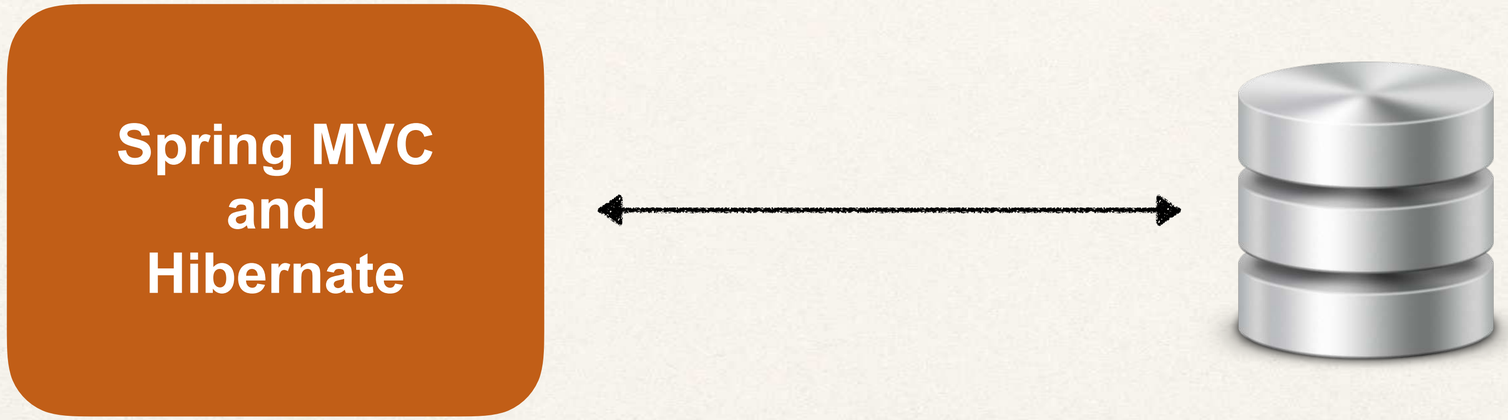
luv2code

# Create a Database Application

# Class Project

**Full working *Spring MVC and Hibernate* application that connects to a database**



# Customer Relationship Management - CRM

# Customer Relationship Management - CRM

- Set up Database Dev Environment
- List Customers
- Add a new Customer
- Update a Customer
- Delete a Customer

Step-By-Step

luv2code

# Setup Database Table

# Two Database Scripts

1. Folder: **sql-scripts**
   - **01-create-user.sql**
   - **02-customer-tracker.sql**

# About: 01-create-user.sql

1. Create a new MySQL user for our application
    - user id: **springstudent**
    - password: **springstudent**

# About: 02-customer-tracker.sql

1. Create a new database table: **customer**

2. Load table with sample data

# Setup Project - Test DB Connection

# Test DB Connection

1. Set up our Eclipse project

2. Add JDBC Driver for MySQL

3. Sanity test … make sure we can connect :-)

luv2code

# Setup Dev Environment - Part 2

# Configuration for Spring + Hibernate

Step-By-Step

1. Define database dataSource / connection pool
2. Setup Hibernate session factory
3. Setup Hibernate transaction manager
4. Enable configuration of transactional annotations

# Placement of Configurations

Add the following configurations in your Spring MVC configuration file

For our example
spring-mvc-crud-demo-servlet.xml

# Step 1: Define database dataSource / connection pool

```
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">

 <property name="driverClass" value="com.mysql.jdbc.Driver" />
 <property name="jdbcUrl"
                value="jdbc:mysql://localhost:3306/web_customer_tracker?useSSL=false" />

 <property name="user" value="springstudent" />
 <property name="password" value="springstudent" />

 <!-- these are connection pool properties for C3P0 -->
 <property name="minPoolSize" value="5" />
 <property name="maxPoolSize" value="20" />
 <property name="maxIdleTime" value="30000" />
</bean>
```

# Step 2: Setup Hibernate session factory

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

   <property name="dataSource" ref="myDataSource" />
   <property name="packagesToScan" value="com.luv2code.springdemo.entity" />

   <property name="hibernateProperties">
     <props>
       <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
       <prop key="hibernate.show_sql">true</prop>
     </props>
   </property>

