



January 2022,
IPT Course Hibernate

Introduction to ORM, Hibernate, Spring and Spring Boot

Trayan Iliev

tiliev@ipproduct.org

<http://ipproduct.org>

Copyright © 2003-2022 IPT - Intellectual
Products & Technologies

Agenda for This Session

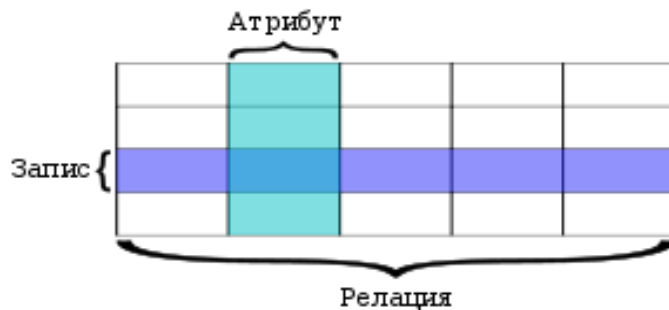
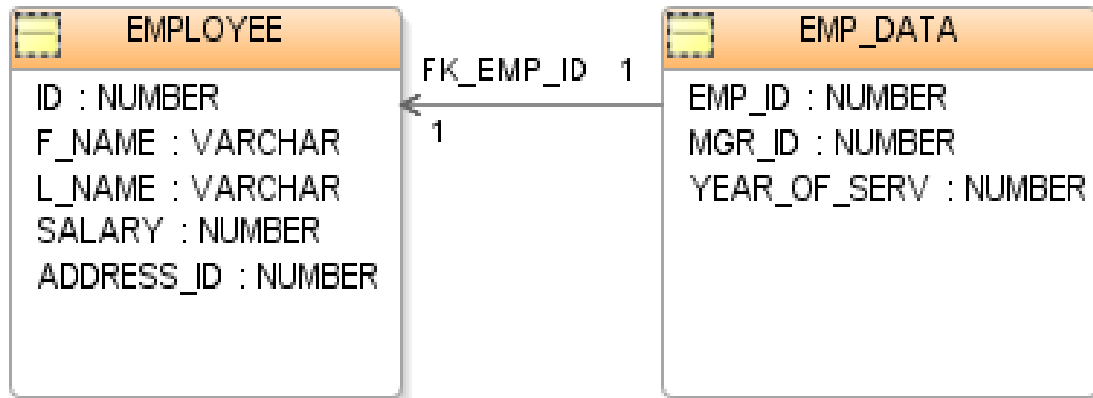
- Relational model, SQL, indexes, transactions
- Domain Driven Design (DDD) – value types, entities, repositories, services, web controllers, mapping
- DAO pattern
- Hibernate and JPA - Spring Boot starter JPA
- Integration with Spring MVC and Spring Boot.
- Database Connections and DataSources
- Configuring connection handling and transaction types
- Spring JDBC – JdbcTemplate, NamedParameterJdbcTemplate
- Practical project: User Management API

Where to Find the Demo Code?

Introduction to Spring 5 demos and examples are available @ GitHub:

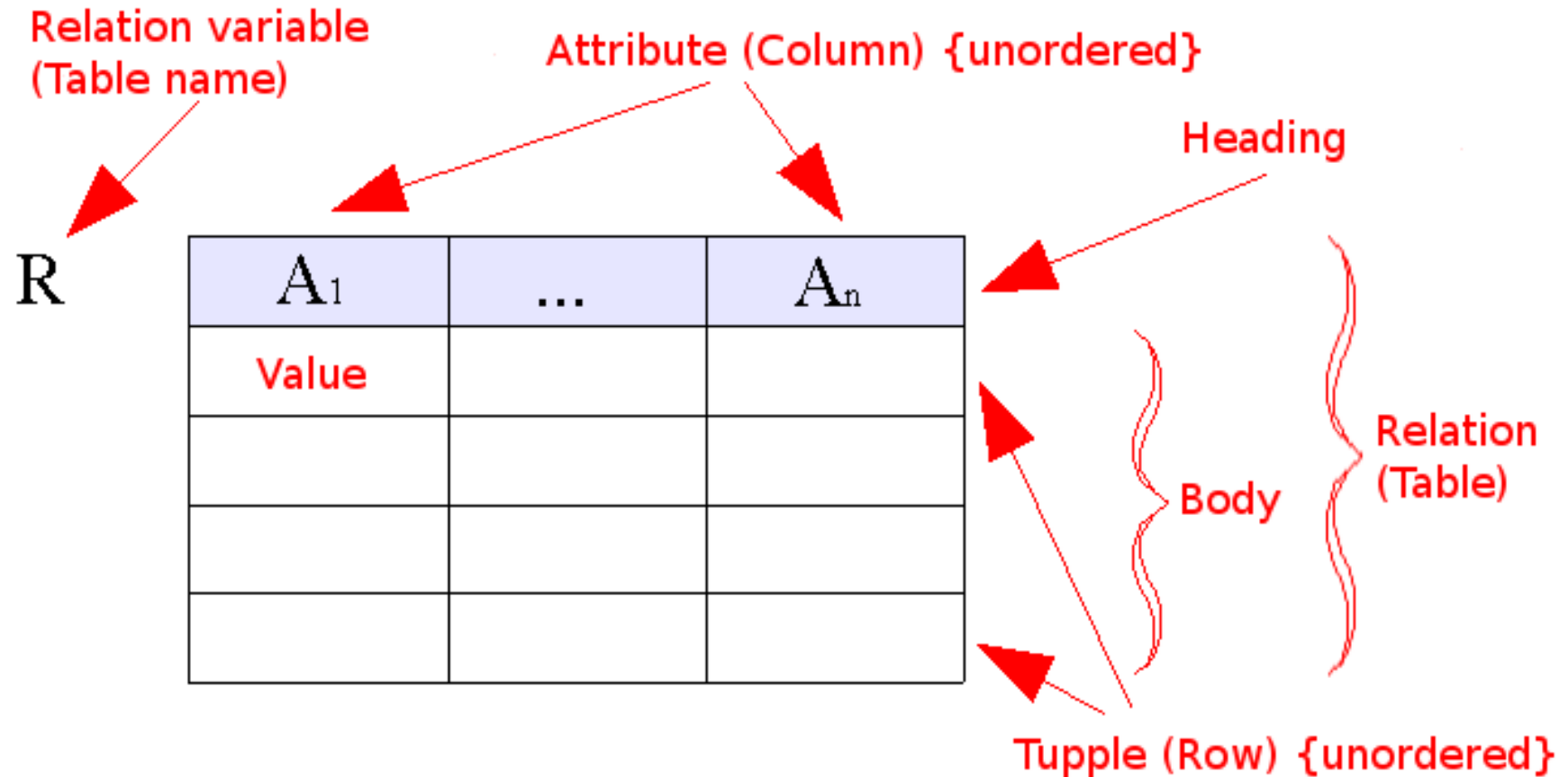
<https://github.com/iproduct/course-hibernate>

Relational Model



- релация, релационна схема (relation) \leftrightarrow таблица (table),
- запис, кортеж (tuple) \leftrightarrow ред (row)
- атрибут, поле (attribute) \leftrightarrow стълб, колона (column)

Relational Model



Views. Domains. Constraints

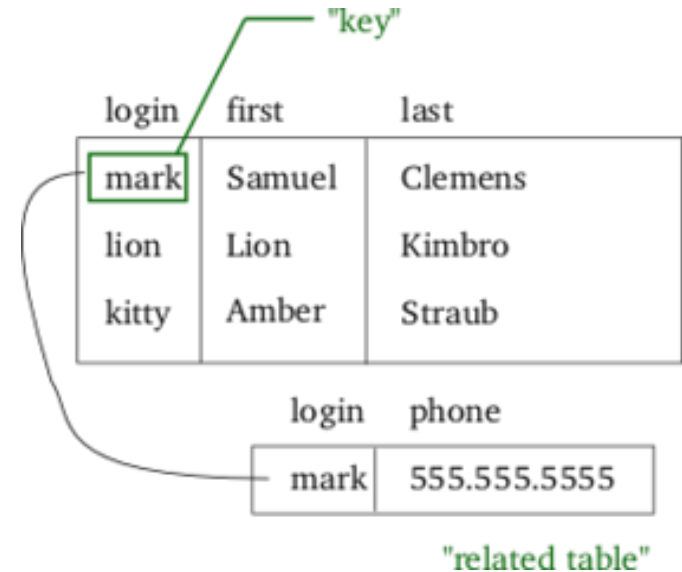
- ❖ **Def:** Relations which store primary data are called **base relations or tables**. Other relations, which are derived from primary relations are **queries** and **views**.
- ❖ **Def: Domain** in database is a set of allowed values for a given attribute in a relation – an existing constraint about valid the type of values for given attribute.
- ❖ **Def: Constraints** allow more flexible specification of values that are valid for given attribute – e.g. from 1 to 10.

Keys

❖ **Key** consists of one or more attributes, such as:

- 1) relation has no two records with the same values for these attributes
- 2) there is no proper subset of these attributes with the same property

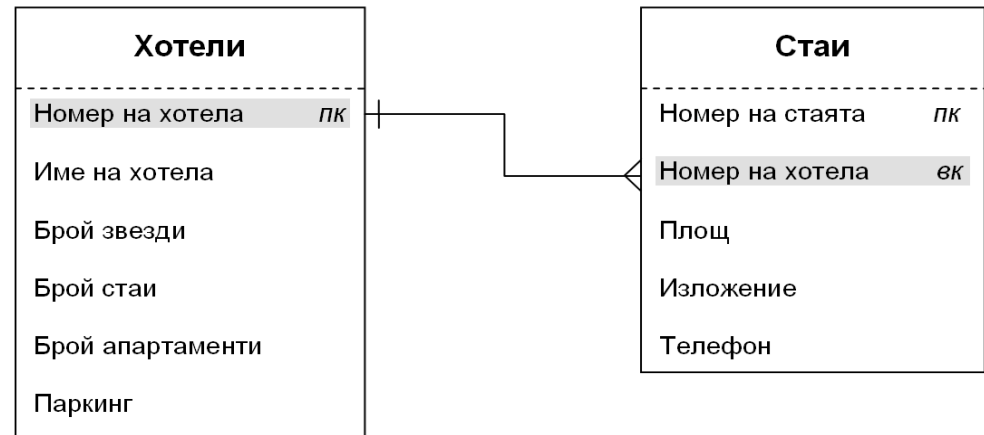
❖ **Primary Key** is an attribute (less frequently a group of attributes), which uniquely identifies each record (tuple) in the relation



❖ **Foreign key** is necessary when **there** exists a relation between two tables

Table Relations. Cardinality

❖ Relationship is a dependency existing between two tables, when the records from first table can be connected somehow with records from second one.

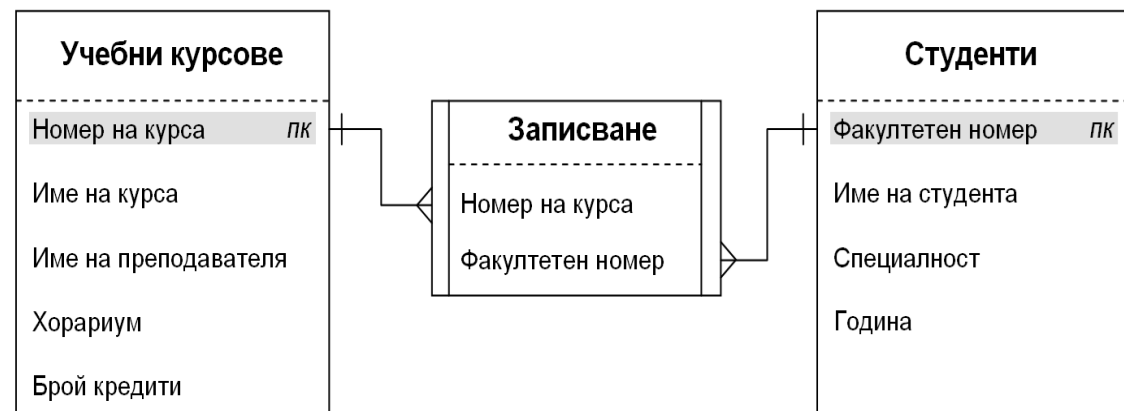


❖ Cardinality:

❖ One to one (1:1),

❖ One to many (1:N),

❖ Many to many (M:N)



Java EE Specs Supported by Spring

- ❖ Servlet API (JSR 340)
- ❖ WebSocket API (JSR 356)
- ❖ Concurrency Utilities (JSR 236)
- ❖ JSON Binding API (JSR 367)
- ❖ Bean Validation (JSR 303)
- ❖ JPA (JSR 338)
- ❖ JMS (JSR 914)
- ❖ JTA/JCA transaction coordination
- ❖ Dependency Injection (JSR 330)
- ❖ Common Annotations (JSR 250)

Which Problems Spring Addresses?

