JAVA EE & JPA (FOUNDATIONS) 2026

Rajeev Gupta

rgupta.mtech@gmail.com

https://www.linkedin.com/in/rajeevguptajavatrainer

Trainer's Profile

- Senior Corporate Trainer & Technical Consultant with 21+ years helping global engineering teams master Java, Spring Boot, Microservices, AWS Cloud, DevOps, and GenAI for Java.

I specialize in enabling teams to build scalable, observable, production-ready microservices using real-world architecture patterns, hands-on labs, and deep technical coaching.

Technical Expertise

• Java 8–25 — Streams, Concurrency, JVM Internals
• Spring Boot Ecosystem — Security, Data, Spring Cloud
• Microservices Architecture — API Design, SAGA, CQRS, resilience, observability
• Event Streaming — Kafka, RabbitMQ
• AWS Cloud-Native Development — Built & deployed Spring Boot microservices using ECS, EKS, API Gateway, SQS/SNS, with CI/CD via CodePipeline, CodeBuild, GitHub Actions, Terraform
• Containers & DevOps — Docker, Kubernetes (EKS), IaC, CI/CD pipelines
• GenAI for Java Developers — LangChain4J, Spring AI, RAG, Vector DBs, Java-based AI Agents

Corporate Client

- Bank Of America
- MakeMyTrip
- GreatLearning
- Deloitte
- Kronos
- Yamaha Moters
- IBM
- Sapient
- Accenture
- Airtel
- Gemalto
- Cyient Ltd
- Fidelity Investment Ltd
- Blackrock
- Mahindra Comviva
- Iris Software
- harman
- Infosys
- Espire
- Steria
- Incedo
- Capgemini
- HCL
- CenturyLink
- Nucleus
- Ericsson
- Ivy Global
- Avaya
- NEC Technologies
- A.T. Kearney
- UST Global
- TCS
- North Shore Technologies
- Incedo
- Genpact
- Torry Harris
- Indian Air force
- Indian railways

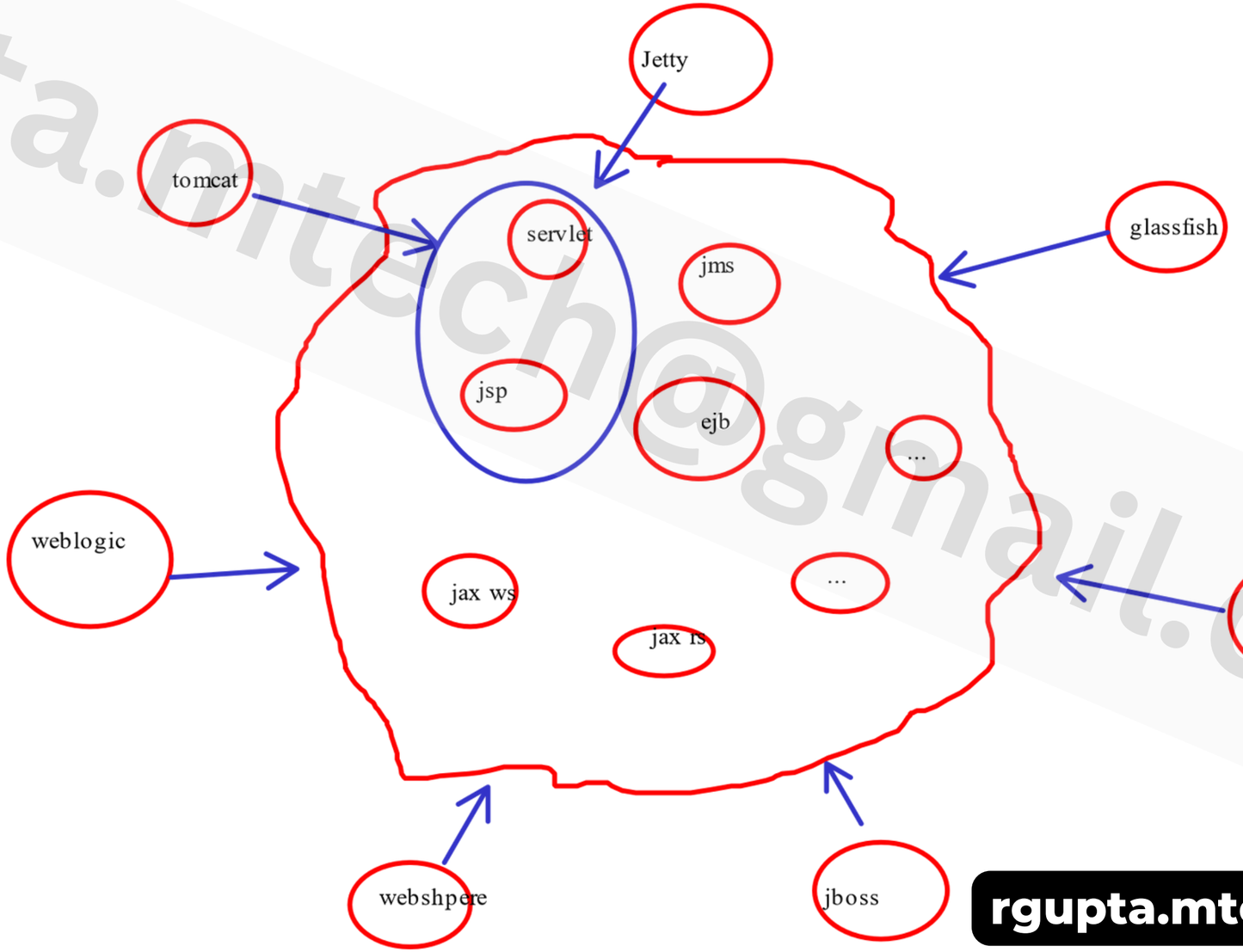
rgupta.mtech@gmail.com

JCP

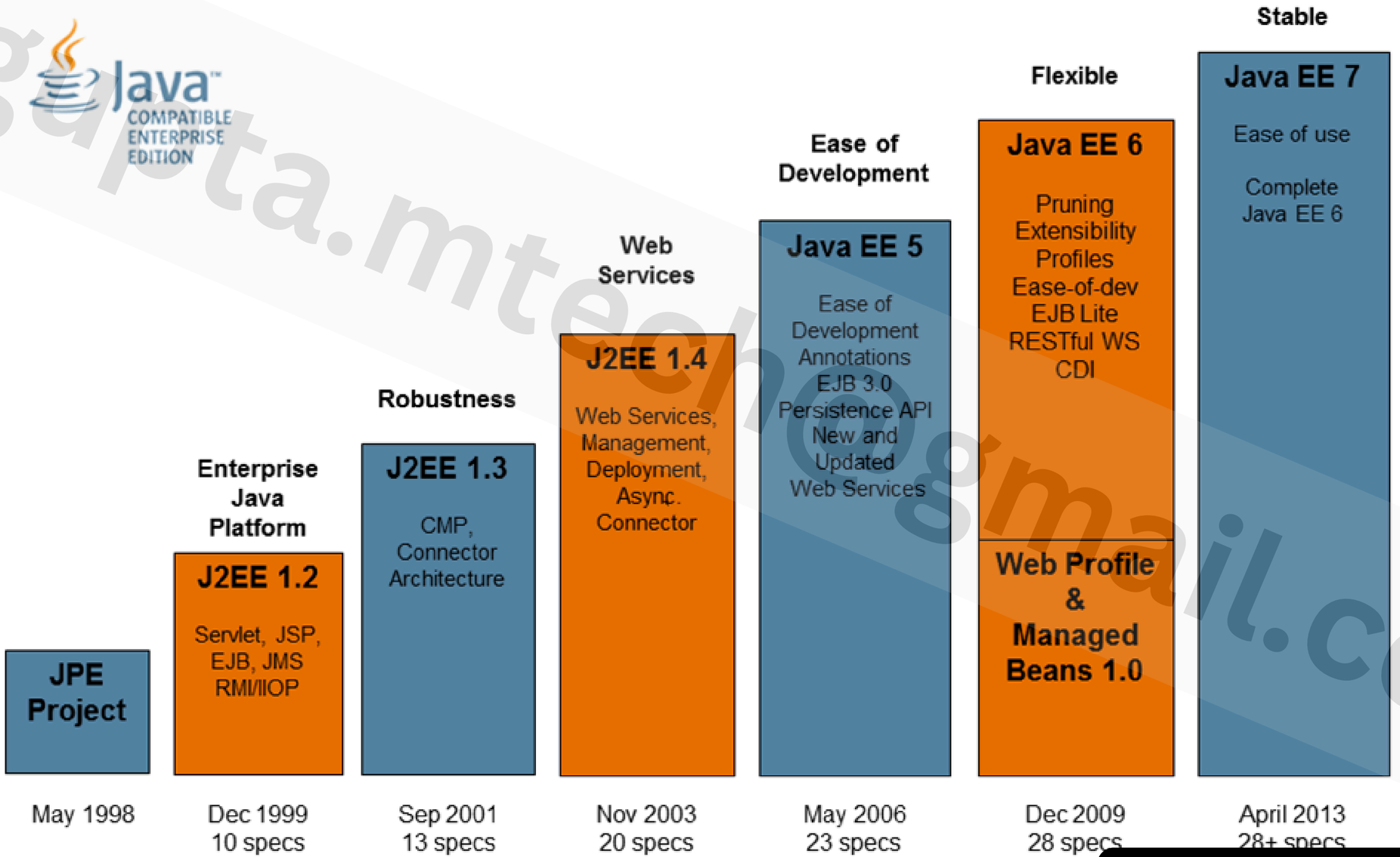
JSR

What is J2EE? Introduction

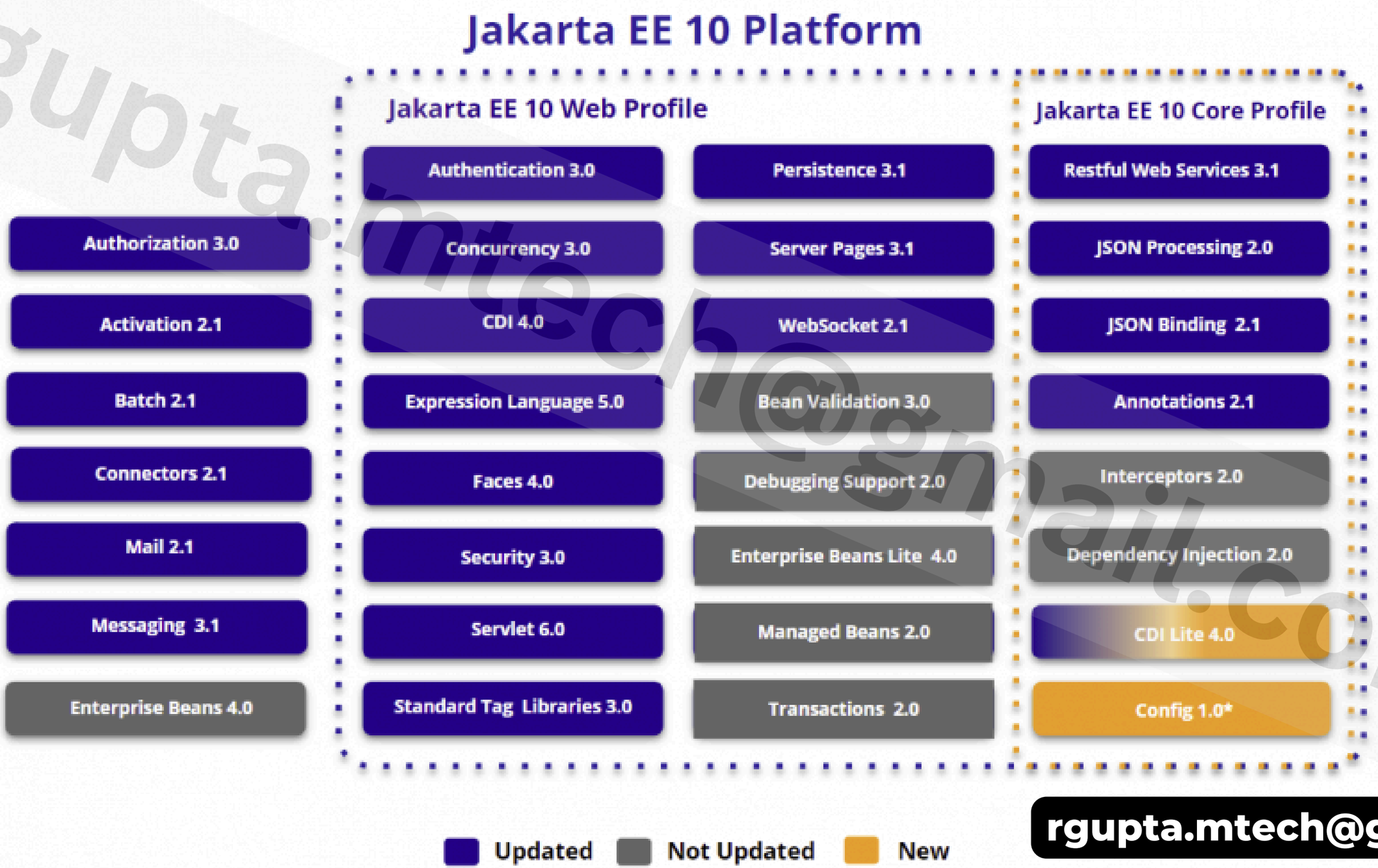
J2EE is group of specification to create dynamic distributed application



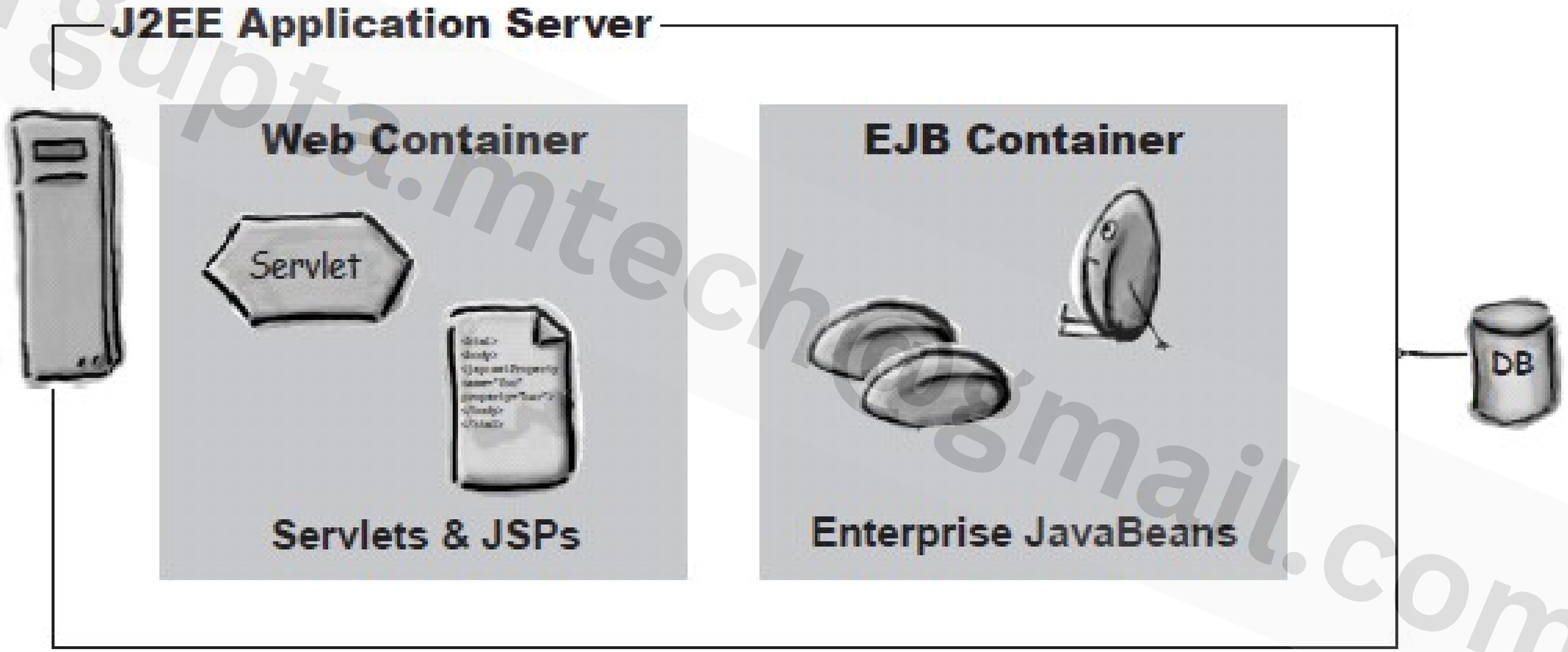
What is J2EE? Introduction



Jakarta EE



J2EE application server



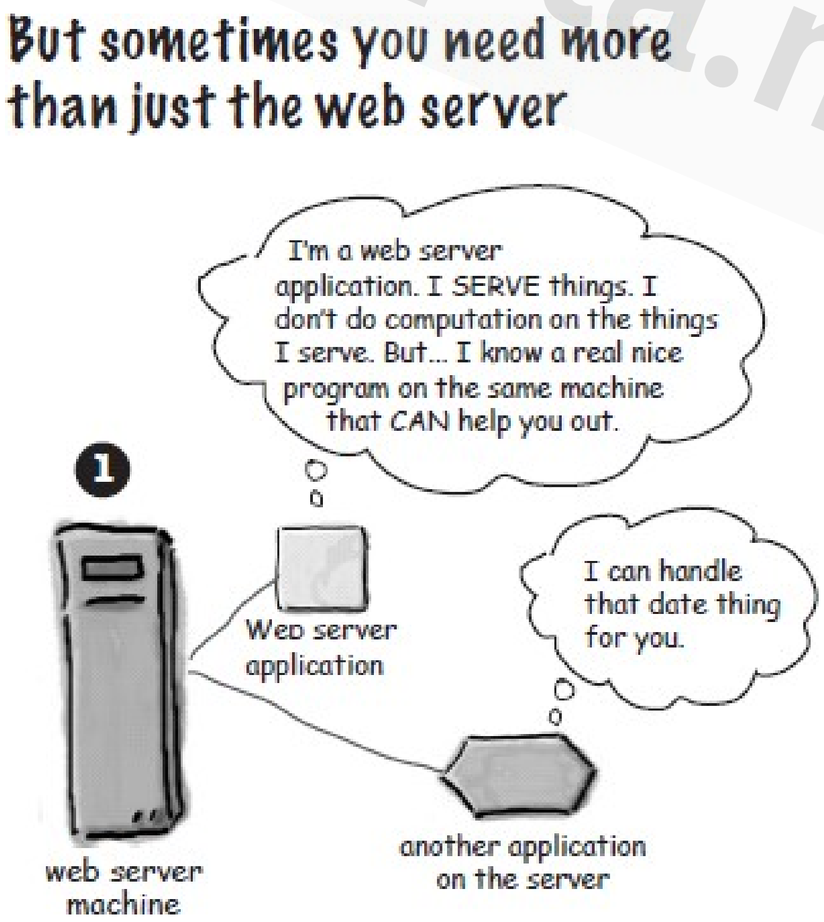
Introduction to Servlet API

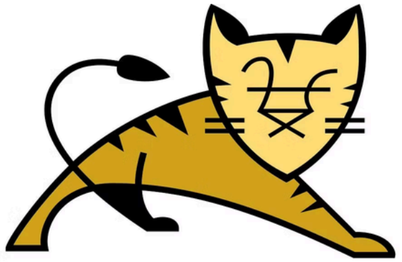
Web Server vs Web Container vs Application Server

Apache Tomcat

rgupta.mtech@gmail.com

Why we need tomcat?