</bean>
```

# Step 3: Setup Hibernate transaction manager

```
<bean id="myTransactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">

    <property name="sessionFactory" ref="sessionFactory"/>

</bean>
```

# Step 4: Enable configuration of transactional annotations

```
<tx:annotation-driven transaction-manager="myTransactionManager" />
```

# Configuration for Spring + Hibernate

Step-By-Step

1. Define database dataSource / connection pool
2. Setup Hibernate session factory
3. Setup Hibernate transaction manager
4. Enable configuration of transactional annotations

luv2code

# Setup Dev Environment

# Setup Dev Environment

Step-By-Step

1. Copy starter config files
    1. web.xml and spring config

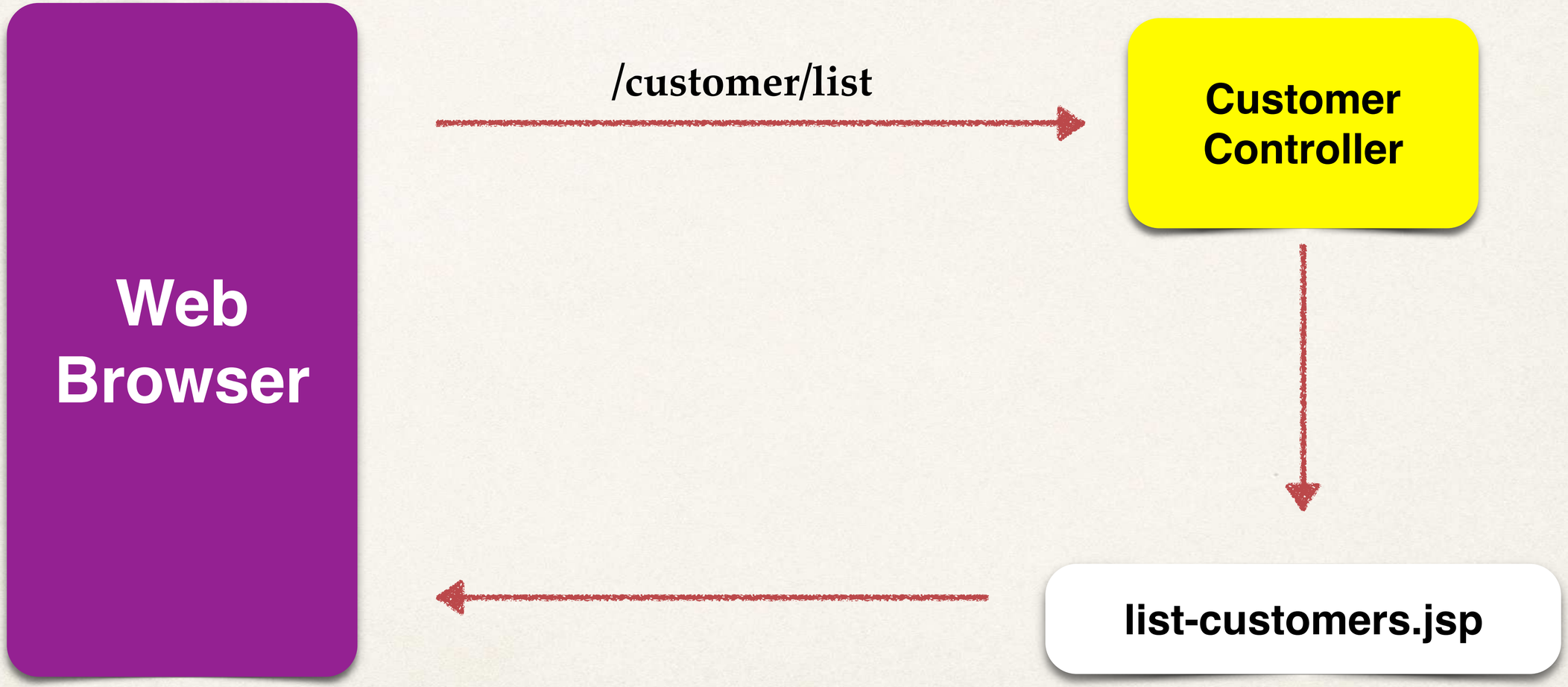
2. Copy over basic libs
    1. Jakarta Commons and JSTL

3. Copy latest Spring JAR files

4. Copy latest Hibernate JAR files

# Test Basic Spring MVC Controller

# Customer Controller

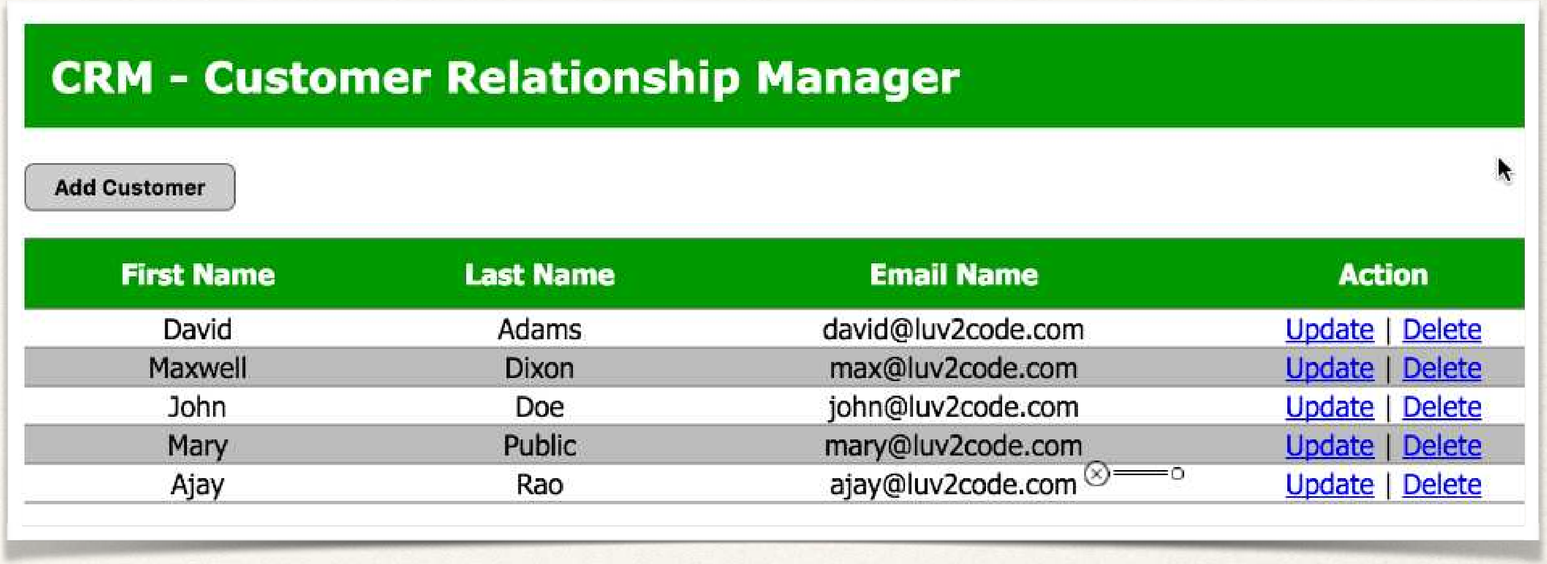


luv2code

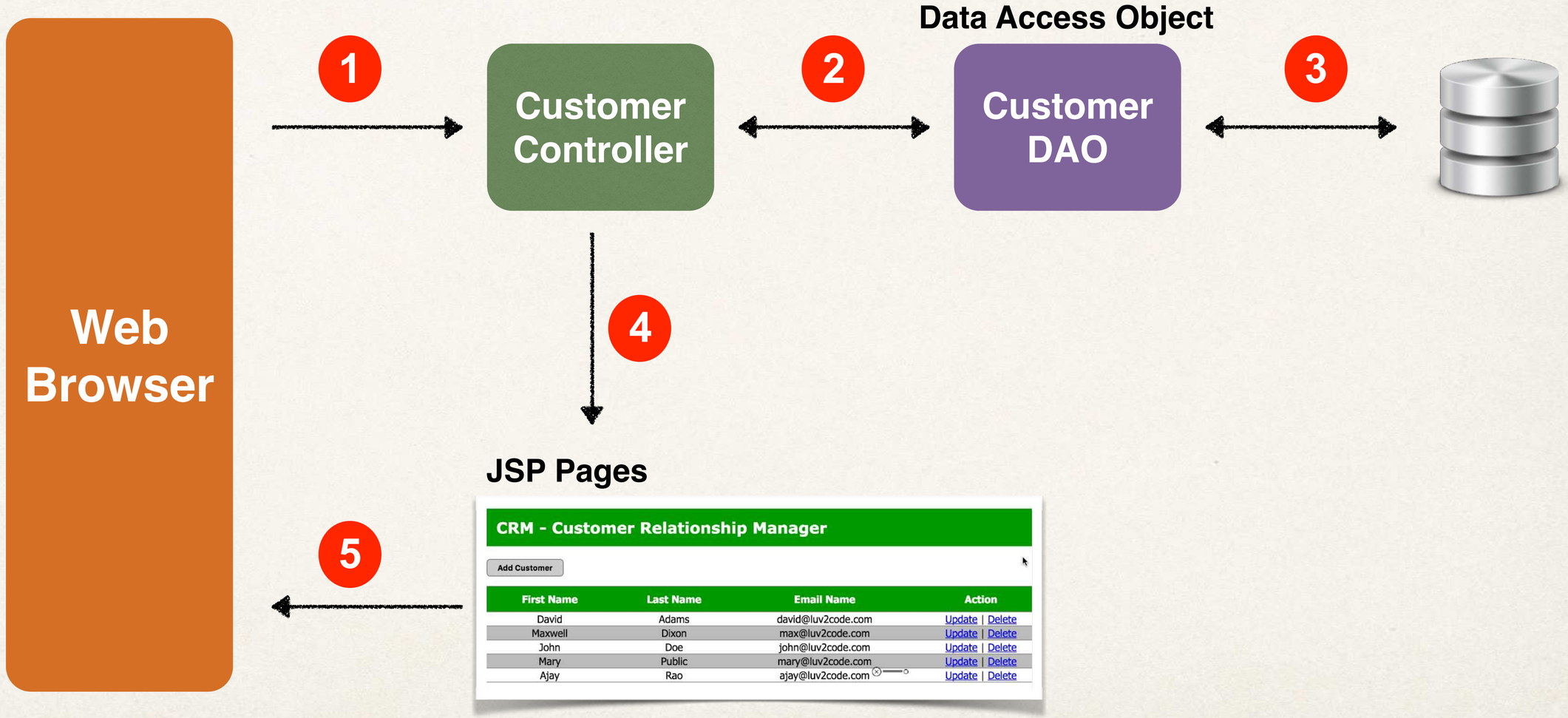
# Sample App Architecture

# Sample App Features - Refresher

- List Customers
- Add a new Customer
- Update a Customer
- Delete a Customer

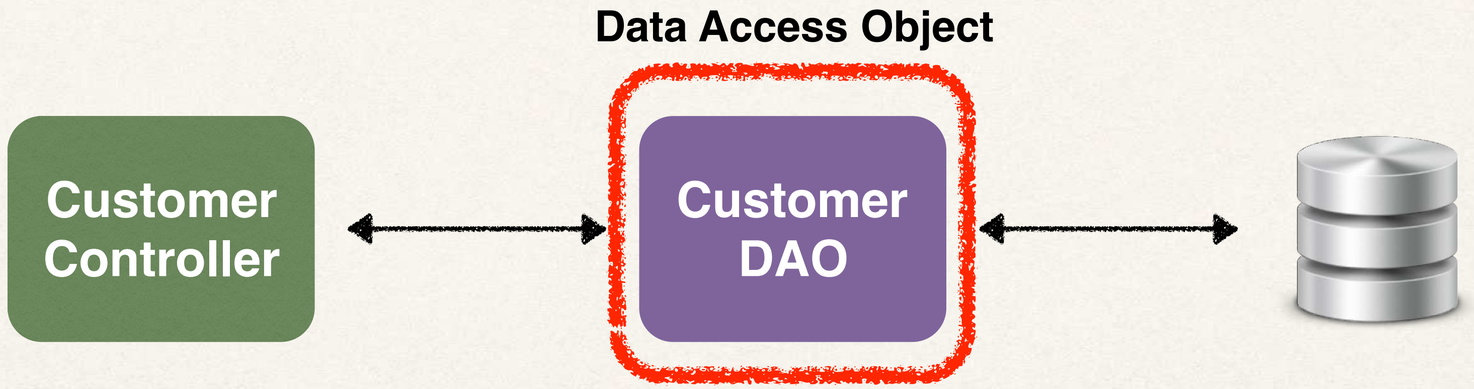


# Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

# Customer Data Access Object

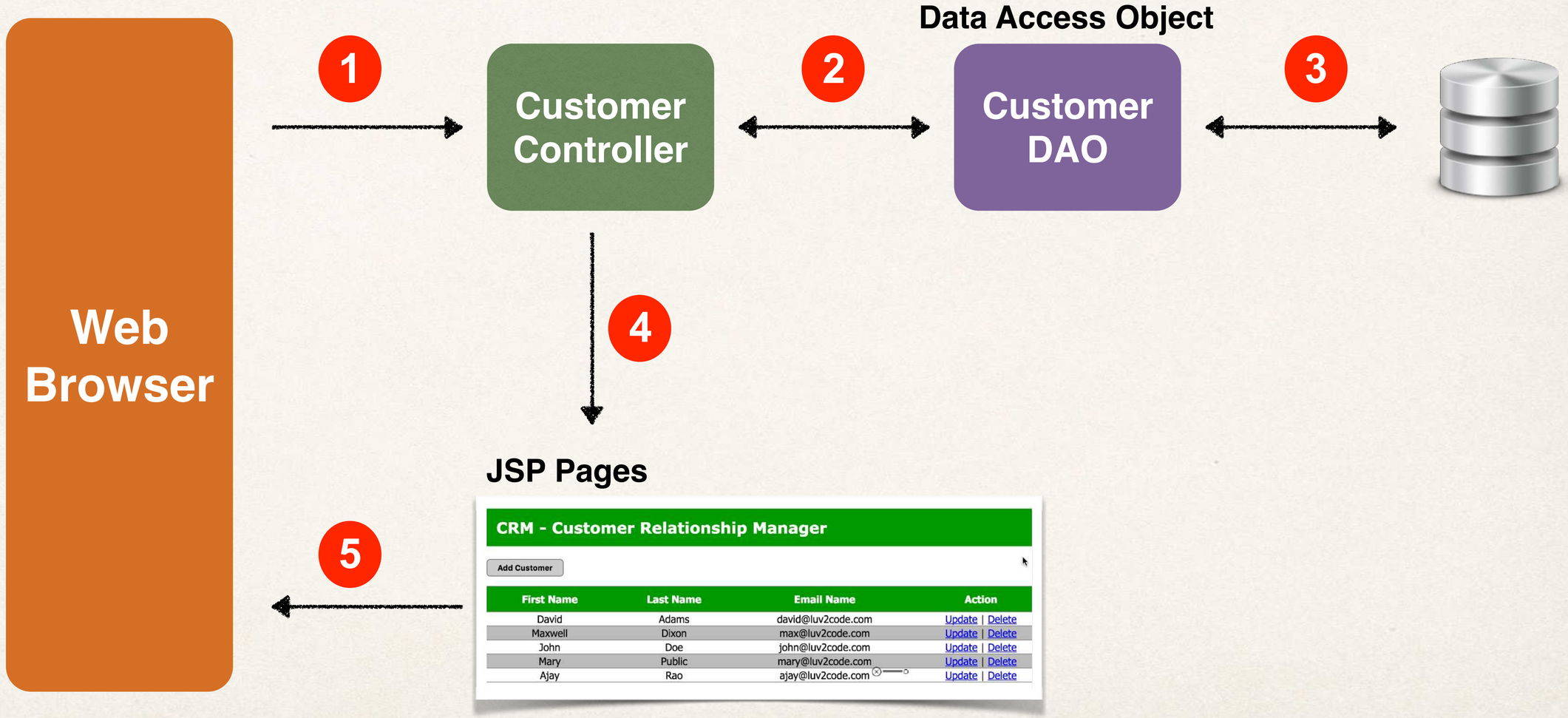
- Responsible for interfacing with the database
- This is a common design pattern:  **Data Access Object (DAO)**



# Customer Data Access Object

| Methods |
| --- |
| saveCustomer(…) |
| getCustomer(…) |
| getCustomers() |
| updateCustomer(…) |
| deleteCustomer(…) |

# Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

# List Customers - Dev Process

# List Customers

Step-By-Step

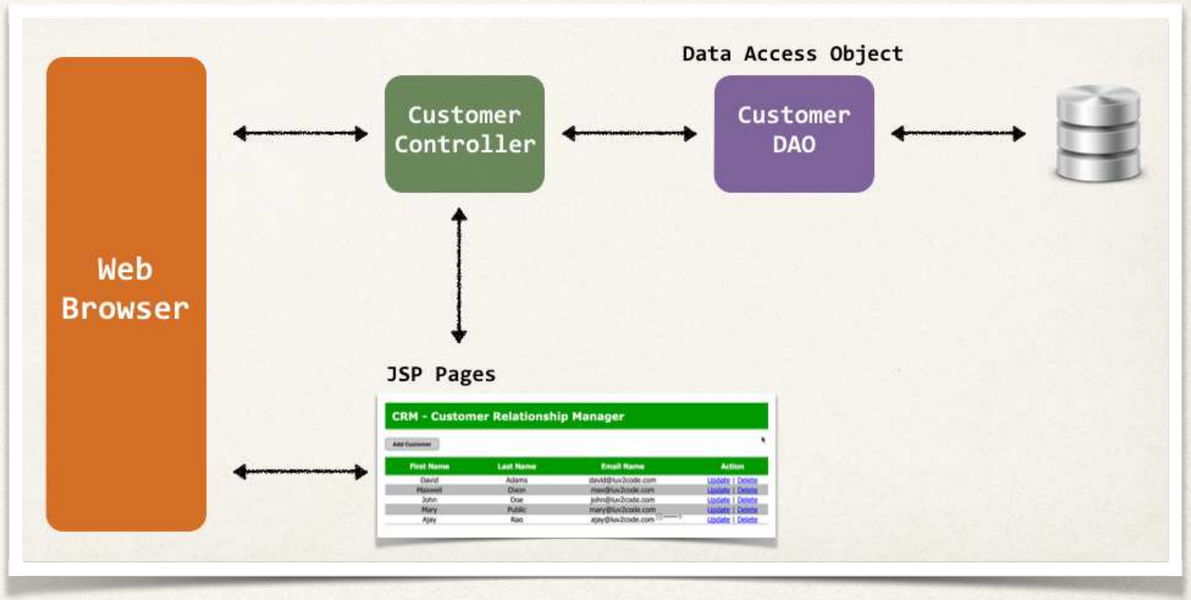
1. Create **Customer.java**

2. Create **CustomerDAO.java**

    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page:  **list-customers.jsp**



# List Customers

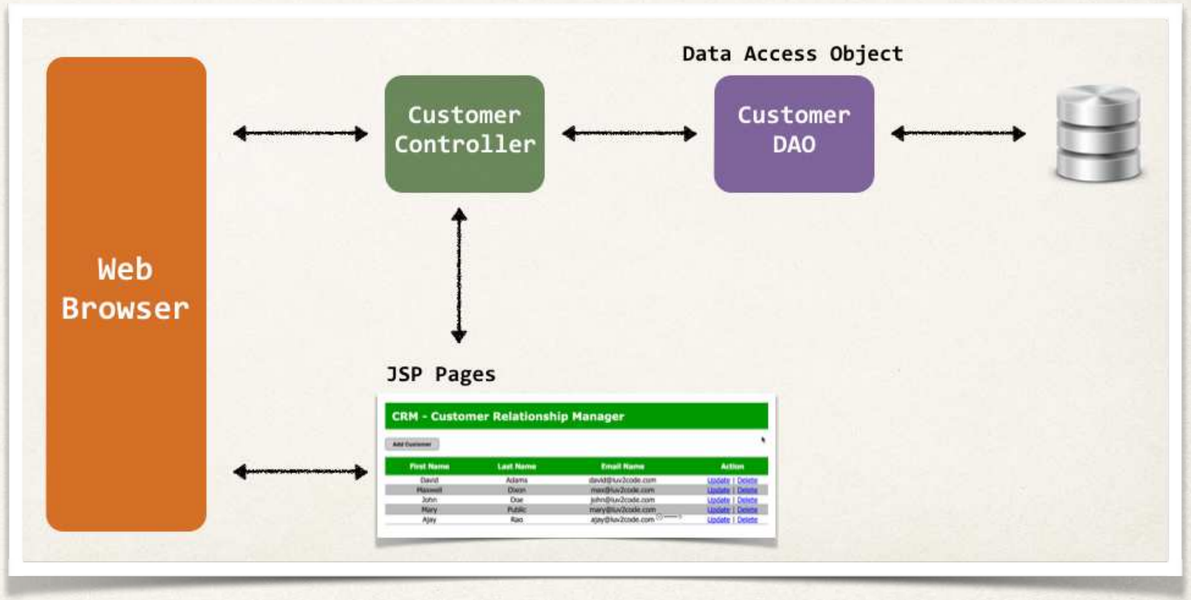
Step-By-Step

1. Create **Customer.java**

2. Create **CustomerDAO.java**
    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page:  **list-customers.jsp**

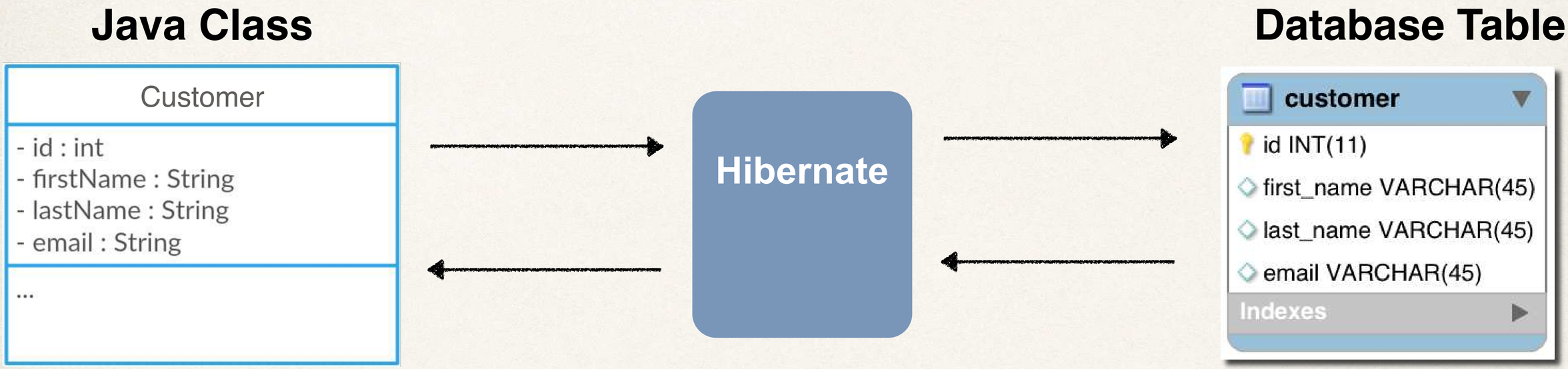


# Hibernate Terminology - Refresh

**Entity Class**

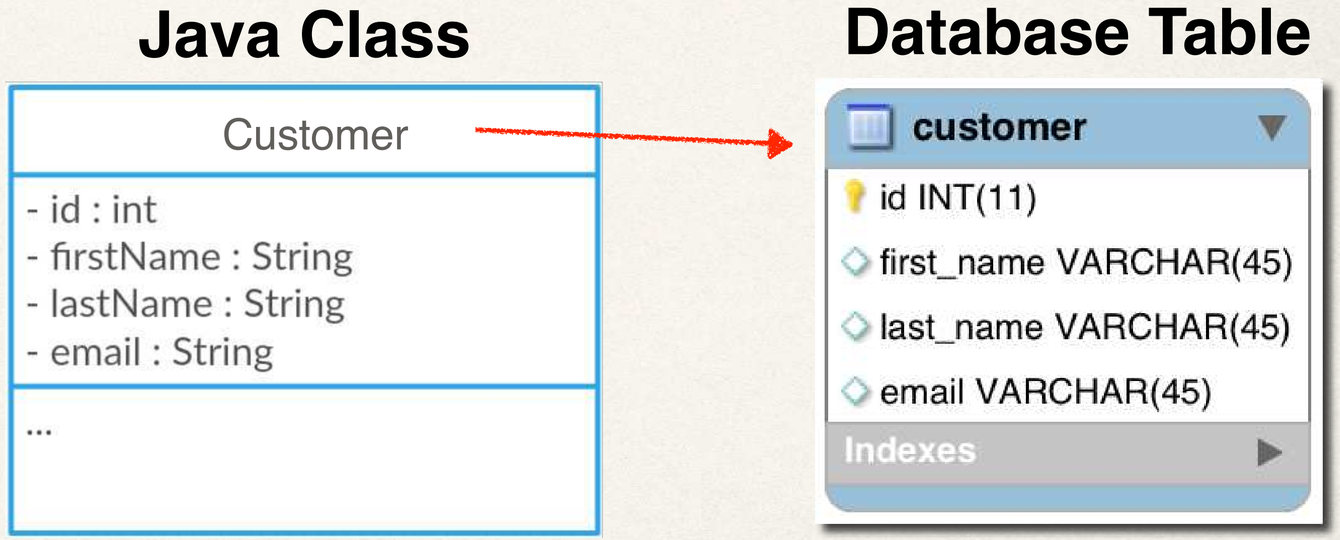
**Java class that is mapped to a database table**

# Object-to-Relational Mapping (ORM)



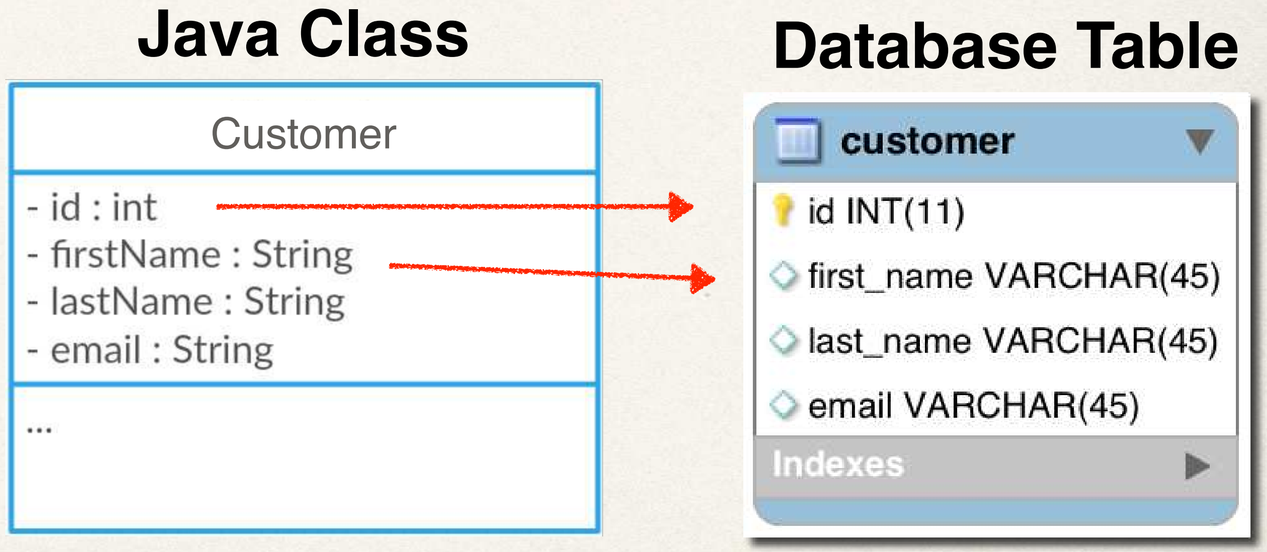
# Step 1: Map class to database table

```
@Entity
@Table(name="customer")
public class Customer {

  ...