- ❖ **Scalability** and **modularity**
- ❖ Boiler plate code – using **templates** (JdbcTemplate, HibernateTemplate) and **aspects** (advises)
- ❖ Handling non-functional requirements – **transactions**, **load scaling**, **security**, **logging**, **testability**, **maintainability**, etc.
- ❖ **Unit testing** and **integration testing**
- ❖ Complex frameworks/application servers – **POJO** vs. **EJB**
- ❖ Code coupling – **interfaces** + **Dependency Injection (DI)**
- ❖ Separating **What?** From **How?** - declarative programming using XML config files, annotations & functional composition



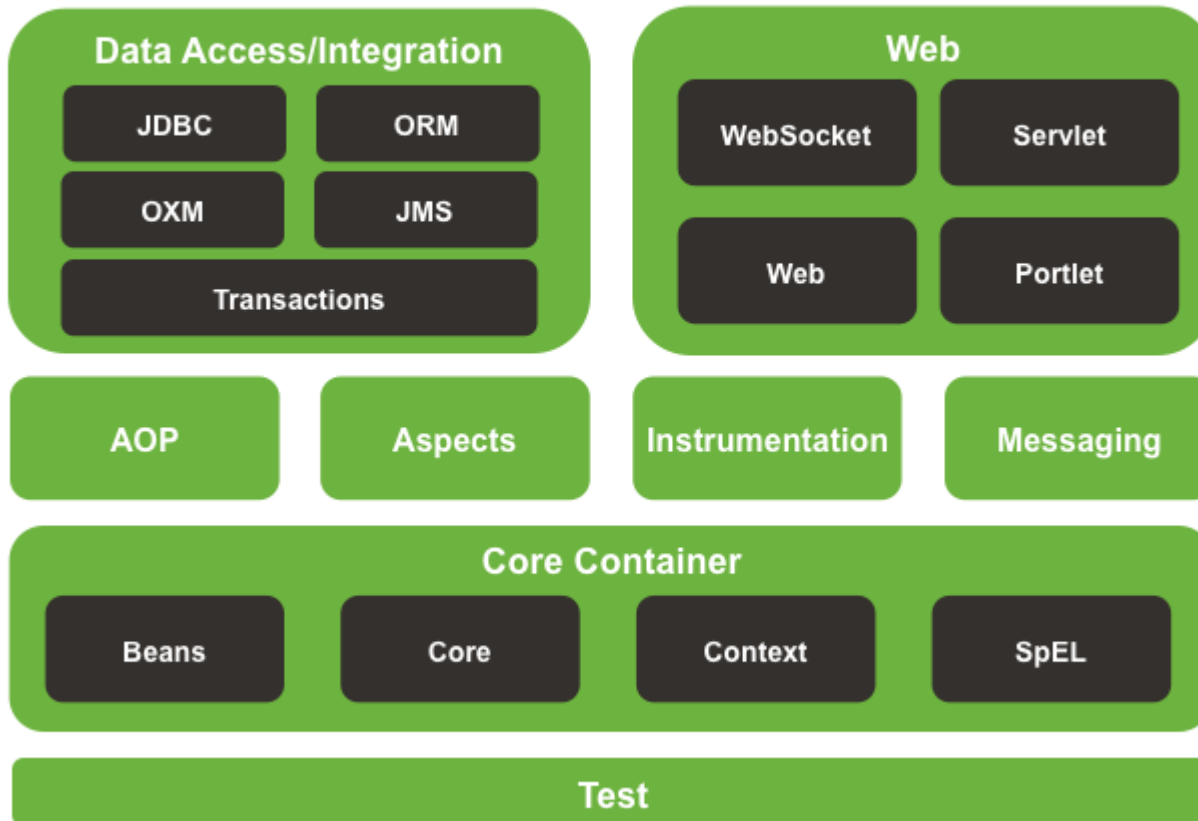
Spring Framework Main Features

- ❖ **Core technologies** – dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
- ❖ **Testing** – mock objects, TestContext framework, Spring MVC Test, WebTestClient.
- ❖ **Data Access** – transactions, DAO support, JDBC, ORM, Marshalling XML.
- ❖ **Spring MVC** and **Spring WebFlux** web frameworks
- ❖ **Integration** – remoting, JMS, JCA, JMX, email, tasks, scheduling, cache.
- ❖ **Languages** – Kotlin, Groovy, dynamic languages.

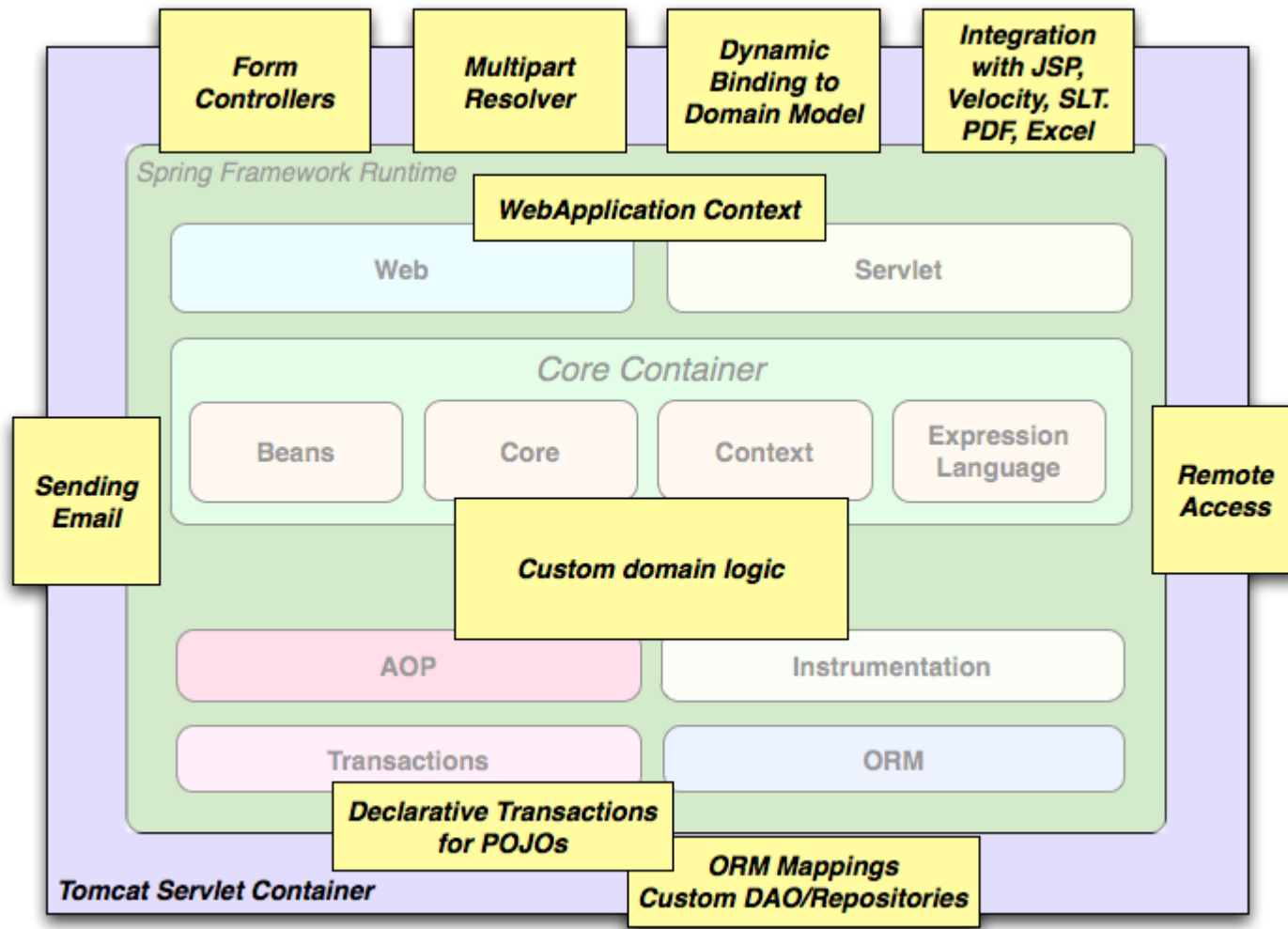
Spring Framework 4.2 Main Modules



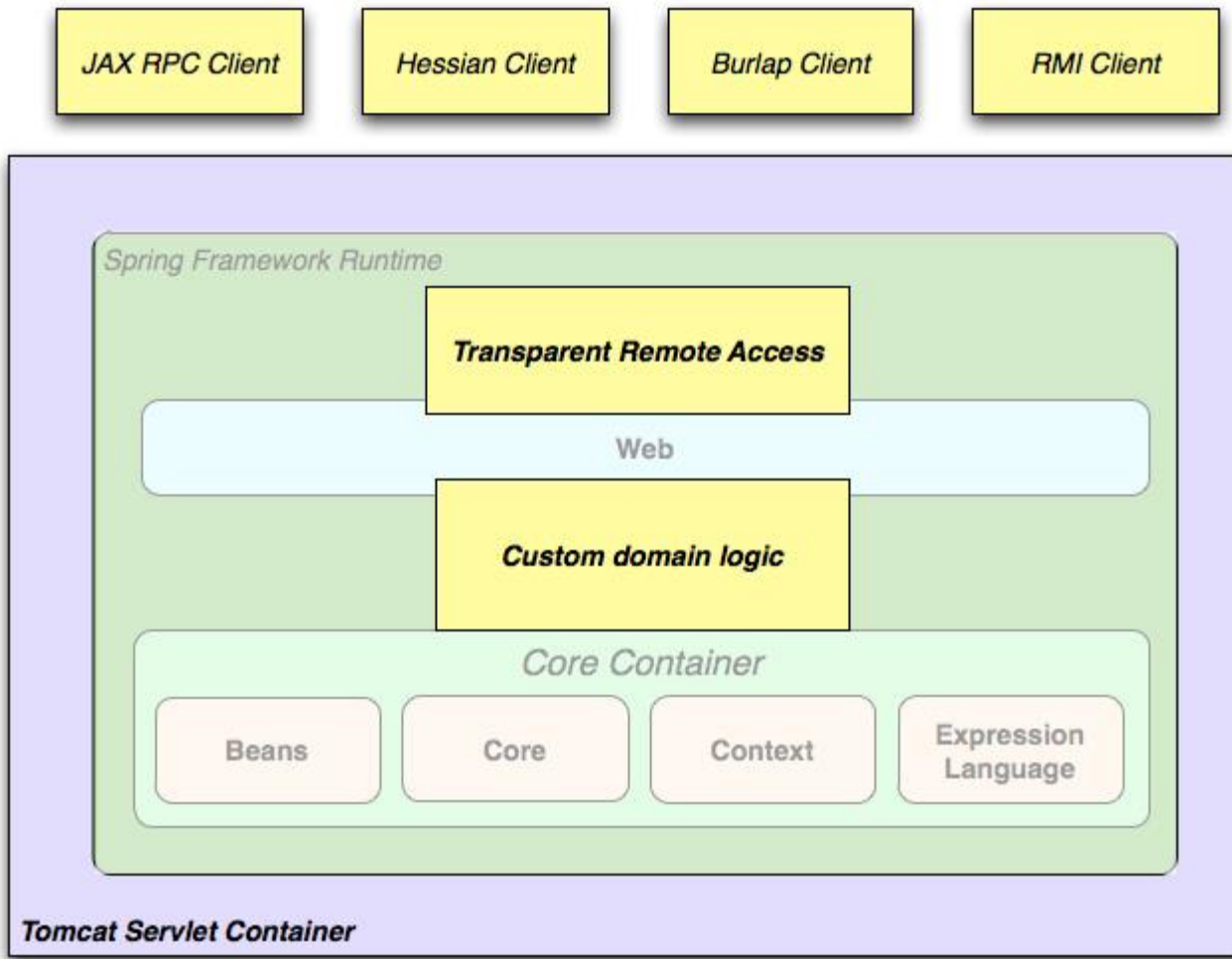
Spring Framework Runtime



Fully Fledged Spring Web Application



Remoting Application



Spring 5 Framework Modules

Spring Boot 2.0

Web Servlet: Spring MVC,
WebSocket, SockJS, STOMP

Web Reactive: Spring WebFlux,
WebClient, WebSocket

Data Access: Transactions,
DAO support, JDBC, ORM, OXM

Integration: Remoting, JMS, JCA, JMX,
Email, Tasks, Scheduling, Cache

Spring Core: IoC container and beans, Events, Resources, i18n,
Validation, Data Binding, Type Conversion, SpEL, AOP