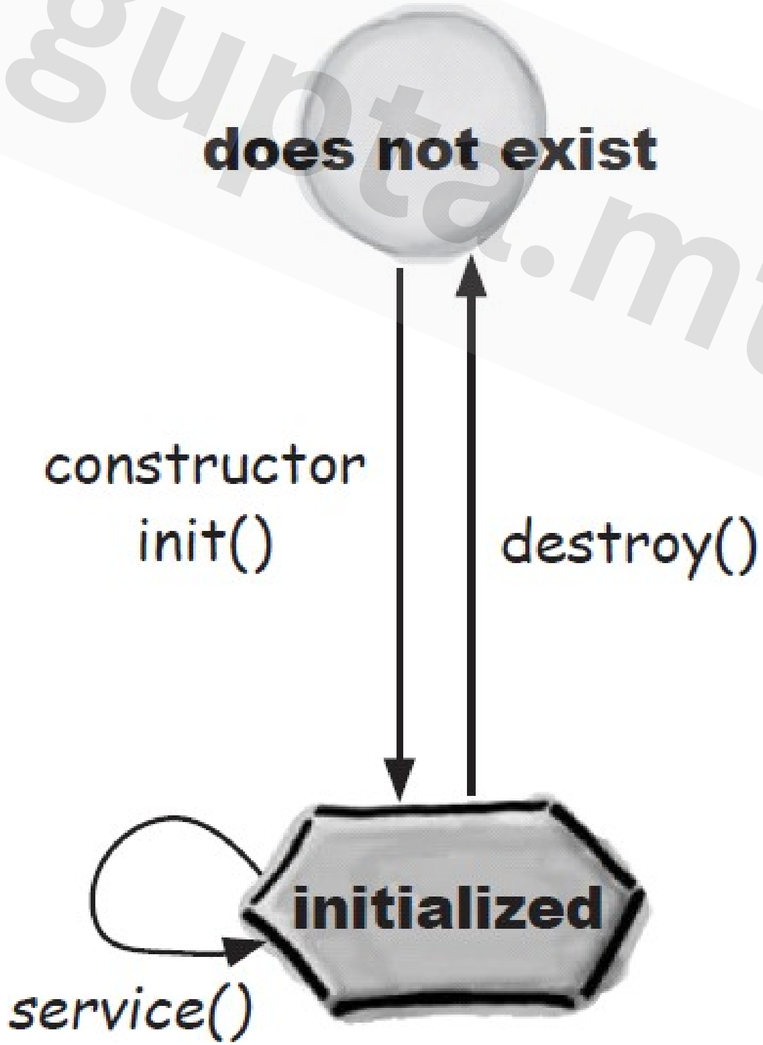


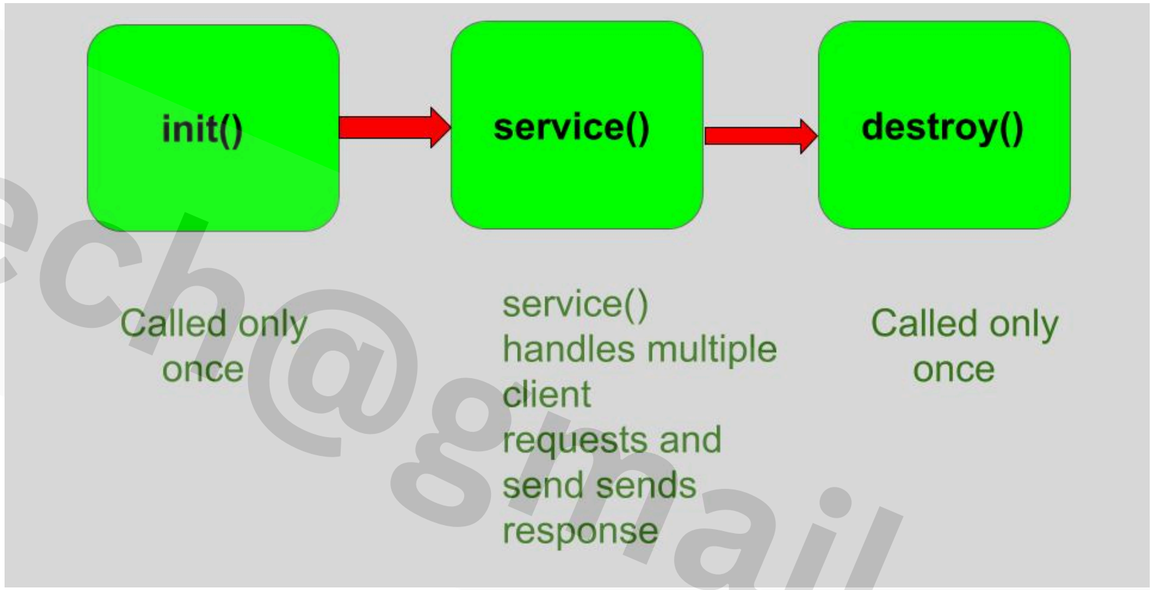


Apache Tomcat

- **Communication support**
- **Lifecycle management**
- **Multithreading support**
- **Declarative security**
- **JSP Support**
- **Dynamic Content**
- **Saving data**

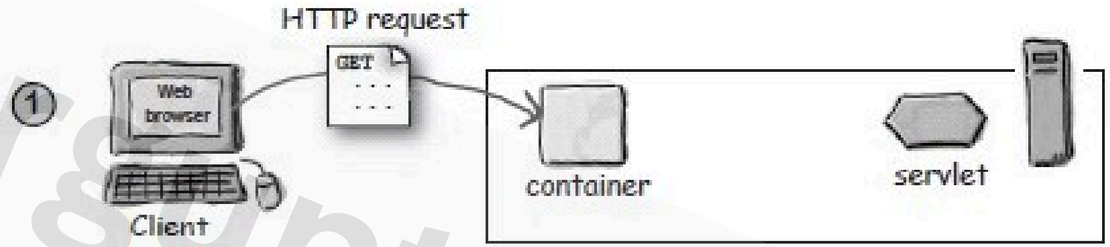
What is Servlet?



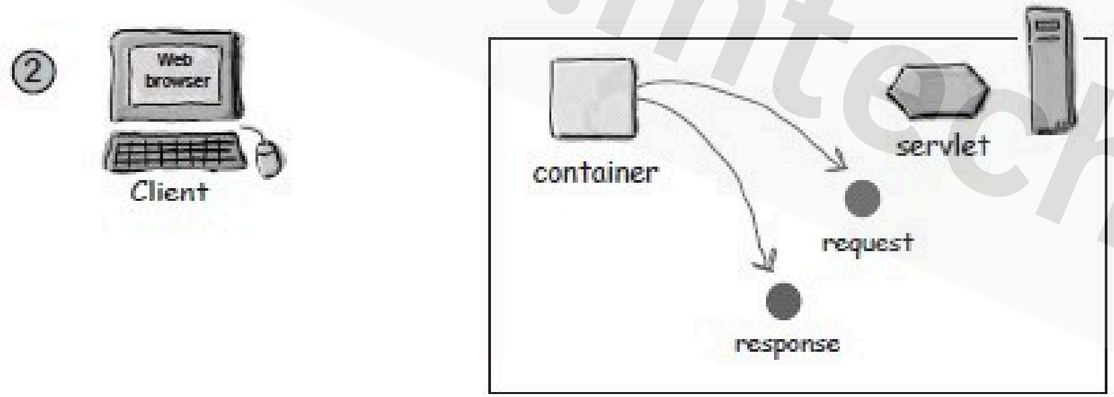


- What is servlet?
- Why we need servlet?

How a container handle a dynamic request?



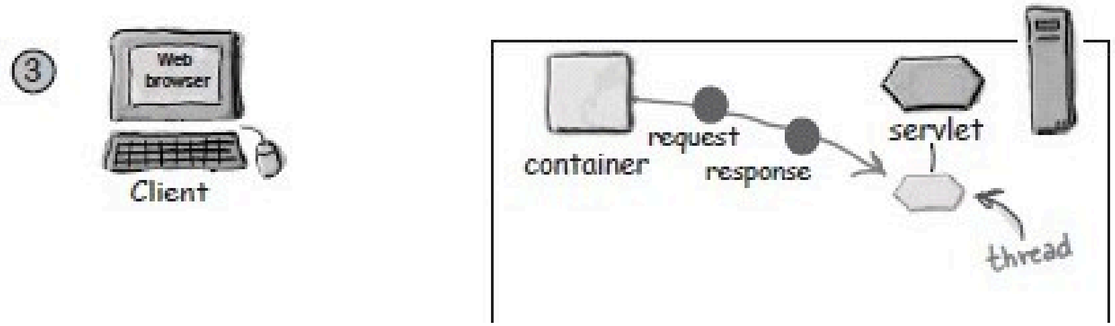
User clicks a link that has a URL to a servlet instead of a static page.



The container "sees" that the request is for a servlet, so the container creates two objects:

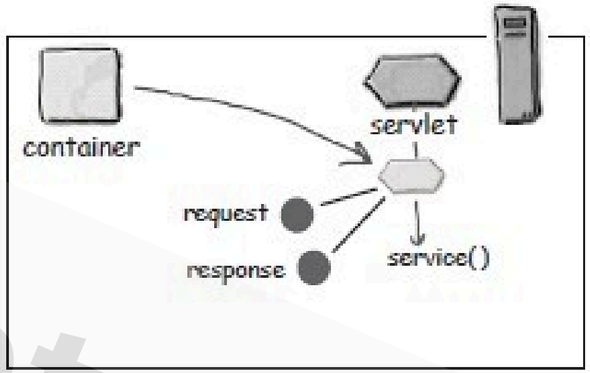
1) HttpServletResponse

2) HttpServletRequest



The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

④

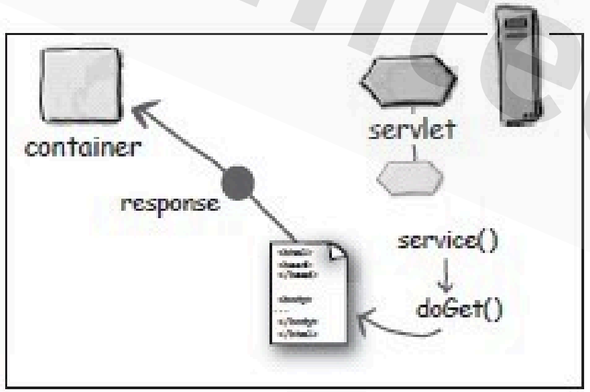


The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.

---

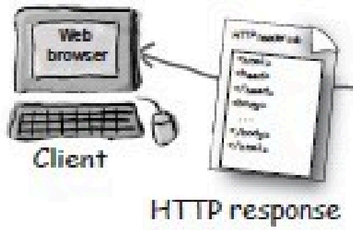
⑤

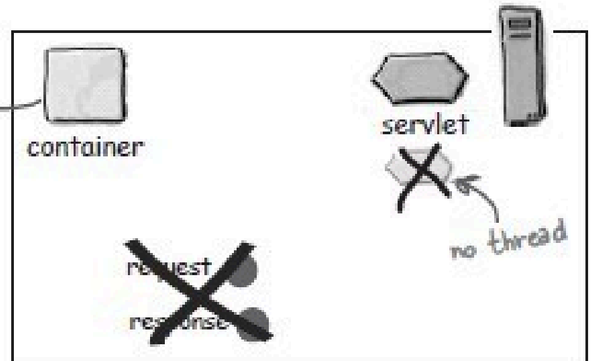


The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

---

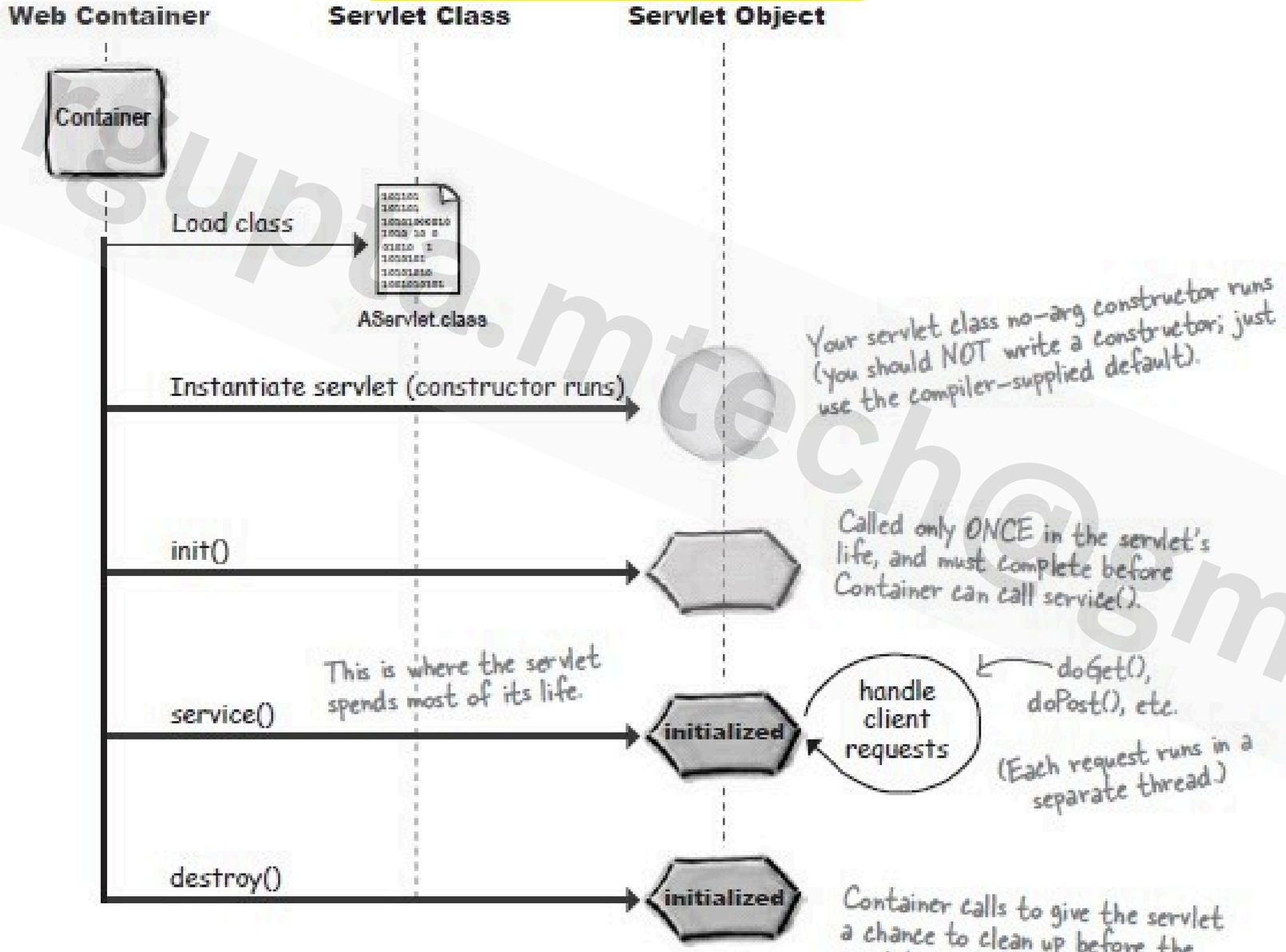
⑥



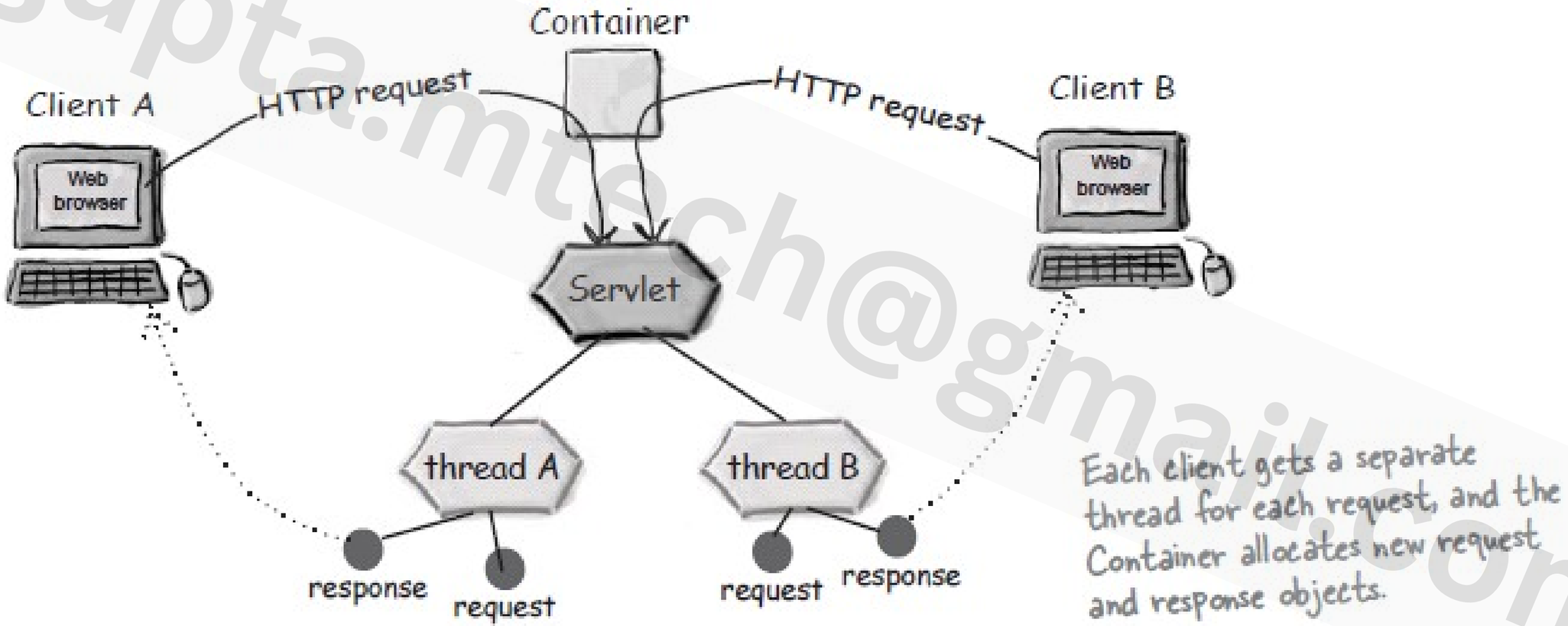


The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

Servlet Life Cycle



Each request runs in a separate thread!