}
```



# Step 2: Map fields to database columns

```
@Entity
@Table(name="customer")
public class Customer {

    @Id
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;
   ...
}
```



# Entity Scanning

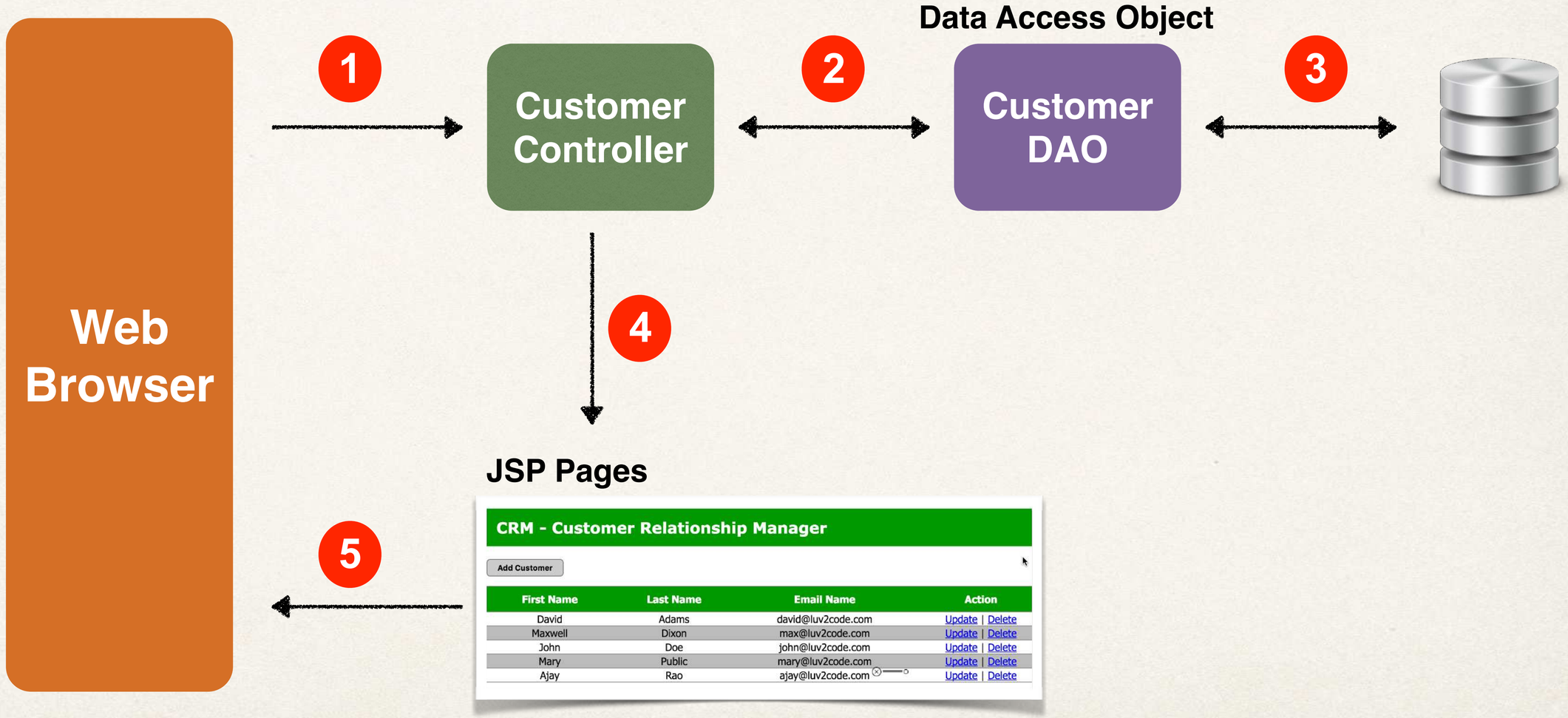
- Remember our Spring MVC config file?

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

   <property name="dataSource" ref="myDataSource" />

   <property name="packagesToScan" value="com.luv2code.springdemo.entity" />

   ...

</bean>
```

luv2code

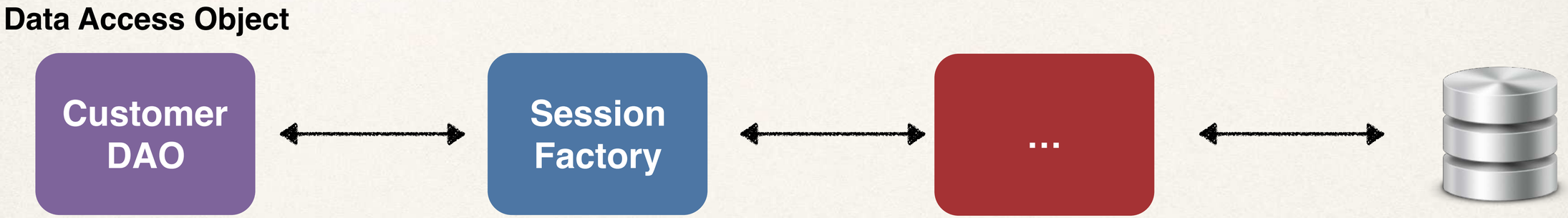
# Define Data Access Object

# Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

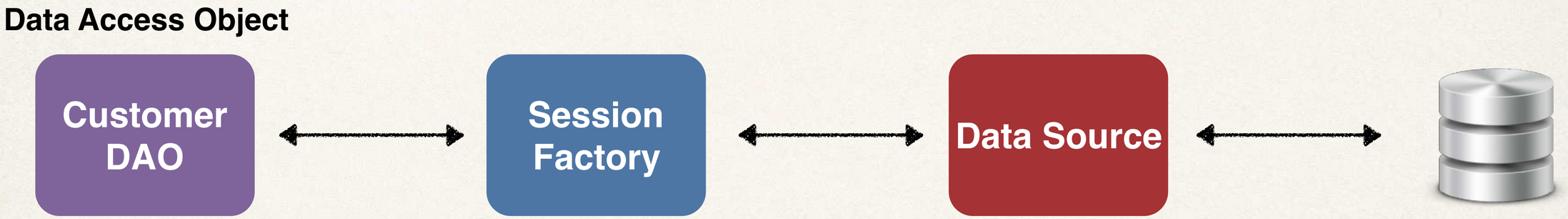
# Customer Data Access Object

- For Hibernate, our DAO needs a Hibernate SessionFactory



# Hibernate Session Factory

- Our Hibernate Session Factory needs a Data Source
  - The data source defines database connection info



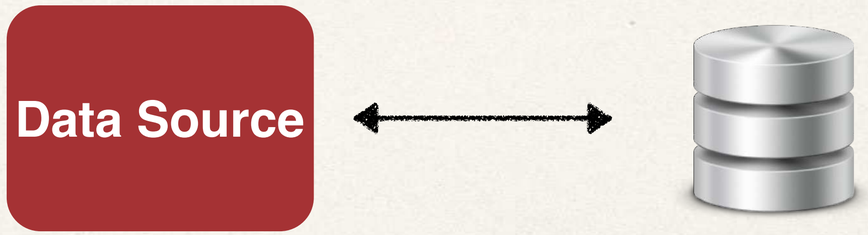
# Dependencies

**These are all dependencies!**

**We will wire them together with Dependency Injection (DI)**

# Data Source

```
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">

 <property name="driverClass" value="com.mysql.jdbc.Driver" />
 <property name="jdbcUrl"
                value="jdbc:mysql://localhost:3306/web_customer_tracker?useSSL=false" />

      … user id, password etc …
</bean>
```



# Session Factory

```
<bean id="sessionFactory"
   class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

   <property name="dataSource" ref="myDataSource" />

   ...

</bean>
```

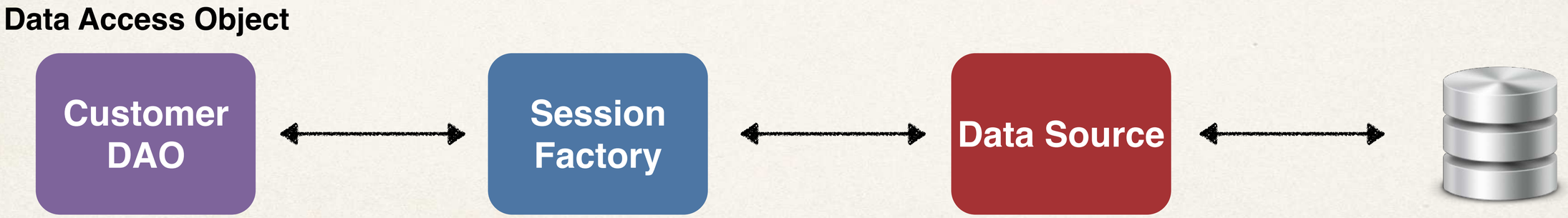
# Customer DAO

Step-By-Step

1. Define DAO interface

2. Define DAO implementation

   - Inject the session factory



# Step 1: Define DAO interface

```
public interface CustomerDAO {

   public List<Customer> getCustomers();

}
```

# Step 2: Define DAO implementation

```
public class CustomerDAOImpl implements CustomerDAO {

   @Autowired
   private SessionFactory sessionFactory;

   public List<Customer> getCustomers() {
     ...
   }

}
```

# Spring @Transactional

- Spring provides an **@Transactional** annotation

- **Automagically** begin and end a transaction for your Hibernate code
  - No need for you to explicitly do this in your code

- This Spring **magic** happens behind the scenes

# Flash back - Standalone Hibernate code

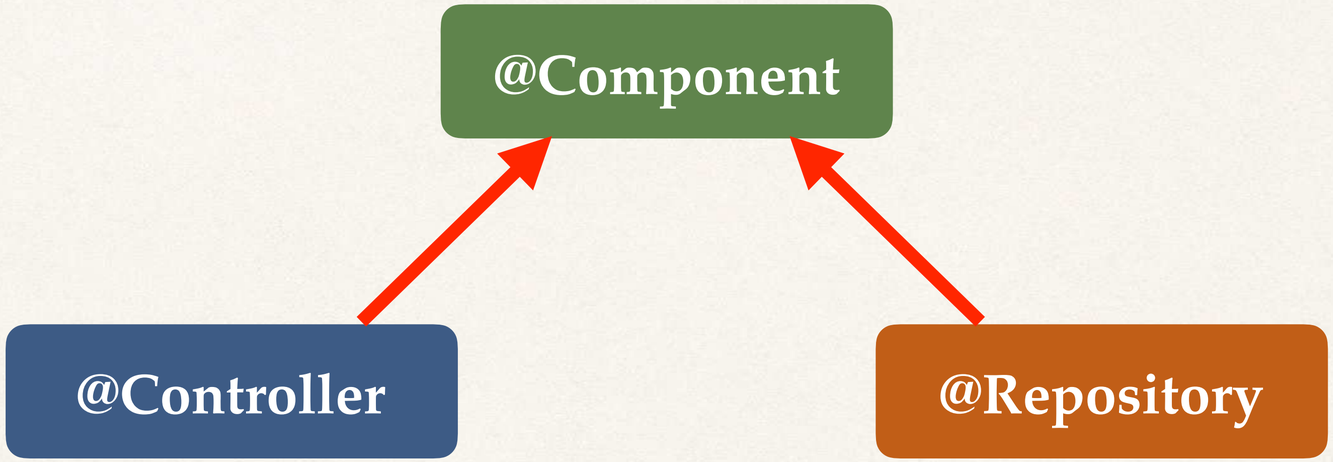
```
// start a transaction
session.beginTransaction();


// DO YOUR HIBERNATE STUFF HERE
// ...


// commit transaction
session.getTransaction().commit();
```

# Spring @Transactional Magic

```
@Transactional
public List<Customer> getCustomers() {

    // get the current hibernate session
    Session currentSession = sessionFactory.getCurrentSession();

    // create a query
    Query<Customer> theQuery =
          currentSession.createQuery("from Customer", Customer.class);

    // get the result list
    List<Customer> customers = theQuery.getResultList();

    return customers;
}
```

# Specialized Annotation for DAOs

- Spring provides the **@Repository** annotation



# Specialized Annotation for DAOs

- Applied to DAO implementations
- Spring will automatically register the DAO implementation
  - thanks to component-scanning
- Spring also provides translation of any JDBC related exceptions

# Updates for the DAO implementation

```
@Repository
public class CustomerDAOImpl implements CustomerDAO {

   @Autowired
   private SessionFactory sessionFactory;

  @Transactional
   public List<Customer> getCustomers() {
     …
   }

}
```