Spring Testing: Mock objects, TestContext, MVC Test, WebTestClient

Evolution of Spring Framework - I

- ❖ **Spring 1.x** – Spring Core, Spring Context, Spring DAO, Spring ORM, Spring AOP, Spring Web, Spring WebMVC
- ❖ **Spring 2.x (2006)** – declarative transactions, @AspectJ, JPA, JMS, MVC form tags, Portlet MVC, Acegi Security
- ❖ **Spring 2.5 (2007)** – @Autowired, JSR-250(@Resource, @PostConstruct, @PreDestroy), stereotype annotations (@Component, @Repository, @Service, @Controller), automatic classpath scanning, AOP updates, TestContext
- ❖ **Spring 3.x (2009)** – Java-based @Configuration model, Spring Expression Language (SpEL), JSR-303:Bean Validation, REST
- ❖ **Spring 3.1.x (2009)** – WebApplicationInitializer, @Cacheable, @Profile, @EnableTransactionManagement..., c: namespace

Evolution of Spring Framework - II

- ❖ **Spring 4.x (2013, Pivotal)** – Java 8, Spring Boot, WebSocket, SockJS, and STOMP messaging, composed annotations, improvements in the core container, CORS, Hibernate 5.0, Spring IO, Spring XD
- ❖ **Spring 5.x (2017)** – JDK 9, Junit 5, XML configuration namespaces streamlined to unversioned schemas, Protobuf 3.0, Java EE7 API level required in Servlet 3.1, Bean Validation 1.1, JPA 2.1, JMS 2.0. Tomcat 8.5+, Jetty 9.4+, Wildfly 10+, Reactor, WebFlux, Spring Vault, Spring Cloud Stream, Micrometer

Spring Design Philosophy

- ❖ Provide choice at every level. Spring lets you defer design decisions as late as possible – e.g. persistence providers, infrastructure, third-party APIs
- ❖ Accommodate diverse perspectives – not opinionated
- ❖ Maintain strong backward compatibility
- ❖ Care about API design – intuitive APIs
- ❖ Code quality – high standards, meaningful, current, and accurate javadoc, clean code structure with no circular dependencies between packages.

Top New Features in Spring 5

- ❖ Reactive Programming Model
- ❖ Spring Web Flux – takes advantage of multi-core processors, handles massive number of connections
- ❖ Reactive DB repositories & integrations + hot event streaming: MongoDB, CouchDB, Redis, Cassandra, Kafka
- ❖ JDK 8+ and Java EE 7+ baseline
- ❖ Testing improvements – WebTestClient (based on reactive WebFlux WebClient)
- ❖ Kotlin functional DSL

Spring 5 Web Application Building Blocks

Spring Boot 2.0



Project Reactor

Servlet Stack
(one request per thread)

Reactive Stack
(async IO)

Every JEE Servlet Container
(tomcat, jetty, undertow, ...)

Nonblocking NIO Runtimes
(Netty, Servlet 3.1 Containers)

Spring Security

Spring Security Reactive

Spring MVC

Spring WebFlux

Spring Data Repositories
JDBC, JPA, NoSQL

Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase

Maven Dependency Management

- ❖ Apache Maven – <https://spring.io/guides/gs/maven/>
- ❖ Common arguments: `mvn compile`, `mvn package`, `mvn install`, `mvn clean` `deploy` `site-deploy`
- ❖ Example configuration:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.iproduct.spring</groupId>
    <artifactId>01-introduction-maven</artifactId>
    <version>1.0-SNAPSHOT</version>
```

Maven Configuration (continued)

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.5.RELEASE</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>io.spring.repo.maven.release</id>
    <url>http://repo.spring.io/release/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

Maven Configuration (continued)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>9</source>
        <target>9</target>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```


Maven Configuration (enhanced)

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.0.5.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
</dependencies>
```

Gradle Dependency Management

- ❖ Gradle – <https://spring.io/guides/gs/gradle/>
- ❖ Init new project/ convert existing from Maven: **gradle init**
- ❖ Build project: **gradle build**
- ❖ Build project: **gradle run**
- ❖ Example configuration:

```
group 'org.iproduct.spring'
version '1.0-SNAPSHOT'
plugins {
    id 'java'
    id 'application'
}
mainClassName='course.spring.coredemo.SpringAnnotationConfigDI'
sourceCompatibility = 11
```

Gradle Configuration (continued)

```
task runApp(type : JavaExec ) {  
    classpath = sourceSets.main.runtimeClasspath  
    main = 'course.spring.coredemo.SpringAnnotationConfigDI'  
}  
  
repositories {  
    mavenLocal()  
    mavenCentral()  
    maven { url "https://repo.spring.io/snapshot" }  
    maven { url "https://repo.spring.io/milestone" }  
}  
  
dependencies {  
    implementation group: 'org.springframework',  
                   name: 'spring-context', version: '5.3.7'  
    testImplementation group: 'junit',  
                      name: 'junit', version: '4.12'  
}
```

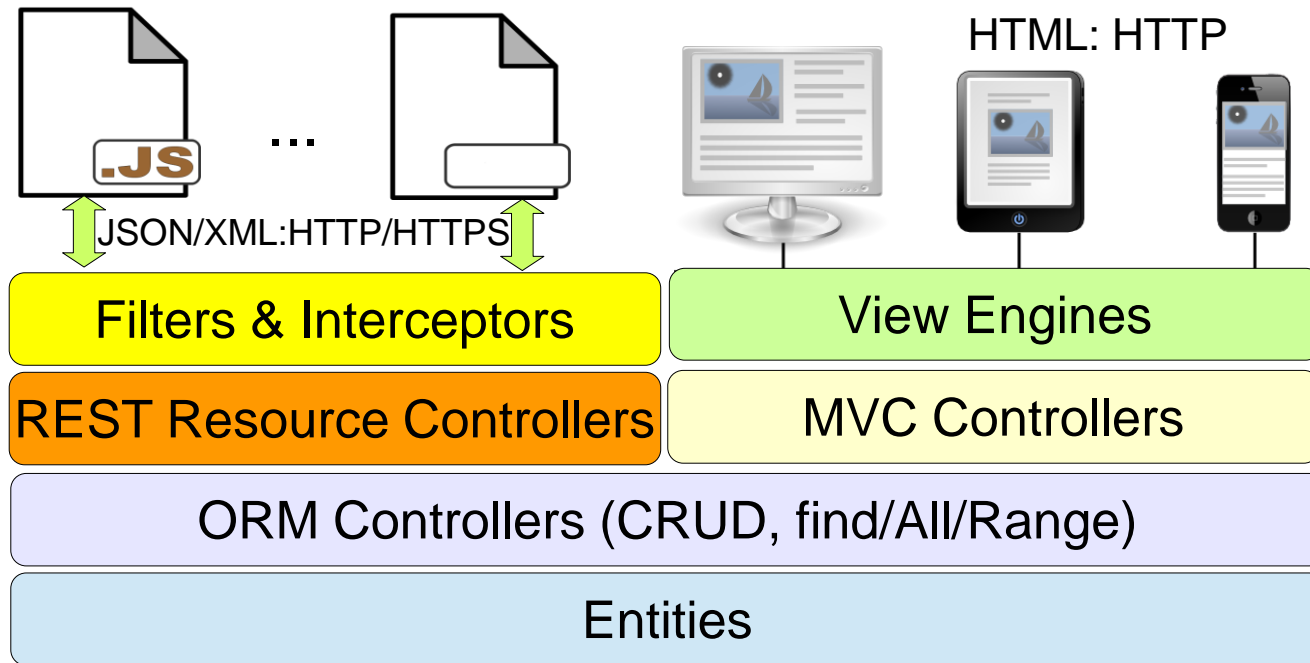
Making Projects Easy: Spring Boot 2

The screenshot shows the Spring Initializr web application in a browser window. The browser's address bar shows the URL `start.spring.io`. The application interface includes a sidebar with a hamburger menu icon and a settings icon. The main content area is divided into several sections:

- Project:** Radio buttons for `Maven Project` and `Gradle Project` (selected).
- Language:** Radio buttons for `Java` (selected), `Kotlin`, and `Groovy`.
- Spring Boot:** Radio buttons for versions `2.6.0 (SNAPSHOT)`, `2.5.3 (SNAPSHOT)`, `2.5.2` (selected), and `2.4.9 (SNAPSHOT)`. Below these are `2.4.8` and `2.3.12`.
- Project Metadata:** Fields for `Group` (course.spring), `Artifact` (02-blogs-api-lab14), `Name` (02-blogs-api-lab14), `Description` (Demo project for Spring Boot, Spring MVC, Spring Data, Spring Security), and `Package name` (course.spring.blogs). There are also fields for `Packaging` (`Jar` selected, `War` unselected) and `Java` version (`16` unselected, `11` selected, `8` unselected).
- Dependencies:** A list of dependencies with tags: `Spring Boot DevTools` (DEVELOPER TOOLS), `Lombok` (DEVELOPER TOOLS), `Spring Web` (WEB), `Spring Boot Actuator` (OPS), `Spring Data JPA` (SQL), and `PostgreSQL Driver` (SQL). An `ADD DEPENDENCIES... CTRL + B` button is present.

At the bottom of the main content area are three buttons: `GENERATE CTRL + G`, `EXPLORE CTRL + SPACE`, and `SHARE...`. The browser's taskbar at the bottom shows various application icons and system information like temperature (88°F) and time (16:09, 7.7.2021).

N-Tier Web Architectures



Domain Driven Design (DDD)

We need tools to cope with all that complexity inherent in robotics and IoT domains.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

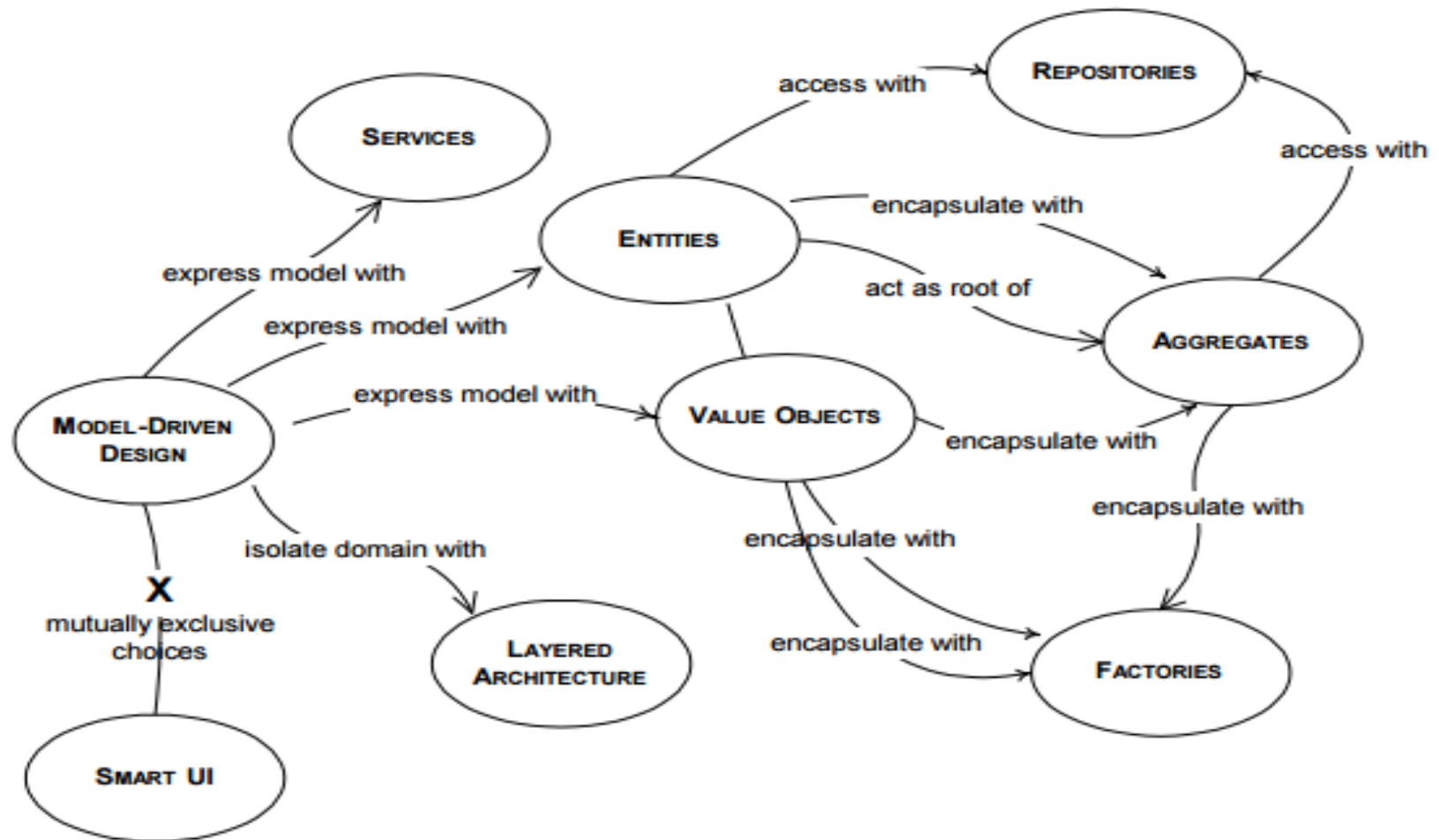
Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design (DDD)