ServletContext vs. ServletConfig

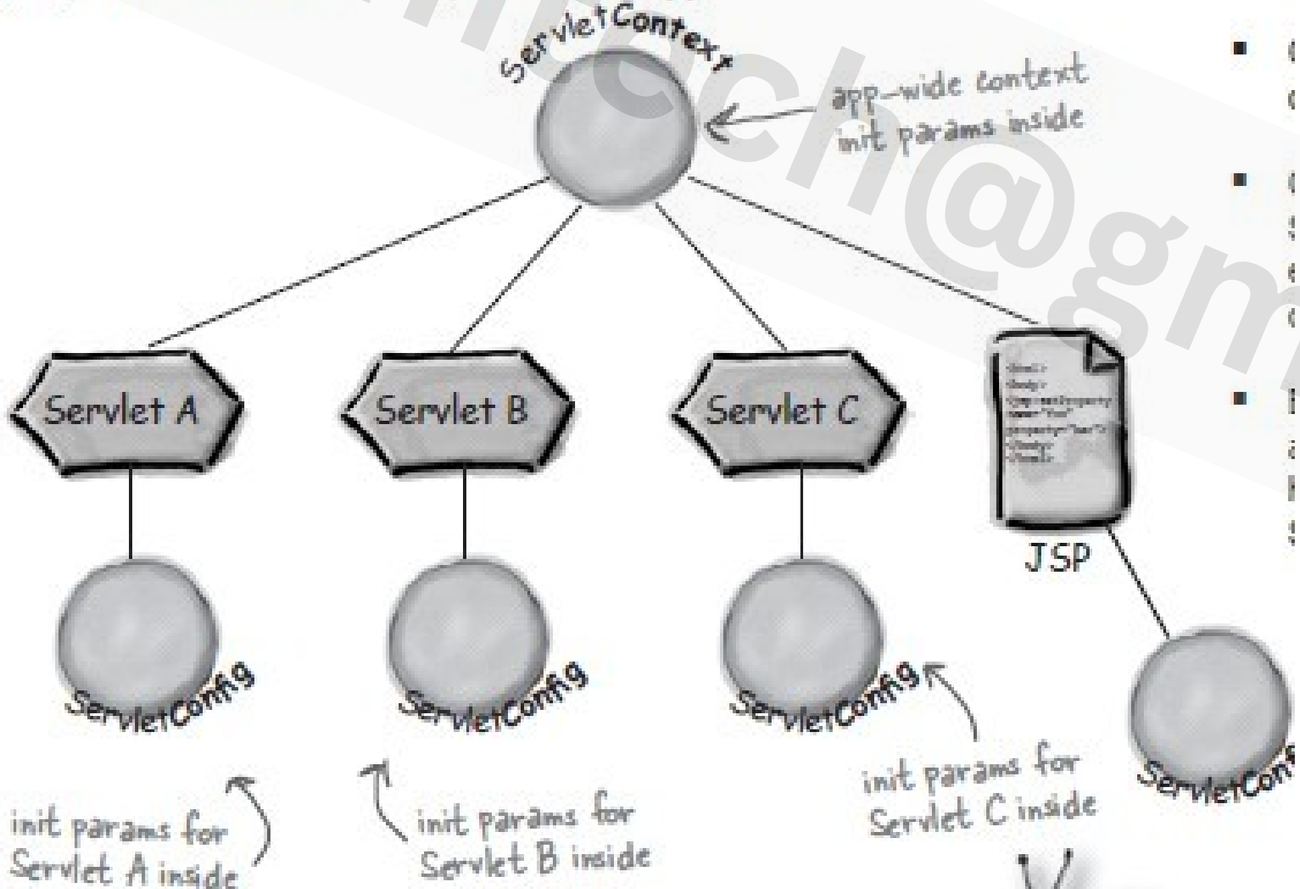
rgupta.mtech@gmail.com

ServletConfig vs ServletContext

ServletConfig is one per servlet

ServletContext is one per web app

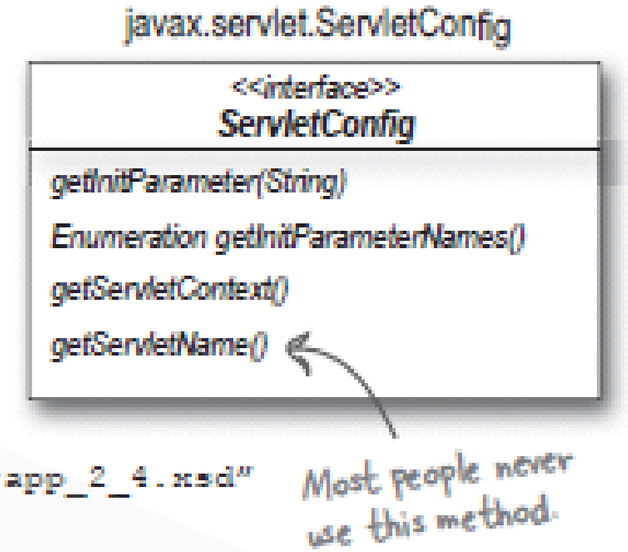
There's only one ServletContext for an entire web app, and all the parts of the web app share it. But each servlet in the app has its own ServletConfig. The Container makes a ServletContext when a web app is deployed, and makes the context available to each Servlet and JSP (which becomes a servlet) in the web app.



Setting ServletConfig

Testing your ServletConfig

ServletConfig's main job is to give you init parameters. It can also give you a ServletContext, but we'll usually get a context in a different way, and the getServletName() method is rarely useful.



In the DD (web.xml) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>com.example.TestInitParams</servlet-class>
    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>mainEmail</param-name>
      <param-value>blooper@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>BeerParamTests</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

and getting it in Servlet...

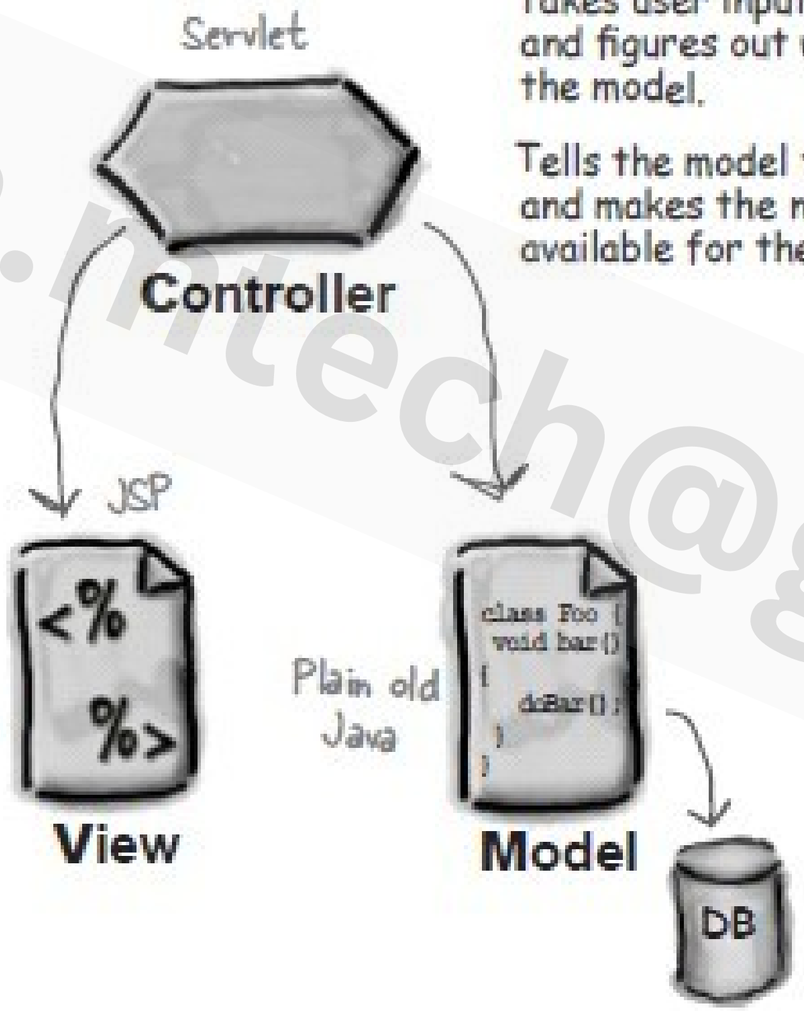
In a servlet class:

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestInitParams extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                    throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test init parameters<br>");

