

Enterprise Application Development

Spring and Hibernate

Agenda

- Spring Inversion of Control and Dependency Injection
 - Spring Context
 - Spring Beans
 - Spring Configuration
- Spring MVC
- Spring Data
 - Hibernate
 - JPA
 - Spring Repository
- More on Spring

Agenda

- Spring Inversion of Control and Dependency Injection
 - Spring Context
 - Spring Beans
 - Spring Configuration
- Spring MVC
- Spring Data
 - Hibernate
 - JPA
 - Spring Repository
- More on Spring

What's 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”*

Spring main reference: <http://docs.spring.io/spring/docs/3.0.x/reference/overview.html>

Inversion of Control (IoC)

Design technique that delegates invoking a behavior to an assembler at runtime

Example: program to get and process information from a user

Command line version

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

Graphical version

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Inversion of Control (IoC)

Example: program to get and process information from a user

Command line version

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

Graphical version

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Control goes from my command line program module to the event manager module, which is instructed via "bind"

This is IoC, aka "Hollywood principle: don't call us, we'll call you"

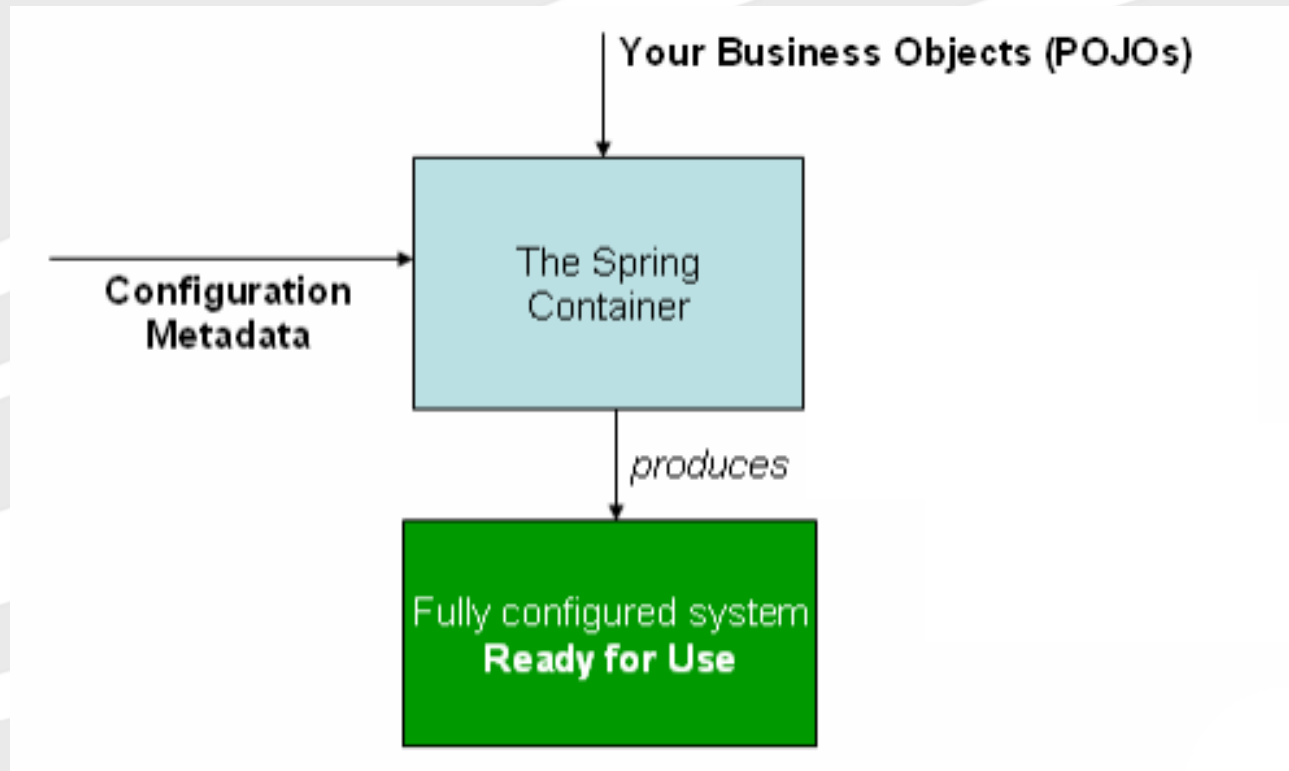
<http://martinfowler.com/bliki/InversionOfControl.html>

Dependency Injection (DI)

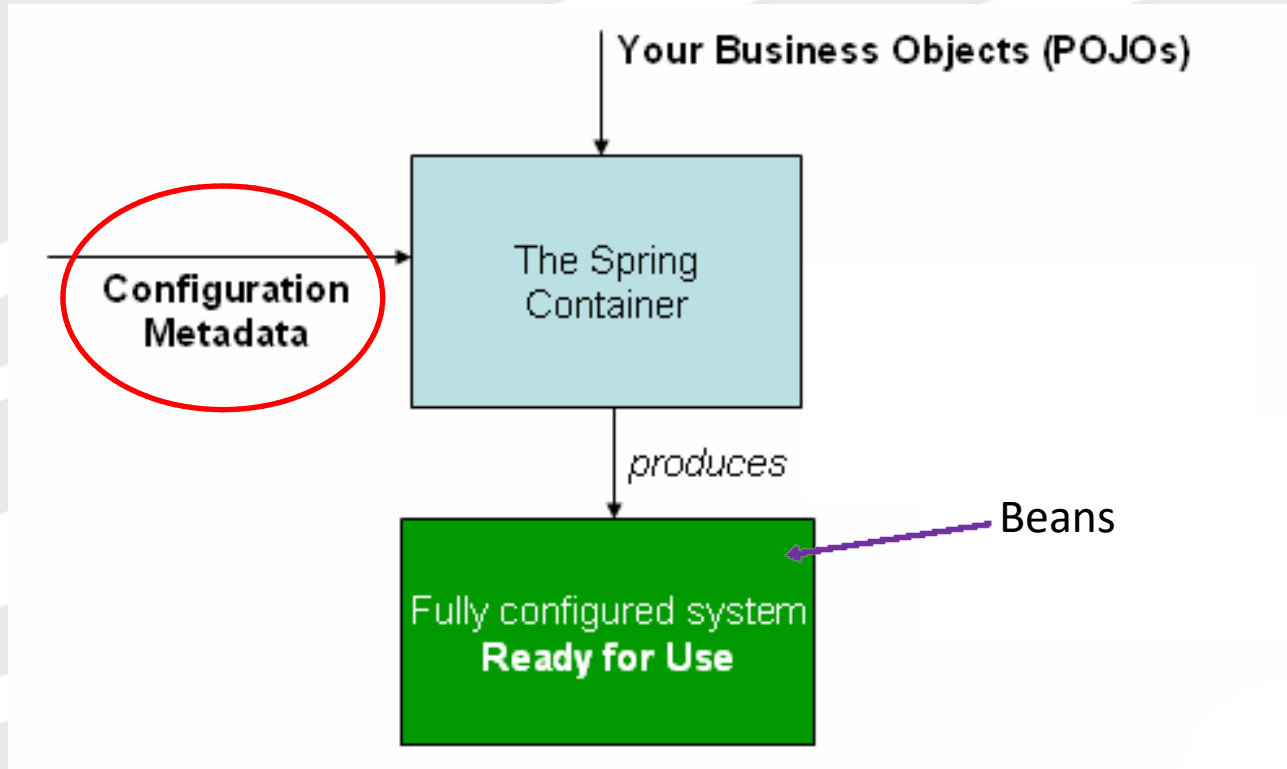
- Design pattern to create an object O1 another object O2 relies on, without knowing, at compile time, which class O1 is instance of
- 3 roles
 - Dependent consumer
 - Interface contract
 - Injector: create instances of classes implementing the interface contract and **inject** the dependency on the dependent consumer
 - The injector selects the class to instantiate

Spring heavily leverages IoC and DI

Spring IoC Container (IoCC)



Spring IoC Container (IoCC)