Main concepts:

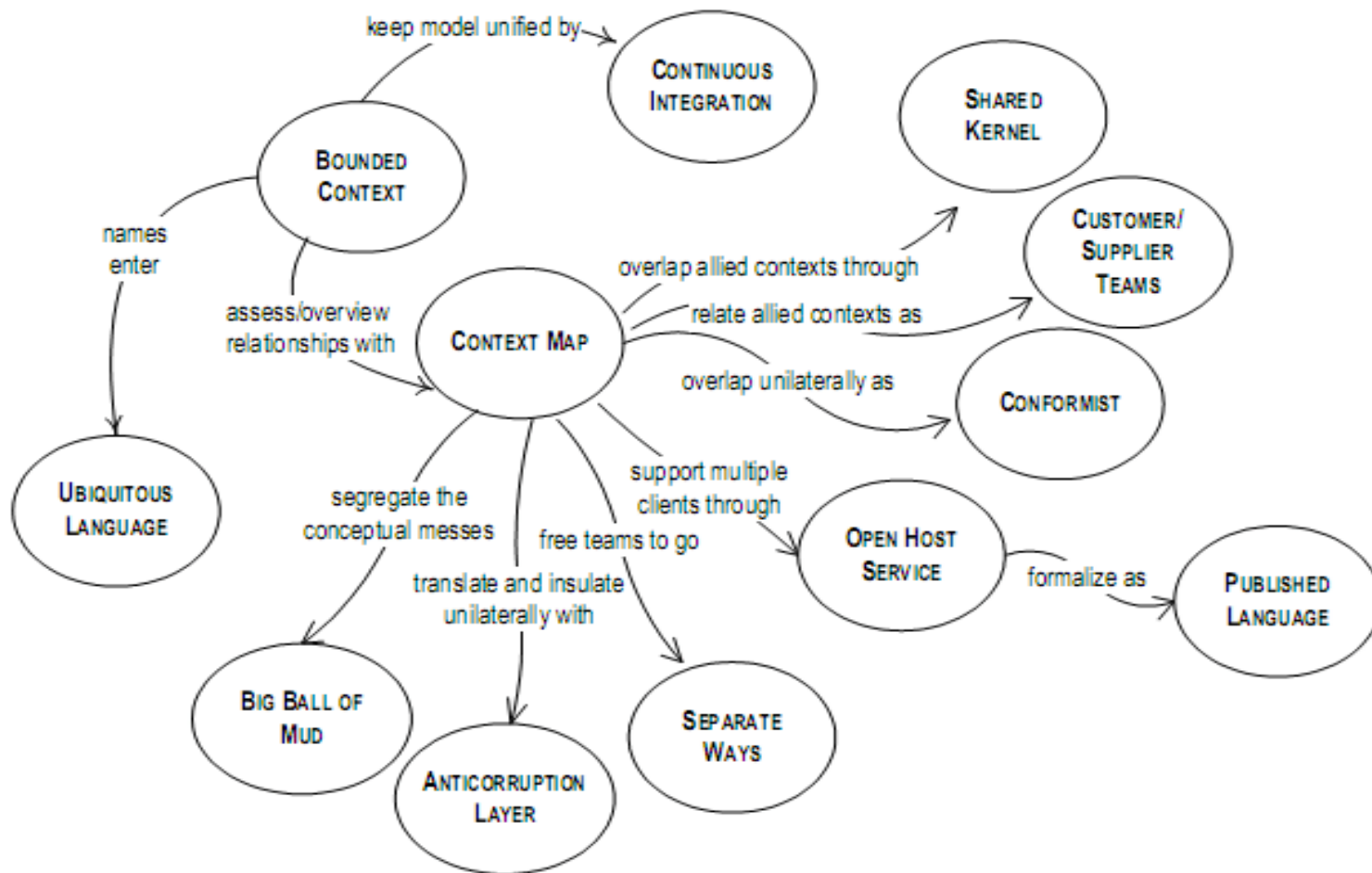
- Entities, value objects and modules
- Aggregates and Aggregate Roots [Haywood]:
value < entity < aggregate < module < BC
- Repositories, Factories and Services:
application services <-> domain services
- Separating interface from implementation

Domain Driven Design (DDD)



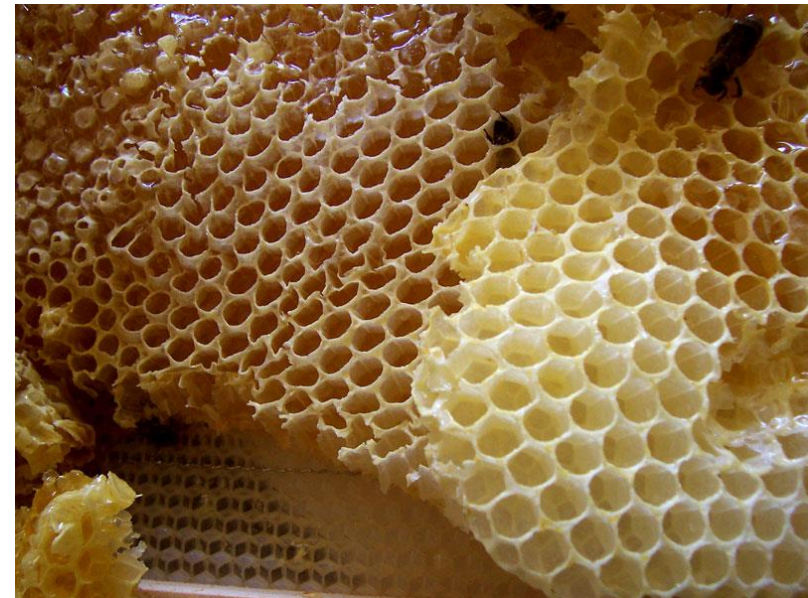
Domain Driven Design (DDD)

Maintaining Model Integrity



Domain Driven Design (DDD)

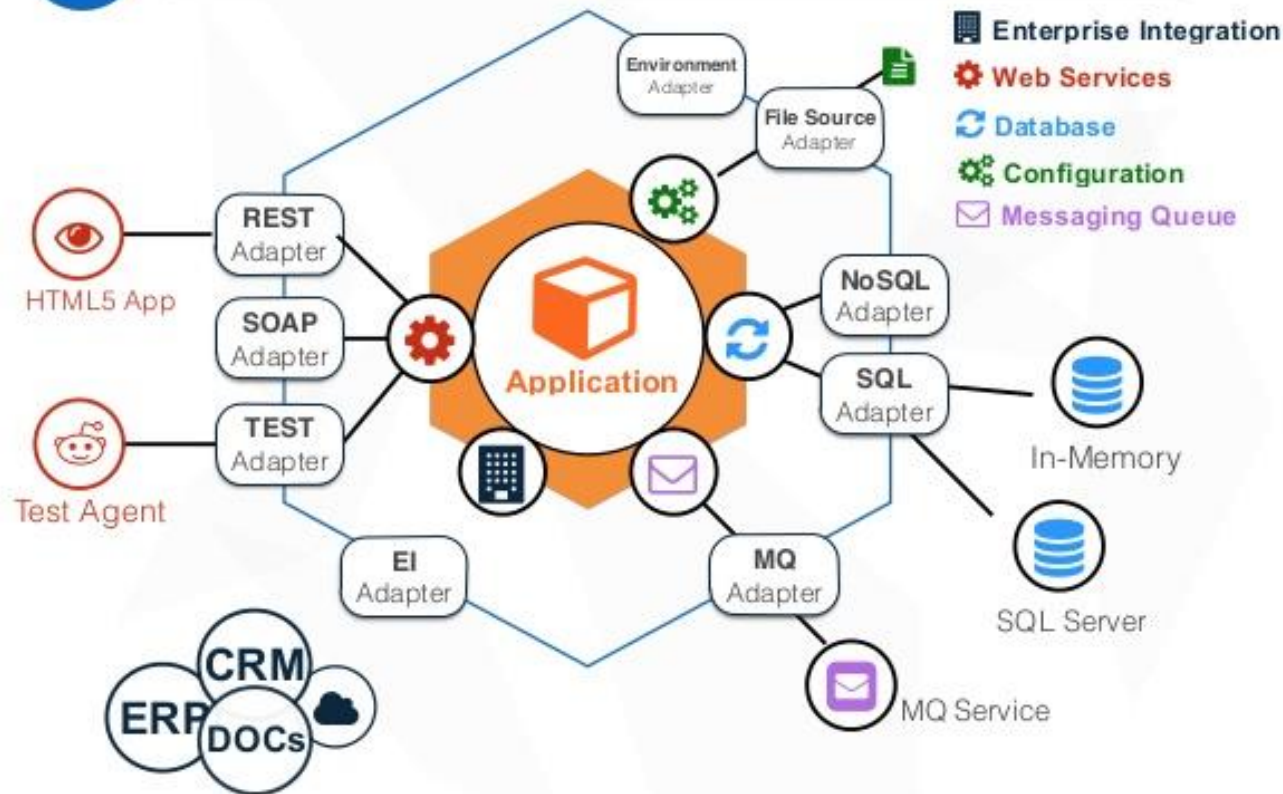
- Ubiquitous language and Bounded Contexts
- DDD Application Layers:
- Infrastructure, Domain, Application, Presentation
- Hexagonal architecture :
OUTSIDE <-> transformer <->
(application <-> domain)
[A. Cockburn]