        java.util.Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
           out.println("<br>param name = " + e.nextElement() + "<br>");
        }
        out.println("main email is " + getServletConfig().getInitParameter("mainEmail"));
        out.println("<br>");
        out.println("admin email is " + getServletConfig().getInitParameter("adminEmail"));
    }
  }
```

MVC

rgupta.mtech@gmail.com

MVC with Servlet JSP



CONTROLLER

Takes user input from the request and figures out what it means to the model.

Tells the model to update itself, and makes the new model state available for the view (the JSP).

VIEW

Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it). It's also the part that gets the user input that goes back to the Controller.

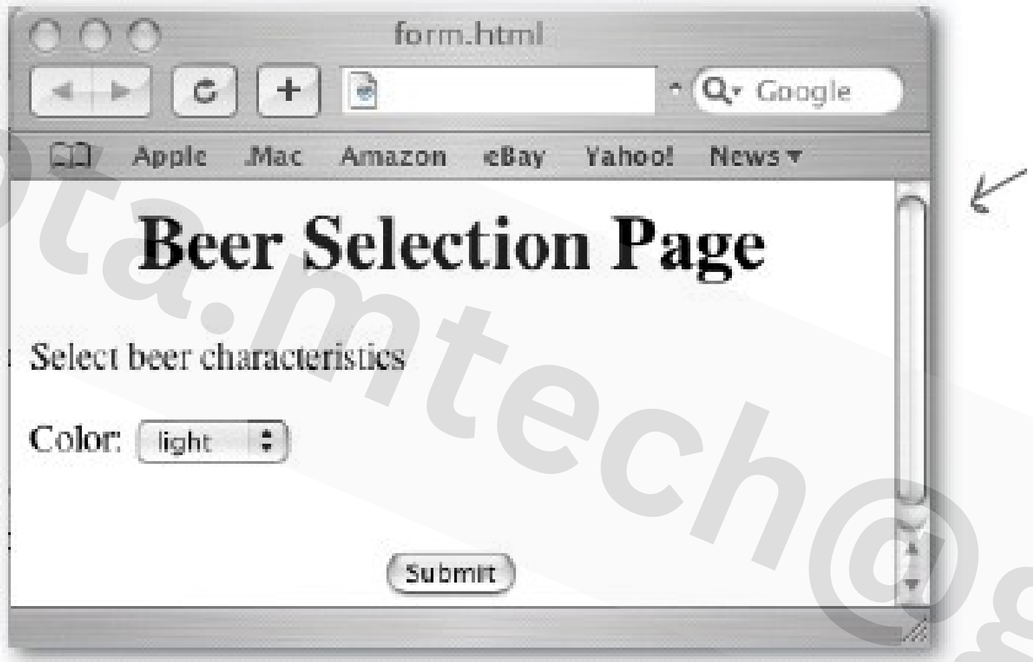
MODEL

Holds the real business logic and the state. In other words, it knows the rules for getting and updating the state.

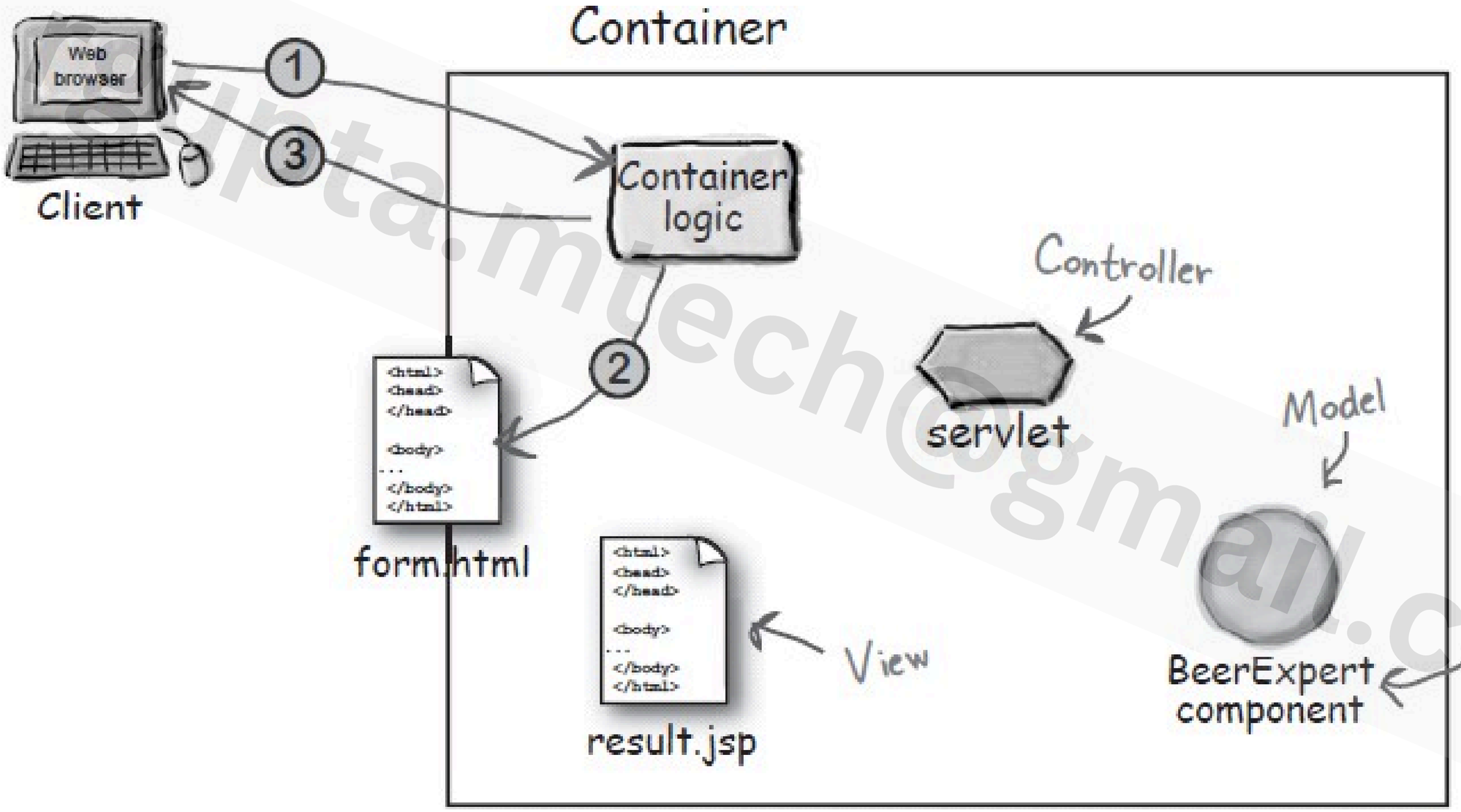
A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.

It's the only part of the system that talks to the database (although it probably uses *another* object for the actual DB communication, but we'll save that pattern for later...)

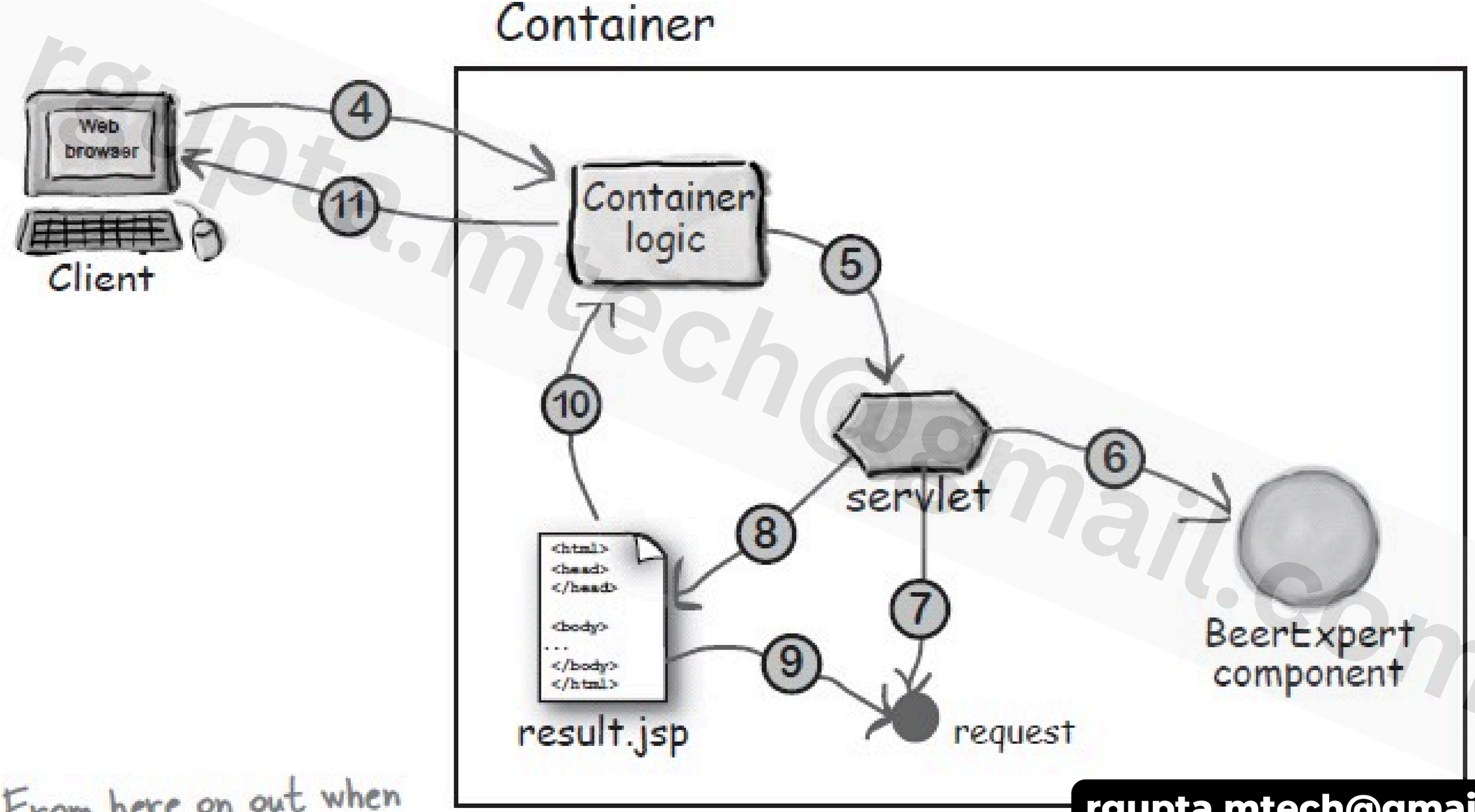
Beer Selection Page



MVC flow



MVC flow



From here on out when

Servlet Listners

rgupta.mtech@gmail.com

Eight Listners

Scenario	Listener interface	Event type
You want to know if an attribute in a web app context has been added, removed, or replaced.	javax.servlet.**ServletContextAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced*	ServletContextAttributeEvent
You want to know how many concurrent users there are. In other words, you want to track the active sessions. (We cover sessions in detail in the next chapter).	javax.servlet.http.**HttpSessionListener** *sessionCreated* *sessionDestroyed*	HttpSessionEvent
You want to know each time a request comes in, so that you can log it.	javax.servlet.**ServletRequestListener** *requestInitialized* *requestDestroyed*	ServletRequestEvent
You want to know when a request attribute has been added, removed, or replaced.	javax.servlet.**ServletRequestAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced*	ServletRequestAttributeEvent
You have an attribute class (a class for an object that will be stored as an attribute) and you want objects of this type to be notified when they are bound to or removed from a session.	javax.servlet.http.**HttpSessionBindingListener** *valueBound* *valueUnbound*	HttpSessionBindingEvent
You want to know when a session attribute has been added, removed, or replaced.	javax.servlet.http.**HttpSessionAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced* Watch out for this naming inconsistency! The Event for HttpSessionAttributeListener is NOT what you expect (you expect HttpSessionAttributeEvent).	HttpSessionBindingEvent
You want to know if a context has been created or destroyed.	javax.servlet.**ServletContextListener** *contextInitialized* *contextDestroyed*	ServletContextEvent
You have an attribute class, and you want objects of this type to be notified when the session to which they're bound is migrating to and from another JVM.	javax.servlet.http.**HttpSessionActivationListener** *sessionDidActivate* *sessionWillPassivate*	HttpSessionEvent It's NOT "HttpSessionActivationEvent"

Servlet Filter API

rgupta.mtech@gmail.com

Servlet Filter API

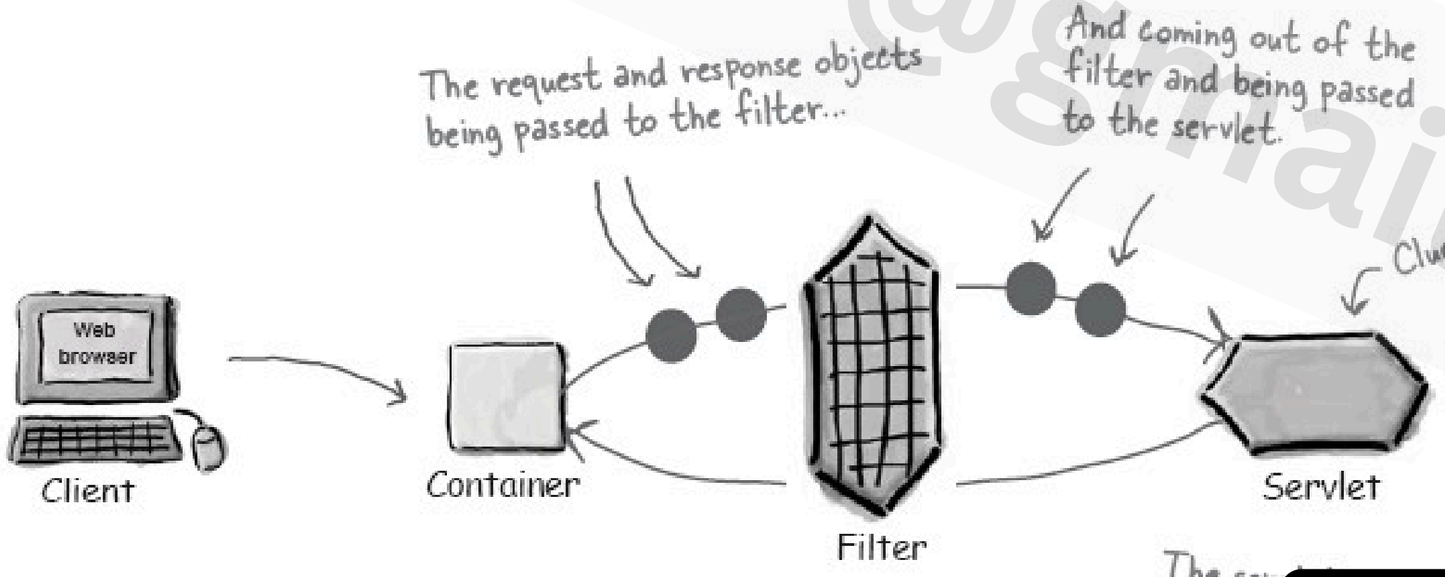
The Power of Filters

Servlet Filter API

How about some kind of "filter"?

Filters are Java components—very similar to servlets—that you can use to intercept and process requests *before* they are sent to the servlet, or to process responses *after* the servlet has completed, but *before* the response goes back to the client.

The Container decides when to invoke your filters based on declarations in the DD. In the DD, the deployer maps which filters will be called for which request URL patterns. So it's the deployer, not the programmer, who decides which subset of requests or responses should be processed by which filters.



Servlet Filter API

Request filters can:

- perform security checks
- reformat request headers or bodies
- audit or log requests

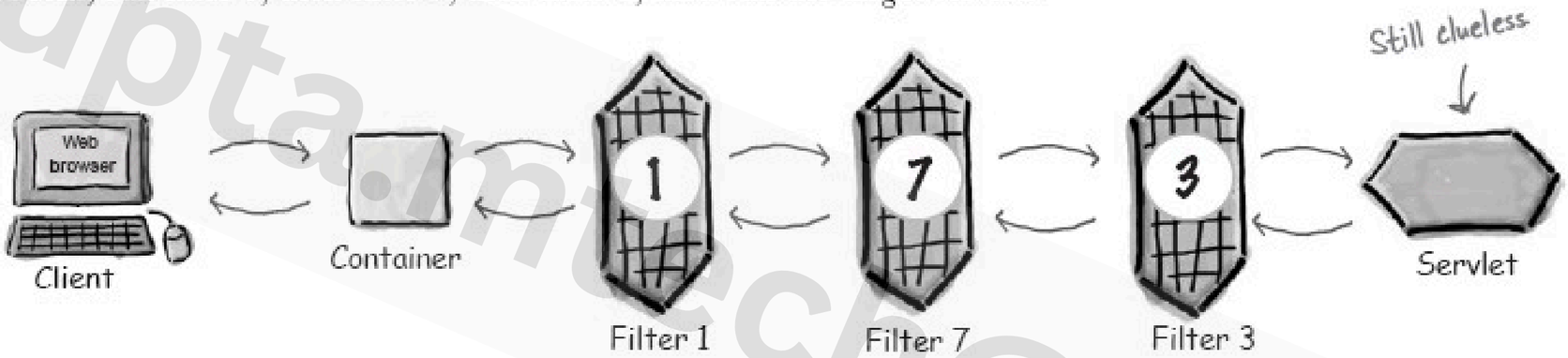
Response filters can:

- compress the response stream
- append or alter the response stream
- create a different response altogether

Filters are modular, and configurable in the DD

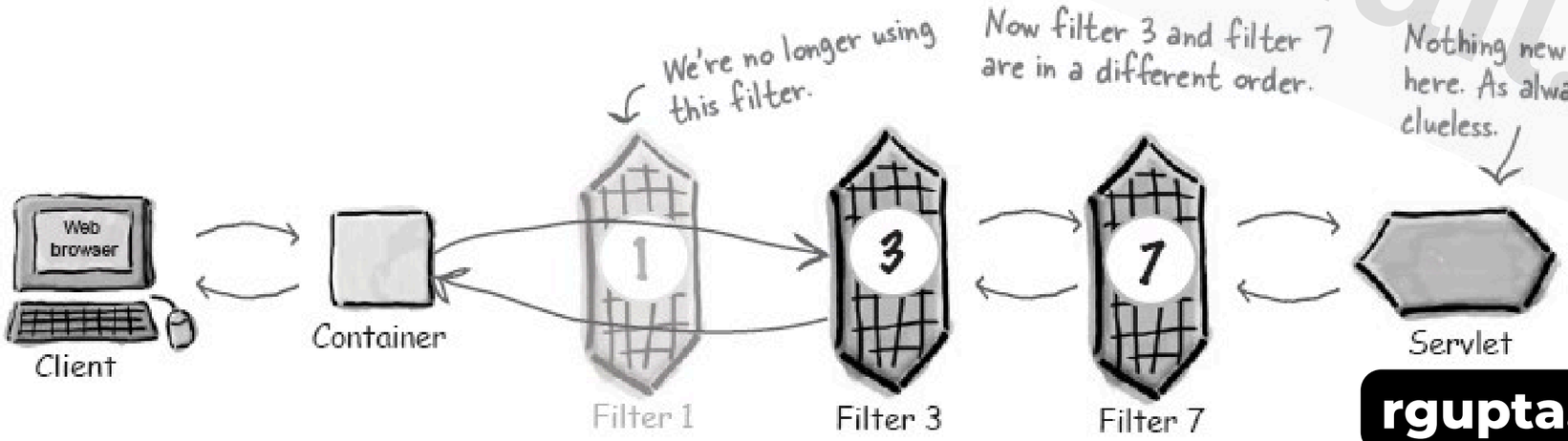
DD configuration 1:

