printMessage() {
  System.out.println(this.service.getMessage());
 }

}
```

An Example

- **hello/Application.java**

```
package hello;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan
public class Application {

  @Bean
  MessageService mockMessageService() {
    return new MessageService() {
      public String getMessage() {
        return "Hello World!";
      }
    };
  }
  public static void main(String[] args) {
    ApplicationContext context = new
              AnnotationConfigApplicationContext(Application.class);
    MessagePrinter printer = context.getBean(MessagePrinter.class);
    printer.printMessage();
  }
}
```

An Example

- **hello/MockMessageService**

```
package hello;
public class MockMessageService implements MessageService
{
    public String getMessage() {
        return "Hello World! Mock Message Service!";
    }
}
```

- **hello/AnotherMessageService**

```
package hello;
public class AnotherMessageService implements MessageService
{
    public String getMessage() {
        return "Hello World! Another Message Service!";
    }
}
```

An Example

- **hello/Application.java**

```
package hello;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan
public class Application {

  @Bean
  MessageService mockMessageService() {
    return new MockMessageService;
    or
    return new AnotherMessageService;
  }
  public static void main(String[] args) {
    ApplicationContext context = new
              AnnotationConfigApplicationContext(Application.class);
    MessagePrinter printer = context.getBean(MessagePrinter.class);
    printer.printMessage();
  }
}
```

Bean overview

- A Spring IoC container manages one or more *beans*.
- These beans are created with the configuration metadata that you supply to the container,
  - for example, in the form of XML <bean/> definitions.
- Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:
  - *A package-qualified class name:* typically the actual implementation class of the bean being defined.
  - Bean behavioral configuration elements, which state how the bean should behave in the container(scope, lifecycle callbacks, and so forth).
  - References to other beans that are needed for the bean to do its work; these references are also called *collaborators* or *dependencies*.
  - Other configuration settings to set in the newly created object,
    - for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.
- This metadata translates to a set of properties that make up each bean definition.

Dependency injection

- Constructor-based dependency injection

```
package x.y;
public class Foo {
   public Foo(Bar bar, Baz baz) {
    // ...
   }
}
```

```
<beans>
  <bean id="foo" class="x.y.Foo">
    <constructor-arg ref="bar"/>
    <constructor-arg ref="baz"/>
  </bean>
  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>
</beans>
```

Dependency injection

- Constructor-based dependency injection

```
package examples;
public class ExampleBean {
  // No. of years to the calculate the Ultimate Answer
  private int years;
  // The Answer to Life, the Universe, and Everything
  private String ultimateAnswer;

  public ExampleBean(int years, String ultimateAnswer) {
    this.years = years;
    this.ultimateAnswer = ultimateAnswer;
  }
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>

<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>

<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg name="years" value="7500000"/>
  <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

Dependency injection

- hello/ExampleBeanClient.java

```
package hello;

public class ExampleBeanClient {

  public static void main(String[] args) {
      ApplicationContext context =
                    new ClassPathXmlApplicationContext(
                          new String[] {"ExampleContext.xml"});
      ExampleBean bean = context.getBean(ExampleBean.class);
      System.out.println(bean.answer());
  }
}
```

Dependency injection

- Setter-based dependency injection.java

```
package hello;

import java.beans.ConstructorProperties;

public class ExampleBean {
     private int years;
     private String ultimateAnswer;

     public void setYears(int years) {
              this.years = years;
     }

     public void setUltimateAnswer(String ultimateAnswer) {
              this.ultimateAnswer = ultimateAnswer;
     }

     public String answer() {
              return ultimateAnswer + " " + years;
     }
}
```

```
<bean id="exampleBean" class="hello.ExampleBean">
  <property name="years">
     <value>7500000</value>
  </property>
  <property name="ultimateAnswer">
     <value>42</value>
  </property>
</bean>
```

Dependency injection

- **Constructor-based or setter-based DI?**
  - Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.
  - The Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy, especially when properties are optional.
    - Setter methods also make objects of that class amenable to reconfiguration or re-injection later.
    - Management through JMX MBeans is a compelling use case.
  - Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state.
    - The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

References

- Spring Document - 31. Working with SQL Databases,
  - https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html

- *Web*开发技术
- *Web Application Development*

Thank You!