Hexagonal Architecture

02

Overview

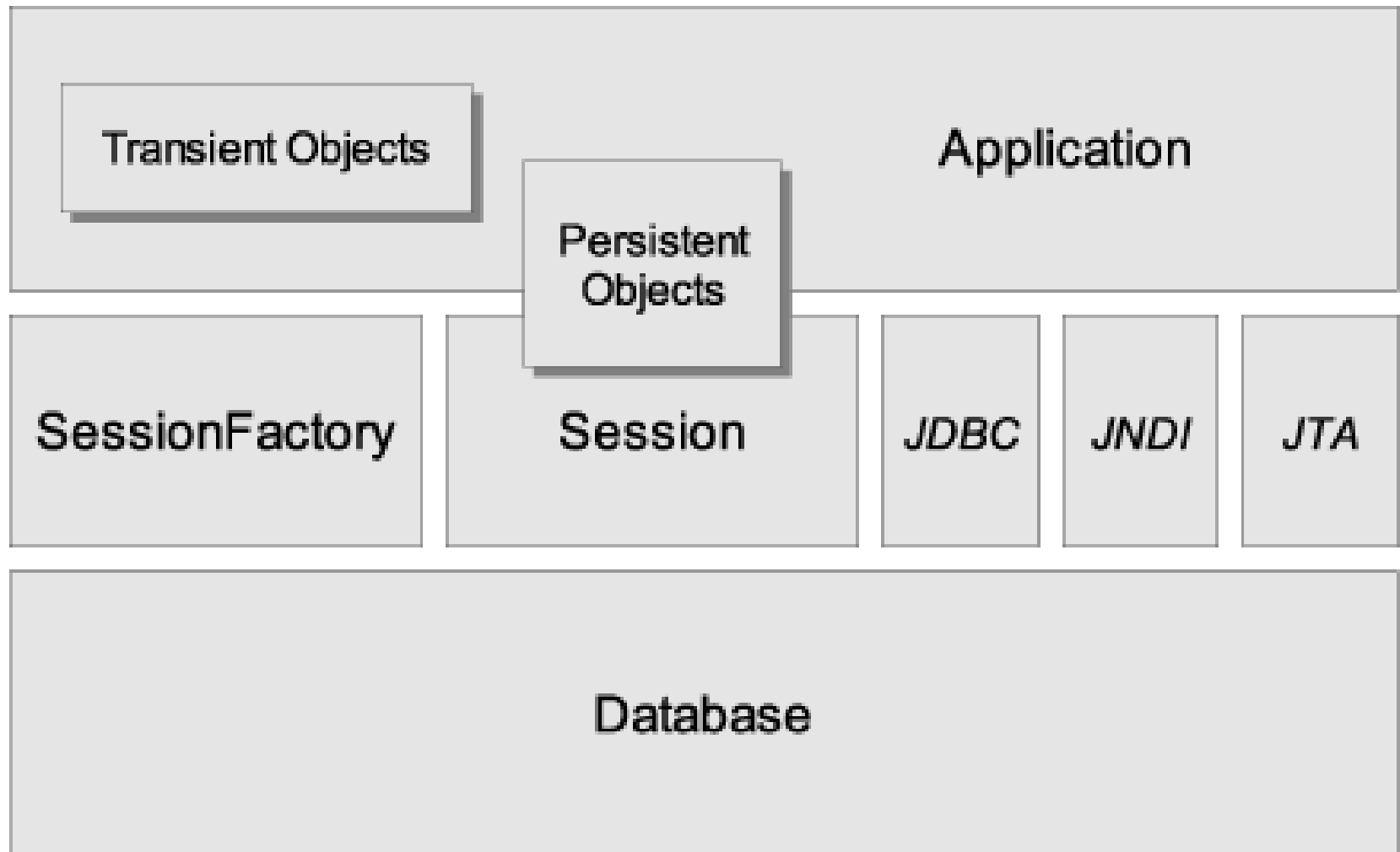


Hexagonal Architecture Principles

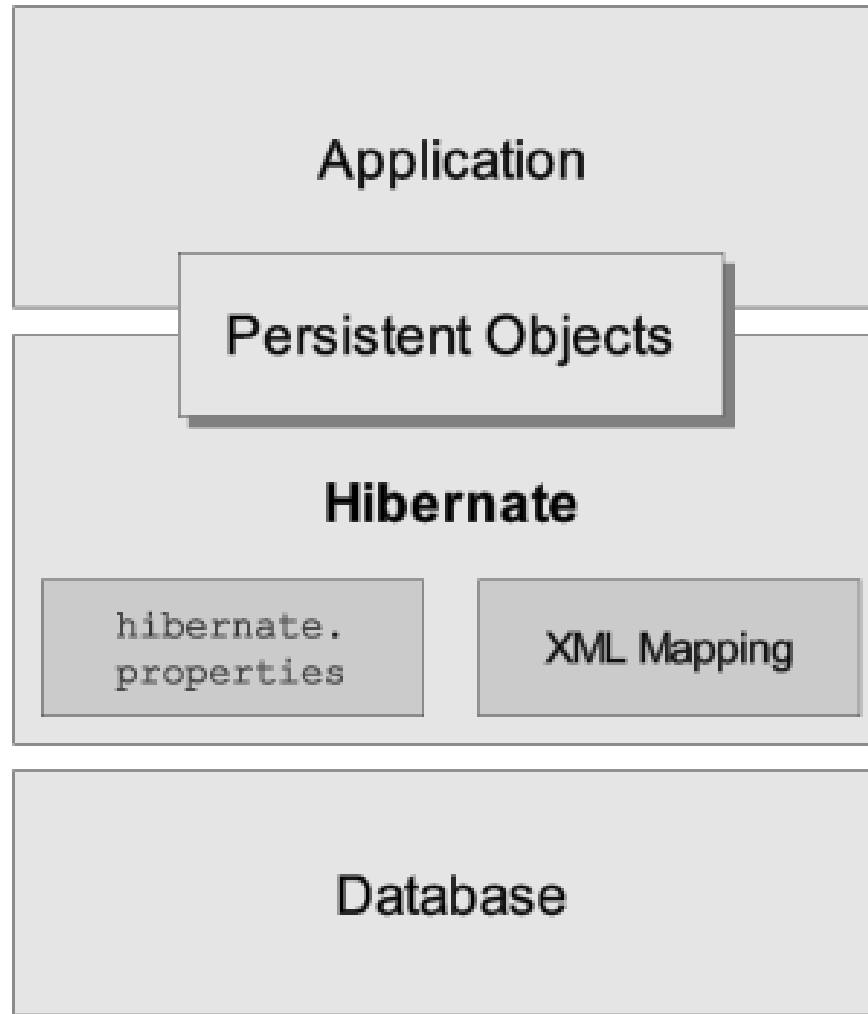
- Allows an application to equally be driven by **users, programs, automated test or batch scripts**, and to be developed and tested in isolation from its eventual run-time devices and databases.
- As events arrive from the outside world at a port, a **technology-specific adapter** converts it into a **procedure call** or **message** and passes it to the application
- Application sends messages through **ports** to **adapters**, which signal data to the receiver (human or automated)
- The application has a **semantically sound interaction** with all the adapters, **without actually knowing the nature of the things** on the other side of the adapters

35

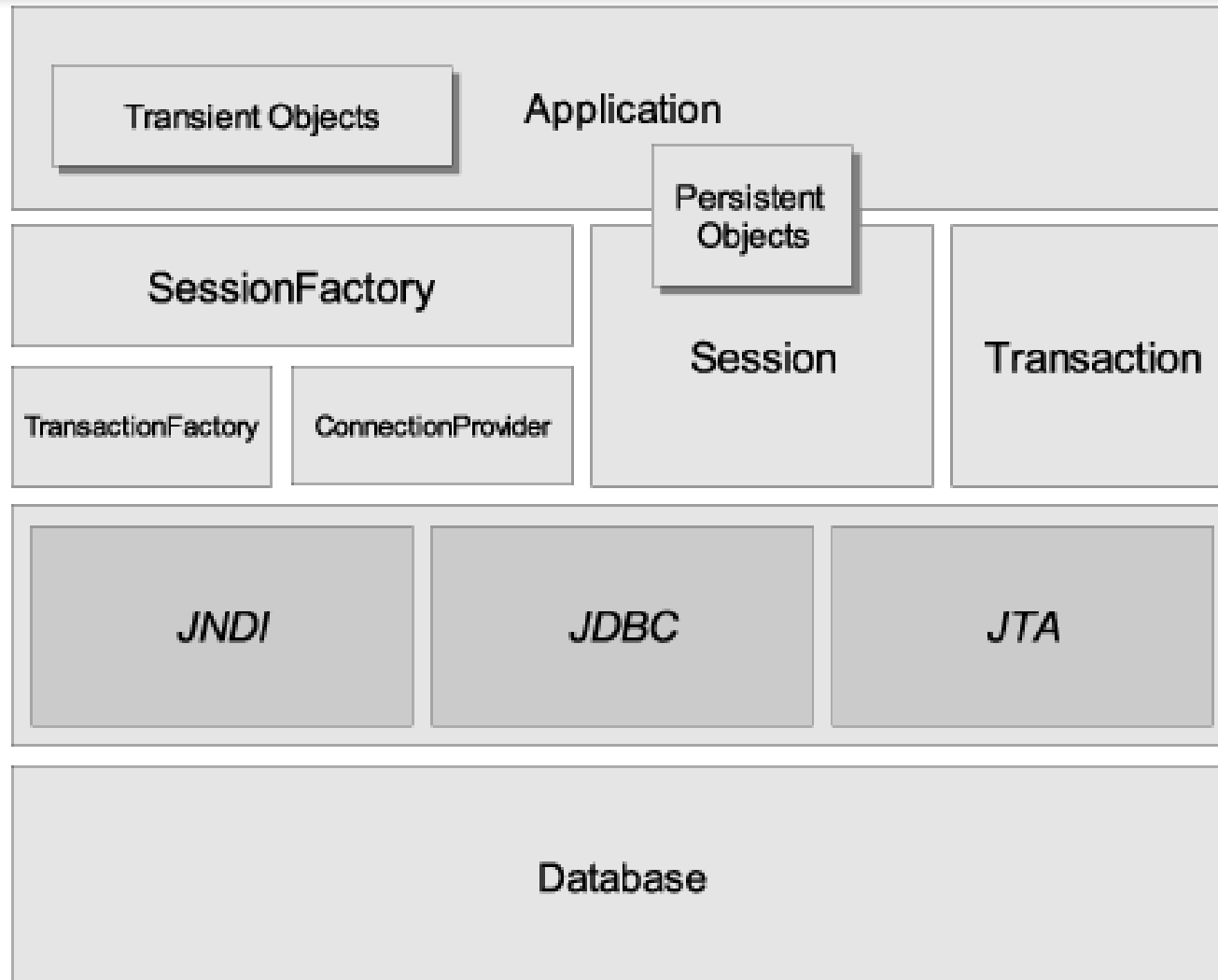
Hibernate Architecture



Hibernate Architecture



Hibernate Architecture

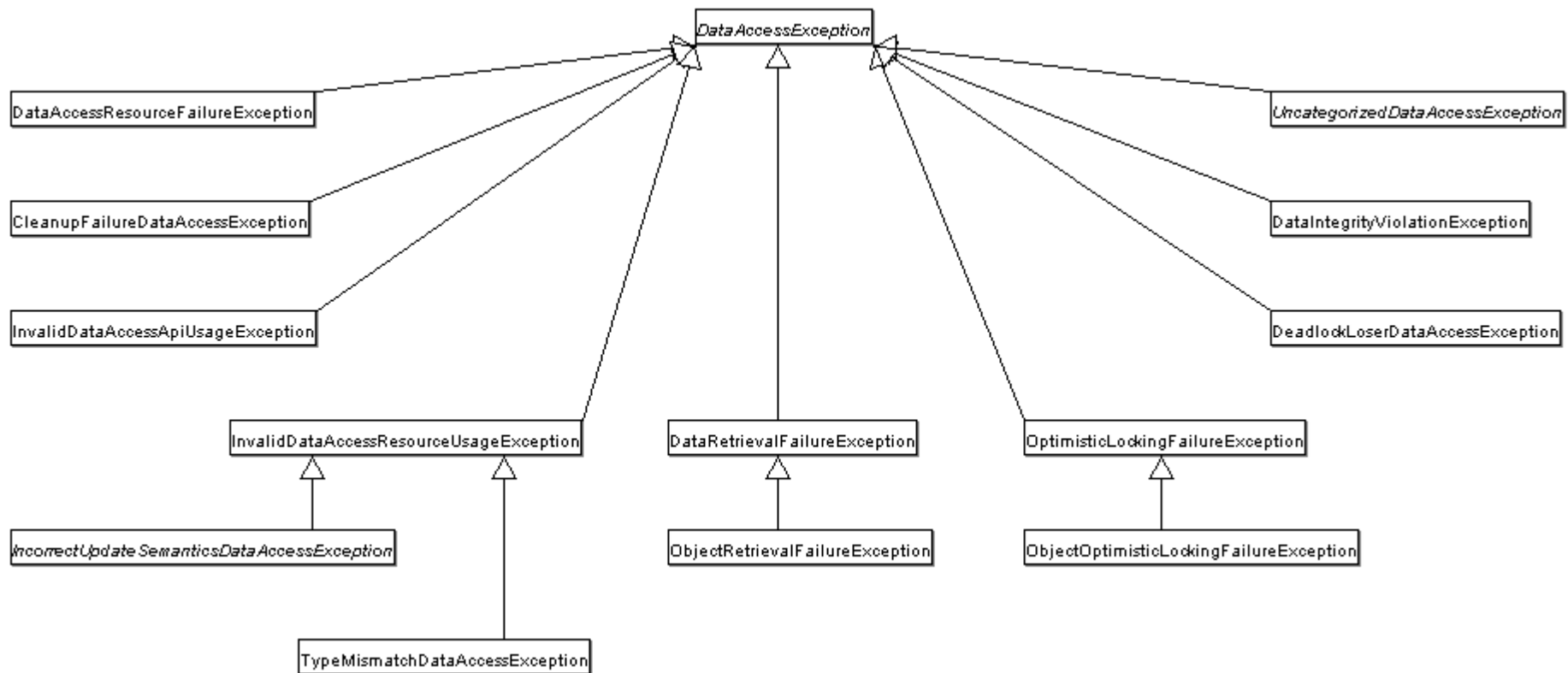


Spring Data Access Objects (DAO)