Using the DD, you can link them together by telling the Container: "For these URLs, run filter 1, then filter 7, then filter 3, then run the target servlet."



DD configuration 2:

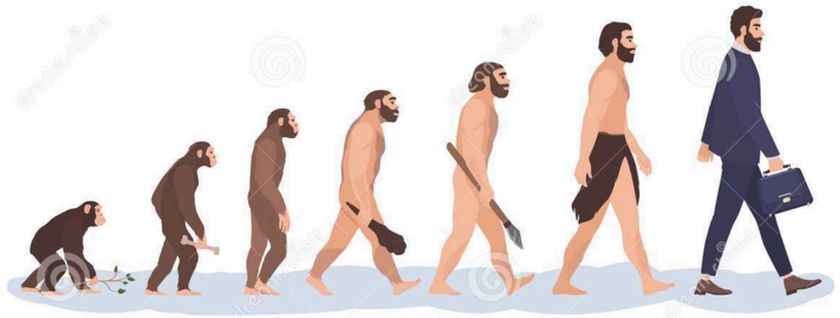
Then, with a quick change to the DD, you can delete and swap them with: "For these URLs, run filter 3, then filter 7, and then the target servlet."

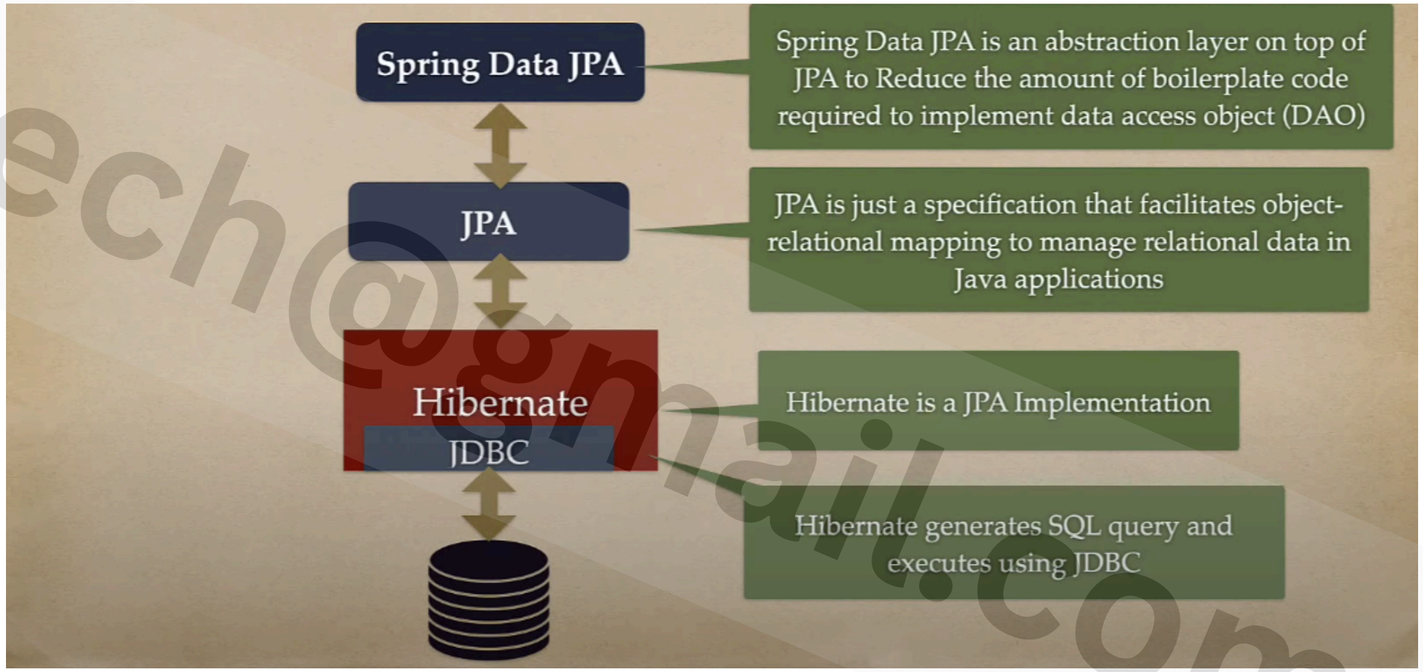


JPA
Java Persistance API

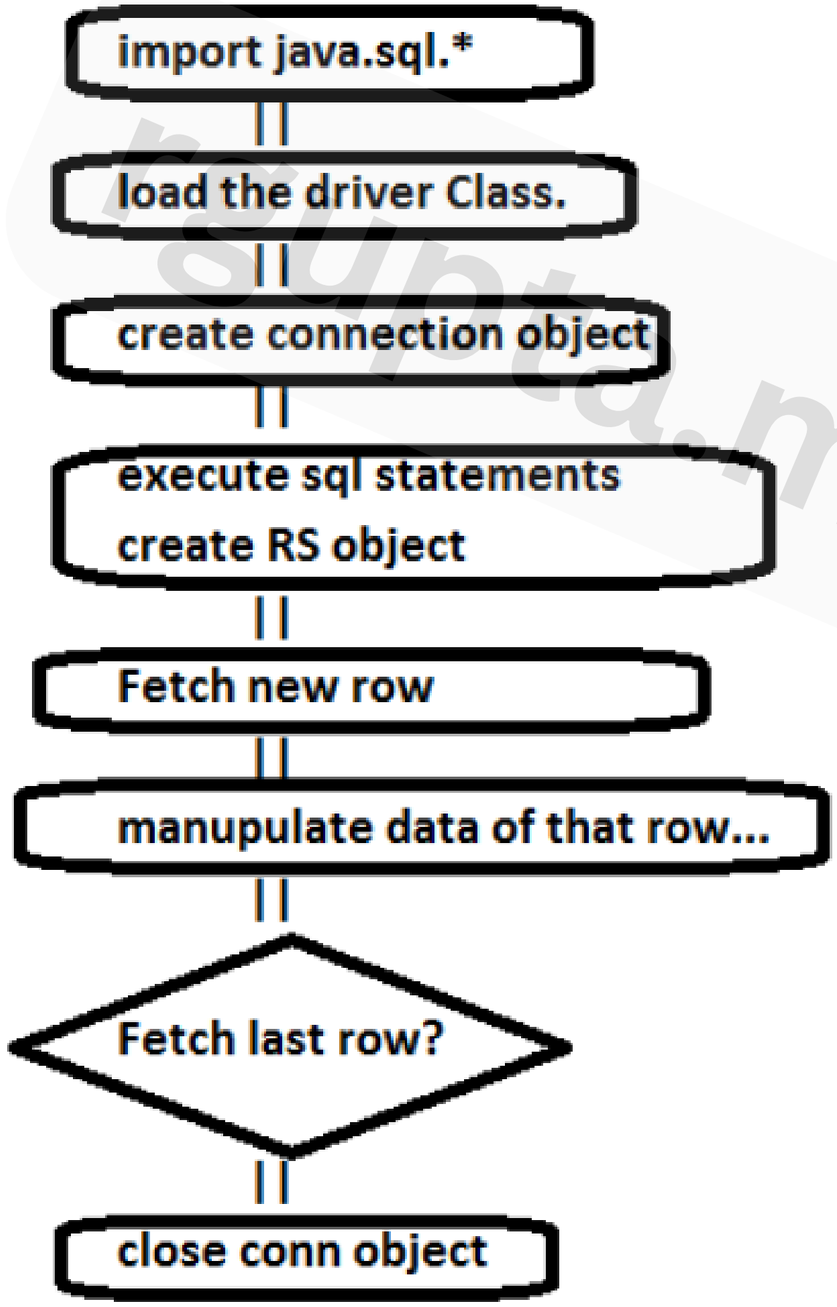
Data Layer Evolution Process

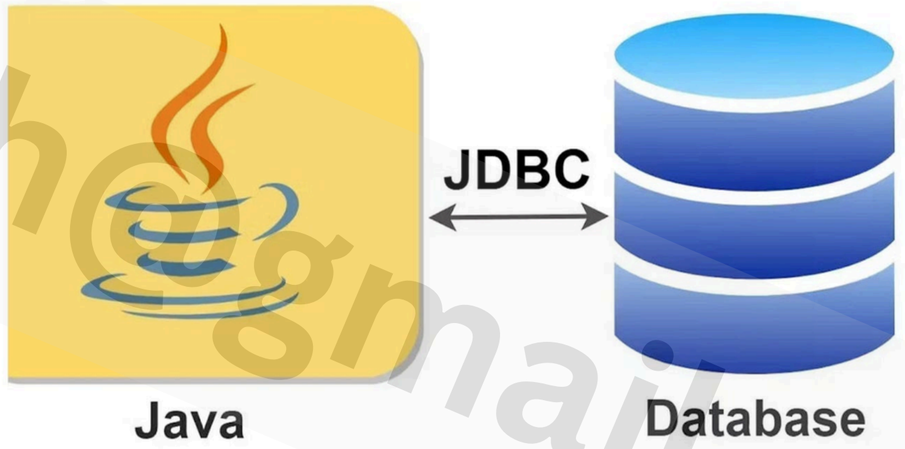
Database layer evolution process is very similer to human evolution process





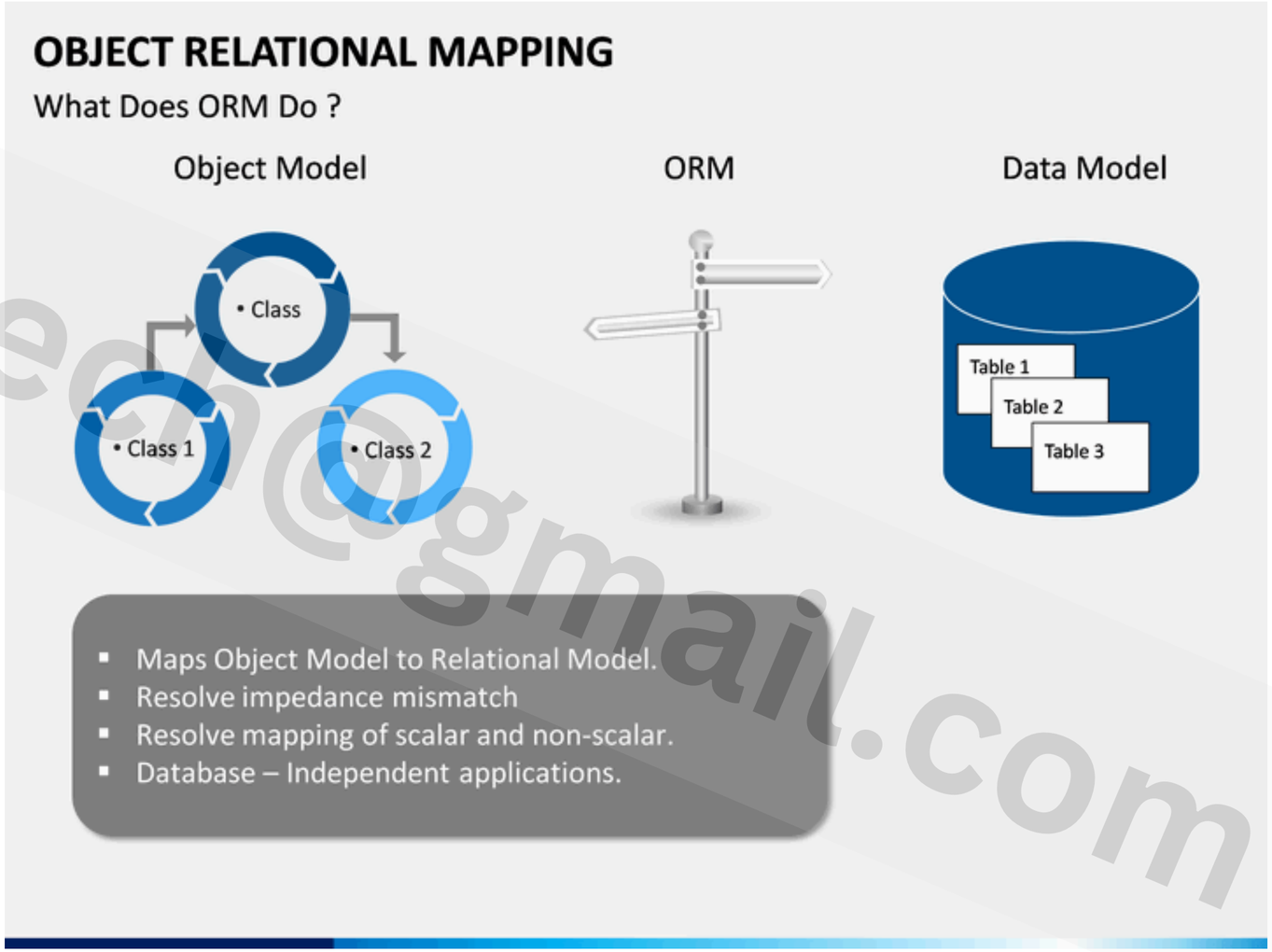
JDBC





What is Object Relational Mapping ORM?

Object Relational Mapping (ORM) is a technique used in creating a "bridge" between object-oriented programs and relational databases.



What is Hibernate Framework?

Hibernate ORM is an object–relational mapping tool for the Java programming language.

It provides a framework for mapping an object-oriented domain model to a relational database

What is JPA?

Jakarta Persistence API is a Jakarta EE (Earlier called J2EE) application programming interface specification that describes the management of relational data in enterprise Java applications

JPA
Java Persistence API

JSR

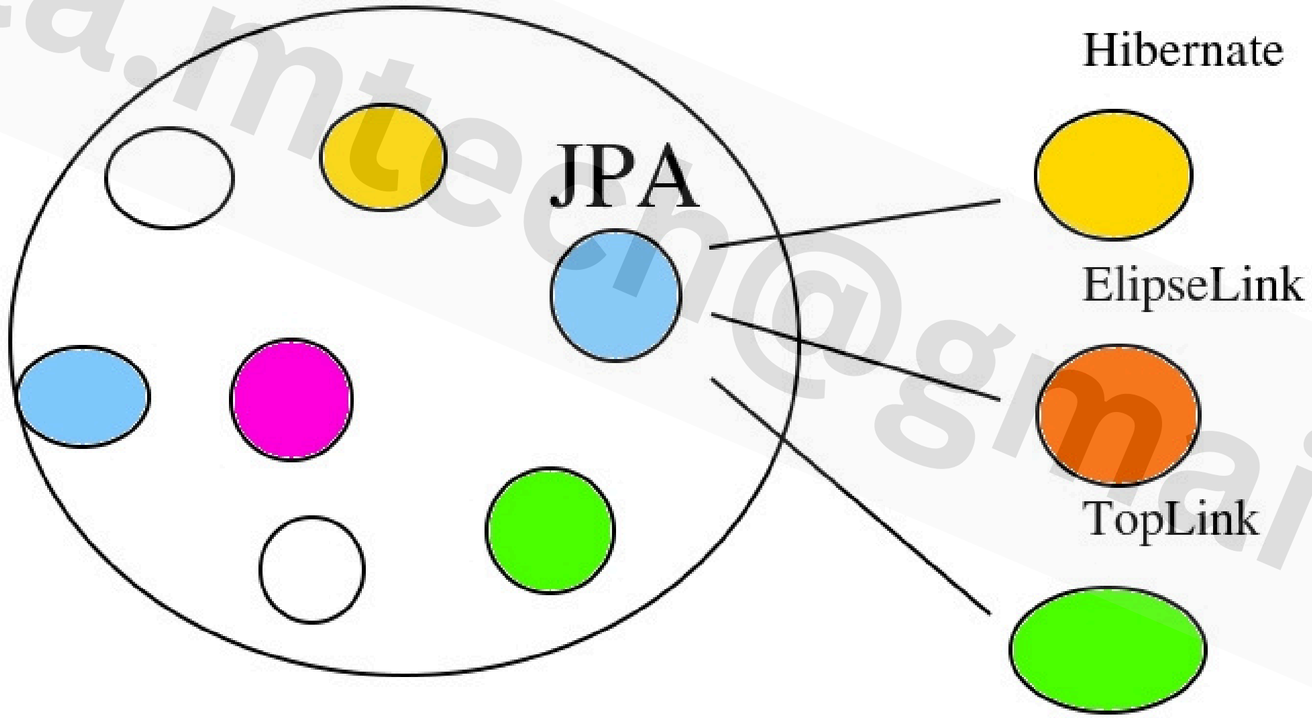
JPA vs Hibernate

The major difference between Hibernate and JPA is that Hibernate is a framework while JPA is API specifications

- **Hibernate is an implementation of JPA guidelines.**
- **It helps in mapping Java data types to SQL data types.**
- **It is the contributor of JPA.**

JPA vs Hibernate

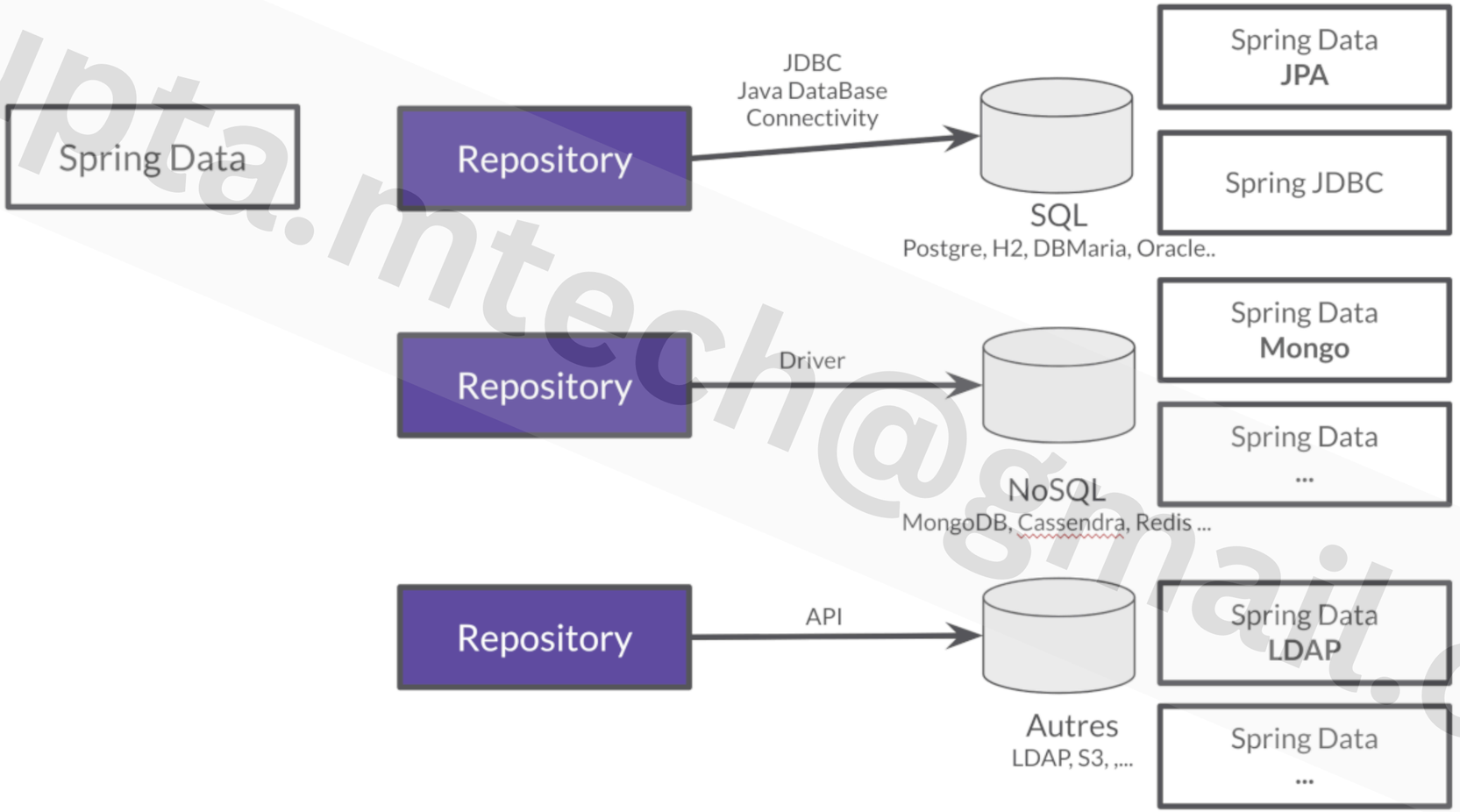
Advantage of JPA is that You can swap ORM in your project without changing the code



JPA vs Hibernate

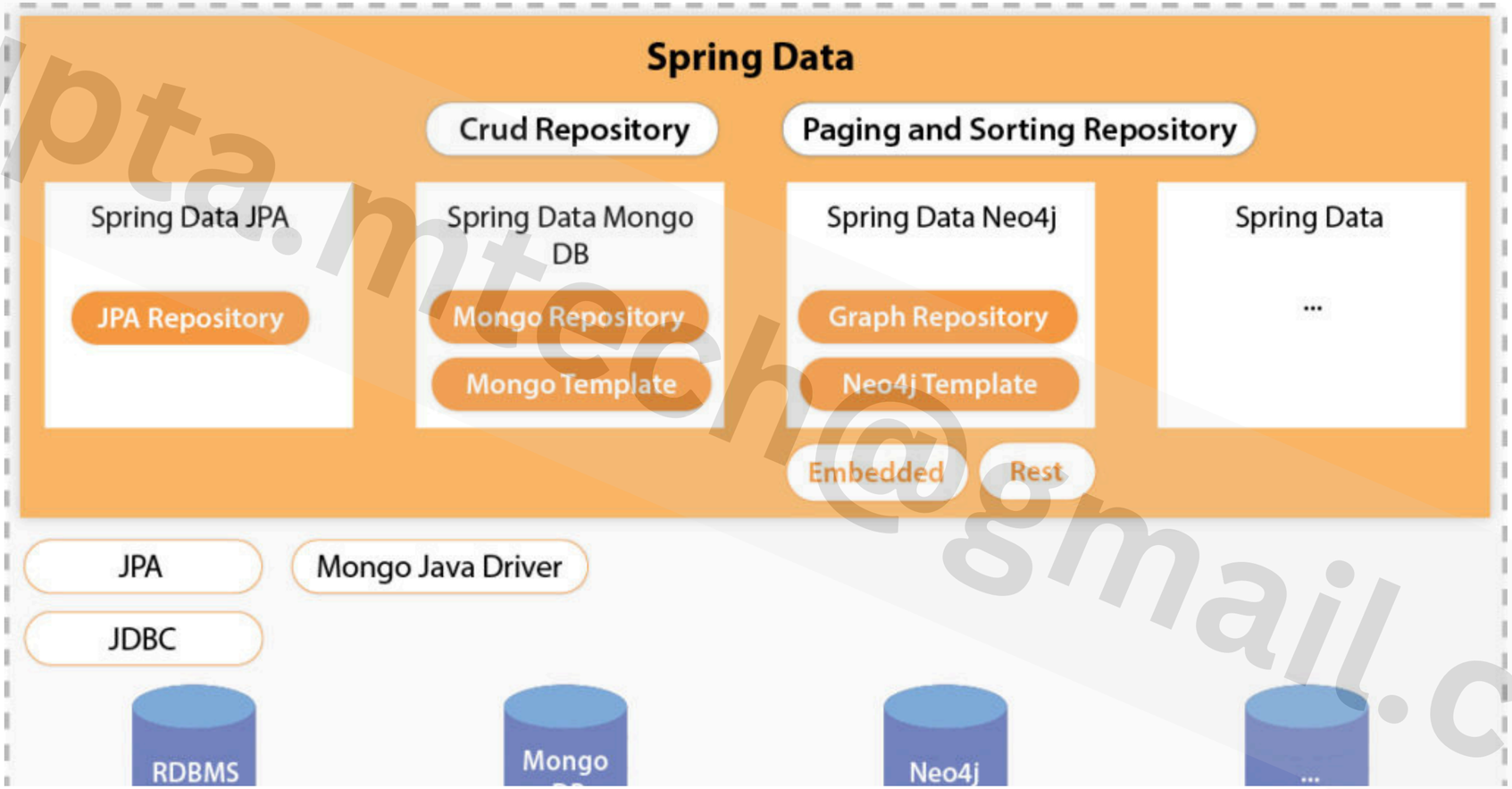
JPA	Hibernate
JPA is described in **javax.persistence** package.	Hibernate is described in **org.hibernate** package.
It describes the handling of relational data in Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system.
It is not an implementation. It is only a Java specification.	Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate.
It is a standard API that permits to perform database operations.	It is used in mapping Java data types with SQL data types and database tables.
As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations.	As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations.
To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager.	To create Session instances, it uses **SessionFactory** interface.
To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnects with the persistence condition.	To make, read, and remove actions for instances of mapped entity classes, it uses **Session** interface. It acts as a runtime interface between a Java application and Hibernate.

What is Spring Data?



The Spring-Data is an umbrella project having many sub-projects or modules to provide uniform abstractions and uniform utility methods for the Data Access Layer in an application and support a wide range of databases and datastores.

What is Spring Data?



By using Spring data We dont have to write Dao layer, we just need to declare Dao layer and it provide uniform interface to interact with polyglot of databases

JPA ?

What is JPA?

JSR 220, JSR-317 ,Java Persistence 2.0
Specification implemented by : Hibernate , eclipseLink, topLink etc JPA abstraction above JDBC
javax.persistence package

Component of JPA?

ORM

Entity manager API CRUD

operations JPQL

Transactions and locking mechanisms when accessing data concurrently provided by:

- Java Transaction API (JTA)
- Resource-local (non-JTA)

Callbacks and listeners to hook business logic into the life cycle of a persistent

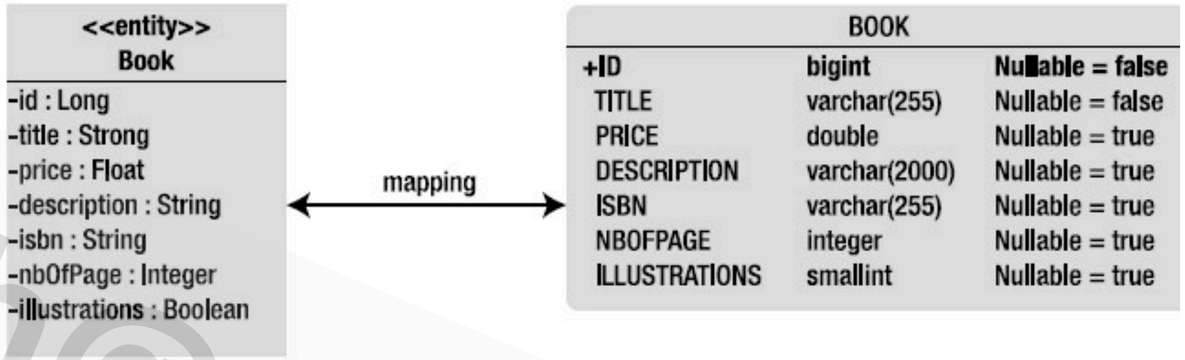
Hay Coming from hibernate World!

JPA	Hibernate
Entity Classes	Persistent Classes
EntityManagerFactory	SessionFactory
EntityManager	Session
Persistence	Configuration
EntityTransaction	Transaction
Query	Query
Persistence Unit	Hibernate Config

Entity

Entity class must be:

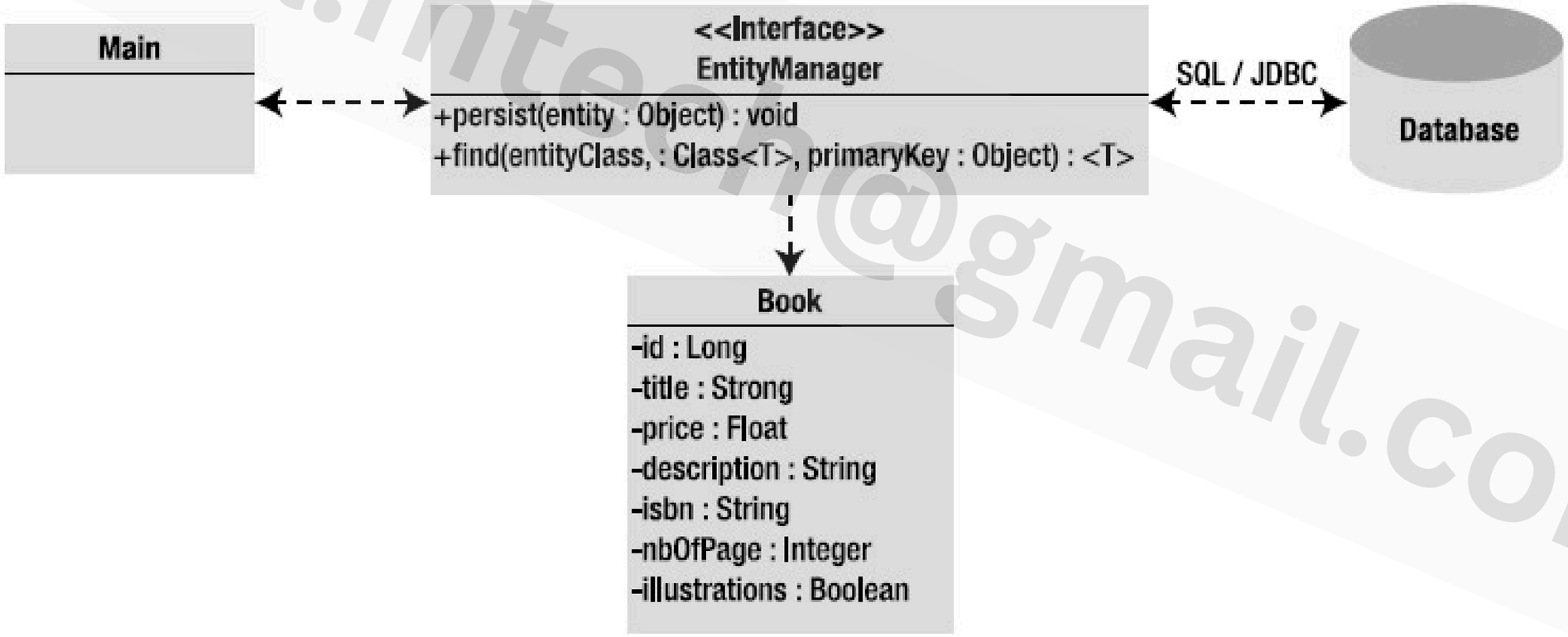
1. Annotated with @javax.persistence.Entity
2. @javax.persistence.Id annotation must be used to denote primary key
3. Must have a no-arg constructor that has to be public or protected.
4. Must be a top-level class.
5. Entity class must not be final.
6. No methods or persistent instance variables of the entity class may be final
7. May implements Serializable interface