Spring Beans

- IoCC represents bean definitions as BeanDefinition instances
 - Package-qualified class name
 - Unique identifiers
 - Behavioral configurations
 - Scope
 - Lifecycle callbacks
 - References to other beans (also called dependencies)
 - Other configurations (e.g. bean-specific properties)

Configuration Metadata for IoCC

- 3 techniques
 - XML-Based configuration
 - Annotation-based configuration
 - Annotating classes, attributes, methods
 - Java-based configuration
 - Meta-data hard-coded in a Java Class

Spring Bean Scope

- Singleton (default)
 - Unique instance shared by
- Prototype
 - Any number of instances and the container does not keep any reference after handing the instance to the client
- Request
 - The bean is alive for an HTTP request lifecycle (only web applications)
- Session
 - The bean is alive for an HTTP session (only web applications)
- GlobalSession (only web applications with portlets)
 - The bean is alive for an HTTP global session
- Custom scopes can be created

Use the scope="value" attribute in the bean definition tag

Bean Lifecycle

- 1 Instantiation
- 2 Property initialization
- 3 Bean name set, if any non-default name
- 4 Initializer is called, if any
- 5 Post-initialization processing
- 6 Pre-disposal processing
- 7 Disposal

Spring Bean Autowiring

- Automatic inspection of Spring-managed beans
 - When a dependency of a bean on another bean is detected, it is resolved by the IoCC
- Disabled by default
 - Can be enabled only for some beans
 - Can be enabled for all beans
- Avoids explicitly specifying dependencies among beans via XML (e.g. via property tag)

Annotation-Based Configuration

- `@Component`
 - Identifies a generic Spring-managed bean
- `@Service`, `@Controller` and `@Repository` are specialization of `@Component` for future use
 - `@Repository` identifies a DAO
 - `@Service` annotates beans of the service layer (i.e. controllers in MVC)
 - `@Controller` annotates beans of the presentation layer (i.e. the layer between web view and service layer, e.g. the one managing navigation among pages)

Annotation-Based Configuration

- **@Autowired**
 - Spring-specific
 - Semantics by-type
- **@Inject**
 - JSR-330
- **@PostConstruct**,
@PreDestroy

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
  
    @Autowired  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
  
    // ...  
}
```

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

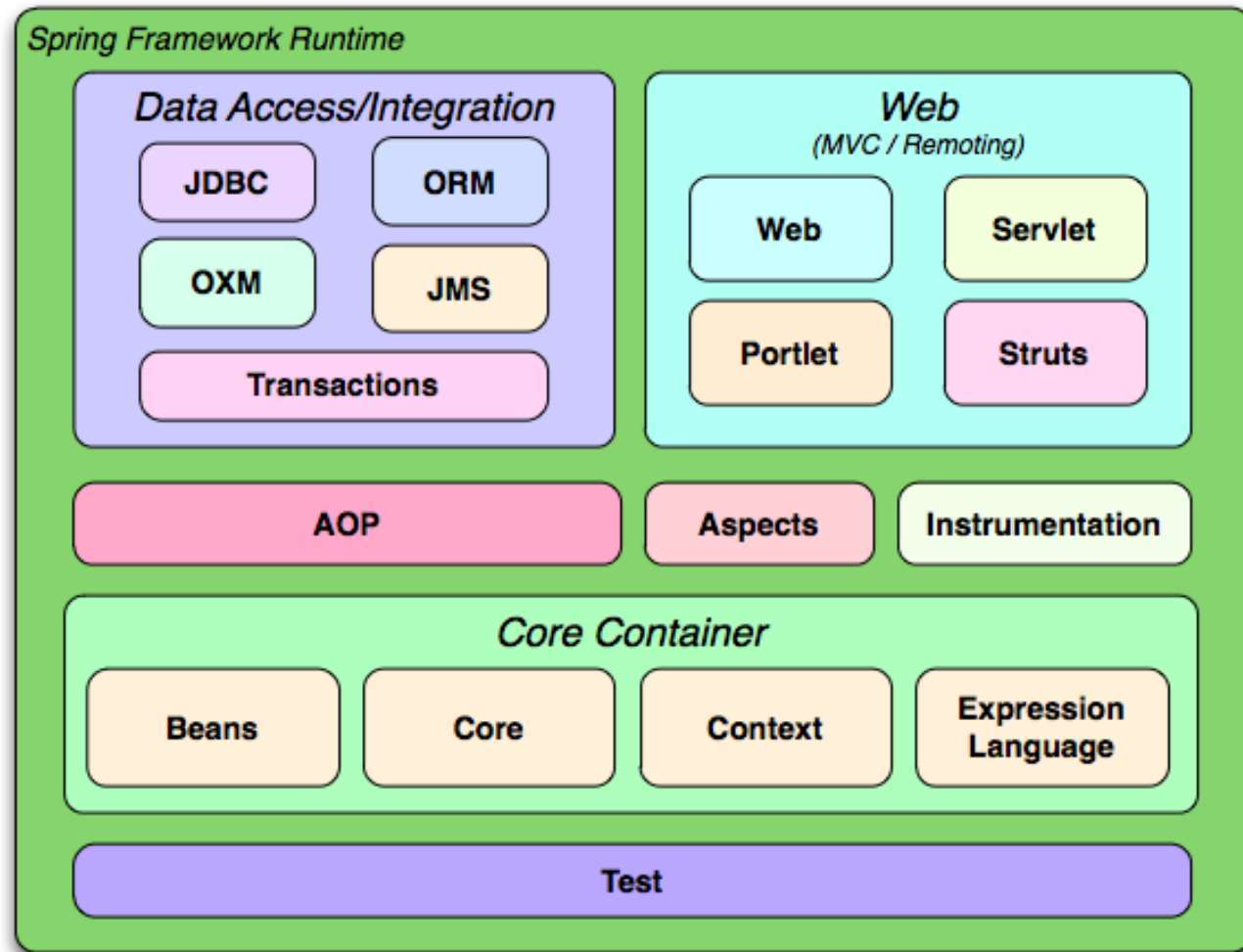
Java-Based Configuration

```
@Configuration
@ComponentScan("net.codejava.spring")
@EnableTransactionManagement
public class ApplicationContextConfig {