# Inject DAO into Controller

# List Customers

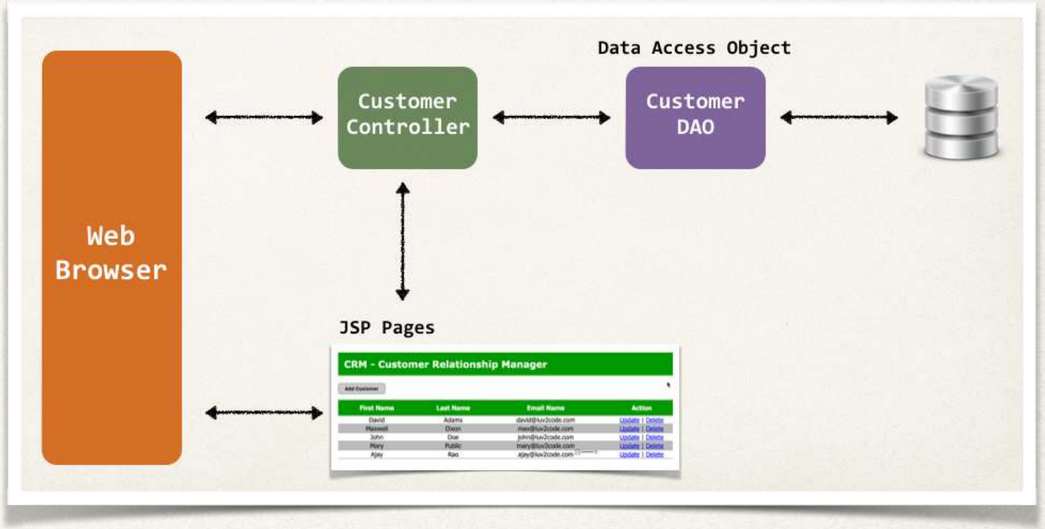
Step-By-Step

1. Create **Customer.java**

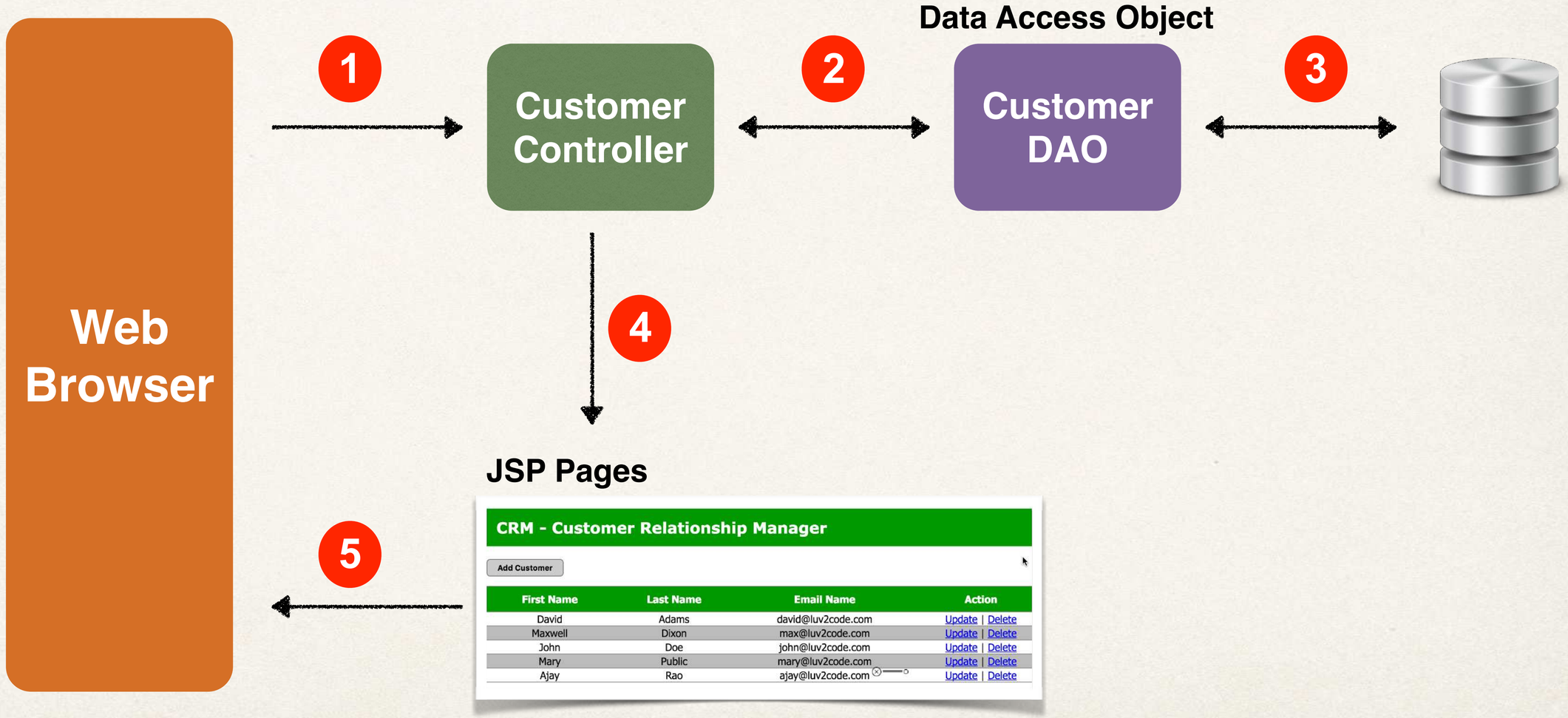
2. Create **CustomerDAO.java**
    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page: **list-customers.jsp**



# Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

luv2code

# Create JSP View Page

# List Customers

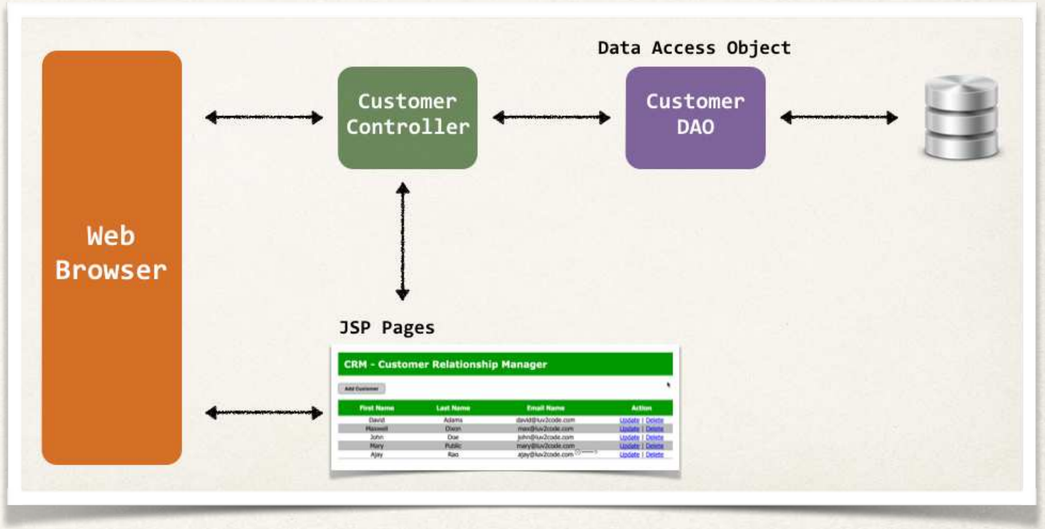
Step-By-Step

1. Create **Customer.java**

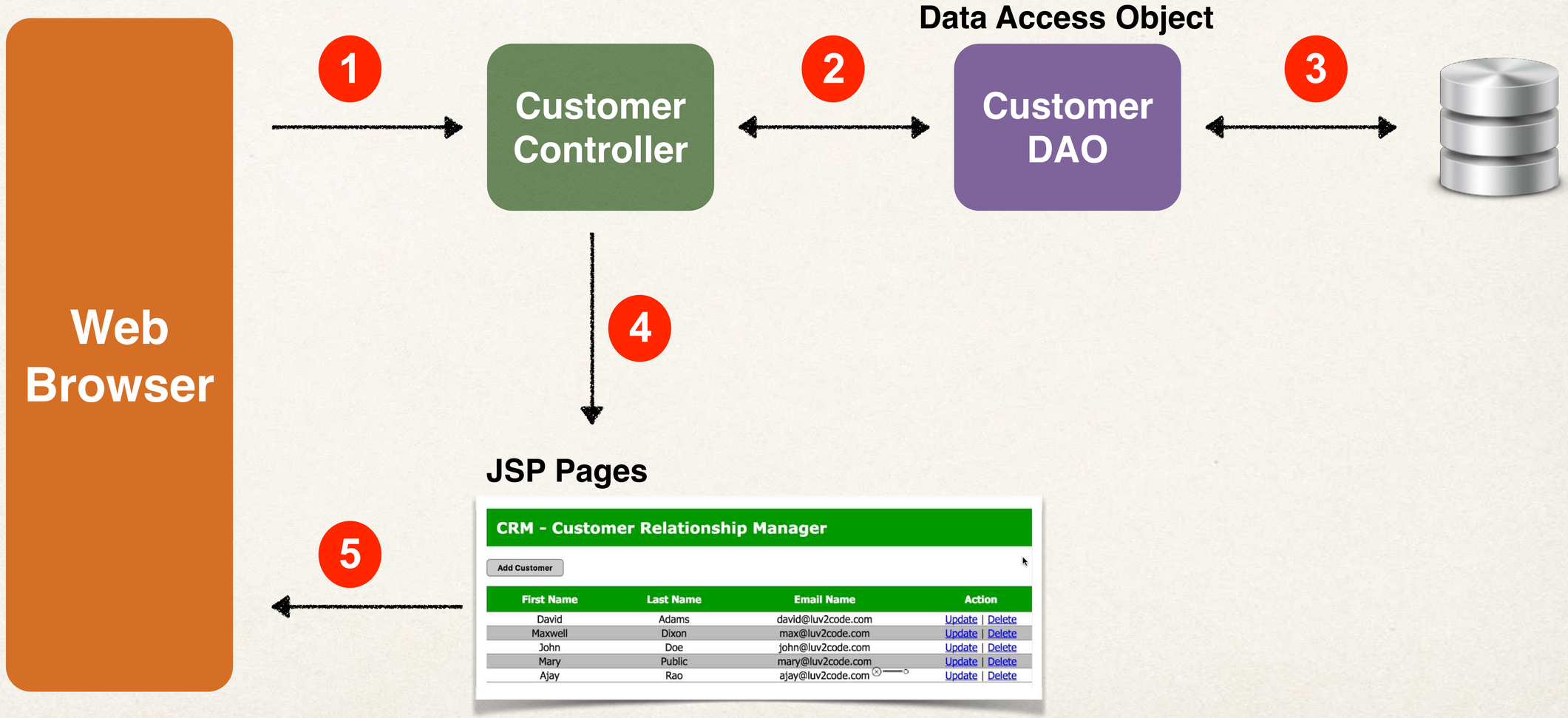
2. Create **CustomerDAO.java**
    1. and **CustomerDAOImpl.java**

3. Create **CustomerController.java**

4. Create JSP page: **list-customers.jsp**



# Big Picture


Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

# First Version - Plain

**CRM - Customer Relationship Manager**

| First Name | Last Name | Email |
|---|---|---|
| David | Adams | david@luv2code.com |
| John | Doe | john@luv2code.com |
| Ajay | Rao | ajay@luv2code.com |
| Mary | Public | mary@luv2code.com |
| Maxwell | Dixon | max@luv2code.com |

luv2code

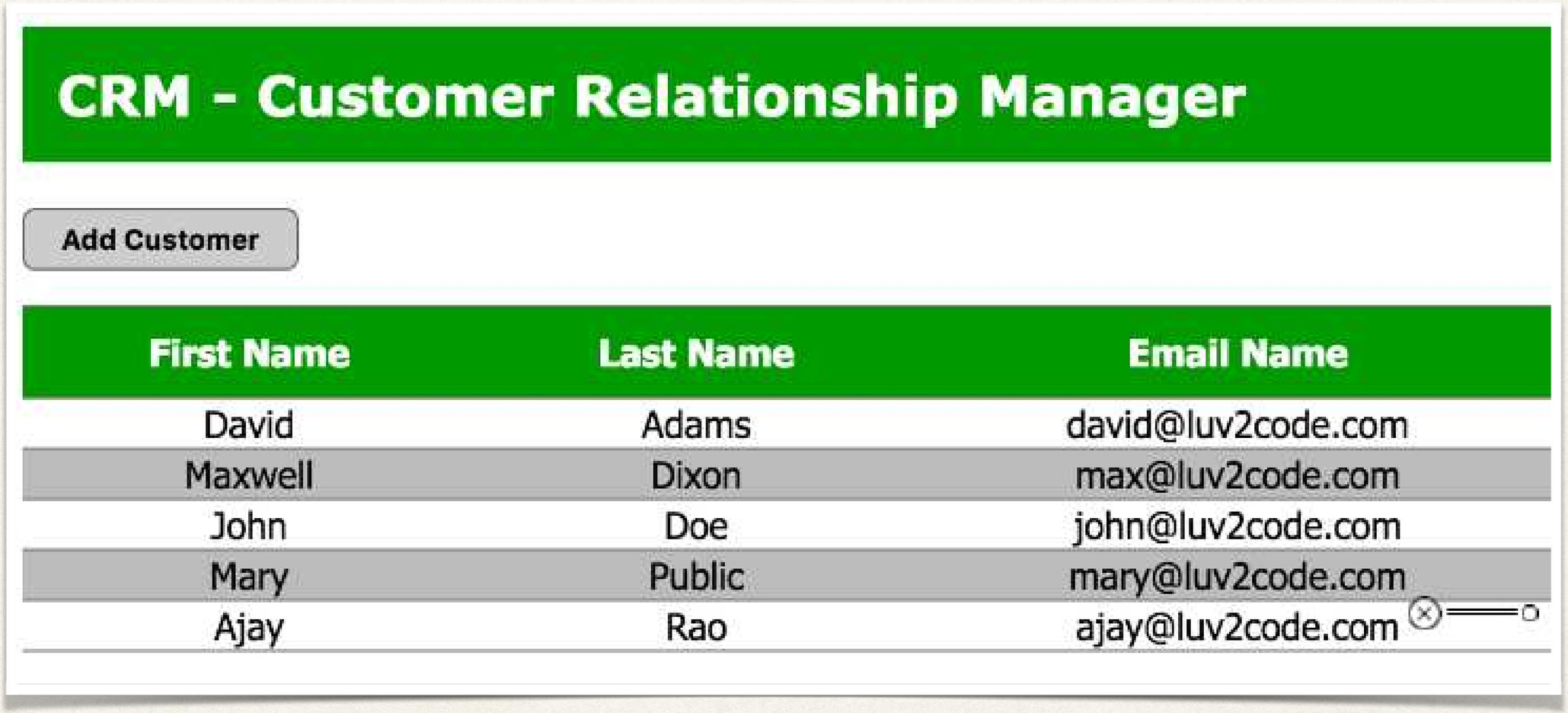
# Adding CSS to Spring MVC App

# First Version - Plain

**CRM - Customer Relationship Manager**

| First Name | Last Name | Email |
|---|---|---|
| David | Adams | david@luv2code.com |
| John | Doe | john@luv2code.com |
| Ajay | Rao | ajay@luv2code.com |
| Mary | Public | mary@luv2code.com |
| Maxwell | Dixon | max@luv2code.com |

# Apply CSS and make it Pretty



# Development Process

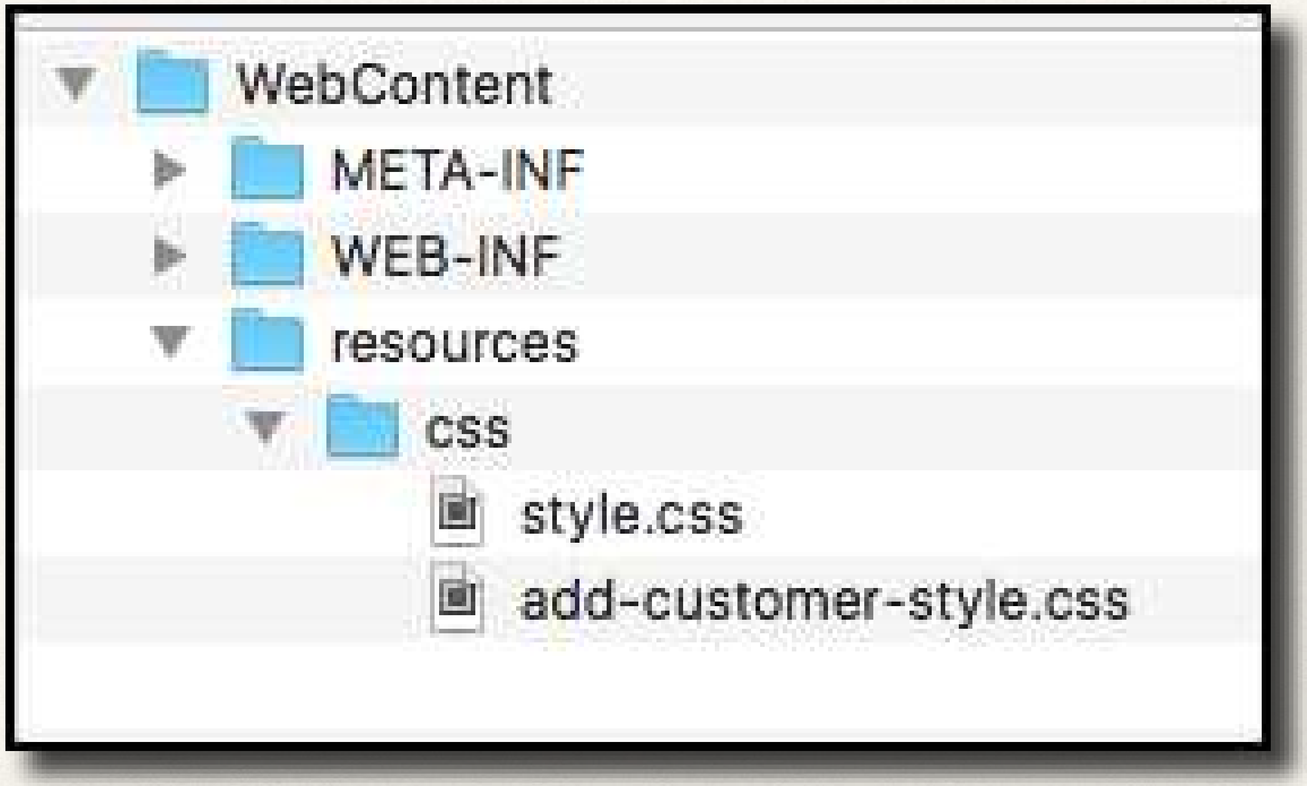
Step-By-Step

1. Place CSS in a ‘resources’ directory

2. Configure Spring to serve up ‘resources’ directory

3. Reference CSS in your JSP

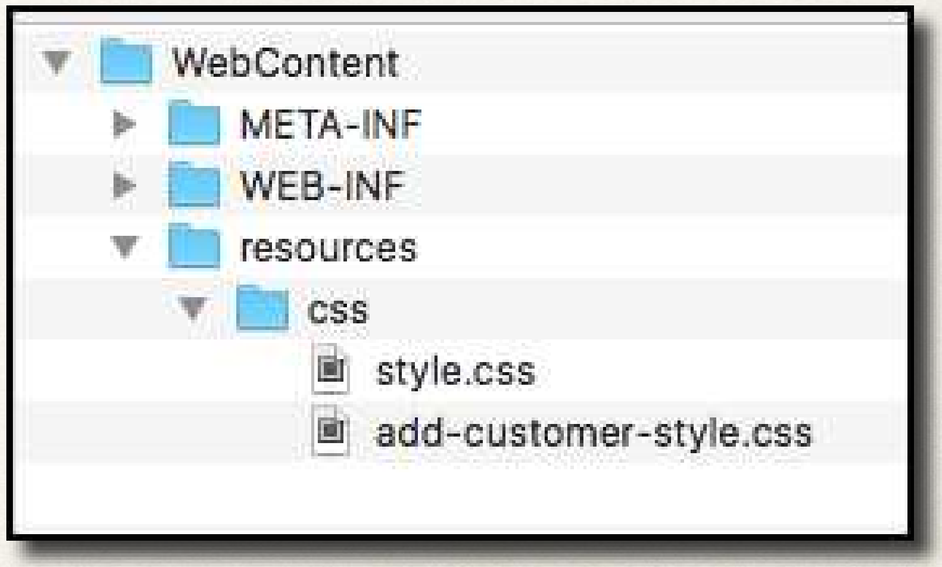
# Step 1: Place CSS in ‘resources’ directory