- **Data Access Object (DAO)** – simplifies work with different data access technologies like **JDBC**, **Hibernate** or **JPA** in a consistent way.
- Consistent exception hierarchy - **RuntimeExceptions**
- Annotations used for configuring **DAO** or **Repository** classes – with automatic exception translation:

```
import org.springframework.stereotype.Repository;  
  
@Repository  
public class SomeMovieFinder implements MovieFinder {  
    // ...  
}
```

DAO Exception Hierarchy



DAO Repository - JDBC

```
import javax.sql.DataSource;

@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...

}
```

DAO Repository - Hibernate

```
import org.hibernate.SessionFactory;
import

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

@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory
sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...
}
```

DAO Repository - JPA

```
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Repository
public class JpaMovieFinder implements MovieFinder {

    @PersistenceContext
    private EntityManager entityManager;

    // ...

}
```

Spring JDBC

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC DB Access Alternatives

- ❖ **JdbcTemplate** - the “classic” Spring JDBC approach and the most popular - “lowest level”, all others use a JdbcTemplate
- ❖ **NamedParameterJdbcTemplate** – wraps a JdbcTemplate to provide named parameters instead of the “?” placeholders
- ❖ **SimpleJdbcInsert** and **SimpleJdbcCall** uses DB metadata, you only need to provide the name of the table or procedure and provide a map of parameters matching column names.
- ❖ **RDBMS Objects** – include **MappingSqlQuery**, **SqlUpdate** and **StoredProcedure**, you create reusable and thread-safe objects during initialization, like JDO Query, wherein you define your query string, declare parameters, and compile the query. Then you can execute methods multiple times.

JDBC Repository Methods - I

@Override

```
public Collection<Article> findAll() {  
    List<Article> articles = this.jdbcTemplate  
        .query("select * from articles", new  
ArticleMapper());  
    log.info("Articles loaded: {}", articles.size());  
    return articles;  
}
```

@Override

```
public Article find(long id) {  
    Article article = this.jdbcTemplate.queryForObject(  
        "select * from articles where id = ?",  
        new Object[]{id}, new ArticleMapper());  
    log.info("Article found: {}", article);  
    return article;  
}
```


JDBC Repository Methods - II

@Override

```
public Article create(Article article) {
    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement
            (Connection connection) throws SQLException {
            PreparedStatement ps = connection
                .prepareStatement(INSERT_SQL, new String[] {"id"});
            ps.setString(1, article.getTitle());
            ps.setString(2, article.getContent());
            ps.setTimestamp(3, new Timestamp(
                article.getCreatedDate().getTime()));
            ps.setString(4, article.getPictureUrl());
            return ps;
        }
    }, keyHolder);
    article.setId(keyHolder.getKey().longValue());
    log.info("Article created: {}", article);
    return article;
}
```

JDBC Repository Methods - III

@Override

```
public Article update(Article article) {  
    int count = this.jdbcTemplate.update(  
        "update articles set (title, content, created_date, picture_url)  
        VALUES (?, ?, ?, ?) where id = ?",  
        article.getTitle(), article.getContent(),  
        article.getCreatedDate(),  
        article.getPictureUrl(), article.getId());  
    log.info("Article updated: {}", article);  
    return article;  
}
```

@Override

```
public boolean remove(long articleId) {  
    int count = this.jdbcTemplate.update(  
        "delete from articles where id = ?",  
        Long.valueOf(articleId));  
    return count > 0;  
}
```

JDBC DataSource - I

@Configuration

@ComponentScan({"org.iproduct.spring.webmvc.service",
"org.iproduct.spring.webmvc.dao"})

@PropertySource("classpath:jdbc.properties")

public class SpringRootConfig {

@Value("\${jdbc.driverClassName:org.postgresql.Driver}")

private String driverClassname;

@Value("\${jdbc.url:jdbc:postgresql://localhost/articles}")

private String url;

@Value("\${jdbc.username:postgres}")

private String username;

@Value("\${jdbc.password:postgres}")

private String password;

(- continues on next slide -)

JDBC DataSource - II

@Bean

```
DataSource getDataSource() {  
    DriverManagerDataSource dataSource =  
        new DriverManagerDataSource();  
    //PostgreSQL database we are using  
    dataSource.setDriverClassName(driverClassname);  
    dataSource.setUrl(url); //change url  
    dataSource.setUsername(username); //change username  
    dataSource.setPassword(password); //change pwd  
  
    //H2 database  
    /*  
    dataSource.setDriverClassName("org.h2.Driver");  
    dataSource.setUrl("jdbc:h2:tcp://localhost/~/test");  
    dataSource.setUsername("sa");  
    dataSource.setPassword(""); */  
    return dataSource;  
}
```

Hibernate DAO

```
import org.hibernate.SessionFactory;
import

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

@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(
        SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    // ...
}
```

Web_INITIALIZER – XML Root Config

```
public class ArticlesWebInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected WebApplicationContext
        createRootApplicationContext() {
        return new XmlWebApplicationContext();
    }

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[0];
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { SpringWebConfig.class };
    }
}
```


WEB-INF/applicationContext.xml - I

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:property-placeholder location="classpath:jdbc.properties" />

    <context:component-scan base-package="org.iproduct.spring.webmvc.dao,
        org.iproduct.spring.webmvc.service"/>

    <context:annotation-config />
```

WEB-INF/applicationContext.xml -II

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list><value>article.hbm.xml</value></list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=org.hibernate.dialect.HSQLDialect
            hibernate.hbm2ddl.auto=update
        </value>
    </property>
</bean>
```

WEB-INF/applicationContext.xml III

```
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>  
  
<tx:annotation-driven/>  
  
</beans>
```

Hibernate Mapping: article.hbm.xml

```
<hibernate-mapping>
  <class name="org.iproduct.spring.webmvc.model.Article" table="ARTICLES">

    <meta attribute="class-description">
      This class contains the articles details.
    </meta>

    <id name="id" type="long" column="id">
      <generator class="identity"/>
    </id>

    <property name="title" column="title" type="string"/>
    <property name="content" column="content" type="string"/>
    <property name="createdDate" column="created_date" type="timestamp"/>
    <property name="pictureUrl" column="picture_url" type="string"/>

  </class>
</hibernate-mapping>
```

ArticlesDaoHibernate Class - I

@Repository

@Transactional

```
public class ArticleDaoHibernate implements ArticleDao {
```

```
    private SessionFactory sessionFactory;
```

@Autowired

```
public void setSessionFactory(SessionFactory sessionFactory) {  
    this.sessionFactory = sessionFactory;  
}
```

@Override

```
public Collection<Article> findAll() {  
    return this.sessionFactory.getCurrentSession()  
        .createQuery("select article from Article article", Article.class)  
        .list();  
}
```

@Override

```
public Article find(long id) {  
    return this.sessionFactory.getCurrentSession()  
        .findById(Article.class).load(id);  
}
```

ArticlesDaoHibernate Class - II

```
@Override
public Article create(Article article) {
    this.sessionFactory.getCurrentSession()
        .persist(article);
    return article;
}

@Override
public Article update(Article article) {
    Article toBeDeleted = find(article.getId());
    if (toBeDeleted == null) {
        throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    return (Article) this.sessionFactory.getCurrentSession()
        .merge(article);
}

@Override
public Article remove(long articleId) {
    Article toBeDeleted = find(articleId);
    if (toBeDeleted == null) {
        throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    this.sessionFactory.getCurrentSession()
        .delete(toBeDeleted);
    return toBeDeleted;
}}
```


Transactions and Concurrency

❖ **Transaction** = Commits as **Business Event**

❖ **ACID rules:**

❖ **Atomicity** – the whole transaction is completed (commit) or no part is completed at all (rollback).

❖ **Consistency** – transaction should preserve existing integrity constraints

❖ **Isolation** – two uncompleted transactions can not interact

❖ **Durability** – successfully completed transactions can not be rolled back

Advantages of Spring Transactions

- ❖ **Consistent** programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, and Java Persistence API (JPA).
- ❖ Support for **declarative transaction** management.
- ❖ Simpler API for **programmatic transaction management** than complex transaction APIs such as JTA.
- ❖ Excellent **integration** with Spring's data access abstractions.

Spring Transaction Management

- ❖ **Global transactions** – enable you to work with multiple transactional resources, typically relational databases and message queues (JTA UserTransaction, JNDI lookup).
- ❖ **Local transactions** – resource-specific, such as a transaction associated with a JDBC connection, but cannot work across multiple transactional resources.
- ❖ **Spring Framework's transactions** – consistent programming model in any environment, write code once, and it can use different transaction management strategies in different environments – both **declarative and programmatic transaction management** (Spring Framework transaction abstraction).

Spring Transaction Abstraction

```
public interface PlatformTransactionManager {  
  
    TransactionStatus getTransaction(TransactionDefinition definition)  
        throws TransactionException;  
  
    void commit(TransactionStatus status) throws TransactionException;  
  
    void rollback(TransactionStatus status) throws TransactionException;  
}
```

❖ TransactionDefinition:

- **Propagation** – what to do when a transactional method is executed when a transaction context already exists)
- **Isolation** – degree to which this transaction is isolated from the work of other transactions (e.g. can this transaction see uncommitted writes from other transactions?)
- **Timeout** – how long run before timing out and being rolled back
- **Read-only status**: used when you read but not modify data

Transaction Isolation Levels

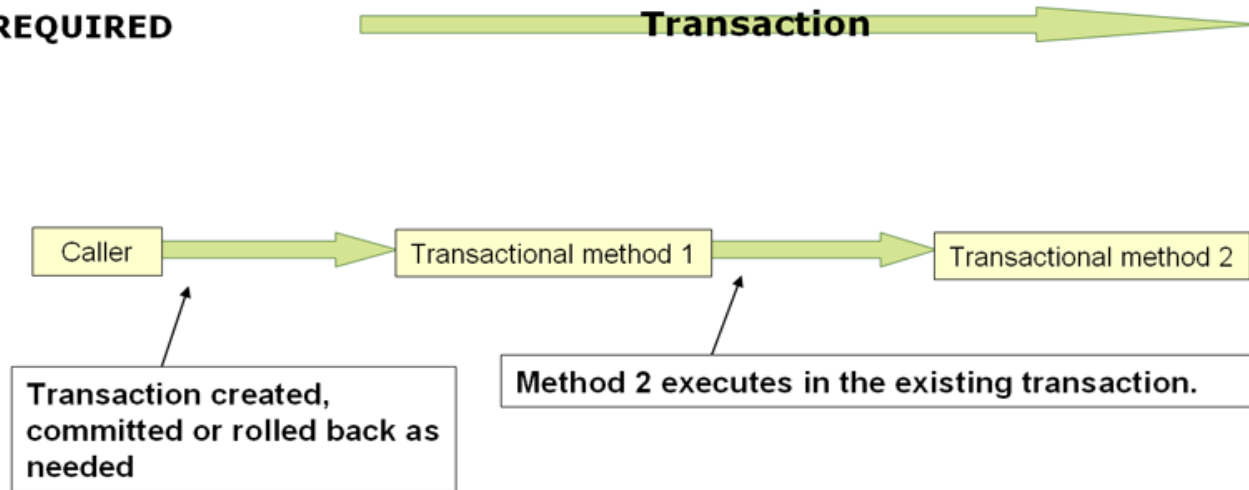
- ❖ **DEFAULT** - use the default isolation level of the underlying datastore
- ❖ **READ_UNCOMMITTED** – dirty reads, non-repeatable reads and phantom reads can occur
- ❖ **READ_COMMITTED** – prevents dirty reads; non-repeatable reads and phantom reads can occur
- ❖ **REPEATABLE_READ** – prevents dirty reads and non-repeatable reads; phantom reads can occur
- ❖ **SERIALIZABLE** – prevents dirty reads, non-repeatable reads and phantom reads