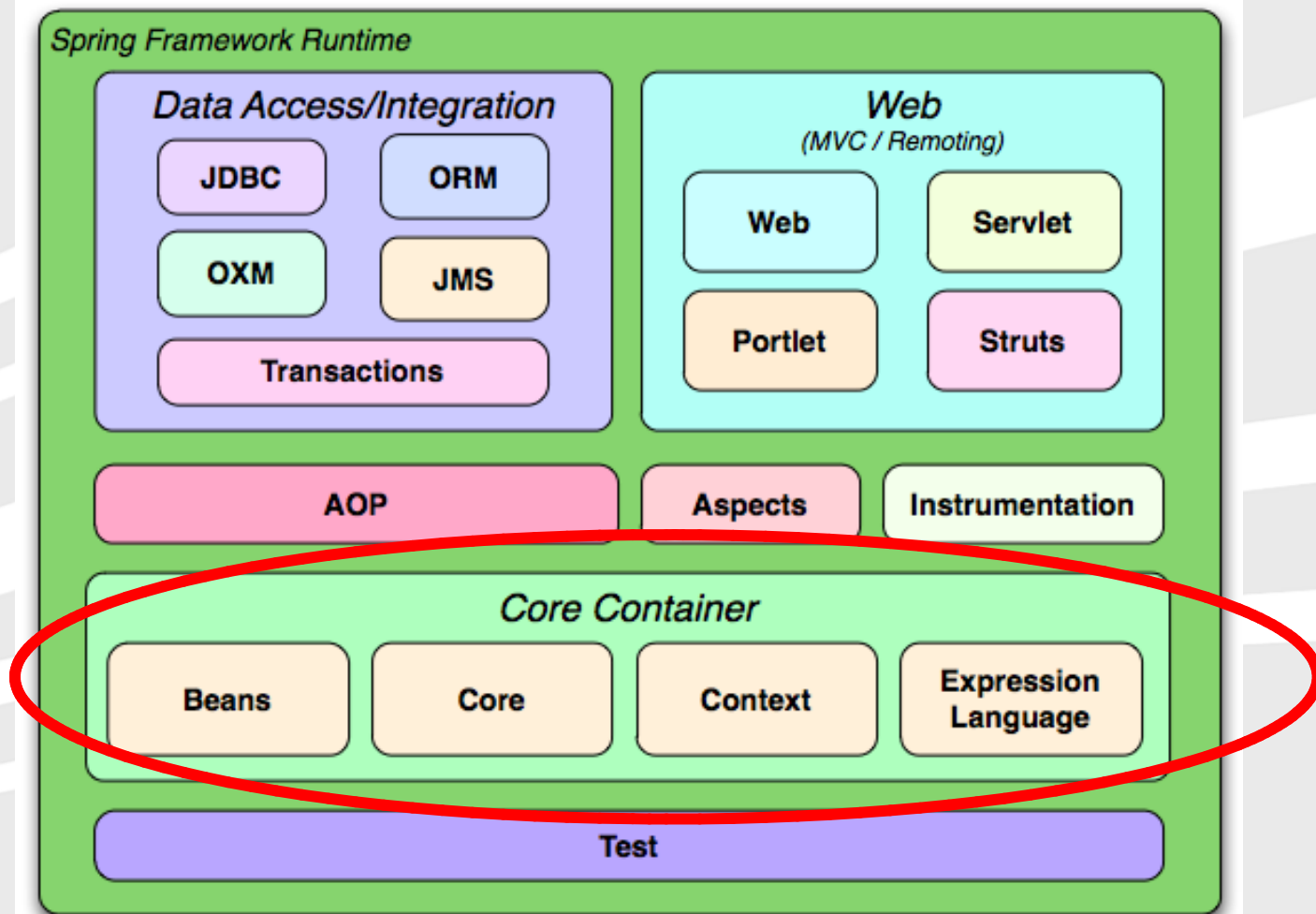
    @Bean(name = "dataSource")
    public DataSource getDataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/usersdb");
        dataSource.setUsername("root");
        dataSource.setPassword("secret");

        return dataSource;
    }
}
```

Spring Framework Overview



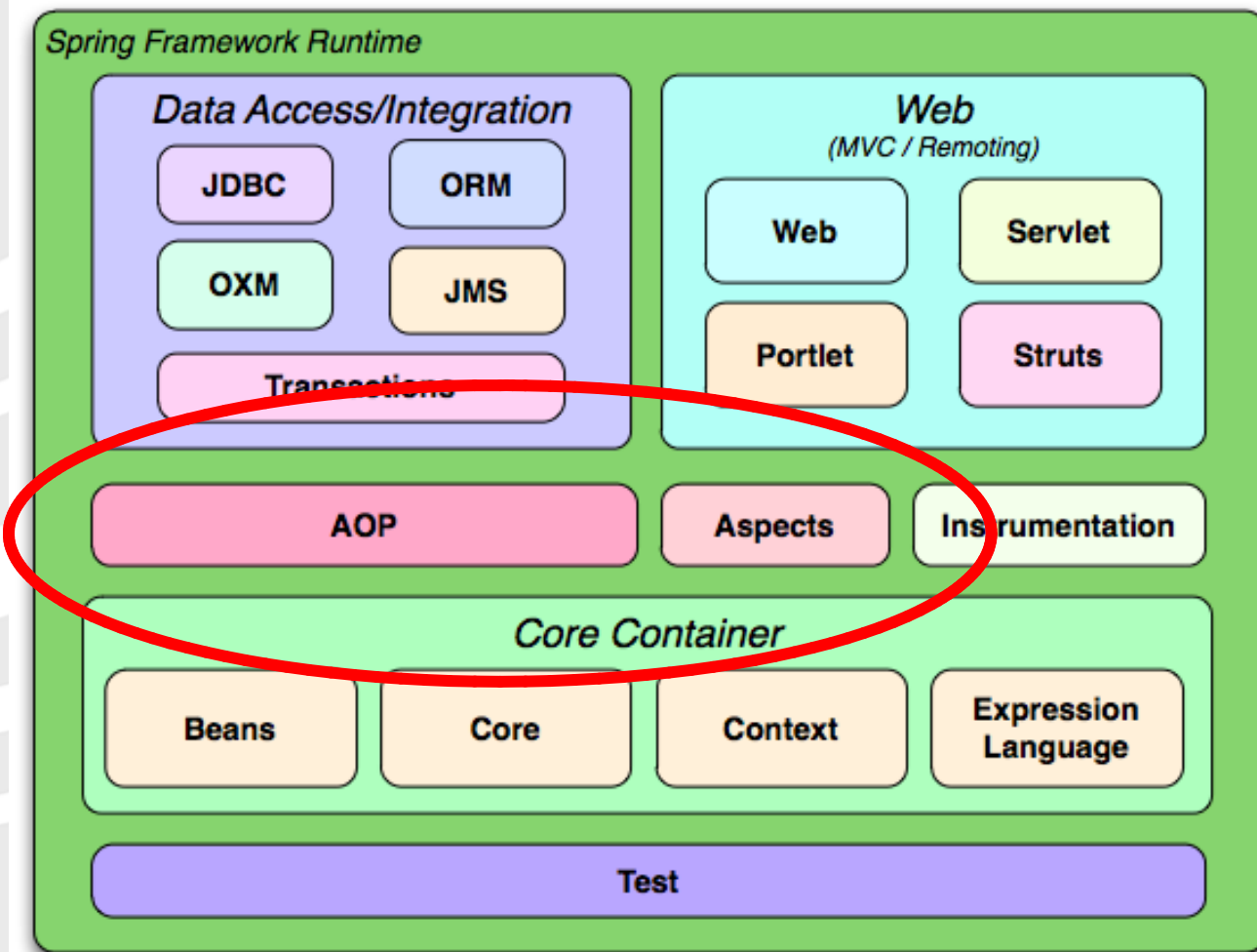
Spring Framework Overview



Spring Framework Overview

- Core Container
 - Beans
 - Bean definitions and management
 - Core
 - Inversion of Control Container and Dependency Injection features
 - BeanFactory is the main interface
 - Context
 - Java EE features for framework-managed objects
 - ApplicationContext and BeanFactory are the main interfaces
 - Expression Language
 - Querying and manipulating framework-managed objects at runtime

Spring Framework Overview



AOP

- *Aspects*, i.e. concepts that apply across multiple type of classes or objects
 - @Aspect annotation or XML configuration in Spring
- *Join point*: a point during the execution of a program (in Spring, a method) that is suitable for aspect weaving
- *Advice*: action taken by an aspect at a particular join point
 - @Before, @After (on smooth termination, on exception, or in all cases), @Around (both before and after)
 - Can receive parameters, e.g. the list of arguments received by the joint point, the implicit object
- *Pointcut*: a predicate that matches join points
 - Associated with Advice to identify when weaving an aspect
- *Weaving*: process of introducing an advice in a matched join points
- In synthesis, pointcuts define which join points get advised, i.e. where aspects get woven.

AspectJ Example

```
package foo;