# Step 2: Configure Spring to serve up ‘resources’ directory

File: spring-mvc-crud-demo-servlet.xml

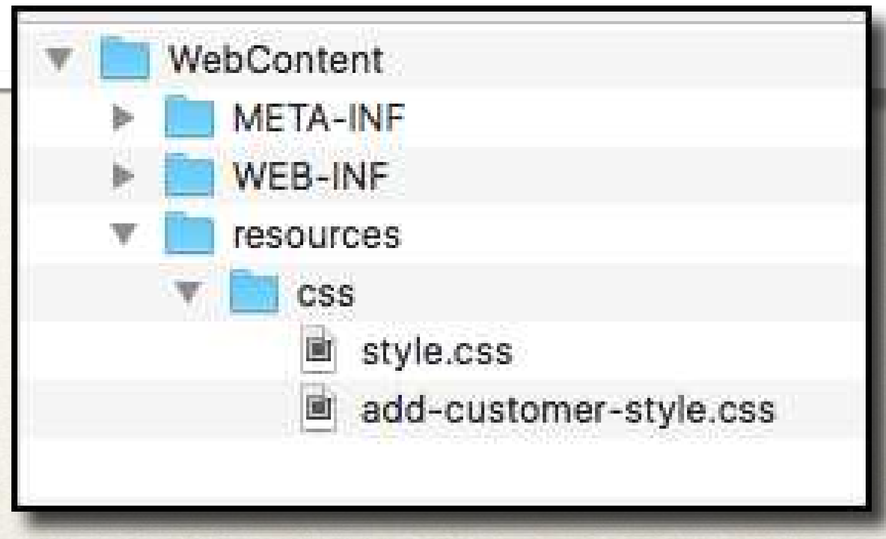
```
<mvc:resources location="/resources/" mapping="/resources/**" />
```



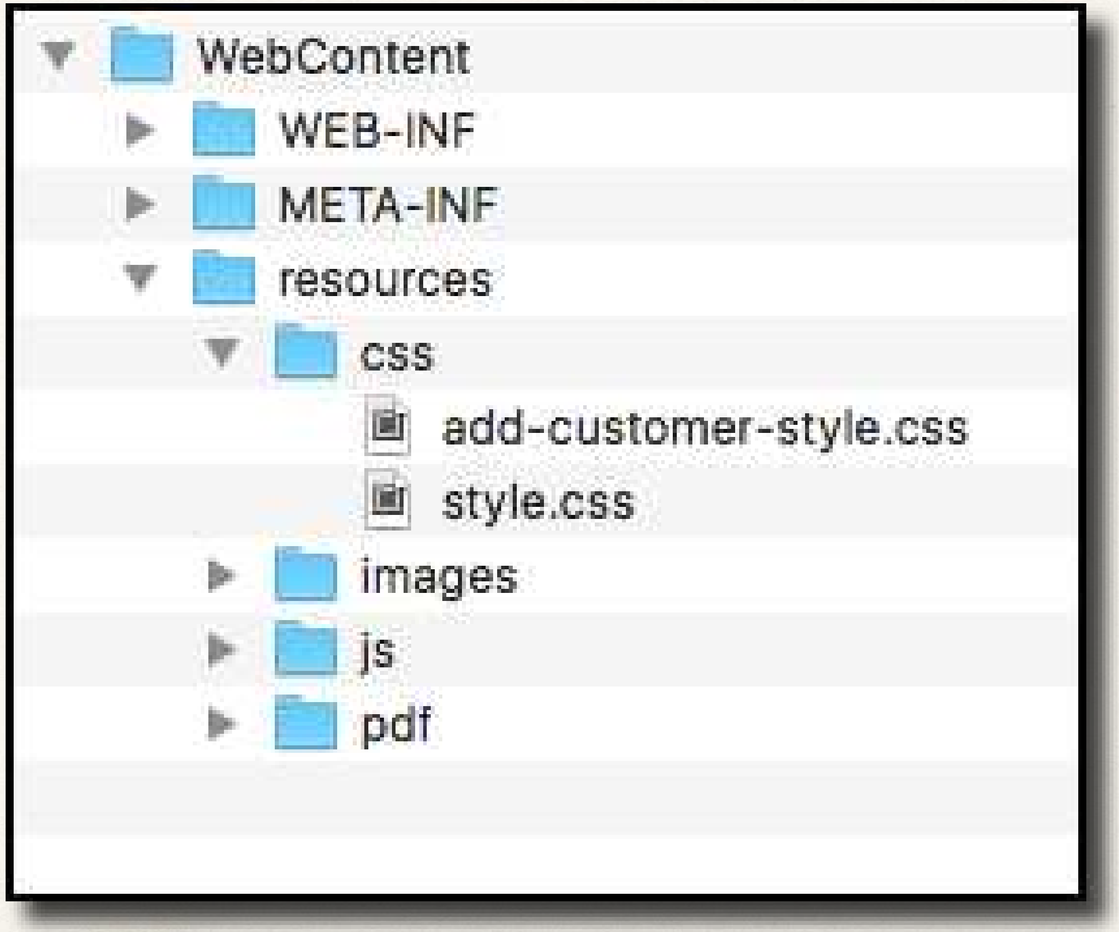
# Step 3: Reference CSS in your JSP

File: list-customers.jsp

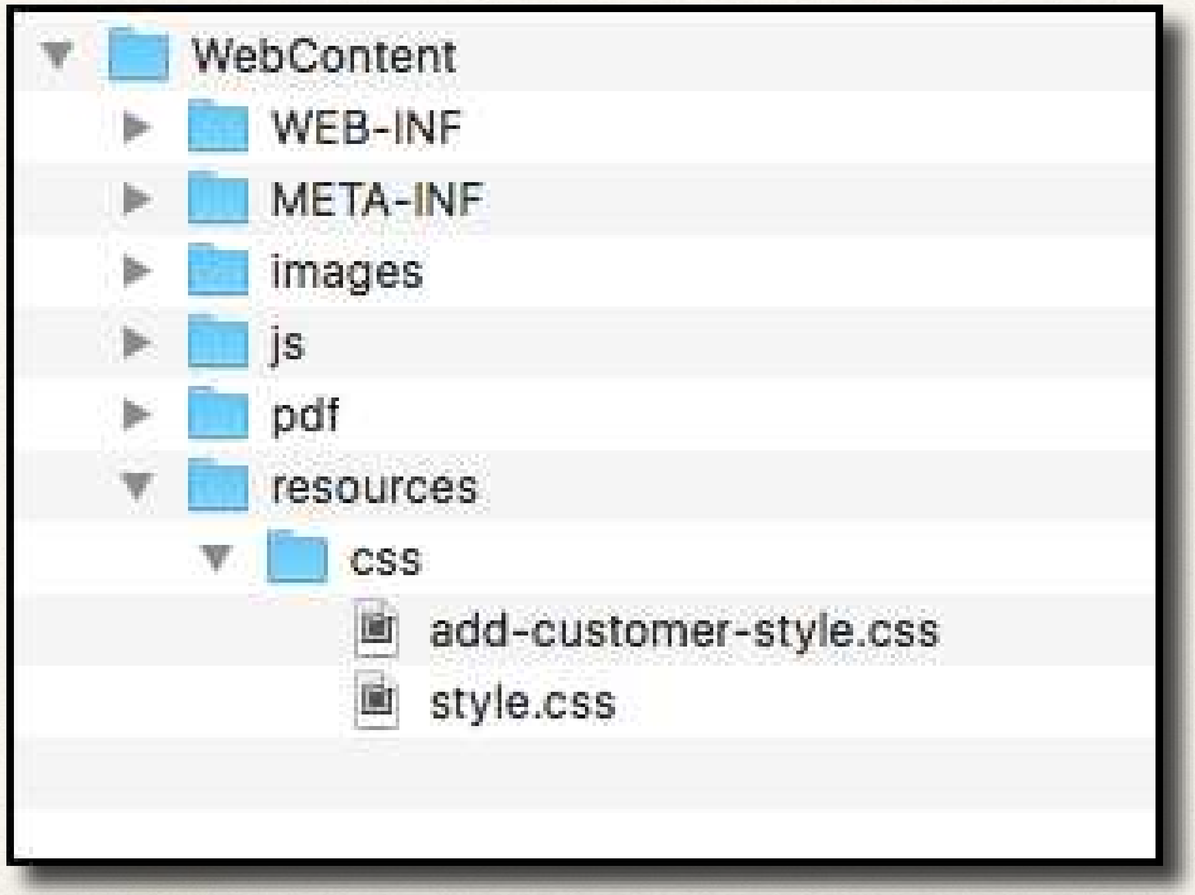
```
<head>
   <title>List Customers</title>

   <link type="text/css"
         rel="stylesheet"
         href="${pageContext.request.contextPath}/resources/css/style.css">

</head>
```



# Applies for JavaScript, images, pdfs etc…



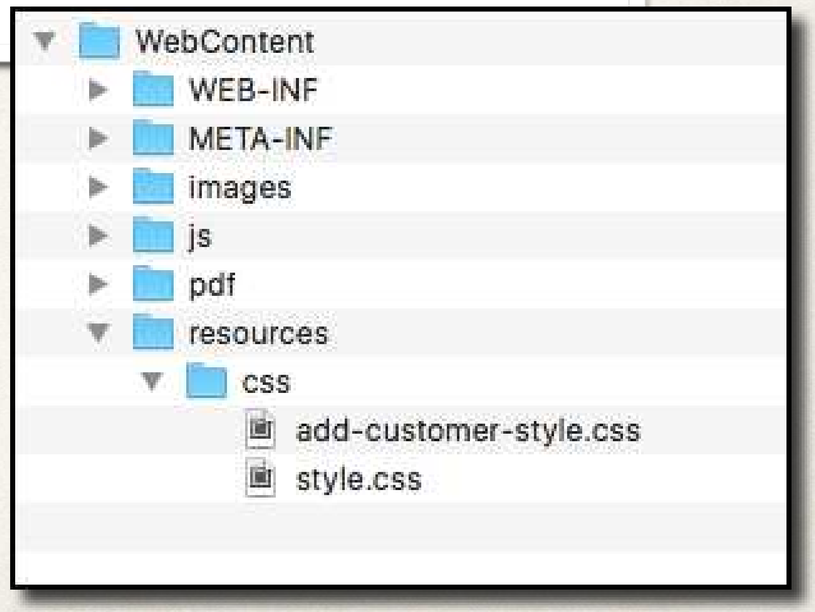
# Alternate Directory Structure



# Alternate Directory Structure

File: spring-mvc-crud-demo-servlet.xml

```
<mvc:resources location="/resources/" mapping="/resources/**" />
<mvc:resources location="/images/" mapping="/images/**" />
<mvc:resources location="/js/" mapping="/js/**" />
<mvc:resources location="/pdf/" mapping="/pdf/**" />
```



luv2code

# Adding a Welcome File

# This is our “welcome page” ???

# Welcome File

**http://localhost:8080/web-customer-tracker**

1. Server will look for a welcome file

2. If it doesn't find one, then you'll get 404 :-(

3. Welcome files are configured in **web.xml**

luv2code

# @GetMapping and @PostMapping

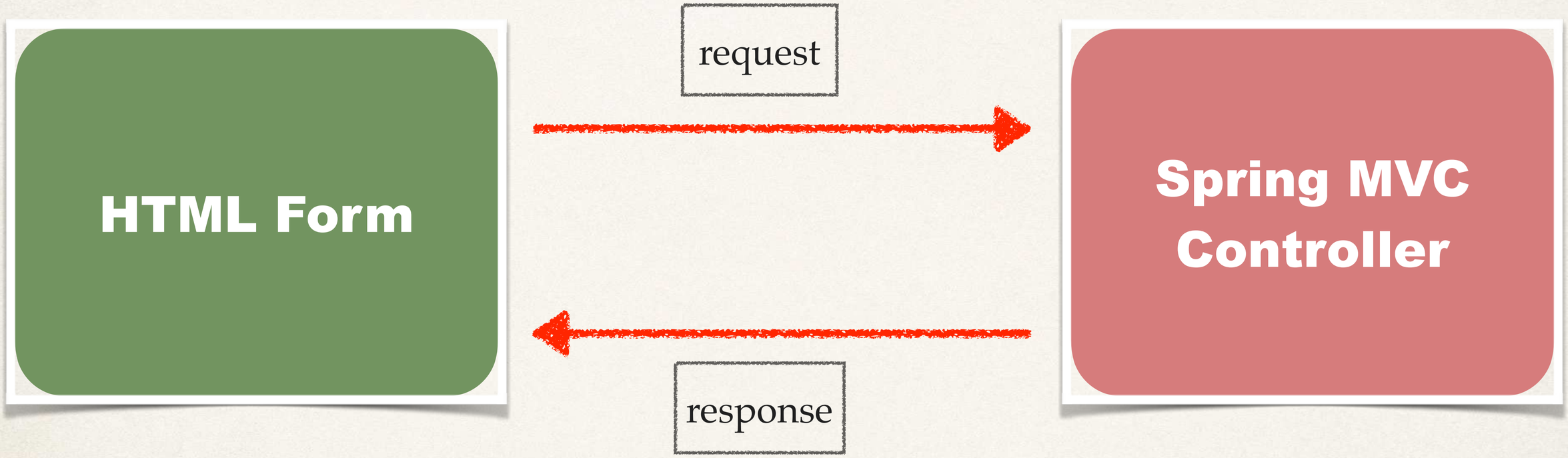
# New Annotations

- @GetMapping

- @PostMapping

Added in Spring 4.3

# HTTP Request / Response



# Most Commonly Used HTTP Methods

| Method | Description |
| --- | --- |
| **GET** | Requests data from given resource |
| **POST** | Submits data to given resource |
| *others* | … |

# Sending Data with GET method

```
<form action="processForm" method="GET" …>
  …
</form>
```

- Form data is added to end of URL as name/value pairs
  - **theUrl?field1=value1&field2=value2…**

# Handling Form Submission

```
@RequestMapping("/processForm")
public String processForm(...) {
   ...
}
```

- This mapping handles **ALL HTTP** methods
  - GET, POST, etc …

# Constrain the Request Mapping - GET

```
@RequestMapping(path="/processForm", method=RequestMethod.GET)
public String processForm(...) {
   ...
}
```

- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

# New Annotation Short-Cut

```
@GetMapping("/processForm")
public String processForm(...) {
    ...
}
```

Added in Spring 4.3

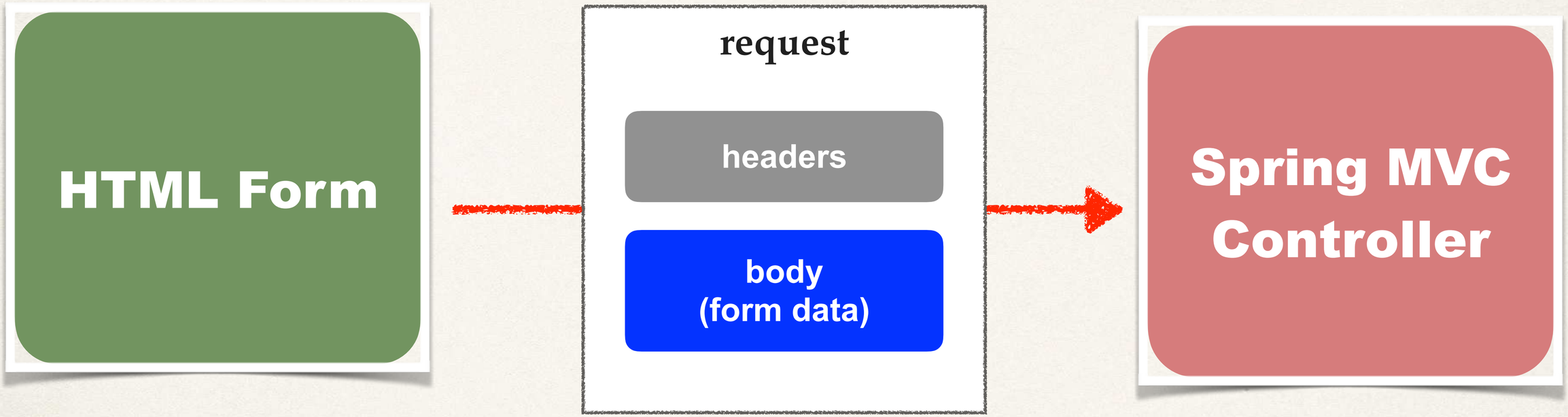
- New annotation: **@GetMapping**
- This mapping **ONLY** handles **GET** method
- Any other HTTP REQUEST method will get rejected

# Sending Data with POST method

```
<form action="processForm" method="POST" …>
  …
</form>
```

- Form data is passed in the body of HTTP request message

# Sending Data with POST method



# Constrain the Request Mapping - POST

```
@RequestMapping(path="/processForm", method=RequestMethod.POST)
public String processForm(...) {
   ...
}
```

- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

# New Annotation Short-Cut

```
@PostMapping("/processForm")
public String processForm(...) {
   ...
}
```

Added in Spring 4.3

- New annotation: **@PostMapping**
- This mapping **ONLY** handles **POST** method
- Any other HTTP REQUEST method will get rejected

# Well …. which one???

## GET

- Good for debugging
- Bookmark or email URL
- Limitations on data length

## POST

- Can't bookmark or email URL
- No limitations on data length
- Can also send binary data

# Recap: New Annotations

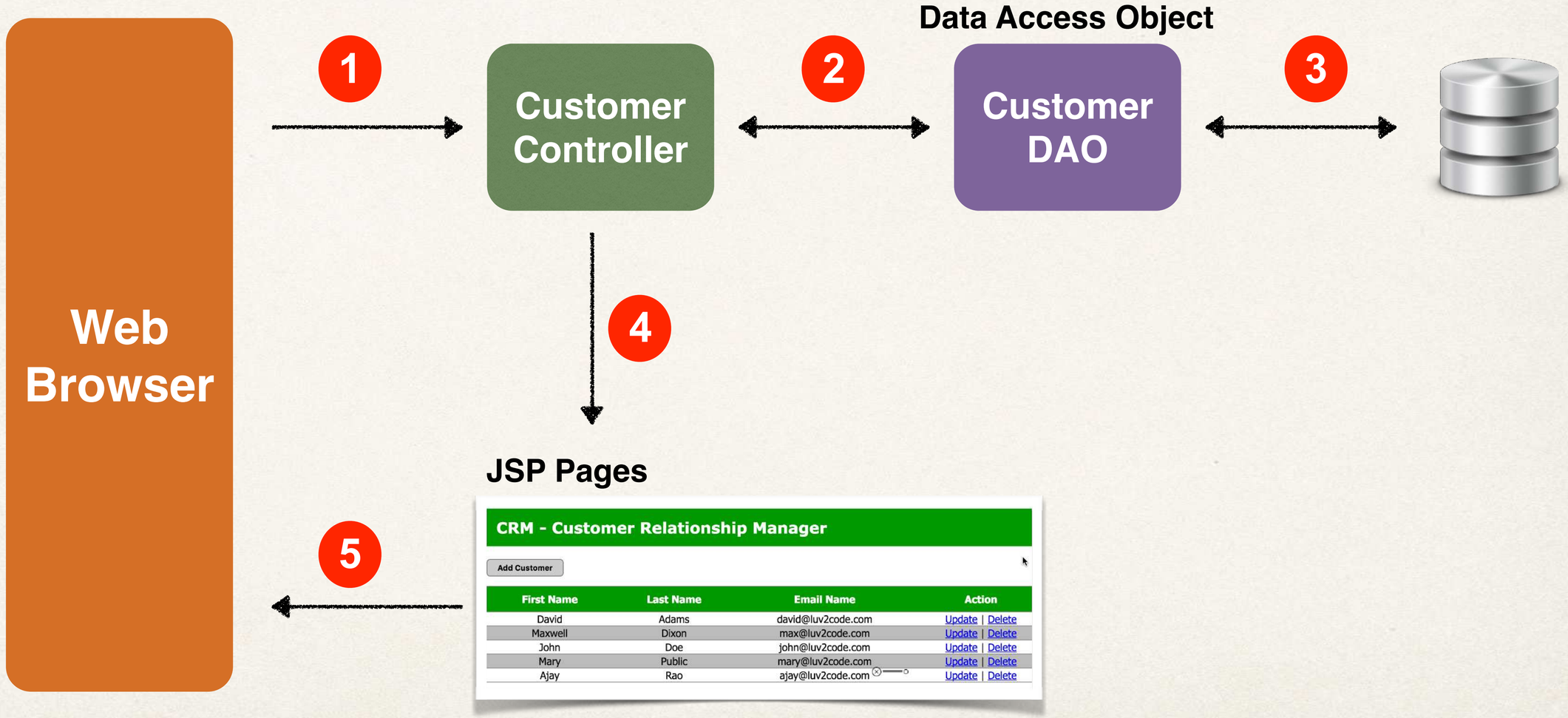
- **@GetMapping**

- **@PostMapping**

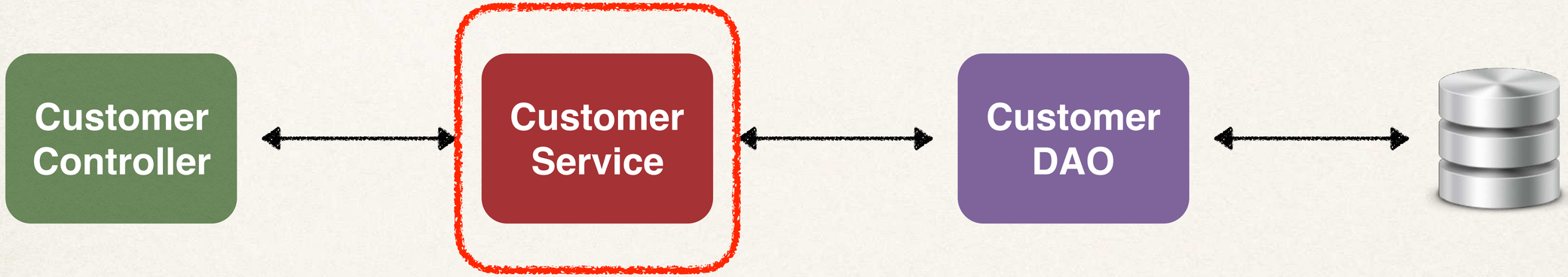
luv2code

# Define Services with @Service

# Big Picture

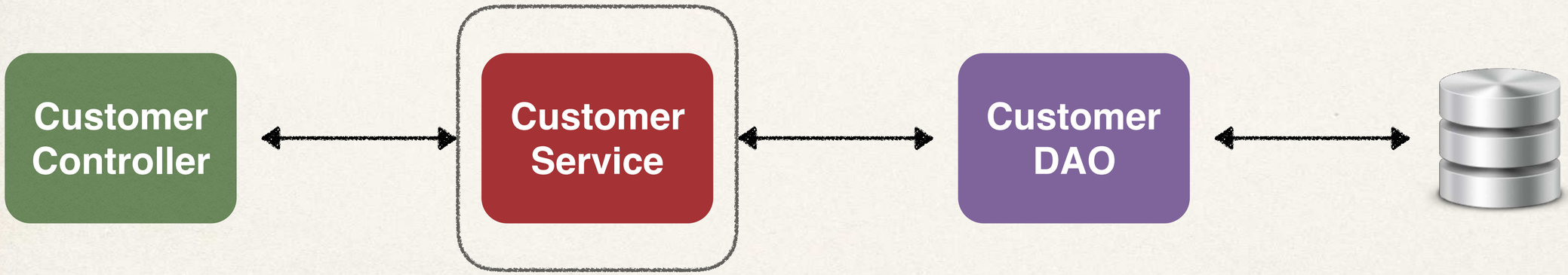

Data Access Object
Web Browser
1
Customer Controller
2
Customer DAO
3
4
JSP Pages
5
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email Name
Action
David
Adams
david@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Mary
Public
mary@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete

# Refactor: Add a Service Layer

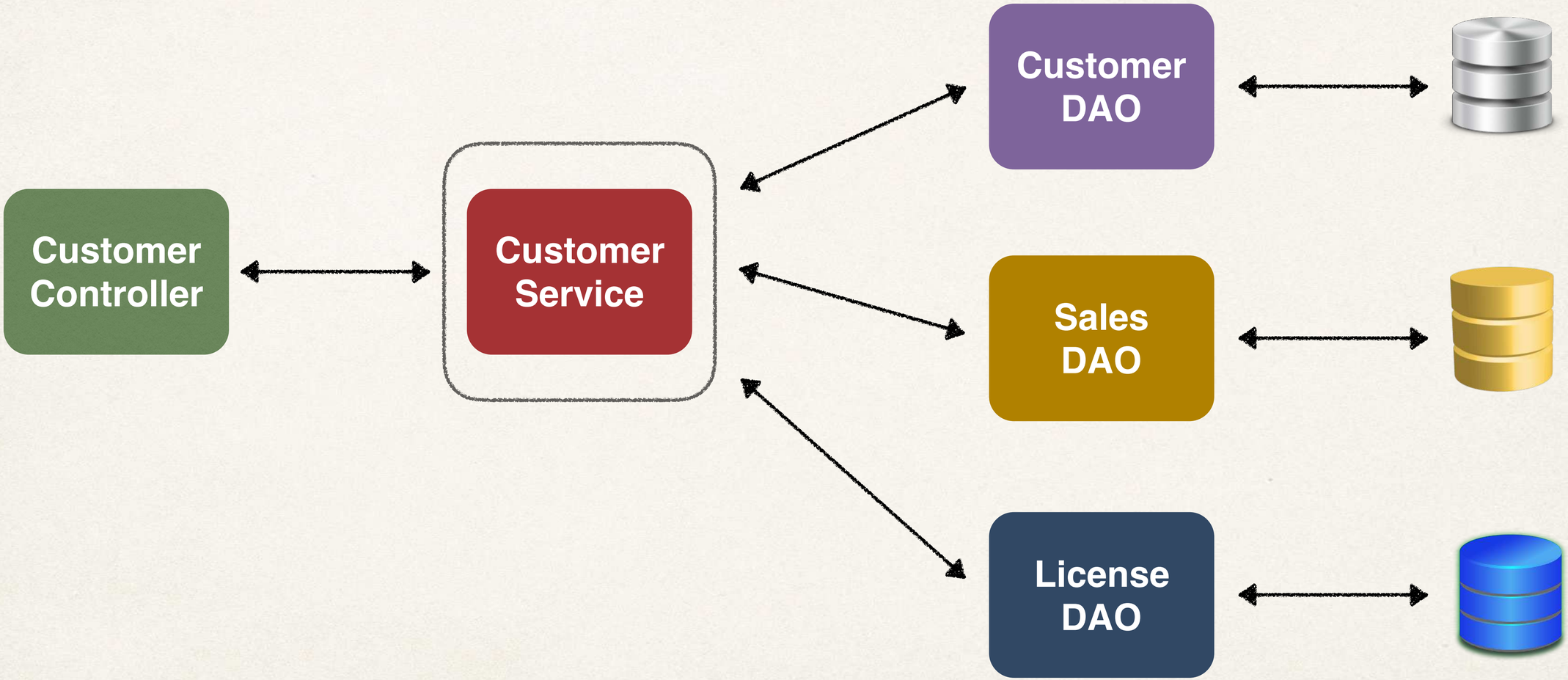


# Purpose of Service Layer

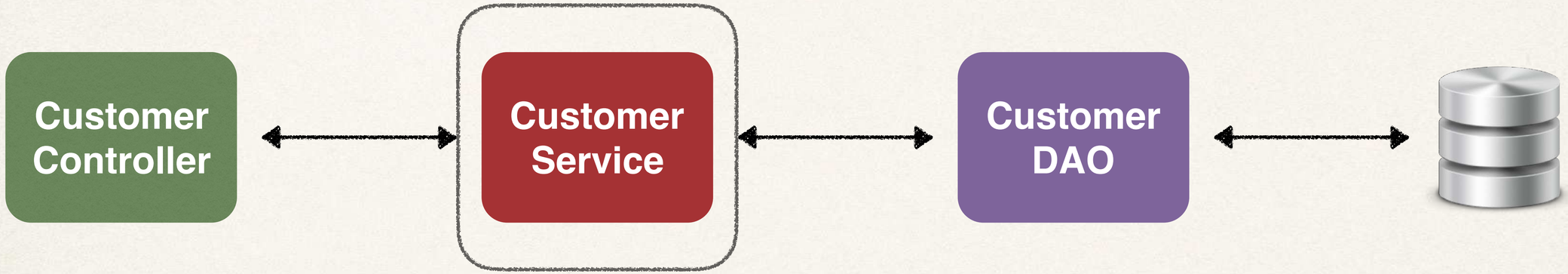
- **Service Facade** design pattern
- Intermediate layer for custom business logic
- Integrate data from multiple sources (DAO/repositories)