```
@Entity
public class Book {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

EntityManager interacts with Entity

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter02PU");
EntityManager em = emf.createEntityManager();
em.persist(book);
```



EntityManager Methods

Method	Description
`void persist(Object entity)`	Makes an instance managed and persistent
`<T> T find(Class<T> entityClass, Object primaryKey)`	Searches for an entity of the specified class and primary key
`<T> T getReference(Class<T> entityClass, Object primaryKey)`	Gets an instance, whose state may be lazily fetched
`void remove(Object entity)`	Removes the entity instance from the persistence context and from the underlying database
`<T> T merge(T entity)`	Merges the state of the given entity into the current persistence context
`void refresh(Object entity)`	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
`void flush()`	Synchronizes the persistence context to the underlying database
`void clear()`	Clears the persistence context, causing all managed entities to become detached
`void detach(Object entity)`	Removes the given entity from the persistence context, causing a managed entity to become detached
`boolean contains(Object entity)`	Checks whether the instance is a managed entity instance belonging to the current persistence

find() vs getReference()

Listing 4-10. Finding a Customer by ID

```
Customer customer = em.find(Customer.class, 1234L)
if (customer!= null) {
    // Process the object
}
```

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

Listing 4-11. Finding a Customer by Reference

```
try {
    Customer customer = em.getReference(Customer.class, 1234L)
    // Process the object
} catch(EntityNotFoundException ex) {
    // Entity not found
```

em.getReference()

=>Takes the same parameters, but it retrieves a reference to an entity (via its primary key) and not its data.
=>It is intended for situations where a managed entity instance is needed, but no data, other than potentially the entity's primary key, being accessed.
=>With getReference(), the state data is fetched lazily,which means that, if you don't access state before the entity is detached, the data might not be there.
=> If the entity is not found, an EntityNotFoundException is thrown

remove() method

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);

tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

tx.begin();
em.remove(customer);
tx.commit();
```

Listing 4-13. The Customer Entity Dealing with Orphan Address Removal

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY, orphanRemoval=true)
    private Address address;

    // Constructors, getters, setters
}
```

persist(), flush(), refresh()

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

```
tx.begin();
em.persist(customer);
em.flush();
em.persist(address);
tx.commit();
```

Forcing persistenc e to flush the data, to synch with DB

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), "Antony");

customer.setFirstName("William");

em.refresh(customer);
assertEquals(customer.getFirstName(), "Antony");
```

The **refresh() method is used for data synchronization in the opposite direction of the flush, meaning it** overwrites the current state of a managed entity with data as it is present in the database.
A typical case
is where the **EntityManager.refresh() method is used to undo changes that have been done to the entity** in memory only. The test class snippet in Listing 4-14 finds a **Customer by ID, changes**

Clear() and Detach()

The clear() method is straightforward: it empties the persistence context, causing all managed entities to become detached.

The detach(Object entity) method removes the given entity from the persistence context. Changes made to the entity will not be sync to the database

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

em.detach(customer);
```

Merging an Entity

A detached entity is no longer associated with a persistence context. If you want to manage it, you need to merge it.

Let's take the example of an entity that needs to be displayed in a JSF page. The entity is first loaded from the database in the persistent layer (it is managed), it is returned from an invocation of a local EJB (it is detached because the transaction context ends), the presentation layer displays it (it is still detached), and then it returns to be updated to the database.

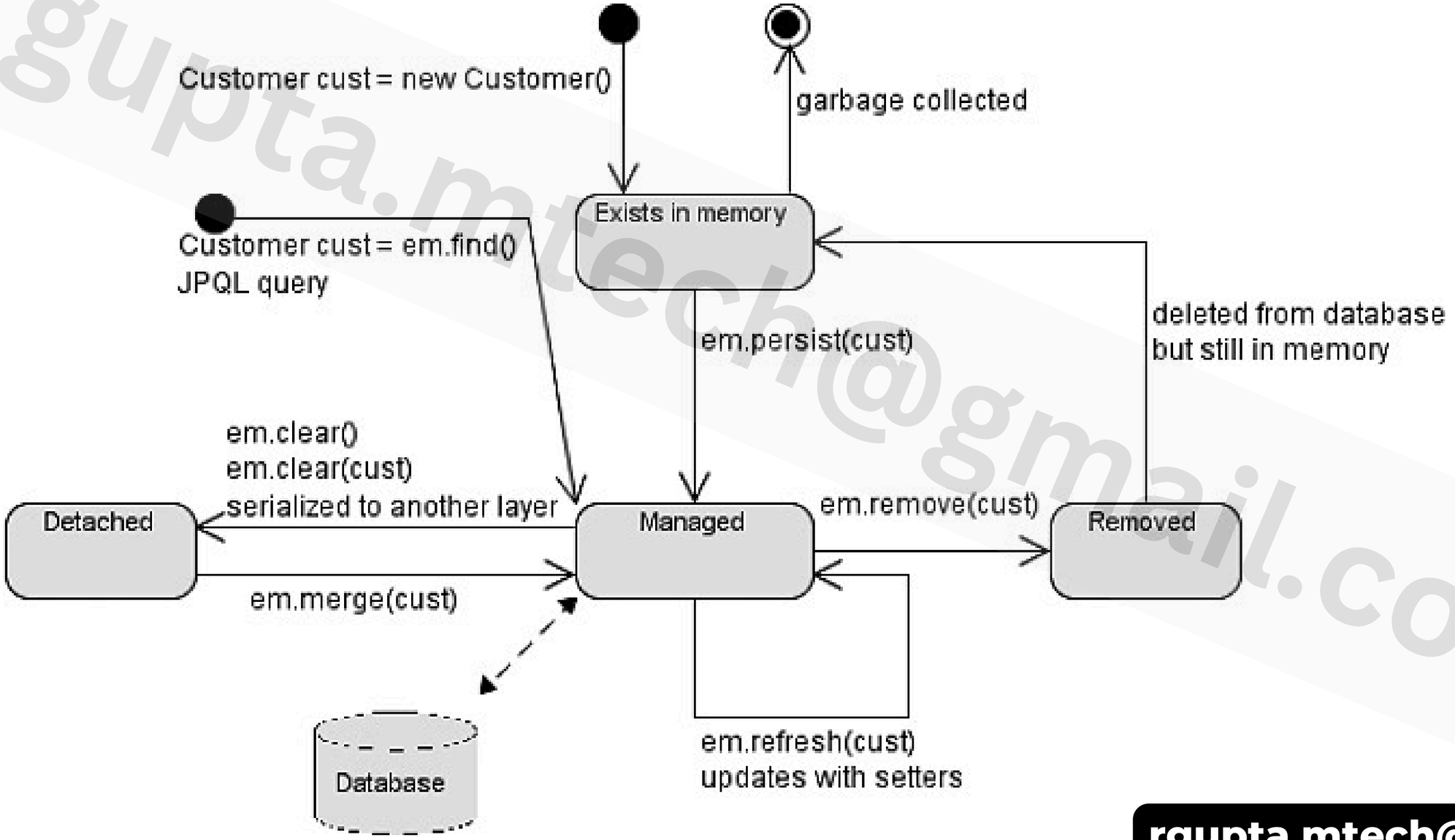
However, at that moment, the entity is detached and needs to be attached again, or merged, to synchronize its state with the database.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);
tx.commit();

em.clear();

// Sets a new value to a detached entity
customer.setFirstName("William");

tx.begin();
em.merge(customer);
tx.commit();
```

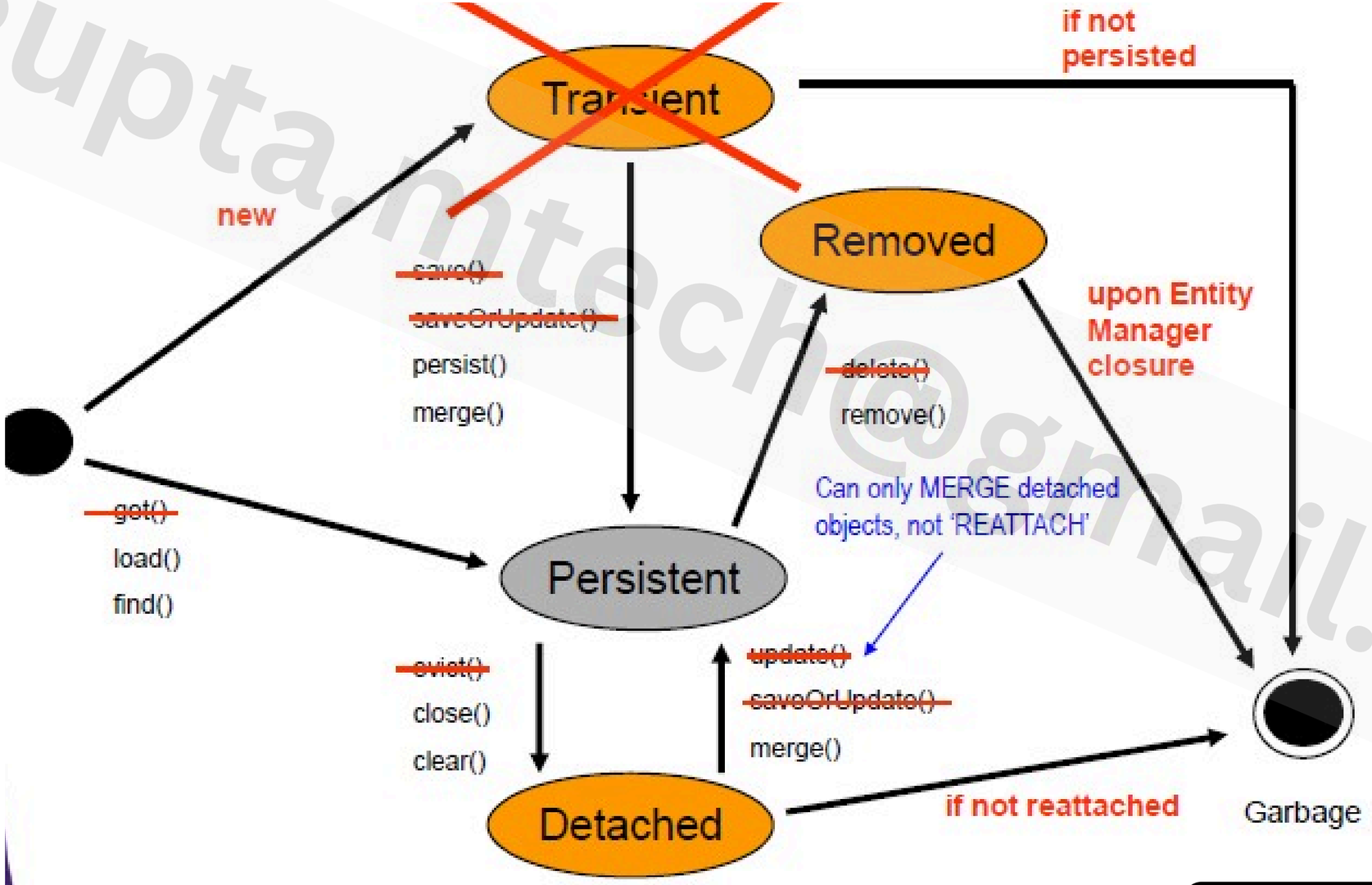
Updating an Entity

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");

tx.begin();
em.persist(customer);

customer.setFirstName("Williman");

tx.commit();
```

Entity Life Cycle



Entity Life Cycle



persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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_2_0.xsd"
             version="2.0">

  <persistence-unit name="chapter02PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.apress.javaee6.chapter02.Book</class>
    <properties>
      <property name="eclipselink.target-database" value="DERBY"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="INFO"/>
      <property name="javax.persistence.jdbc.driver" ↪
                value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.url" ↪
         value="jdbc:derby://localhost:1527/chapter02DB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="APP"/>
      <property name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
  </persistence-unit>
</persistence>
```

Inserting Record

```
// Creates an instance of book
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy book");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
book.setIllustrations(false);

// Gets an entity manager and a transaction
EntityManagerFactory emf = ↪
        Persistence.createEntityManagerFactory("chapter02PU");
EntityManager em = emf.createEntityManager();

// Persists the book to the database
EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(book);
tx.commit();

em.close();
emf.close();
}
```

JPQL

Under the hood, JPQL uses the mechanism of mapping to transform a JPQL query into language comprehensible by an SQL database.
The query is executed on the underlying database with SQL and JDBC calls, and then entity instances have their attributes set and are returned to the

```
SELECT b
FROM Book b
```

simplest JPQL query selects all the instances of a single entity

```
SELECT b
FROM Book b
WHERE b.title = 'H2G2'
```

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Vincent' AND c.address.country = 'France'
```

```
SELECT c
FROM Customer c
```

A simple SELECT returns an entity. For example, if a Customer entity has an alias called c, SELECT c will return an entity or a list of entities

```
SELECT c.firstName, c.lastName
FROM Customer c
```

```
SELECT c.address.country.code
FROM Customer c
```

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Vincent'
```

Binding Parameters

```
SELECT c
FROM Customer c
WHERE c.firstName = ?1 AND c.address.country = ?2
```

Positional parameters **are designated by the question mark (?) followed by an integer (e.g., ?1)**