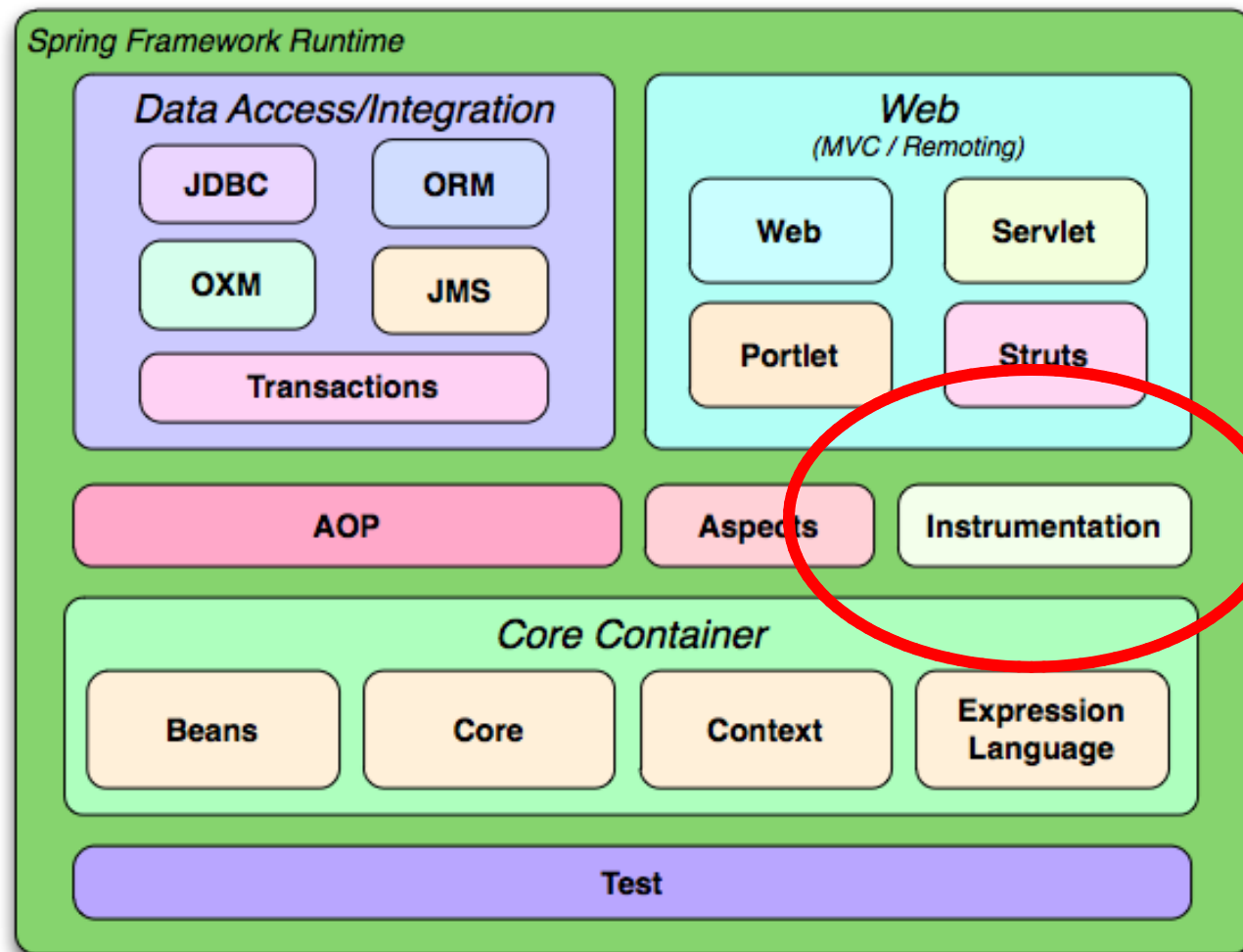
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled() {}
}
```

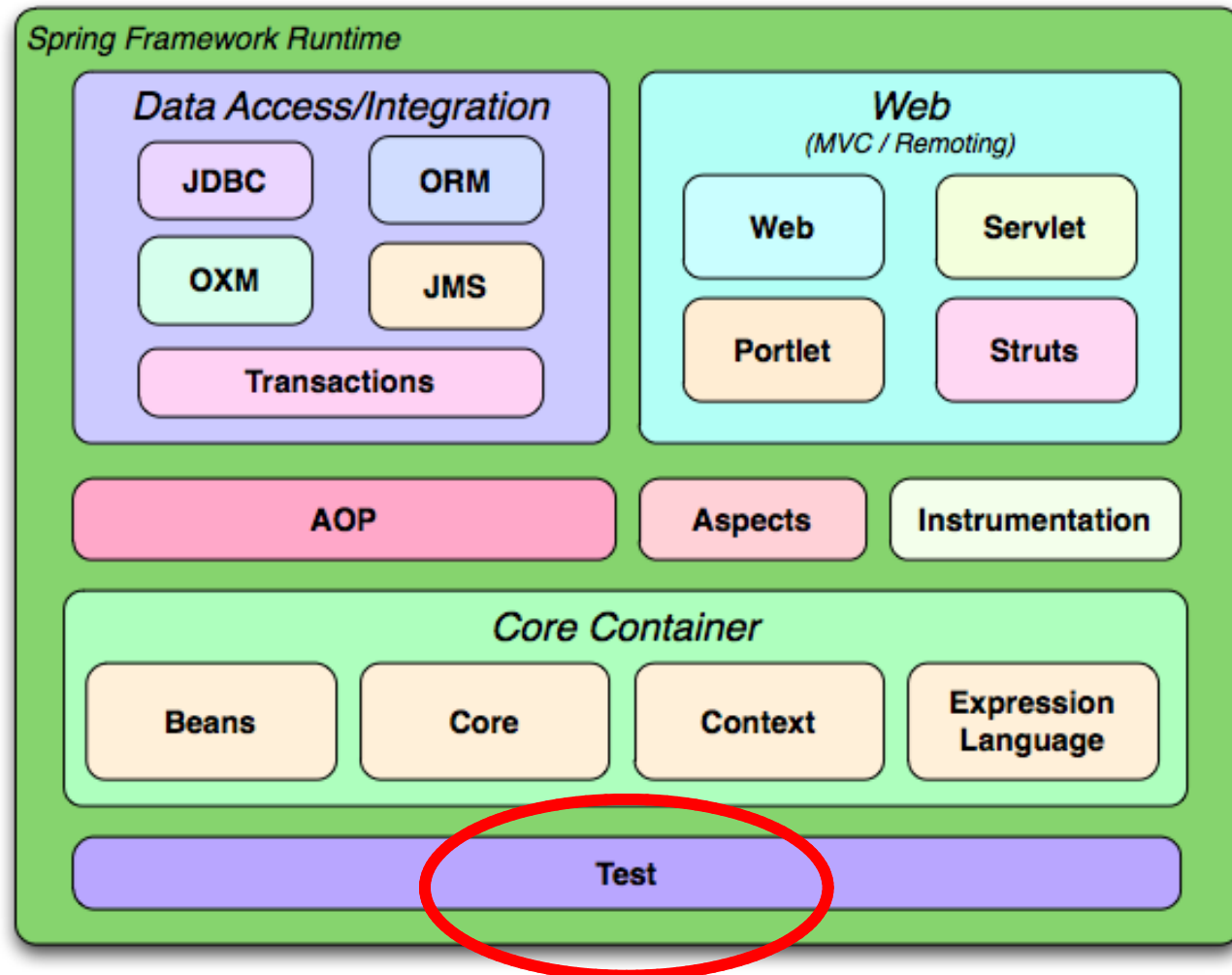
Spring Framework Overview



Spring Instrumentation Module

- Support to class loading
- Class weaving
- Utility class to deploy on specific application servers
- Low-level API, rarely used by application software developers

Spring Framework Overview



Spring Test Module

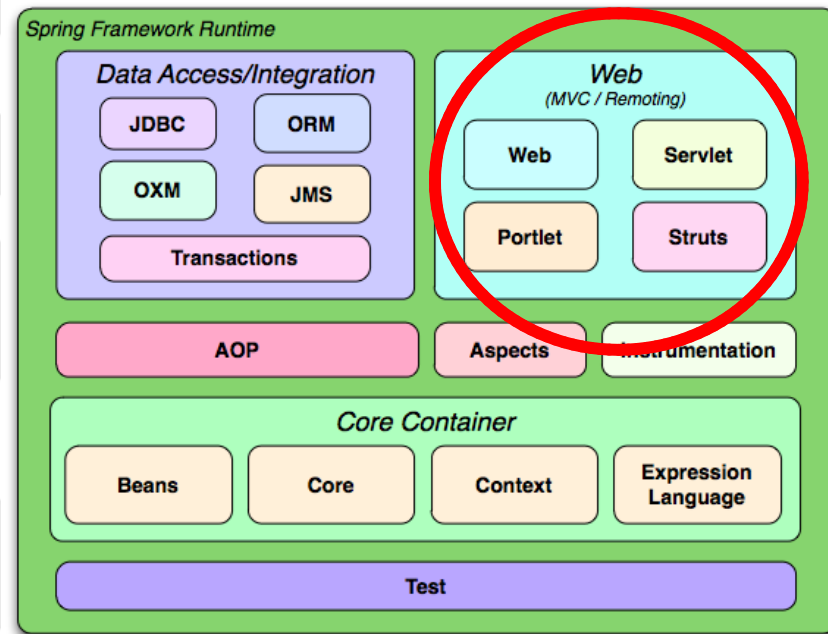
- Support to run tests within a Spring Container
 - Leverage Spring features during testing, e.g. annotations
 - Access to ApplicationContext
- Additional annotations for tests, e.g.
 - **@Timed(t)**
 - **@Repeat(n)**
- Configurability of tests for different architectural layers, e.g.
 - Disable rollback when testing a transaction
 - Manage web session when testing a controller
 - Build an HTTP request for a REST API
- Full integration with JUnit, support for TestNG

Agenda

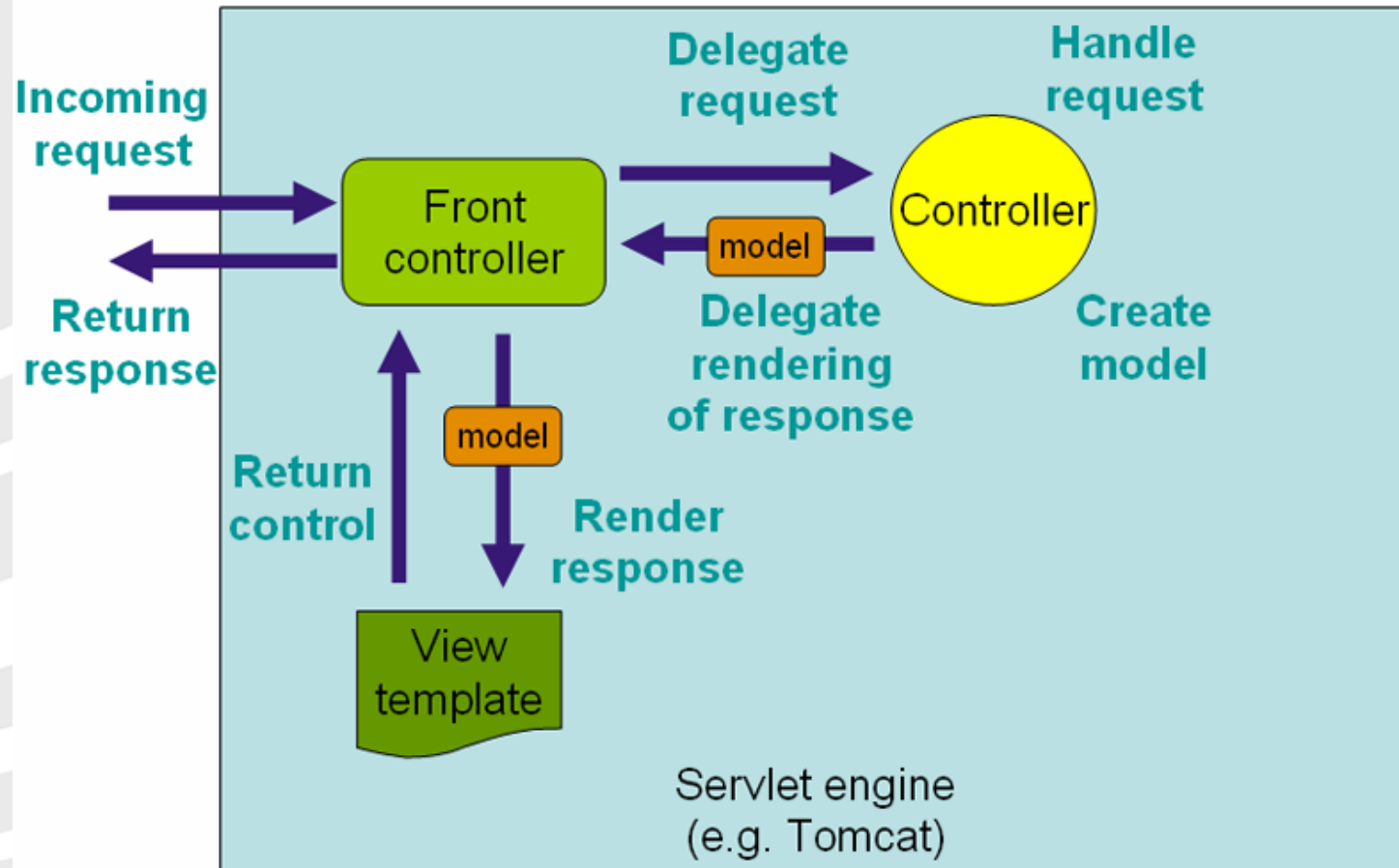
- Spring Inversion of Control and Dependency Injection
 - Spring Context
 - Spring Beans
 - Spring Configuration
- **Spring MVC**
- Spring Data
 - Hibernate
 - JPA
 - Spring Repository
- Additional Spring Components and Projects

Spring Framework Overview

- Web
 - Web
 - Features for multipart file management, web services...
 - Servlet
 - Spring's MVC implementation
 - Portlet
 - Struts



Spring MVC



Spring MVC

- DispatcherServlet
 - Actual servlet that extends HttpServlet
 - URL mapping of requests to be managed by such servlet (via web.xml)
 - @RequestMapping
 - Can access beans from the root application context or from the context associated to it (if any dispatcher-scoped context exists)
 - Requires a WebApplicationContext as ApplicationContext

Spring MVC

- Controllers
- HandlerMapping: handle the execution of pre-processors, post-processors and controllers
- ViewResolver: resolves view names to views
- LocaleResolver: resolves the locale a client is using
- ThemeResolver: resolves the view theme to use
- MultipartResolver: processes file uploads from HTML forms
- HandlerExceptionResolver: map exceptions to views or handle exceptions