Transactions Propagation

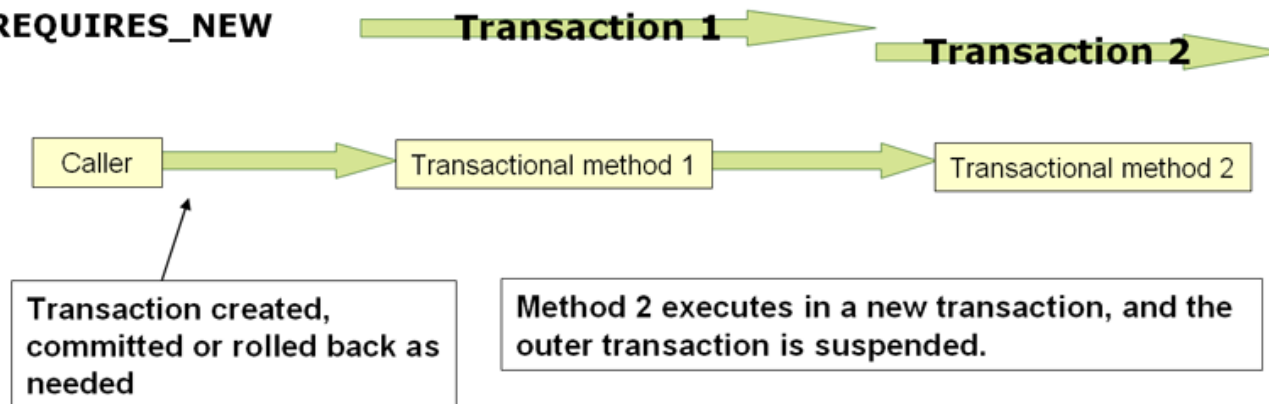
- ❖ **SUPPORTS** – supports transaction if existing, executes non-transactionally if not
- ❖ **REQUIRED** – supports transaction if existing, creates new if not
- ❖ **REQUIRES_NEW** – always create a new transaction, and suspend the current transaction if one exists
- ❖ **MANDATORY** – supports the current transaction, throws an exception if none exists
- ❖ **NEVER** – execute non-transactionally, throw an exception if a transaction exists
- ❖ **NOT_SUPPORTED** - execute non-transactionally, suspend the current transaction if one exists
- ❖ **NESTED** – executes within a nested transaction if current transaction exists, else does like **PROPAGATION_REQUIRED**

Transactions Propagation

REQUIRED



REQUIRES_NEW



TransactionStatus

```
public interface TransactionStatus extends SavepointManager {  
  
    boolean isNewTransaction();  
  
    boolean hasSavepoint();  
  
    void setRollbackOnly();  
  
    boolean isRollbackOnly();  
  
    void flush();  
  
    boolean isCompleted();  
  
}
```

Transactions and Concurrency

- ❖ **DataSourceTransactionManager** – JDBC local transactions, allows thread bound connections, obtained

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

- ❖ **JtaTransactionManager** – using global JTA transactions

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/articles"/>
<bean id="txManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

- ❖ **@Transactional** – declarative transactions

- ❖ **TransactionTemplate** or directly using **PlatformTransactionManager** – programmatic transactions

Declarative Transaction Demarcation

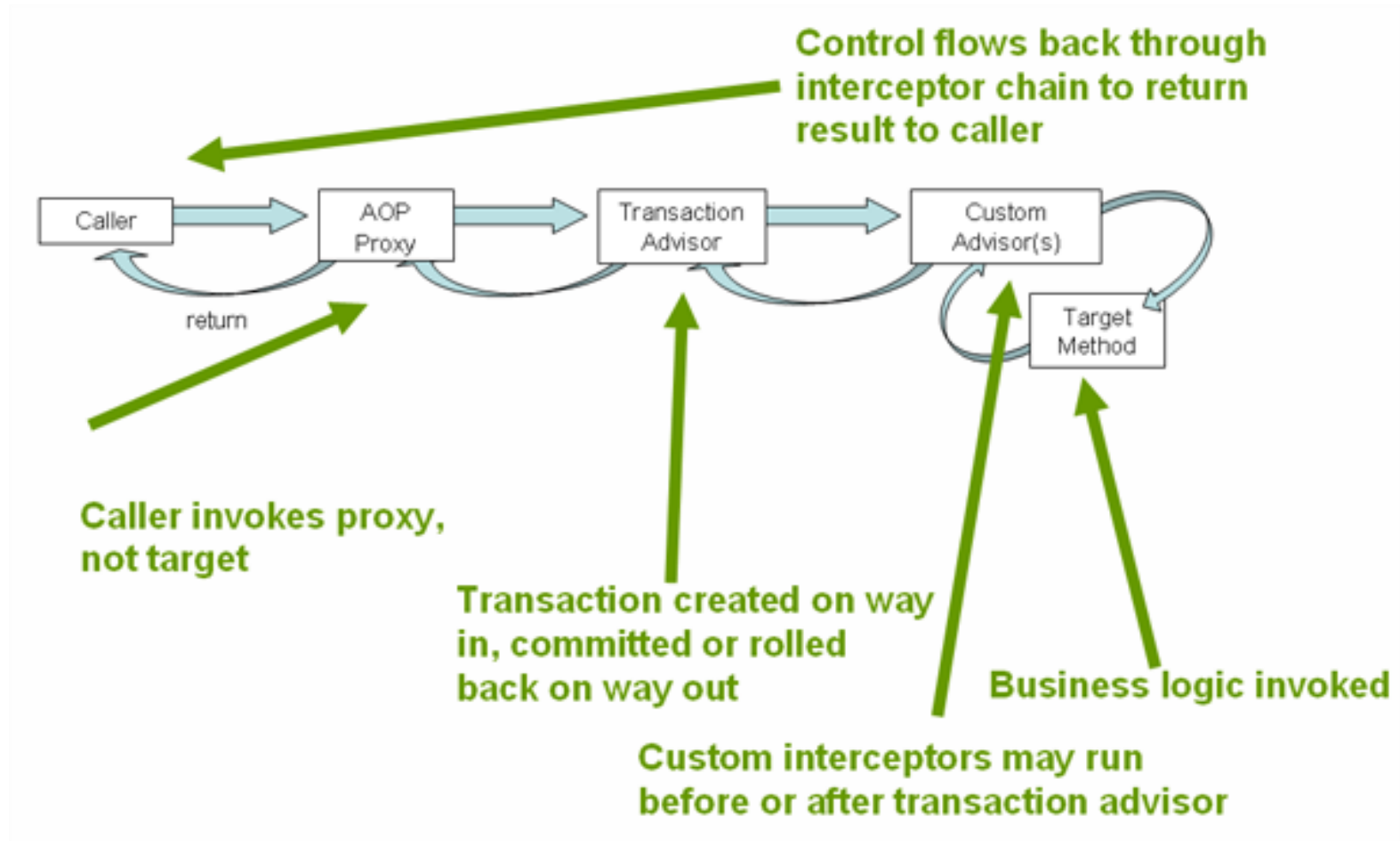
❖ Enabling declarative transactions:

- **@EnableTransactionManagement**
- **<tx:annotation-driven/>**

❖ **@Transactional** attributes: **value** (optional qualifier specifying the transaction manager to be used), **propagation**, **isolation**, **readOnly**, **timeout** (in seconds), **rollbackFor** (optional array of exception classes that must cause rollback), **rollbackForClassName**, **noRollbackFor** (optional array of exception classes that must not cause rollback), **noRollbackForClassName**

```
@Transactional(propagation = Propagation.REQUIRED)
public List<Article> createArticlesBatch(List<Article>
articles) {
    List<Article> created = articles.stream()
        .map(article -> addArticle(article))
        .collect(Collectors.toList());
    return created;
}
```

Transactions via AOP Proxies



Customizing Transactions using AOP

```
<aop:config>
  <aop:pointcut id="entryPointMethod"
    expression="execution(* x.y..*Service.*(..))"/>

  <aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod"
    order="2"/>

  <aop:aspect id="profilingAspect" ref="profiler">
    <aop:pointcut id="methodWithReturn"
      expression="execution(!void x.y..*Service.*(..))"/>
    <aop:around method="profile" pointcut-ref="methodWithReturn"/>
  </aop:aspect>
</aop:config>

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```


Programmatic Transactions - I

```
public List<Article> createArticlesBatch(List<Article> articles)
{
    return transactionTemplate.execute(
        new TransactionCallback<List<Article>>() {
            public List<Article> doInTransaction(
                TransactionStatus status)
            {
                List<Article> created = articles.stream()
                    .map(article -> {
                        try {
                            return addArticle(article);
                        } catch (ConstraintViolationException ex) {
                            log.error("Error:{}", ex.getMessage());
                            status.setRollbackOnly();
                            return null;
                        }
                    })
                    .collect(Collectors.toList());
                return created;
            }
        });
}
```

Programmatic Transactions - II

```
public List<Article> createArticlesBatch(List<Article> articles) {
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    def.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
    def.setTimeout(5);

    TransactionStatus status = transactionManager.getTransaction(def);
    List<Article> created = articles.stream()
        .map(article -> {
            try {
                Article resultArticle = addArticle(article);
                applicationEventPublisher.publishEvent(
                    new ArticleCreationEvent(resultArticle));
                return resultArticle;
            } catch (ConstraintViolationException ex) {
                log.error("Error: {}", ex.getMessage());
                transactionManager.rollback(status); // ROLLBACK
                throw ex;
            }
        })
        .collect(Collectors.toList());
    transactionManager.commit(status); // COMMIT
    return created;
}
```

@TransactionalEventListener

```
@TransactionalEventListener
```

```
public void
```

```
handleArticleCreatedTransactionCommit(ArticleCreationEvent  
creationEvent) {
```

```
    log.info(">>> Transaction COMMIT for article: {}",  
            creationEvent.getArticle());
```

```
}
```

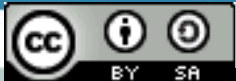
```
@TransactionalEventListener(phase = TransactionPhase.AFTER_ROLLBACK)
```

```
public void
```

```
handleArticleCreatedTransactionRollback(ArticleCreationEvent  
creationEvent) {
```

```
    log.info(">>> Transaction ROLLBACK for article: {}",  
            creationEvent.getArticle());
```

```
}
```



Source: <https://docs.spring.io/spring-framework/docs/>

Java Persistence API (JPA)

❖ JPA four main parts:

- Java Persistence API
- JPA Query Language
- Java Persistence Criteria API
- Object to Relational Mapping (ORM) metadata

❖ JPA Entity Classes

- persistent fields
- persistent properties

❖ @Entity annotation

Object-Relational Mapping (ORM)

- ❖ Package: javax.persistence
- ❖ Simple keys - **@Id** annotation
- ❖ Composite keys
 - **Primary Key Class** – requirements and structure
 - Annotations – **@EmbeddedId, @IdClass**
- ❖ Relations between entity objects –
 - **uni- and bi-directional,**
 - **1:1, 1:many, many:1 many:many**

Advantages of Spring ORM

- ❖ Easier testing
- ❖ Common data access exceptions
- ❖ General resource management
- ❖ Integrated transaction management

ORM Cascade Updates

❖ Entities that have a dependency relationship can be managed declaratively by JPA using **CascadeType**:

- **ALL** – всички операции са каскадни
- **DETACH** – каскадно отстраняване
- **MERGE** – каскадно сливане
- **PERSIST** – каскадно персистиране
- **REFRESH** – каскадно обновяване
- **REMOVE** – каскадно премахване

**@OneToMany(cascade=REMOVE,
mappedBy="customer")**

```
public Set<Order> getOrders() { return orders; }
```

Entity Embeddables

- ❖ **@Embeddable** – аотира клас, който не е Entity, но може да бъде част от Entity
- ❖ **@Embedded** – embeds Embeddable class into Entity class
- ❖ Embedding can be hierarchical on multiple levels
- ❖ Annotations: **@AttributeOverride**, **@AttributeOverrides**, **@AssociationOverride**, **@AssociationOverrides**

Entity Inheritance

- ❖ Entity / Abstract entity
- ❖ Mapped superclass
- ❖ Non-entity superclass
- ❖ Entity -> DB tables mapping strategies
 - SingleTable per Class Hierarchy
 - TheTable per Concrete Class
 - The Joined Subclass Strategy

Persistent Units

❖ Persistent Unit description in **persistence.xml** file:

- description
- provider
- jta-data-source
- non-jta-data-source
- mapping-file
- jar-file
- class
- exclude-unlisted-classes
- properties

Persistent Unit Example 1

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="CustomerDBPU" transaction-type="JTA">
    <jta-data-source>jdbc/sample</jta-data-source>
    <class>customerdb.Customer</class>
    <class>customerdb.DiscountCode</class>
    <properties/>
  </persistence-unit>
</persistence>
```