# Integrate Multiple Data Sources

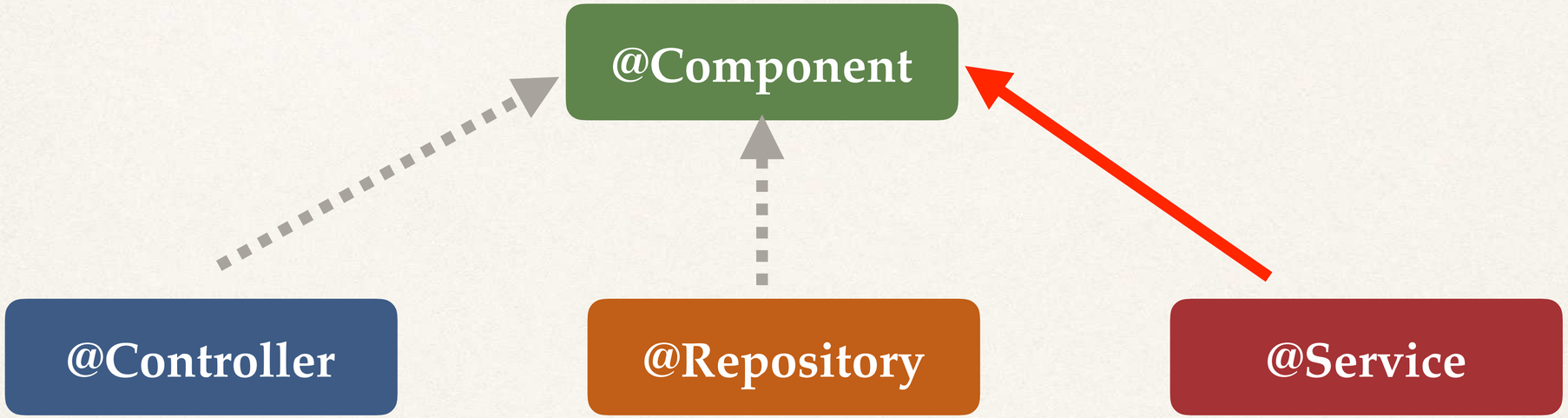


# Most Times - Delegate Calls



# Specialized Annotation for Services

- Spring provides the **@Service** annotation



# Specialized Annotation for Services

- **@Service** applied to Service implementations

- Spring will automatically register the Service implementation
  - thanks to component-scanning

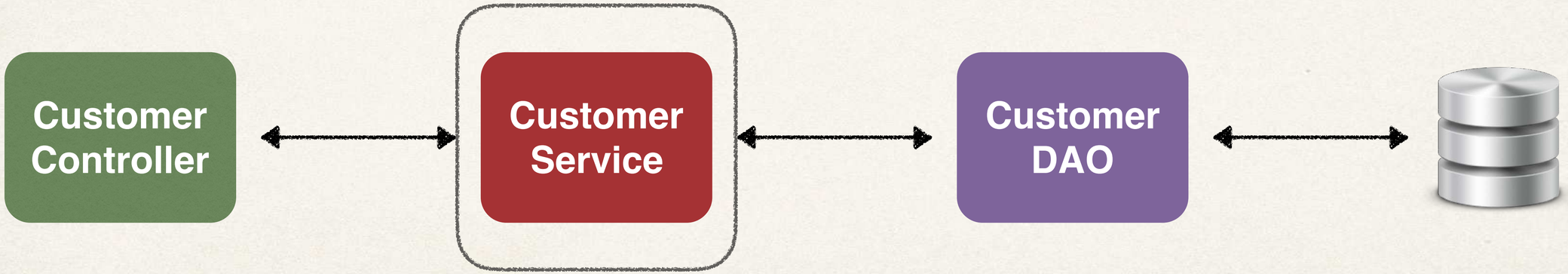
# Customer Service

Step-By-Step

1. Define Service interface

2. Define Service implementation

   - Inject the CustomerDAO



# Step 1: Define Service interface

```
public interface CustomerService {

   public List<Customer> getCustomers();

}
```

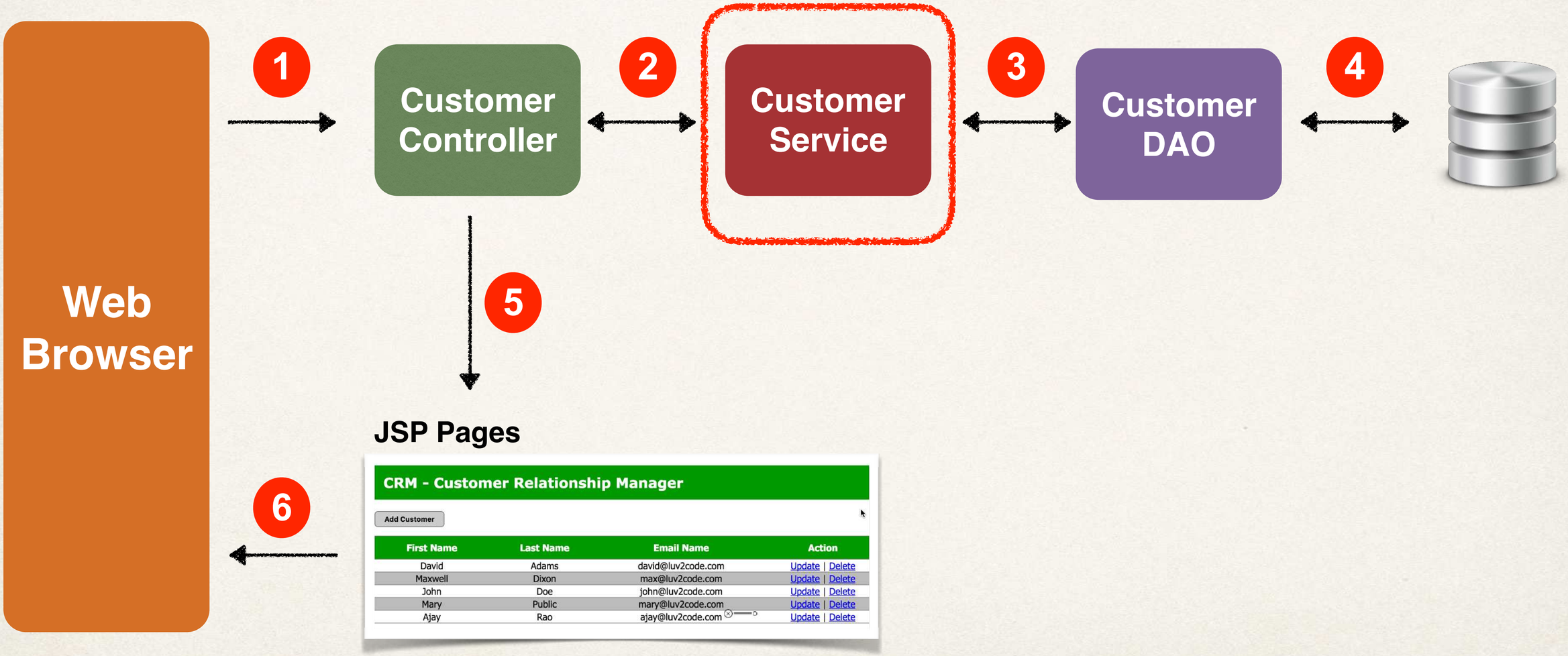
# Step 2: Define Service implementation

```
@Service
public class CustomerServiceImpl implements CustomerService {

   @Autowired
   private CustomerDAO customerDAO;

   @Transactional
   public List<Customer> getCustomers() {
     ...
   }
}
```

# Updates for the DAO implementation

```
@Repository
public class CustomerDAOImpl implements CustomerDAO {

   @Autowired
   private SessionFactory sessionFactory;

   public List<Customer> getCustomers() {
     ...
   }
}
```

# Revised Big Picture



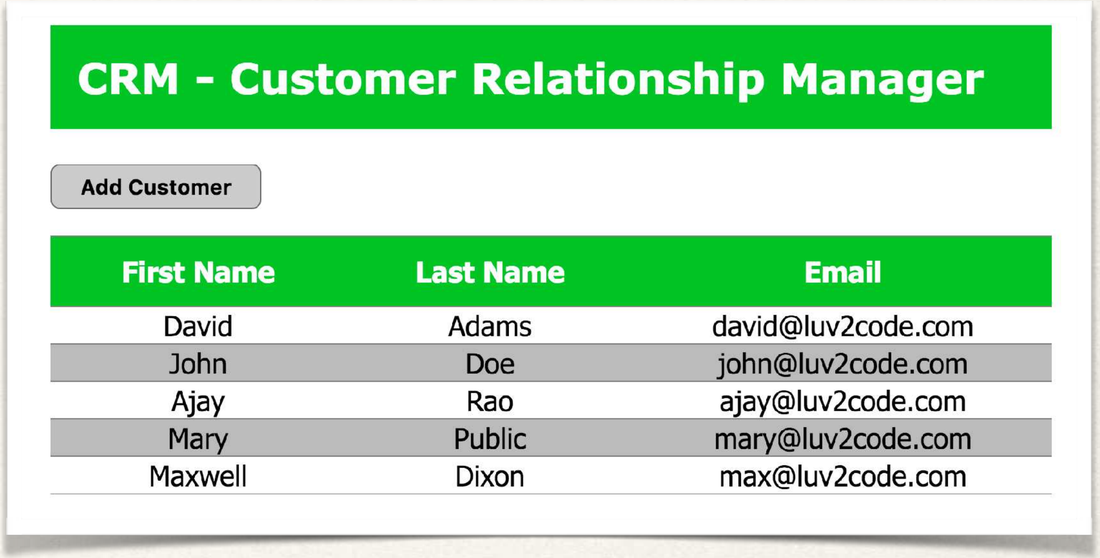
luv2code

# Add Customer

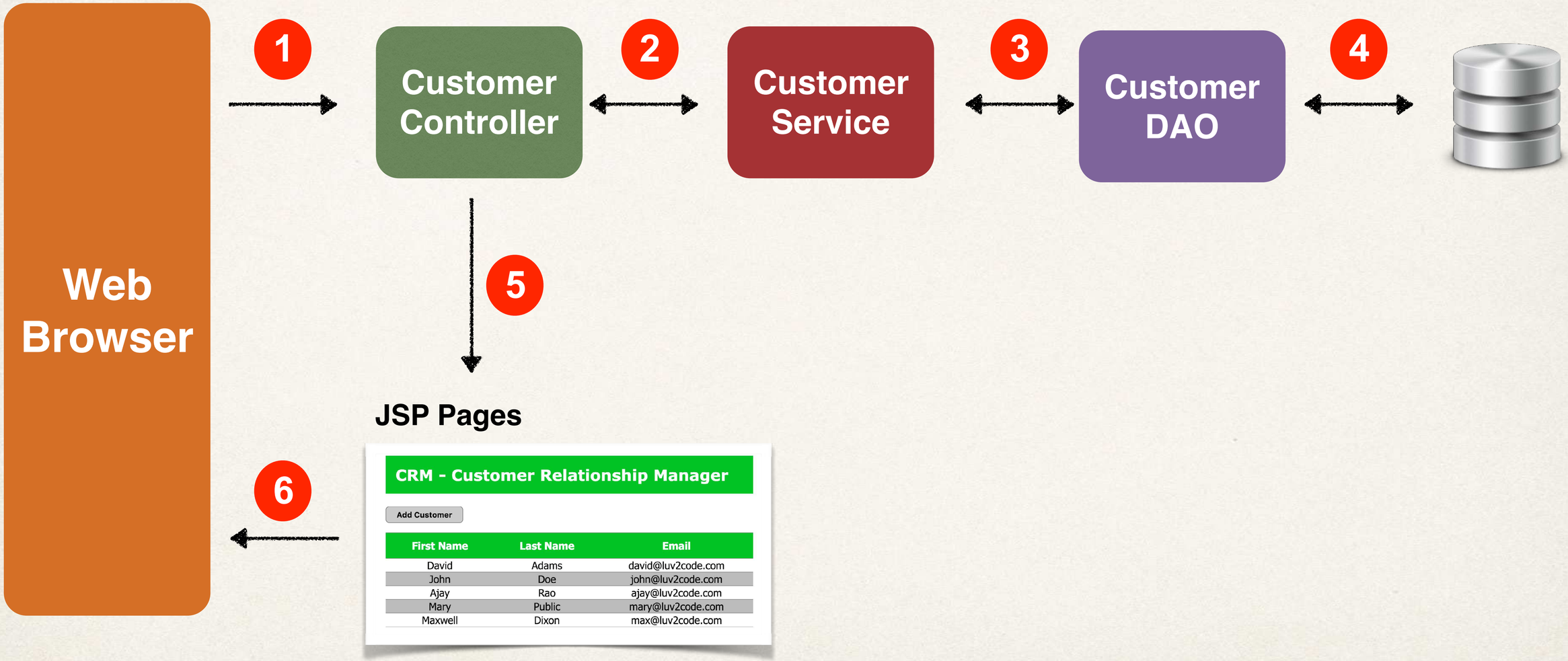
# Add Customer

Step-By-Step

1. **Update list-customer.jsp**
    1. New “Add Customer” button
2. **Create HTML form for new customer**
3. **Process Form Data**
    1. Controller -> Service -> DAO



# Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email
David
Adams
david@luv2code.com
John
Doe
john@luv2code.com
Ajay
Rao
ajay@luv2code.com
Mary
Public
mary@luv2code.com
Maxwell
Dixon
max@luv2code.com