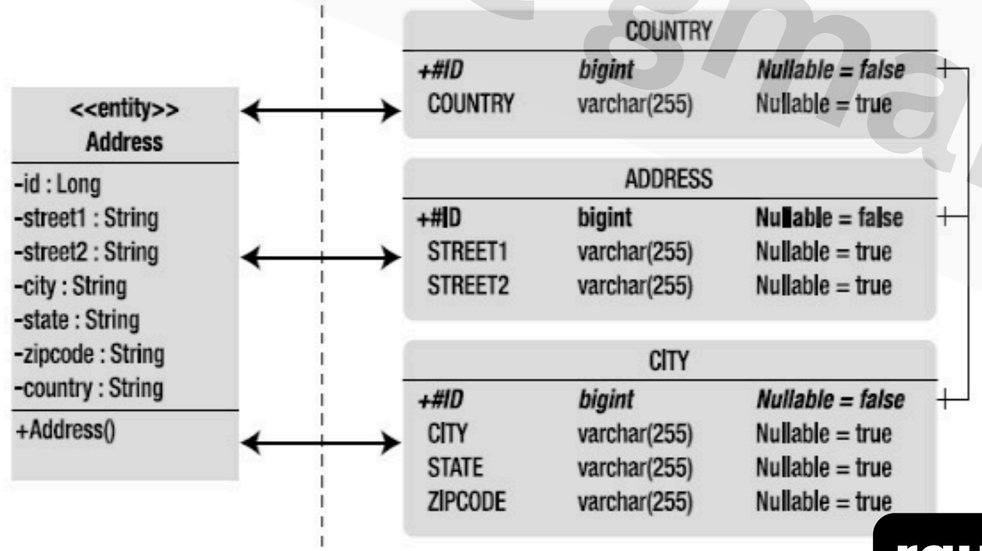
```
SELECT c
FROM Customer c
WHERE c.firstName = :fname AND c.address.country = :country
```

Named parameters **can also be used and are designated by a String identifier that is prefixed by the colon (:) symbol. When the query is executed, the parameter names that should be replaced need to be specified**

@SecondaryTable

- Data need to spread across multiple tables called
- secondary tables
- Use annotation @SecondaryTable to associate a secondary
- SecondaryTables (with an “s”) for les
- Entity Address Mapped to 3 tables

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "city"),
    @SecondaryTable(name = "country")
})
public class Address {

    @Id
    private Long id;
    private String street1;
    private String street2;
    @Column(table = "city")
    private String city;
    @Column(table = "city")
    private String state;
    @Column(table = "city")
    private String zipcode;
    @Column(table = "country")
    private String country;

    // Constructors, getters, setters
}
```



@Id and @GeneratedValue

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}
```

SEQUENCE
IDENTITY
TABLE
AUTO

Composite Primary Keys

The Primary Key Class Is Annotated with `@Embeddable`

```
@Embeddable
public class NewsId {

    private String title;
    private String language;

    // Constructors, getters, setters, equals, and hashcode
}
```

The Entity Embeds the Primary Key Class with `@EmbeddedId`

```
@Entity
public class News {

    @EmbeddedId
    private NewsId id;
    private String content;

    // Constructors, getters, setters
}
```

```
NewsId pk = new NewsId("Richard Wright has died", "EN")
News news = em.find(News.class, pk);
```

@Basic

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}

@Entity
public class Track {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float duration;
    @Basic(fetch = FetchType.LAZY)
    @Lob
    private byte[] wav;
    private String description;

    // Constructors, getters, setters
}
```

@Temporal

```java
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
```

```sql
create table CUSTOMER (
    ID BIGINT not null,
    FIRSTNAME VARCHAR(255),
    LASTNAME VARCHAR(255),
    EMAIL VARCHAR(255),
    PHONENUMBER VARCHAR(255),
    DATEOFBIRTH DATE,
    CREATIONDATE TIMESTAMP,
    primary key (ID)
);
```


rgupta.mtech@gmail.com

@Transient

```
@Entity
public class Customer {

    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
```

Don’t get stored in DB

rgupta.mtech@gmail.com

@Enumerated

```
public enum CreditCardType {
    VISA,
    MASTER_CARD,
    AMERICAN_EXPRESS
}
```

```
@Entity
@Table(name = "credit_card")
public class CreditCard {

    @Id
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    private CreditCardType creditCardType;

    // Constructors, getters, setters
}
```

Mapping an Enumerated Type with ***String***

```
@Entity
@Table(name = "credit_card")
public class CreditCard {

    @Id
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    @Enumerated(EnumType.STRING)
    private CreditCardType creditCardType;

    // Constructors, getters, setters
}
```

Collection of Basic Types

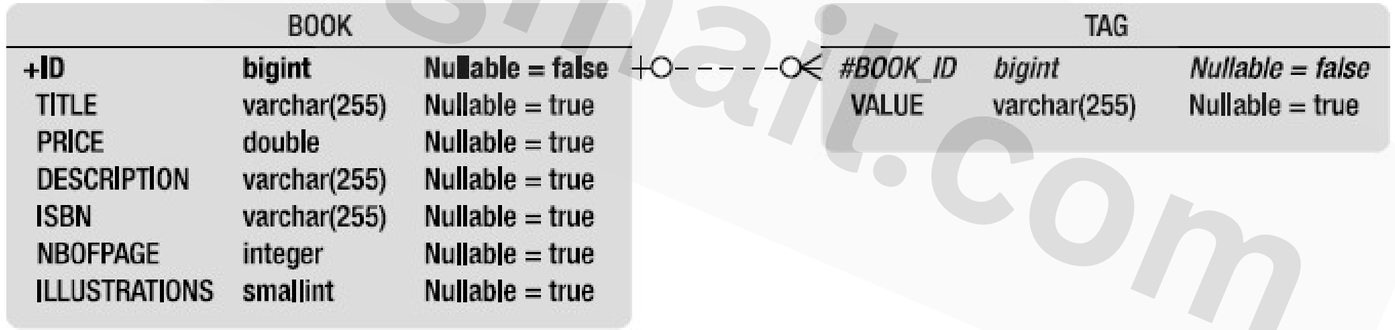
@ElementCollection annotation is used to indicate that an attribute of type java.util.Collection contains a collection of instances of basic types (i.e., nonentities)

Attribute can be of the following types:

- **java.util.Collection: Generic root interface in the collection hierarchy.**
- **java.util.Set: Collection that prevents the insertion of duplicate elements.**
- **java.util.List: Collection used when the elements need to be retrieved in some user-defined order.**

```java
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag")
    @Column(name = "Value")
    private List<String> tags = new ArrayList<String>();

    // Constructors, getters, setters
}
```

Book Entity with collection of Strings



Embeddables

```java
@Embeddable
public class Address {

    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Embedded
    private Address address;

    // Constructors, getters, setters
}
```

Listing 3-35. Structure of the CUSTOMER Table with All th

```
create table CUSTOMER (
  ID BIGINT not null,
  LASTNAME VARCHAR(255),
  PHONENUMBER VARCHAR(255),
  EMAIL VARCHAR(255),
  FIRSTNAME VARCHAR(255),
  STREET2 VARCHAR(255),
  STREET1 VARCHAR(255),
  ZIPCODE VARCHAR(255),
  STATE VARCHAR(255),
  COUNTRY VARCHAR(255),
  CITY VARCHAR(255),
  primary key (ID)
);
```

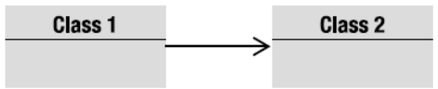
Relationship Mapping

OO relations

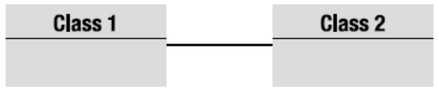
Association between the Objects

IS-A, HAS-A, USE-A

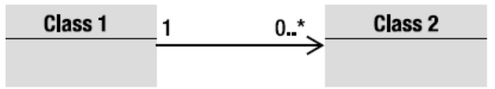
An association has a direction:



Unidirectional



Bidirectional

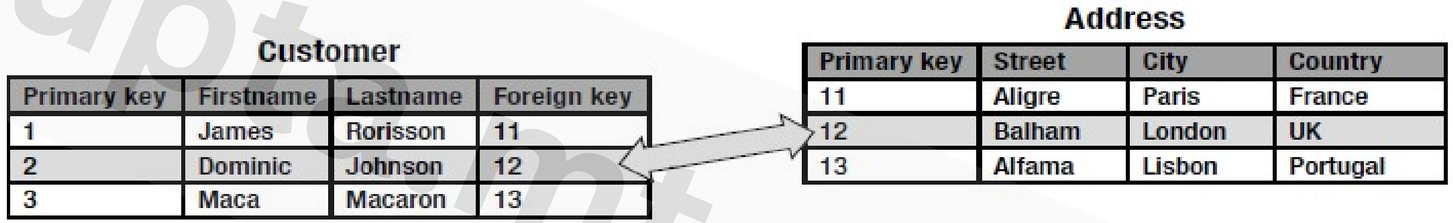


Multiplicity on class associations

Entity Relationships

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Relationships in Relational Databases



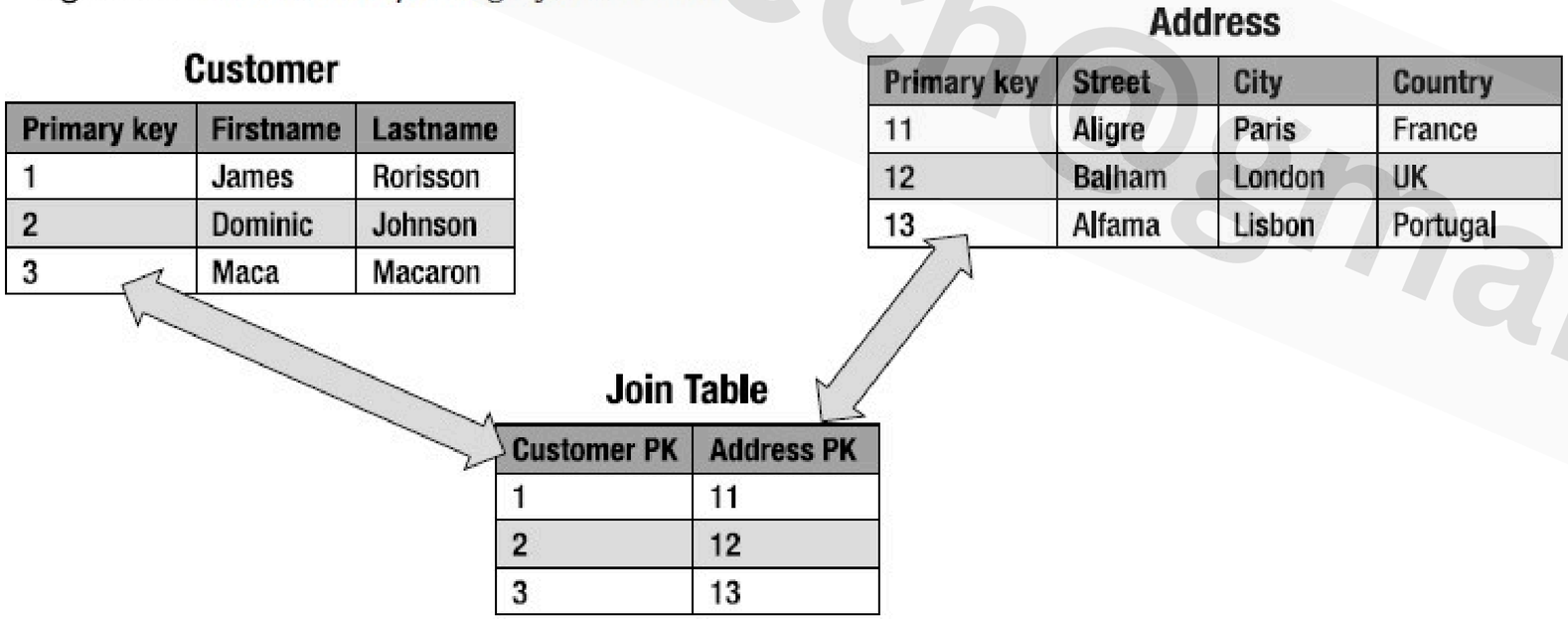
Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Figure 3-9. A relationship using a join column



Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

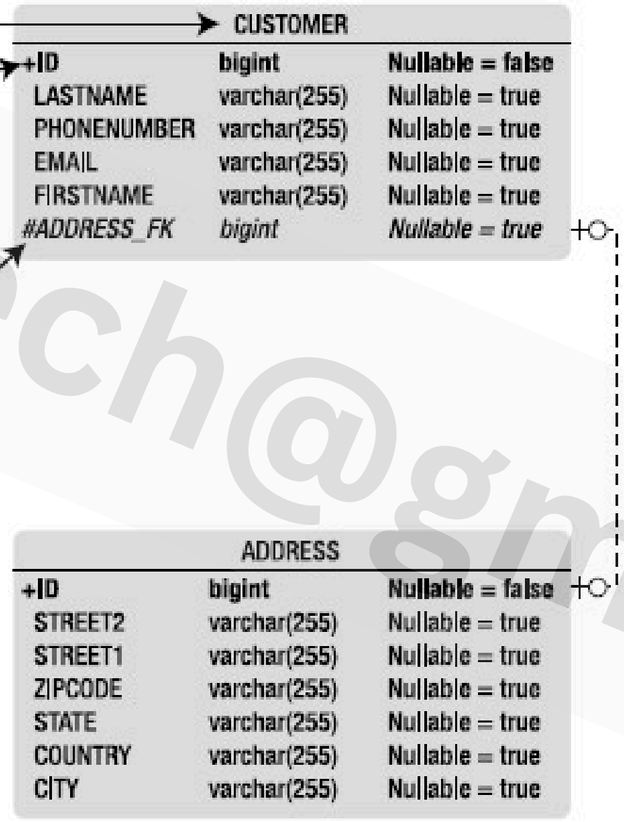
Join Table

Customer PK	Address PK
1	11
2	12
3	13

Figure 3-10. A relationship using a join table

One to one Bidirectional

```java
@Entity
public class Customer

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```

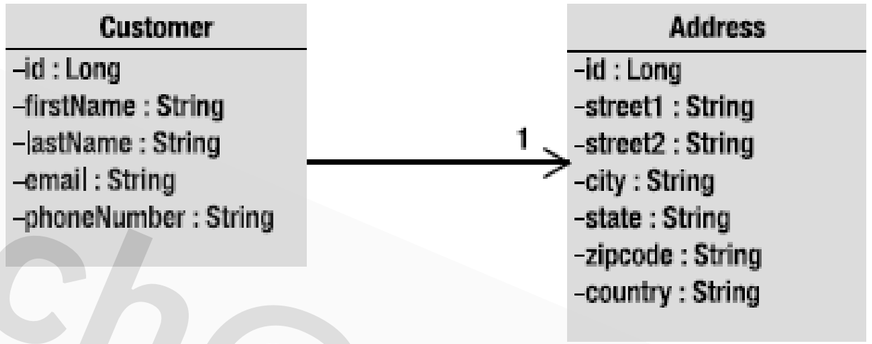


One to one Unidirectional

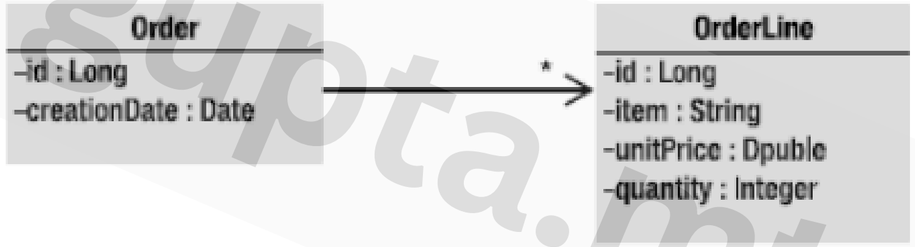
```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    private Address address;

    // Constructors, getters, setters
}
```

Listing 3-39. An Address Entity

```
@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;

    // Constructors, getters, setters
}
```



One to many unidirectional




```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}
```

Listing 3-45. An OrderLine

```
@Entity
@Table(name = "order_line")
public class OrderLine {

    @Id @GeneratedValue
    private Long id;
    private String item;
    private Double unitPrice;
    private Integer quantity;

    // Constructors, getters, setters
}
```

One to many unidirectional

- Previous annotations leads to mapping that relies on the configuration-by exception paradigm.
- By default relationships use a join table to keep the relationship information, with two foreign key columns.
- One foreign key column refers to the table ORDER and has the same type as its primary key, and the other refers to ORDER_LINE. The name of this joined table is the name of both entities, separated by the _ symbol.