Spring MVC

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(value="/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(value="/new", method = RequestMethod.GET)
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

→ All handling methods on this controller are relative to the /appointments (not required)

→ Only accepts GET requests, meaning that an HTTP GET for /appointments invokes this method

@PathVariable binds a parameter to the value of a URI template variable, i.e. {day}

→ GET requests for appointments/new are handled by this method

@Valid requires the parameter to pass the validation via default or custom validator

JSON

- Javascript Object Notation
- Open standard to exchange data between applications
- Used to exchange data between server and client of a web application
 - Alternative to XML
- Data types: number, string, boolean, array and complex object
 - null as special value

JSON Example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd
Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Spring REST

- REST
 - REpresentational
 - State
 - Transfer
- Main REST constraints
 - Client server (on the web)
 - Stateless (no state stored between requests)
 - Uniform interface for communication
- `@RestController` annotations is the same as `@Controller` + `@ResponseBody` for all methods

REST Controller Example

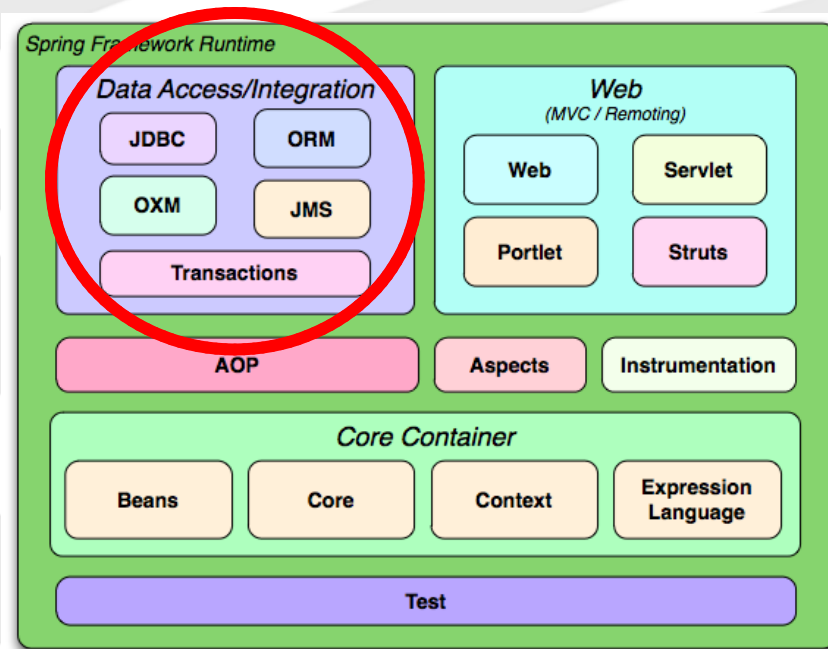
- Web Controller
- Business Logic Controller
- Entity
- Client resources
 - HTML
 - JSP
 - JSON

Agenda

- Spring Inversion of Control and Dependency Injection
 - Spring Context
 - Spring Beans
 - Spring Configuration
- Spring MVC
- **Spring Data**
 - Hibernate
 - JPA
 - Spring Repository
- Additional Spring Components and Projects

Spring Framework Overview

- Data Access/Integration
 - JDBC
 - Abstraction layer from vendor-specific coding (e.g. exceptions)
 - ORM
 - Integration with popular Object-Relational mapping APIs, e.g. Hibernate
 - OXM
 - Integration with popular Object-XML mapping APIs, e.g. JAXB
 - JMS
 - Features for message exchange
 - Transactions
 - Feature for declarative and programmatic transactions management