# Add Customer

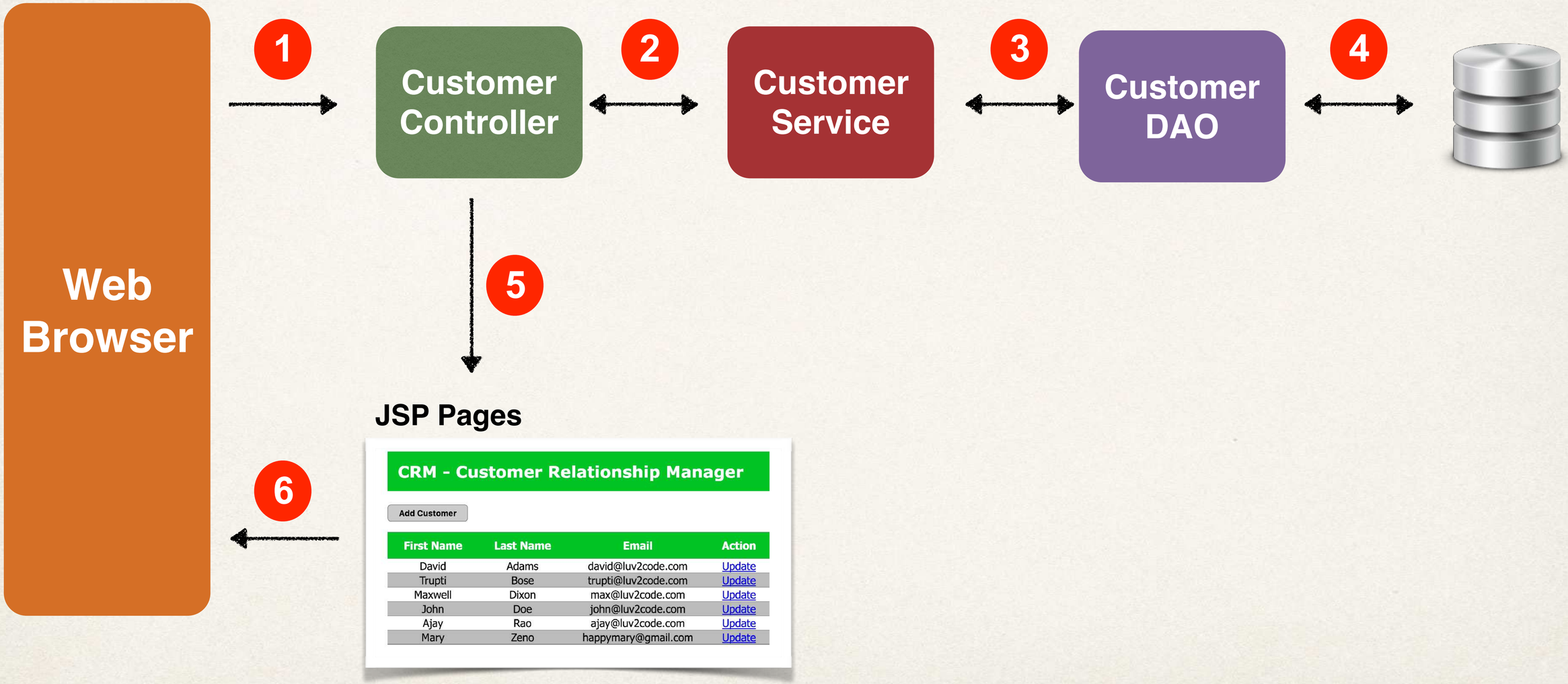

1. **Update list-customer.jsp**
    1. New “Add Customer” button

2. **Create HTML form for new customer**

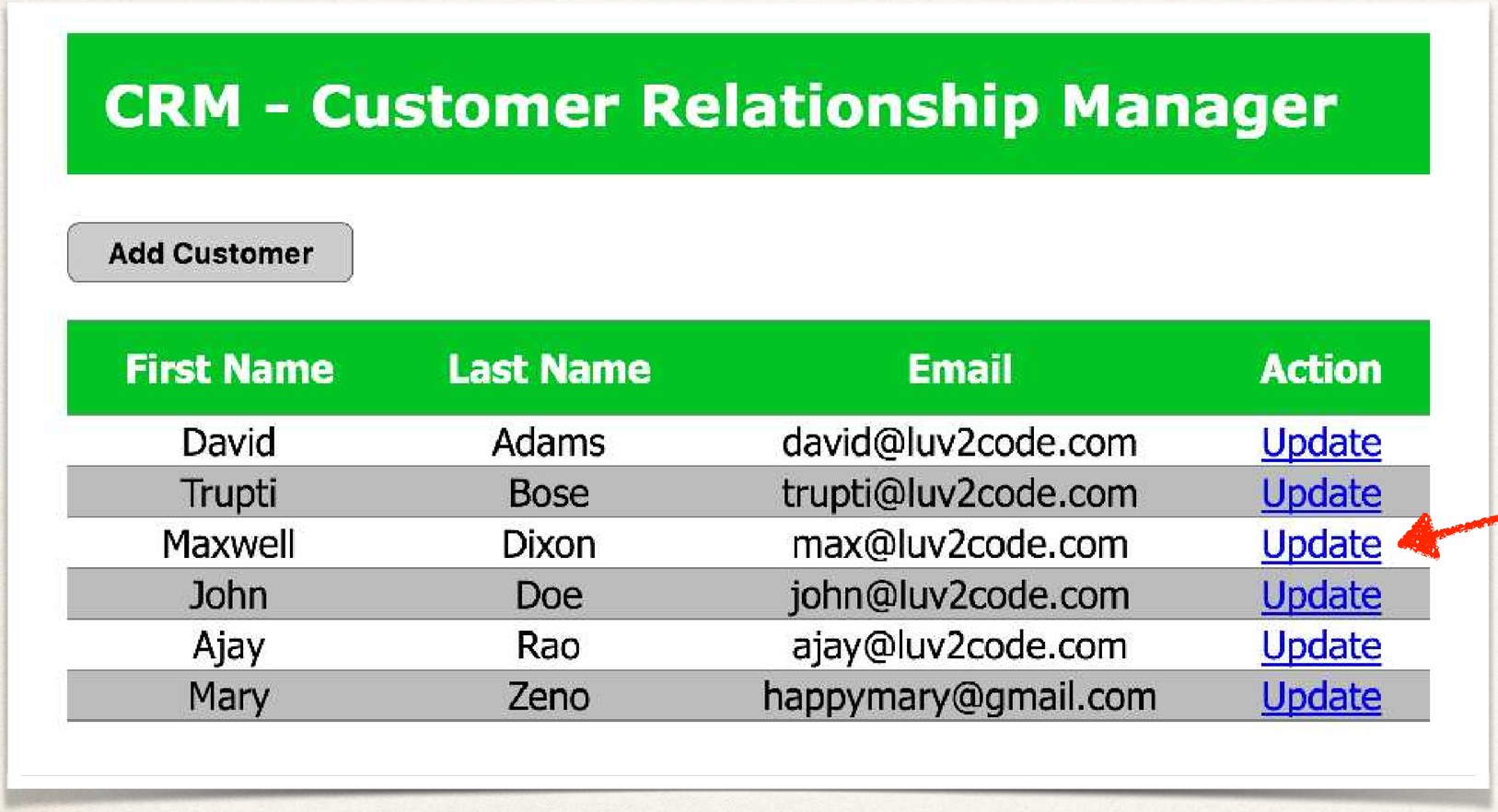
3. **Process Form Data**
    1. Controller -> Service -> DAO

# Update Customer

# Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email
Action
David
Adams
david@luv2code.com
Update
Trupti
Bose
trupti@luv2code.com
Update
Maxwell
Dixon
max@luv2code.com
Update
John
Doe
john@luv2code.com
Update
Ajay
Rao
ajay@luv2code.com
Update
Mary
Zeno
happymary@gmail.com
Update

# Action Link - Update




Each row has an **Update** link

- current customer id embedded in link

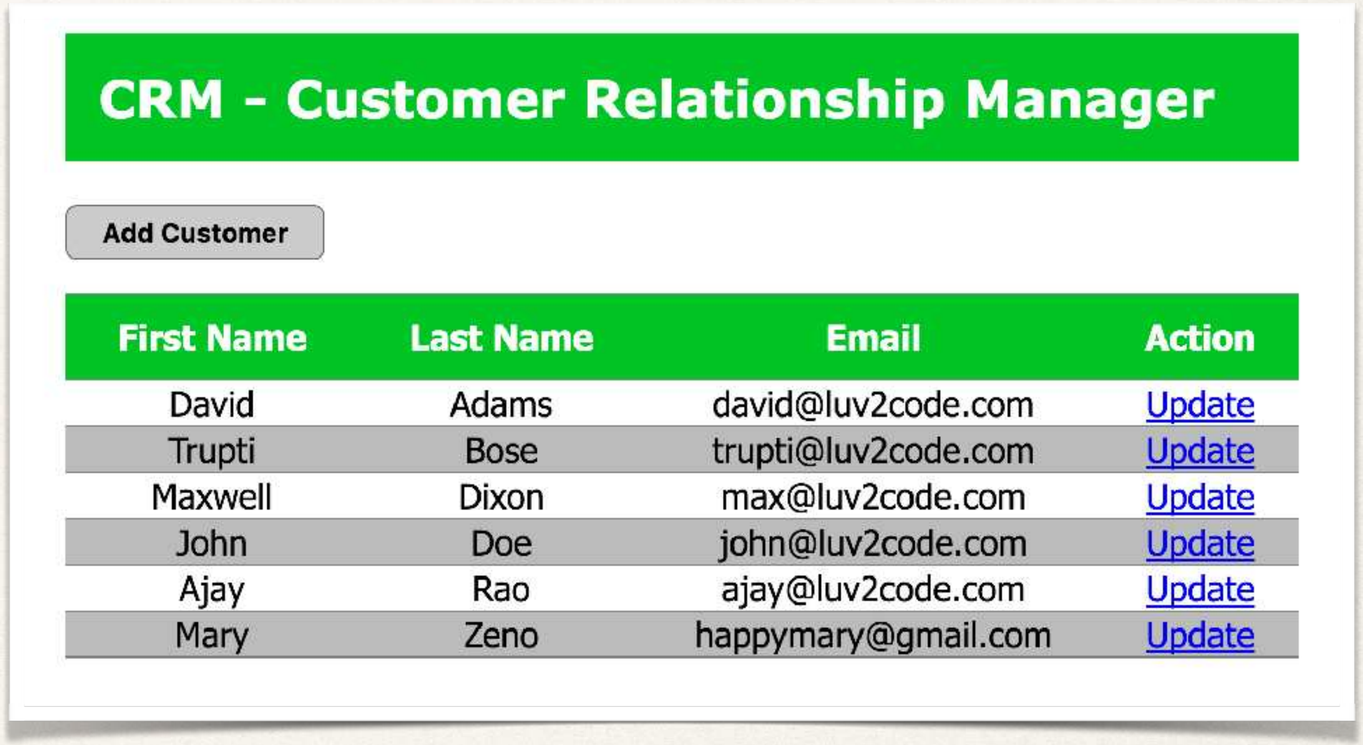
When **clicked**

- will load the customer from database
- prepopulate the form

# Update Customer

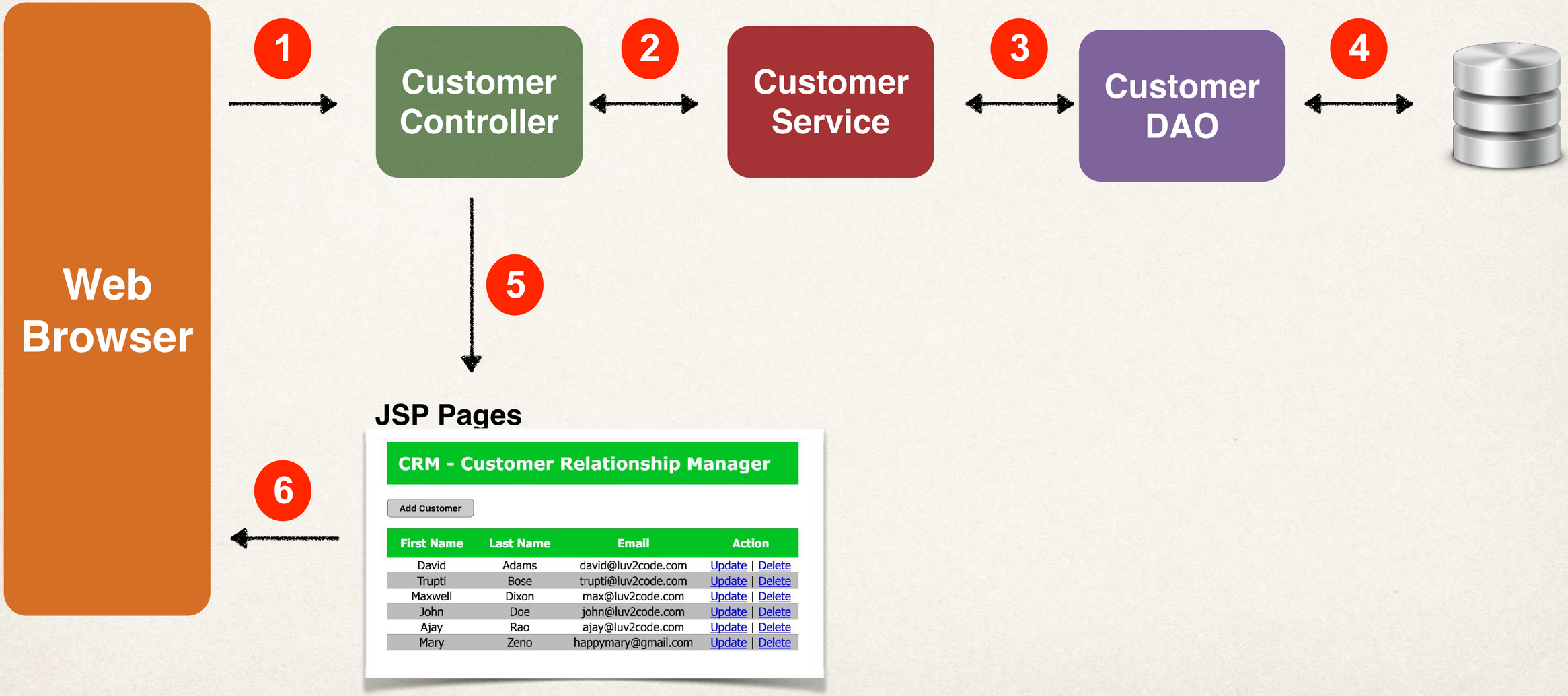
Step-By-Step

1. **Update list-customers.jsp**
    1. New “Update” link
2. **Create customer-form.jsp**
    1. Prepopulate the form
3. **Process form data**
    1. **Controller > Service > DAO**

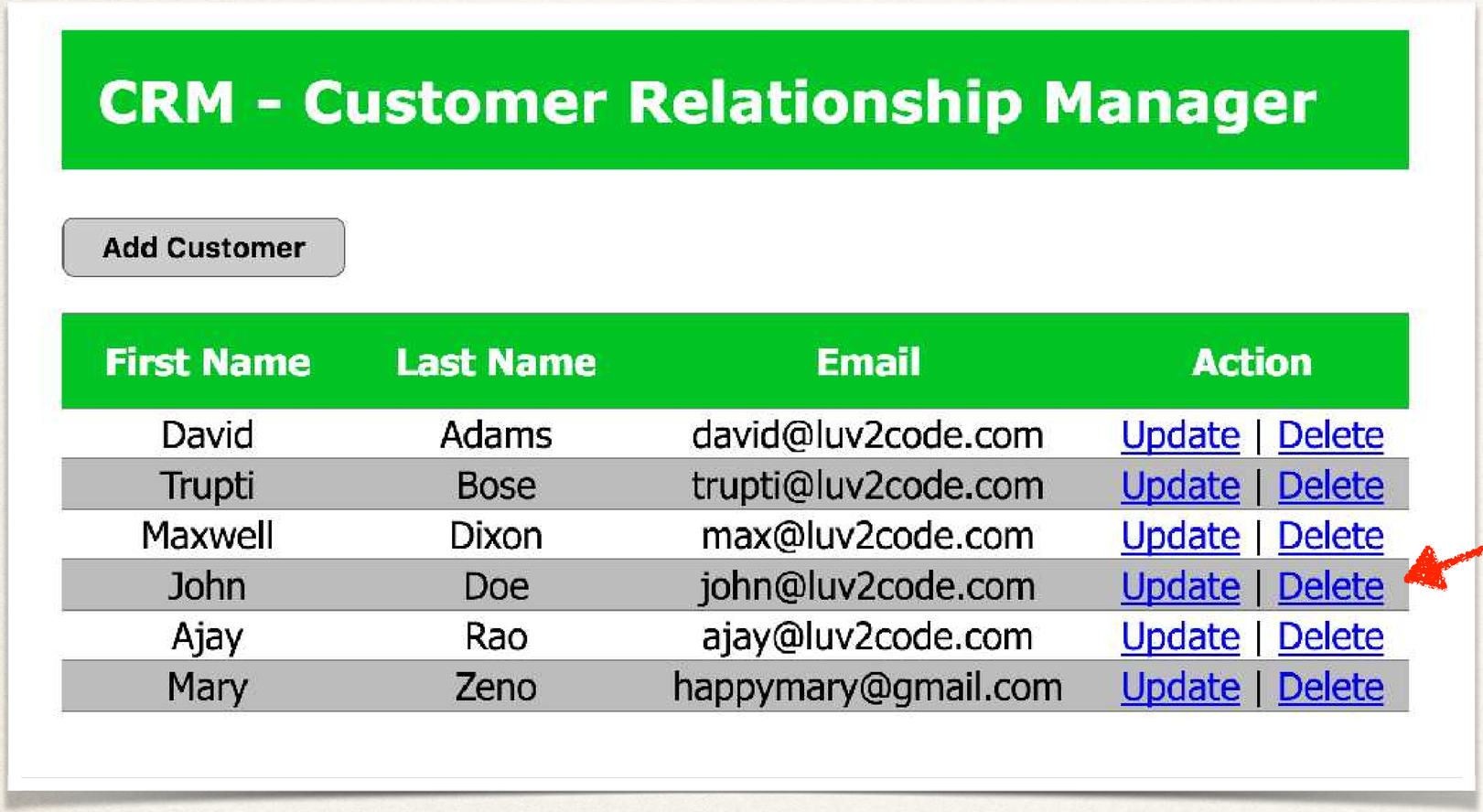


# Delete Customer

# Big Picture


Web Browser
1
Customer Controller
2
Customer Service
3
Customer DAO
4
5
JSP Pages
6
CRM - Customer Relationship Manager
Add Customer
First Name
Last Name
Email
Action
David
Adams
david@luv2code.com
Update | Delete
Trupti
Bose
trupti@luv2code.com
Update | Delete
Maxwell
Dixon
max@luv2code.com
Update | Delete
John
Doe
john@luv2code.com
Update | Delete
Ajay
Rao
ajay@luv2code.com
Update | Delete
Mary
Zeno
happymary@gmail.com
Update | Delete

# Action Link - Delete




Each row has a **Delete** link

- current customer id embedded in link

When **clicked**

- prompt user
- will delete the customer from database

# Delete Customer

Step-By-Step

1. Add “Delete” link on JSP

2. Add code for “Delete”

    1. Controller > Service > DAO