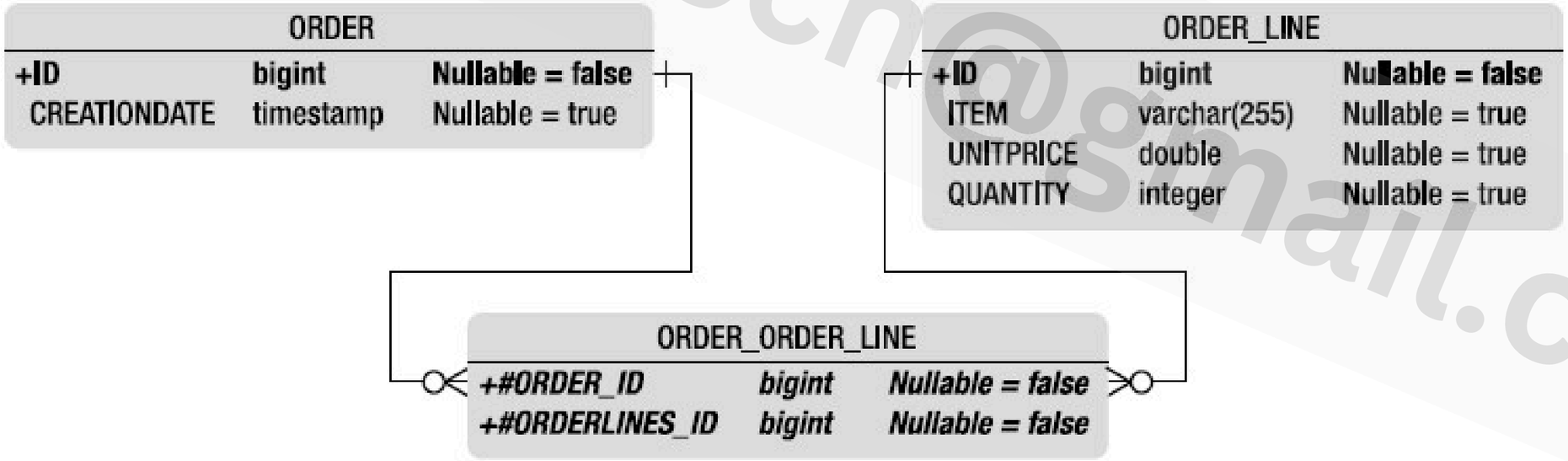


Figure 3-17. Join table between ORDER and ORDER_LINE

One to many unidirectional

Controlling join tables name, fields etc

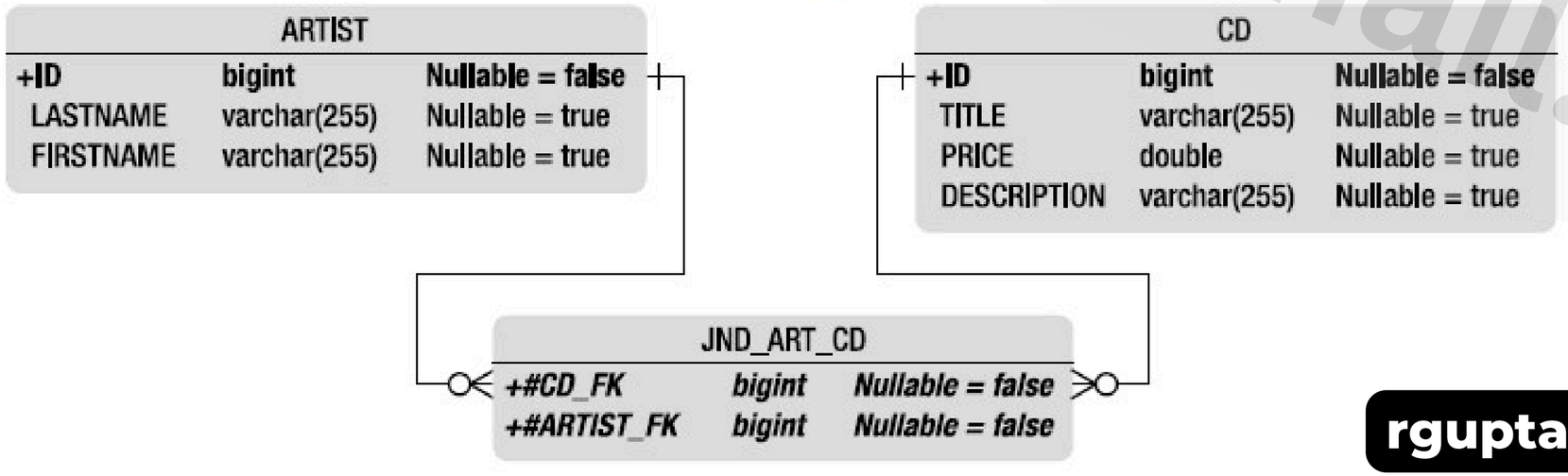
```java
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany
    @JoinTable(name = "jnd_ord_line",
        joinColumns = @JoinColumn(name = "order_fk"),
        inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
    private List<OrderLine> orderLines;

    // Constructors, getters, setters
}
```

```sql
create table JND_ORD_LINE (
    ORDER_FK BIGINT not null,
    ORDER_LINE_FK BIGINT not null,
    primary key (ORDER_FK, ORDER_LINE_FK),
    foreign key (ORDER_LINE_FK) references ORDER_LINE(ID),
    foreign key (ORDER_FK) references ORDER(ID)
);
```

Many to Many Bi-Directional

```java
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ↪
        joinColumns = @JoinColumn(name = "artist_fk"), ↪
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

    // Constructors, getters, setters
}
```



Fetching Relationships

Consider four entities related by eager

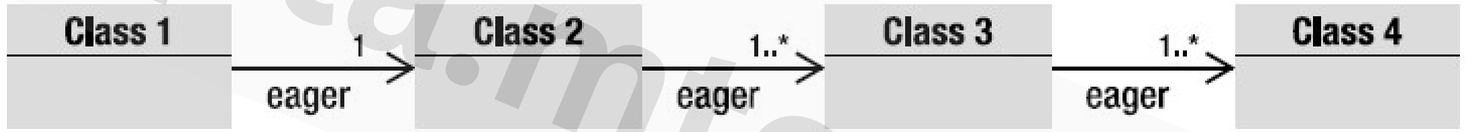


Figure 3-20. Four entities with eager relationships

```
class1.getClass2().getClass3().getClass4()
```

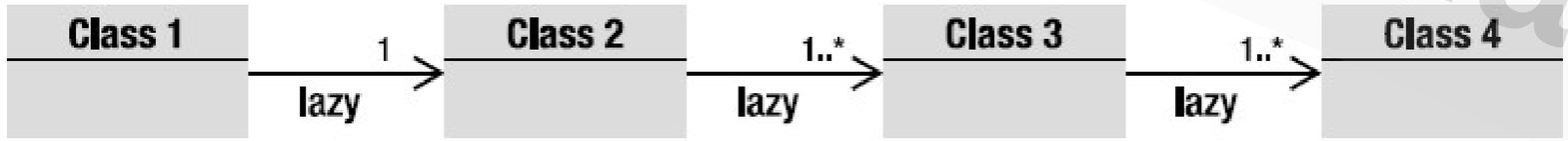


Figure 3-21. Four entities with lazy relationships

Fetching Relationships

Listing 3-52. An Order with an Eager Relationship to `OrderLine`

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)

      private Date creationDate;
      @OneToMany(fetch = FetchType.EAGER)
      private List<OrderLine> orderLines;

      // Constructors, getters, setters
  }
```

Table 3-2. Default Fetching Strategies

Annotation	Default Fetching Strategy
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

@OrderBy

Dynamic ordering can be done with the @OrderBy annotation. "Dynamically" means that the ordering of the elements of a collection is made when the association is retrieved

```
@Entity
public class Comment {

    @Id @GeneratedValue
    private Long id;
    private String nickname;
    private String content;
    private Integer note;
    @Column(name = "posted_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date postedDate;

    // Constructors, getters, setters
}
```

```
@Entity
public class News {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String content;
    @OneToMany(fetch = FetchType.EAGER)
    @OrderBy("postedDate DESC")
    private List<Comment> comments;

    // Constructors, getters, setters
}
```

Inheritance Mapping

☐ JPA has three different strategies to choose from:-

- ***A single-table-per-class hierarchy strategy***
  - *The sum of the attributes of the entire* entity hierarchy is flattened down to a single table
  - This is the default strategy
- ***A joined-subclass strategy***
  - *In this approach, each entity in the hierarchy, concrete* or abstract, is mapped to its own dedicated table.
- ***A table-per-concrete-class strategy***
  - *This strategy maps each concrete entity* hierarchy to its own separate table

Inheritance Mapping

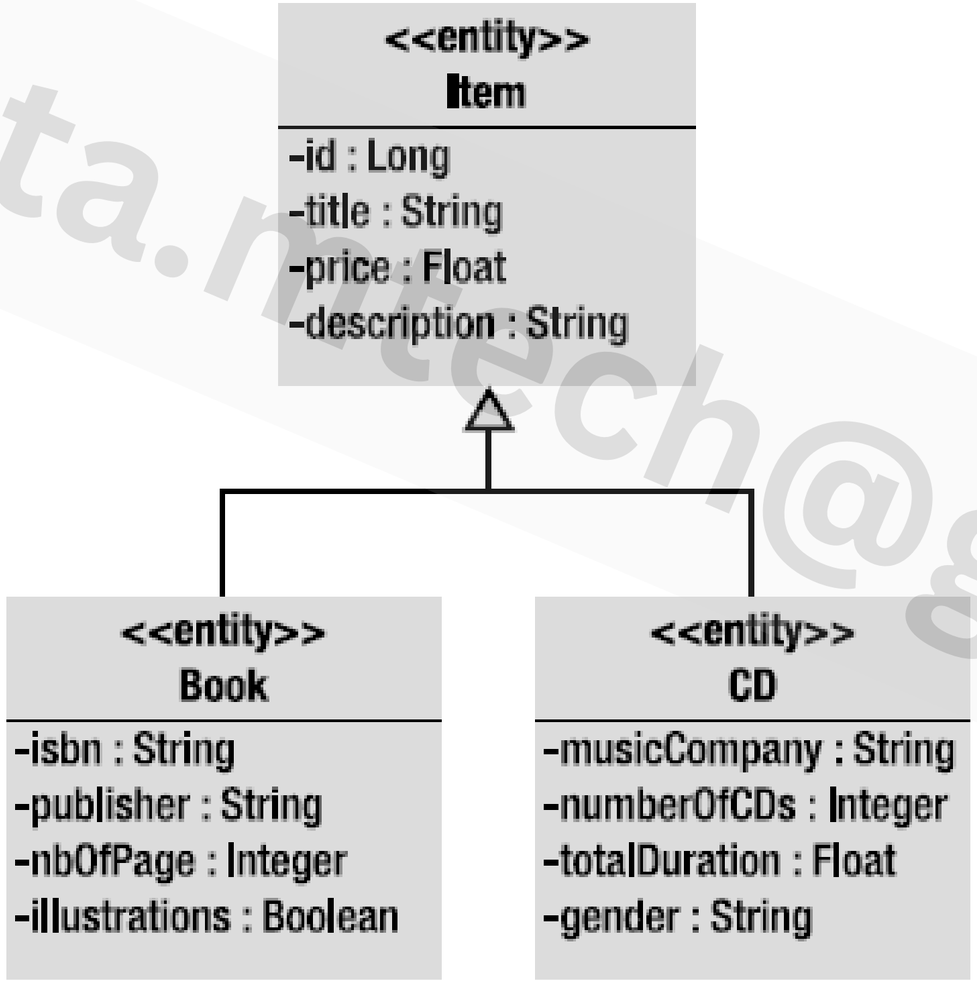


Figure 3-22. Inheritance hierarchy between CD, Book,

Single-Table-per-Class Hierarchy Strategy

```
@Entity
public class Item {

    @Id @GeneratedValue
    protected Long id;
    @Column(nullable = false)
    protected String title;
    @Column(nullable = false)
    protected Float price;
    protected String description;

    // Constructors, getters, setters
}

@Entity
public class Book extends Item {

    private String isbn;
    private String publisher;
    private Integer nbOfPage;
    private Boolean illustrations;

    // Constructors, getters, setters
}


@Entity
public class CD extends Item {

    private String musicCompany;
    private Integer numberOfCDs;
    private Float totalDuration;
    private String gender;

    // Constructors, getters, setters
}
```

ITEM		
+ID	bigint	Nullable = false
DTYPE	varchar(31)	Nullable = true
TITLE	varchar(255)	Nullable = false
PRICE	double	Nullable = false
DESCRIPTION	varchar(255)	Nullable = true
ILLUSTRATIONS	smallinit	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENDER	varchar(255)	Nullable = true

Figure 3-23. ITEM table structure

ID	DTYPE	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	Item	Pen	2.10	Beautiful black pen			...
2	CD	Soul Train	23.50	Fantastic jazz album	Prestige		...
3	CD	Zoot Allures	18	One of the best of Zappa	Warner		...
4	Book	The robots of dawn	22.30	Robots everywhere		0-554-156	...
5	Book	H2G2	17,50	Funny IT book ;o)		1-278-983	...

Figure 3-24. Fragment of the ITEM table filled with data

Single-Table-per-Class Hierarchy Strategy

- Discriminator column is called DTYPE by default, is of type String (mapped to a VARCHAR), and contains the name of the entity.
- If the defaults don't suit, the @DiscriminatorColumn annotation allows you to change the name and the data type.
- By default, the value of this column is the entity name to which it refers, although an entity may override this value using the @DiscriminatorValue annotation.

Listing 3-61. Item Redefines the Discriminator Column

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn (name="disc", ➥
                      discriminatorType = DiscriminatorType.CHAR)
@DiscriminatorValue("I")
public class Item {

    @Id @GeneratedValue
    protected Long id;
    protected String title;
    protected Float price;
    protected String description;

    // Constructors, getters, setters
}
```

```
@Entity
@DiscriminatorValue("B")
public class Book extends Item {

    private String isbn;
```

```
@Entity
@DiscriminatorValue("C")
public class CD extends Item {

    private String musicCompany;
    private Integer numberOfCDs;
    private Float totalDuration;
```

ID	DTYPE	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	I	Pen	2.10	Beautiful black pen			...
2	C	Soul Train	23.50	Fantastic jazz album	Prestige		...
3	C	Zoot Allures	18	One of the best of Zappa	Warner		...
4	B	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	B	H2G2	17.50	Funny IT book ;o)		1-278-983	...

Joined-Subclass Strategy

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {

    @Id @GeneratedValue
    protected Long id;
    protected String title;
    protected Float price;
    protected String description;

    // Constructors, getters, setters
}
```

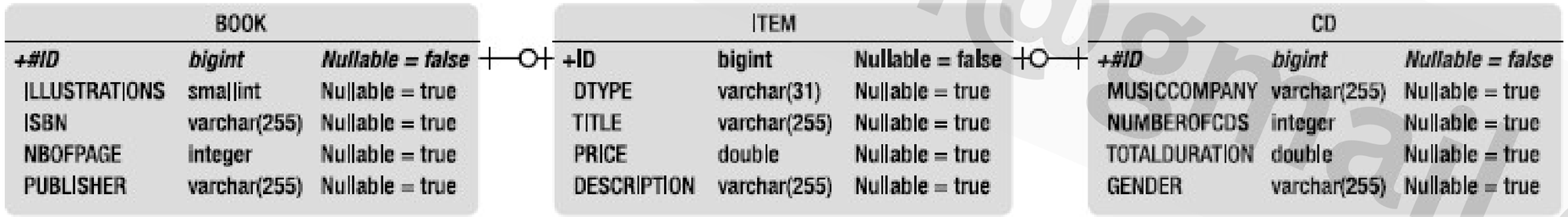


Table-per-Concrete-Class Strategy

```java
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Item {

    @Id @GeneratedValue
    protected Long id;
    protected String title;
    protected Float price;
    protected String description;

    // Constructors, getters, setters
}
```

BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true

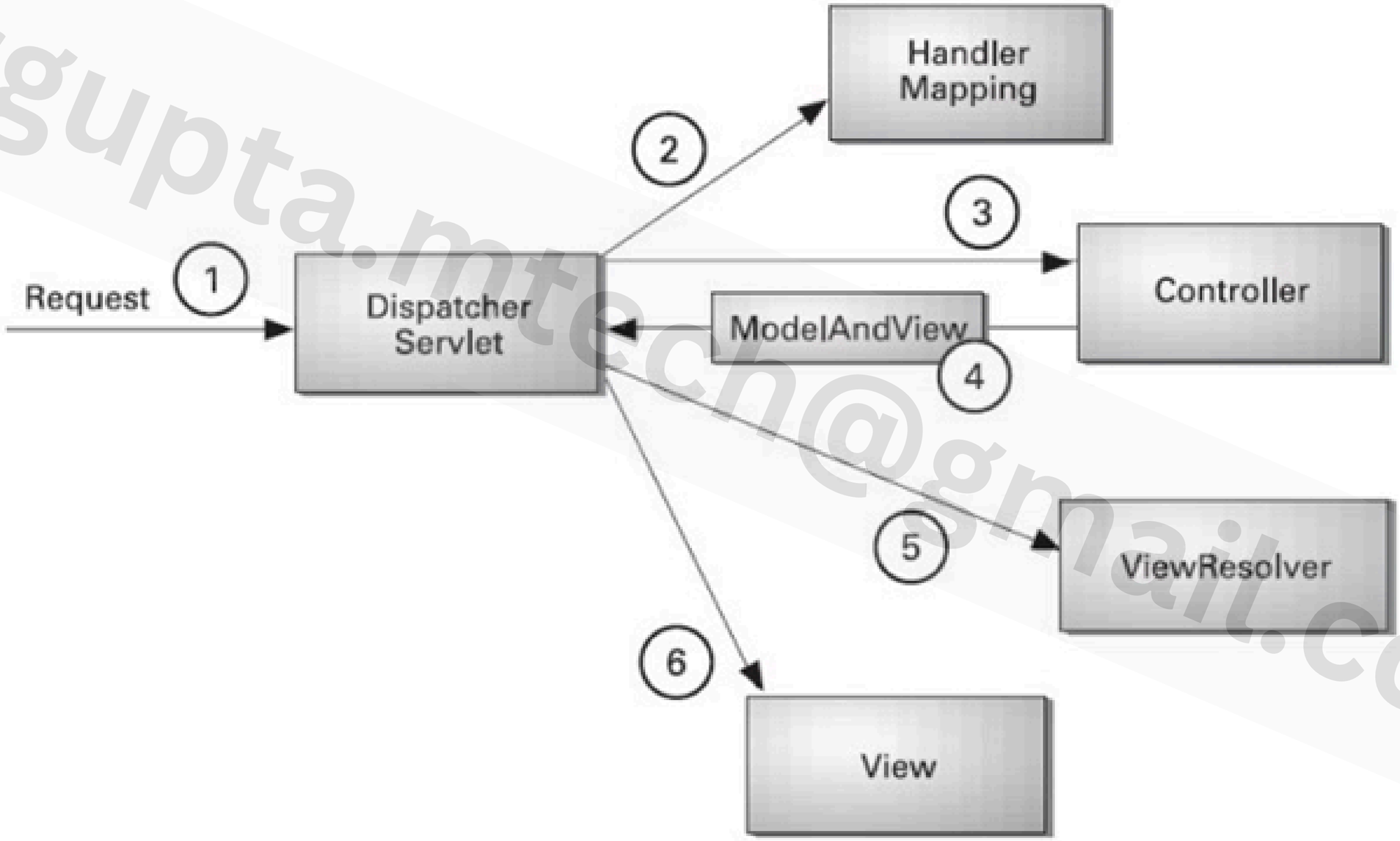
ITEM		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true

CD		
+ID	bigint	Nullable = false
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TITLE	varchar(255)	Nullable = true
TOTALDURATION	double	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
GENDER	varchar(255)	Nullable = true

Spring MVC

rgupta.mtech@gmail.com

Spring MVC architecture



Spring MVC Filter vs Interceptor