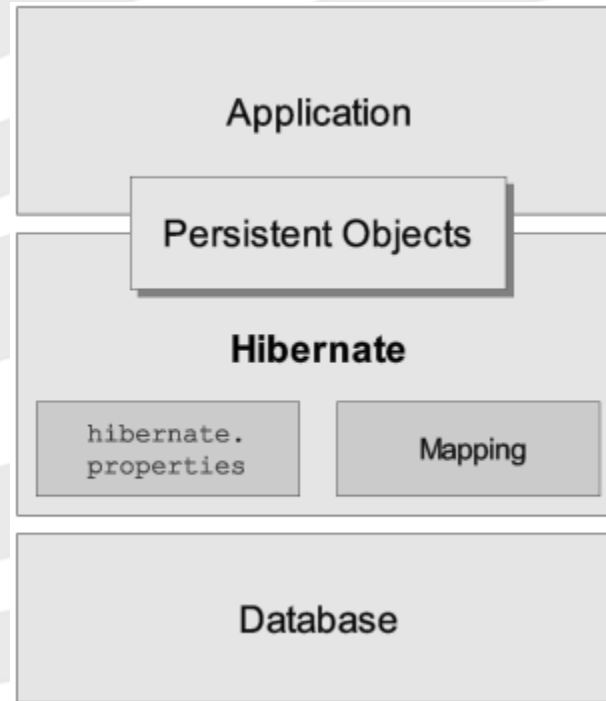


What is Hibernate?

- Framework for Java
- For ORM: Object-Relational Mapping
 - Mapping of Classes to Tables
 - Generates SQL for the schema creation
 - SQL generation from Java instructions
 - No need of conversion from query results to objects

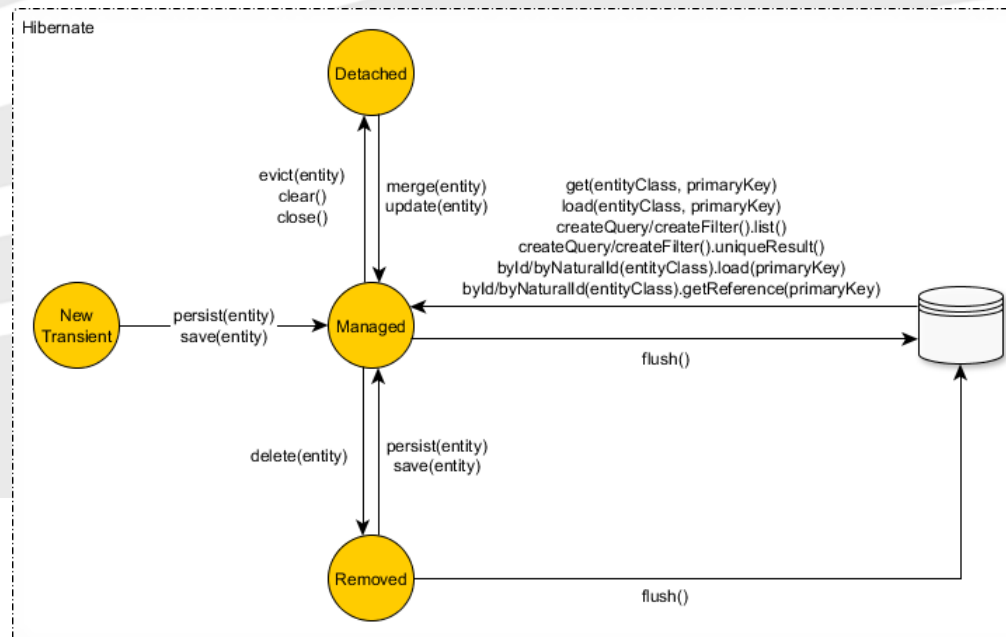


Hibernate Architecture

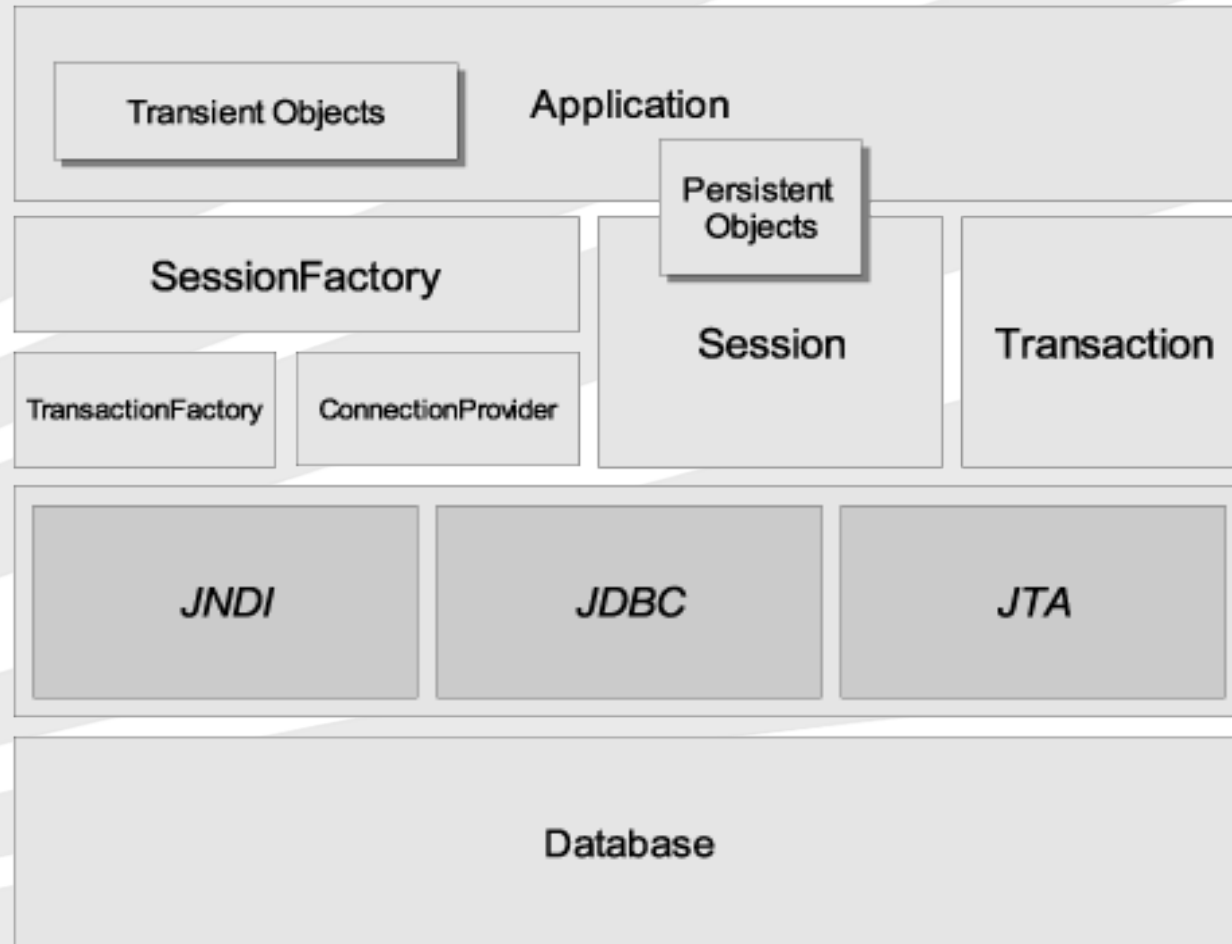


Hibernate Object States

- Transient
 - Instantiated using the **new** operator, and not associated with a Session
 - No persistent representation in the database
- Persistent
 - Has a representation in the database and an identifier value
 - Hibernate detects changes made to an object and synchronizes the state
- Detached
 - Has been persistent and its Session has been closed
 - Java reference still valid and might be modified
 - Can be reattached to a new Session making it persistent again



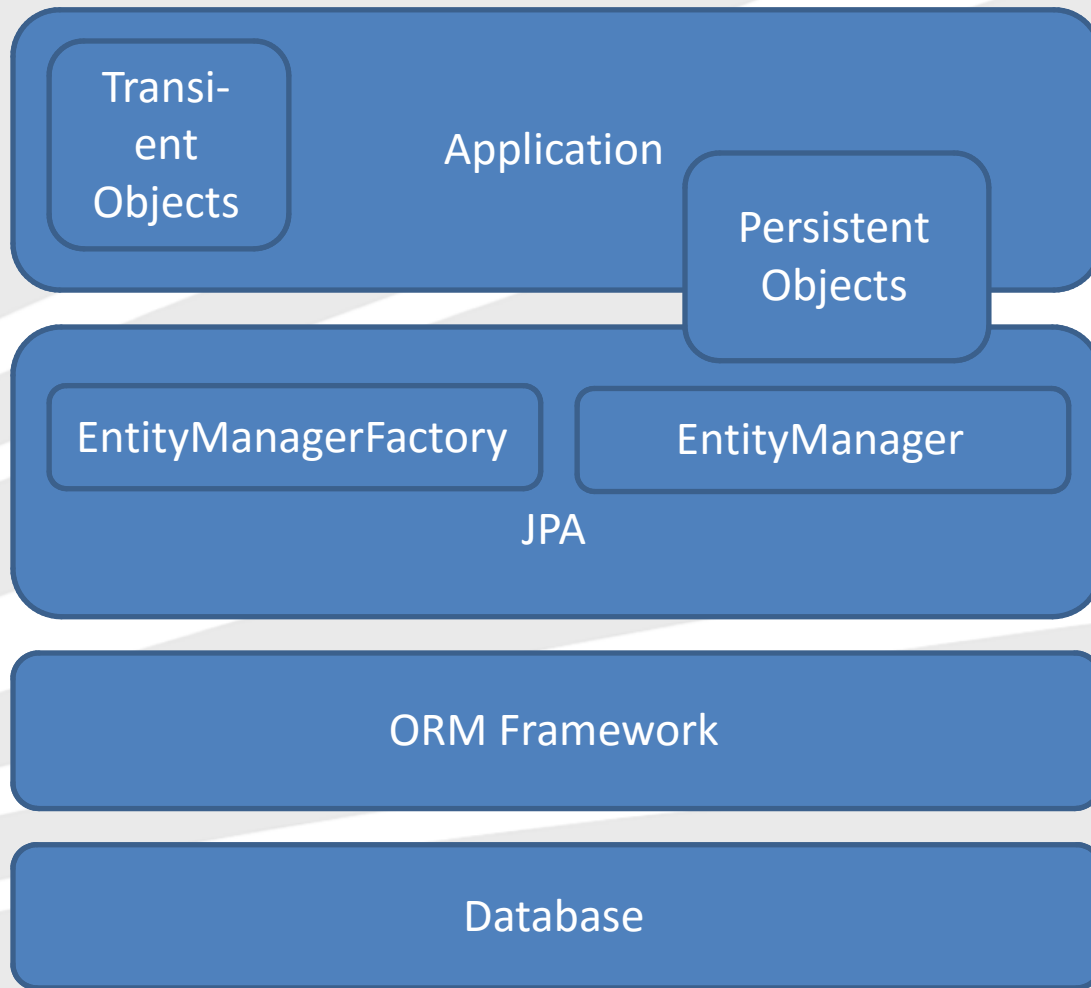
Main Hibernate Components



Java Persistence API (JPA)

- Standard Java interface to setup and manage persistence
 - Provides a set of interfaces and annotations
 - Different JPA implementations can be configured
 - Hibernate version 4.3 (Dec 2013) implements latest JPA (2.1, May 2013)
- Decouples the Java application business logic from the ORM-specific components
 - Possible as long as no vendor-specific feature is used by the application

JPA Architecture



Basic Examples

- Check the example code for the four alternatives
 - Configuration
 - Store (insert)
 - Update
 - Delete
 - Find (select)

Basic Examples: Config Comparison

Hibernate

- Configuration file
 - hibernate.hbm.xml
 - List all mapped classes, either annotated or configured via hbm files
- Entity mapping
 - Annotations
 - <entity_name>.hbm.xml
 - One per entity
 - Required for each entity with no annotations



JPA

- Configuration file
 - persistence.xml
 - Reference the entity mapping file, not annotated classes
- Entity mapping
 - Annotations
 - orm.xml
 - One section per entity
 - Required for each entity with no annotations

Basic Examples: Init Comparison

Hibernate