Persistent Unit Example 2 - I

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="invoicingPU"
    transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>myinvoice.dbentities.ProductDB</class>
    <class>myinvoice.dbentities.PositionDB</class>
    <class>myinvoice.dbentities.InvoiceDB</class>
```


Persistent Unit Example 2 - II

```
<class>myinvoice.dbentities.ContragentDB</class>
```

```
<properties>
```

```
  <property name="toplink.jdbc.user" value="root"/>
```

```
  <property name="toplink.jdbc.password" value="root"/>
```

```
  <property name="toplink.jdbc.url"
    value="jdbc:mysql://localhost:3306/invoicing"/>
```

```
  <property name="toplink.jdbc.driver"
    value="com.mysql.jdbc.Driver"/>
```

```
</properties>
```

```
</persistence-unit>
```

```
</persistence>
```

Collection Type Persistent Fields

❖ Field or properties should be of **Collection** or **Map** type (usually generic):

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

❖ **@ElementCollection**

❖ **@CollectionTable** – name of additional table

❖ **@Embeddable**, **@Column**

❖ **@AttributeOverride**, **@AttributeOverrides**

Main JPA Annotations

- ❖ @PersistenceUnit,
- ❖ @PersistenceContext
- ❖ @Entity
- ❖ @Id
- ❖ @OneToOne
- ❖ @OneToMany
- ❖ @ManyToMany
- ❖ @DiscriminatorColumn
- ❖ @Column
- ❖ @JoinTable
- ❖ @JoinColumn
- ❖ @Embeddable
- ❖ @Embedded

JPA Entity Annotations Example

```
@Entity
public class Article {
    @Id
    @GeneratedValue
    private Long id;

    @Length(min=3, max=80)
    private String title;

    @Length(min=3, max=2048)
    private String content;

    @NotNull
    @ManyToOne
    @JoinColumn(name="AUTHOR_ID", nullable=false)
    private User author;

    @Length(min=3, max=256)
    private String pictureUrl;

    @Temporal(TemporalType.TIMESTAMP)
    private Date created = new Date();

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated = new Date();

    ... }

```



```
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue
    private long id;

    @NotNull
    @Length(min = 3, max = 30)
    private String username;
    ...

    @NotNull
    private String roles = "ROLE_USER";

    @OneToMany(mappedBy = "author",
        cascade = CascadeType.ALL,
        orphanRemoval=true)
    Collection<Article> articles =
        new ArrayList<>();

    @Temporal(TemporalType.TIMESTAMP)
    private Date created = new Date();

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated = new Date();

    ... }

```

JPA Entities: @ManyToMany

```
@Entity
public class Book {
    @Id @GeneratedValue
    private int id;

    @NotNull
    private String title;

    @ManyToOne
    @JoinColumn(name = "PUBLISHER_ID",
        referencedColumnName = "id")
    private Publisher publisher;

    @Column(name = "PUBLISHED_DATE") @PastOrPresent
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private LocalDate publishedDate;

    @Pattern(regexp = "\\d{10}|\\d{13}")
    private String isbn;

    @NotNull @Min(0)
    private double price;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name="BOOK_AUTHOR", joinColumns=
        @JoinColumn(name="BOOK_ID",referencedColumnName="ID"),
        inverseJoinColumns=
        @JoinColumn(name="AUTHOR_ID",referencedColumnName="ID")
    )
    private List<Author> authors = new ArrayList<>();
}
```

```
@Entity
public class Author {
    @Id @GeneratedValue
    private int id;

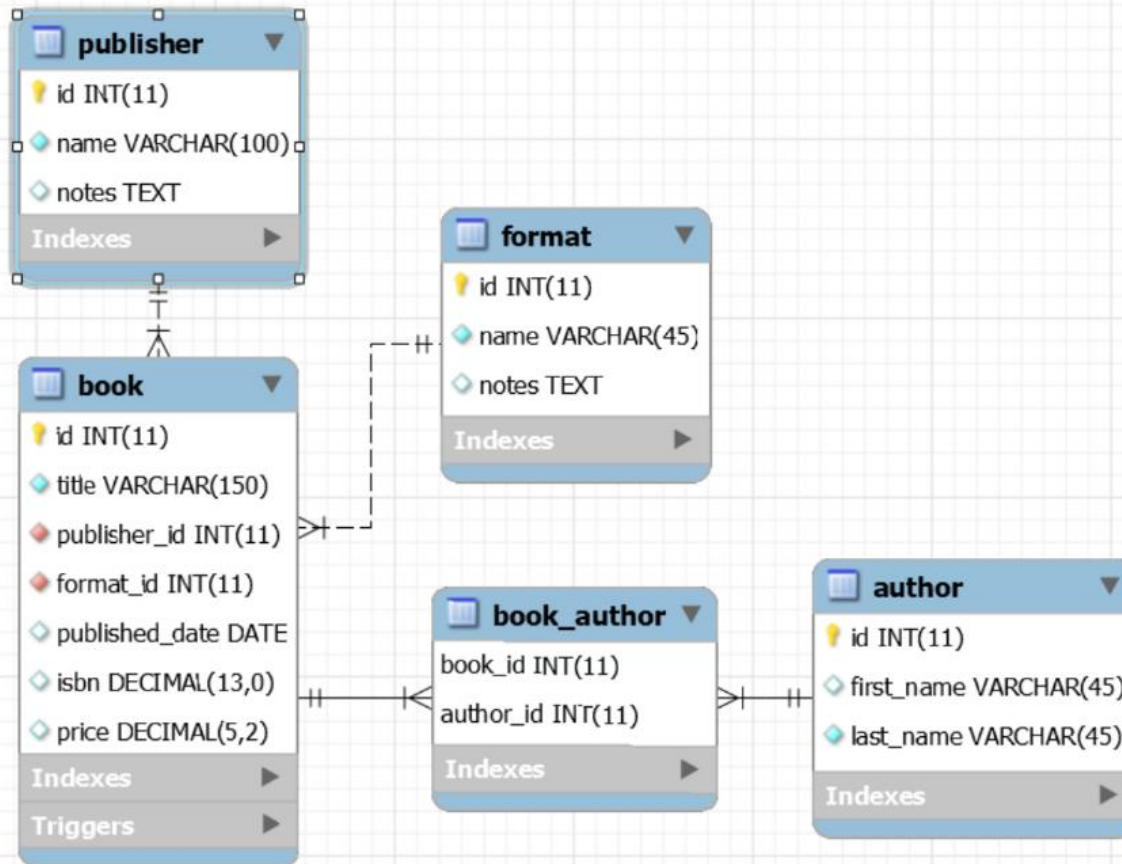
    @NotNull
    @Length(min=2, max=60)
    @Column(name = "first_name")
    private String firstName;

    @NotNull
    @Length(min=2, max=60)
    @Column(name = "last_name")
    private String lastName;

    @ManyToMany(mappedBy = "authors",
        fetch = FetchType.EAGER)
    List<Book> books = new ArrayList<>();
}
```



JPA Entities: ER Diagram



Java Persistence Query Language

- ❖ Object-oriented database queries
- ❖ Navigation
- ❖ Abstract schema
- ❖ Path expression
- ❖ State field
- ❖ Relationship field

Java Persistence Query Language

❖ SELECT

❖ FROM

❖ WHERE

❖ GROUP BY

❖ HAVING

❖ ORDER BY

❖ UPDATE

❖ DELETE

❖ AS, IN

❖ LIKE

❖ EXISTS, ANY, ALL

❖ NEW

JPA Setup in Spring

```
<beans>
  <bean id="myEmf"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="myPersistenceUnit"/>
  </bean>
</beans>
```

```
<beans>
  <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

```
<beans>
  <bean id="myEmf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="someDataSource"/>
    <property name="loadTimeWeaver">
      <bean
class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
    </property>
  </bean>
</beans>
```

JSR-303: Bean Validation (1)

- ❖ Bean Validation стартира през юли 2006 - JSR 303
- ❖ Финализирана е на 16 ноември 2009
- ❖ Валидацията е обща задача, която се осъществява през всички слоеве на приложението – от презентационния до персистирането на данните
- ❖ Често една и съща логика за валидация се реализира многократно във всеки слой, което води до чести грешки е несъответствия, както и до дублиране на усилия
- ❖ За да се справят с проблема, често разработчиците кодират валидационната логика директно в домейн модела, което води до смесване на бизнес логика и метаданни за валидиране на отделните свойства

JSR-303: Bean Validation (2)

- ❖ JSR 303: Bean Validation предлага набор от стандартни ограничения (constraints) относно данните, под формата на анотации, които обозначават полета, методи или класове на **JavaBean** компоненти, като например **JPA Entities** или **JSF Managed Beans**
- ❖ Има множество предварително дефинирани анотации, както и възможност за създаване на собствени такива и свързването им с клас, който да реализира валидационната логика
- ❖ Вградените анотации са дефинирани в пакет **javax.validation.constraints**

Bean Validation Annotations (1):

- ❖ **@AssertFalse** – елемент от булев тип трябва да е лъжа
- ❖ **@AssertTrue** – елемент от булев тип трябва да е истина
- ❖ **@Min, @DecimalMin** – минимална стойност на елемент от ЧИСЛОВ ТИП
- ❖ **@Max, @DecimalMax** – максимална стойност на елемент от ЧИСЛОВ ТИП
- ❖ **@Digits** – атрибути fraction и integer за дробната и цялата част на елемент от числов тип
- ❖ **@Future** – валидиране на бъдеща дата (Date и Calendar)
- ❖ **@Past** – валидиране на минала дата (Date и Calendar)
- ❖ **@Size** – min и max размер на String, Collection, Map или Array

Bean Validation Annotations (2):

- ❖ **@NotNull** – елементът трябва да е различен от null
- ❖ **@Null** – елементът трябва е null
- ❖ **@Pattern** – елементът трябва да съответствува на посочения в атрибута regex регулярен израз
- ❖ **@Valid** – анотация в пакета javax.validation, която указва, че трябва да се извърши рекурсивна валидация на всички обекти свързани с посочения обект
- ❖ Възможно е създаване на нови собствени анотации и композитни анотации с използване на **@Constraint**, **@GroupSequence**, **@ReportAsSingleViolation**, **@OverridesAttribute**

Bean Validation Examples:

```
public class Email {  
    @NotEmpty @Pattern(".*+@.*+\.[a-z]+")  
    private String from;  
    @NotEmpty @Pattern(".*+@.*+\.[a-z]+")  
    private String to;  
    @NotEmpty  
    private String subject;  
    @Min(1) @Max(10)  
    private Integer priority;  
    @NotEmpty  
    private String body;  
    ...  
}
```

Bean Validation – Custom Annotation:

@Size(min=4, max=4)

@ConstraintValidator(validatedBy = **PostCodeValidator.class)**

@Documented

@Target({ANNOTATION_TYPE, METHOD, FIELD})

@Retention(RUNTIME)

```
public @interface PostCode {  
    public abstract String message() default  
        "{package.name.PostCode.message}";  
    public abstract Class<?>[] groups() default {};  
    public abstract Class<? extends ConstraintPayload>[]  
        payload() default {};  
}
```

Bean Validation – Class PostCodeValidator

```
public class PostCodeValidator implements
    ConstraintValidator<PostCode, String> {
    private final static Pattern POSTCODE_PATTERN =
        Pattern.compile("\\d{4}");
    public void initialize(PostCode constraintAnnotation) { }
    public boolean isValid(String value,
        ConstraintValidatorContext context) {
        return POSTCODE_PATTERN.matcher(value).matches();
    }
}
```

Bean Validation – композитна анотация:

```
@ConstraintValidator(validatedBy = {}) @Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RUNTIME)
@Pattern(regexp = "\\d{4}")
@ReportAsSingleViolation
public @interface PostCode {
    public abstract String message() default
        "{package.name.PostCode.message}";
    public abstract Class<?>[] groups() default {};
    public abstract Class<? extends ConstraintPayload>[]
        payload() default {};
```

Additional Examples

Learning Spring 5 book examples are available @
GitHub:<https://github.com/PacktPublishing/Learning-Spring-5.0>

Spring 5 Core Referenc Documentation:
<https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html>

JPA in Java EE 6 Tutorial –
<https://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html>

Thank's for Your Attention!



Trayan Iliev

CEO of IPT – Intellectual Products
& Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>

<https://plus.google.com/+IproductOrg>