```
public class DBResourcesManager {  
    private static Configuration configuration;  
    private static ServiceRegistry serviceRegistry;  
    private static SessionFactory sessionFactory;  
  
    public static void initHibernate() {  
        // load hibernate configuration  
        configuration = new Configuration();  
        configuration.configure();  
  
        // use JNDI to bind Hibernate configuration and  
        // datasource  
        serviceRegistry = new  
        StandardServiceRegistryBuilder().applySettings(  
        configuration.getProperties()).build();  
  
        /* Retrieve the one session factory that will manage  
        sessions, connections and transaction*/  
        sessionFactory =  
        configuration.buildSessionFactory(serviceRegistry);  
    }  
}
```



JPA

```
public class DBResourcesManager {  
    private static EntityManager entityManager;  
    private static EntityManagerFactory entityManagerFactory;  
  
    public static void initPersistence() {  
        entityManagerFactory =  
        Persistence.createEntityManagerFactory("pu");  
        entityManager =  
        entityManagerFactory.createEntityManager();  
    }  
}
```

Basic Examples: Store Comparison

Hibernate

```
public class EventDaoHibernate {  
  
    public static void store(Event e) {  
        Session s =  
            DBResourcesManager.getSession();  
  
        s.beginTransaction();  
        s.save(e);  
        s.getTransaction().commit();  
  
        s.close();  
    }  
}
```

JPA

```
public class EventDaoJPA {  
  
    public static void store(Event e) {  
        EntityManager em =  
            DBResourcesManager.getEntityManager();  
  
        em.getTransaction().begin();  
        em.persist(e);  
        em.getTransaction().commit();  
    }  
}
```



Basic Examples: Update Comparison

Hibernate

```
public class EventDaoHibernate {  
  
    public static void updateEvent(Event toUpdate) {  
        Session s = DBResourcesManager.getSession();  
        s.beginTransaction();  
        s.update(toUpdate);  
        s.getTransaction().commit();  
    }  
}
```



JPA

```
public class EventDaoJPA {  
  
    public static void updateEventB(Event toUpdate) {  
        EntityManager em = DBResourcesManager.getEntityManager();  
        em.getTransaction().begin();  
        Event loaded = em.find(Event.class, toUpdate.getId());  
        loaded.updateEvent(toUpdate); // manual update of fields required  
        em.getTransaction().commit();  
    }  
  
    public static void updateEventA(Event toUpdate) {  
        EntityManager em = DBResourcesManager.getEntityManager();  
        em.getTransaction().begin();  
  
        String sql = "UPDATE Event SET eventDate = :d, title = :t WHERE id  
        = :i";  
        Query query = em.createQuery(sql);  
        query.setParameter("d", toUpdate.getEventDate());  
        query.setParameter("t", toUpdate.getTitle());  
        query.setParameter("i", toUpdate.getId());  
        query.executeUpdate();  
  
        em.getTransaction().commit();  
    }  
}
```

Basic Examples: Delete Comparison

Hibernate

```
public class EventDaoHibernate {
```

```
    public static void deleteEvent(Event toDelete) {  
        Session s = DBResourcesManager.getSession();  
        s.beginTransaction();  
        s.delete(toDelete);  
        s.getTransaction().commit();  
    }  
}
```

JPA

```
public class EventDaoJPA {
```

```
    public static void deleteEvent(Event toDelete) {  
        EntityManager em = DBResourcesManager.getEntityManager();  
        em.getTransaction().begin();  
        Event loaded = em.find(Event.class, toDelete.getId());  
        em.remove(loaded);  
        em.getTransaction().commit();  
    }  
}
```



Basic Examples: Find Comparison

Hibernate

```
public class EventDaoHibernate {  
  
    public static List<Event> findAllEventsA() {  
        Session s = DBResourcesManager.getSession();  
  
        @SuppressWarnings("unchecked")  
        List<Event> events = s.createQuery("from Event").list();  
  
        return events;  
    }  
  
    public static List<Event> findAllEventsB() {  
        Session s = DBResourcesManager.getSession();  
  
        @SuppressWarnings("unchecked")  
        List<Event> events =  
            s.createCriteria(Event.class).list();  
  
        return events;  
    }  
}
```



JPA

```
public class EventDaoJPA {  
  
    public static List<Event> findAllEventsA() {  
        EntityManager em =  
            DBResourcesManager.getEntityManager();  
        List<Event> events = em.createQuery("from Event",  
            Event.class).getResultList();  
  
        return events;  
    }  
  
    public static List<Event> findAllEventsB() {  
        EntityManager em =  
            DBResourcesManager.getEntityManager();  
        CriteriaQuery<Event> query =  
            em.getCriteriaBuilder().createQuery(  
                Event.class);  
        List<Event> events = em.createQuery(query).getResultList();  
  
        return events;  
    }  
}
```



Annotations (1)

- **@Entity**
 - Marks a class to be persisted
 - All non-static non-transient fields are persisted
 - If a non-static non-transient fields has not to be persisted, you can annotate it with @Transient
- **@Table**
 - Optionally adds information on the table corresponding the annotated class
 - Properties
 - *name*: defaults to the entity name
 - *indexes*: defines an array of indexes via @Index(columnList)
 - *uniqueConstraints*: defines an array of unique constraints via @UniqueConstraint(columnList)

Annotations (2)

- **@Id**
 - Marks the id field
 - Can be: a primitive type, a wrapper type, String, Date, BigDecimal, BigInteger
 - JPA supports a single field as ID, Hibernate more fields can be the ID
- **@GeneratedValue**
 - Adds information on the ID generation
 - Properties
 - *strategy*: way of generating IDs, including:
 - AUTO: delegates the decision to the persistence provider
 - IDENTITY: supported by many DBMS, including MySQL

Annotations (3)

- **@Column**
 - Optionally adds information on the column corresponding the annotated field
 - Properties
 - *name*: defaults to the field name
 - *nullable* (can be null or not)
 - *unique* (is a key or not)
 - *length*

Relationships Mapping Types

- @OneToOne
- @ManyToOne/@OneToMany
- @ManyToMany
- Each can be unidirectional or bidirectional

One-to-One Mapping

- Mapped with a foreign key (join column)
- Unidirectional
 - Field annotated `@OneToOne` on the owning side
- Bidirectional
 - Field annotated `@OneToOne` on the owning side
 - `mappedBy` specified in the `@OneToOne` in non-owning side

One-to-Many/Many-to-One Mapping

- Used to map collections
- One-to-Many mapped with a join table
- Many-to-One mapped with a foreign key (join column)
- Unidirectional
 - Add `@OneToMany/@ManyToOne` on the field
- Bidirectional
 - With “one” owning the relation
 - Add `@OneToMany` in the owning side
 - *mappedBy* specified in the `@OneToMany` in non-owning side
 - With “many” owning the relations
 - Add `@ManyToOne` in the owning side
 - Add `@OneToMany` in the non-owning side

Many-to-Many Mapping

- Used to map collections
- Mapped with a join table
- Unidirectional
 - Add `@ManyToMany`
- Bidirectional
 - Add `@ManyToMany`
 - *mappedBy* specified in the non-owning side

Relationships Mappings Attributes

- *optional*: whether the reference can be null
- *fetchType*: fetching strategy set via `@FetchType`
 - EAGER: related entity is immediately fetched
 - LAZY: related entity is automatically fetched as soon as first accessed in the application
 - Not usable when classes are final
- *cascade*: sets triggers on operations according to its value
 - PERSIST (ON INSERT)
 - REFRESH (ON UPDATE)
 - REMOVE (ON DELETE)
 - ALL
- *orphanRemoval*: whether the referenced entity removal at application-level (i.e. setting the field to null) should cause a removal in the database of the formerly related entity
 - Cascade.DELETE produces an ON DELETE trigger, while orphanRemoval does not require the removal of the parent entity in order to remove its child

Inheritance Mapping

- **@Inheritance**
 - Three strategies to map inheritance
 - SINGLE_TABLE: single table for all entities with the addition of a discriminating column
 - JOINED: a subclass persists only its own properties and keeps a reference to its parent
 - TABLE_PER_CLASS: each concrete class has its own table containing all attributes defined by parent classes and its own attributes
- **@MappedSuperclass**
 - Fields of this class are mapped (copied) into its subclasses
 - A table for this class is not created, as its instances are not persistent objects

Type Mappings

- @Enumerated
 - Either STRING or ORDINAL
- @Lob
 - Mapped to MySQL TEXT or BLOB, according to the field type
- @Temporal
 - Either DATE, TIME or TIMESTAMP

Component Mapping

- **@Embedded**
 - Marks a field whose fields will become part of the owning entity table (i.e. no foreign key to the embedded entity)
- **@Embeddable**
 - Marks an entity that can be embedded into another entity
- **@EmbeddedId**
 - Used to mark an ID that is not a supported ID type, but another complex type
 - The complex type is a class that implements `Serializable` and that is marked with `@Embeddable`

Type Mapping: Collections

- If using a List, optionally add `@OrderBy` to store it according to some order
 - Columns to order by are specified as attribute
- If using a Map, add `@MapKeyColumn`, `@MapKeyEnumerated`, `@MapKeyTemporal` or `@MapKeyJoinColumn` to indicate the name of the key column
 - The correct annotation depends on the type of the map key (Basic, Enumeration, Temporal or Entity)
- If the collection contains basic type or embeddable type objects, use `@ElementCollection`

Spring Data Annotations

- **@Repository**
 - Mark a class/interface as DAO
 - Can be a class
 - Implement JpaRepository and define custom methods
 - Leverage the EntityManager
 - Leverage ORM specific features
 - Can be an interface
 - Define operations according to some “convention”
 - Obtain their implementations automatically
 - Generated and provided by Spring
 - E.g. findByUsernameAndPassword(String username, String password)
 - E.g. findByNameLike(String nameLike)

More on Spring: Spring Boot

- “Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.”
- Easy creation of a “Hello World web example”
 1. Create a Maven project (or Gradle)
 2. Add the following dependencies

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.3.RELEASE</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

3. Create a Controller class to manager HTTP requests

- ... or clone this repository: <https://github.com/spring-guides/gs-spring-boot.git>
- or use the Spring initilaizer: <http://start.spring.io/>

More on Spring

- Spring Security
- Spring Integration
- Book reference: ***Spring in Action, 3rd Edition***, by C. Walls, Manning