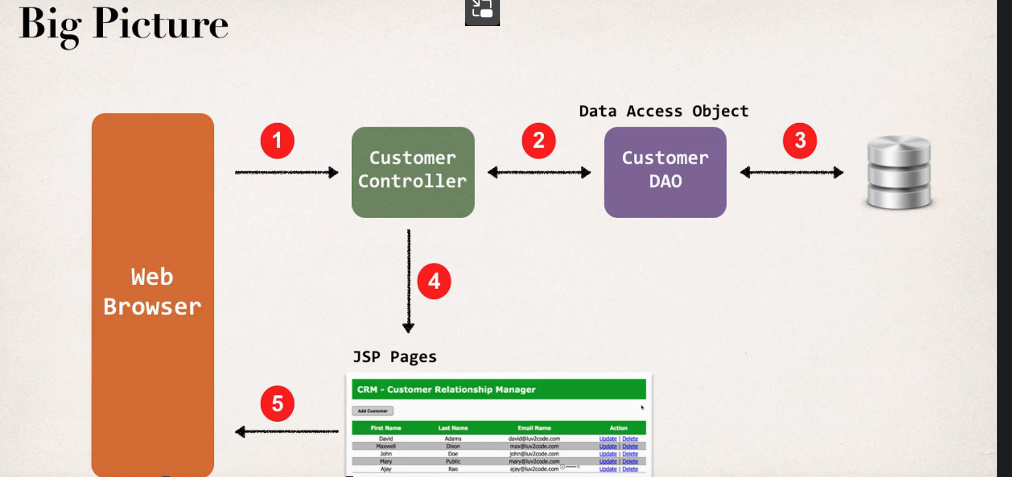
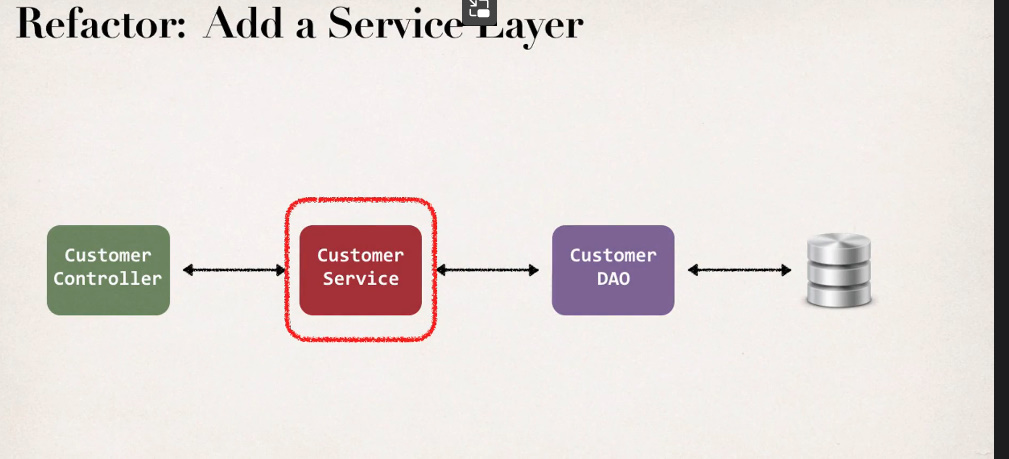
* **Service Facade** design pattern
* Layer intermediar pentru custom bussines logic
* Integreaza date din multiple surse(DAO/repozitorii)

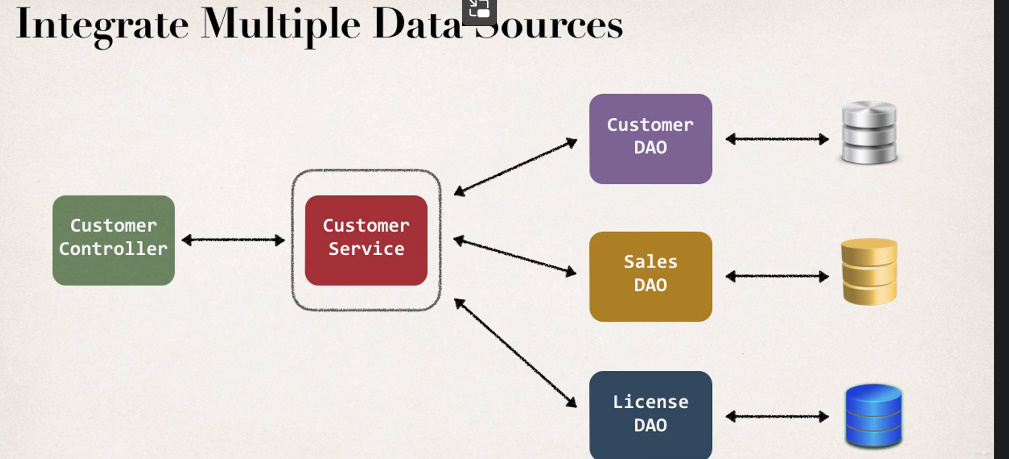


Service Controller sta intre Controller si DAO.



**Utilizare**

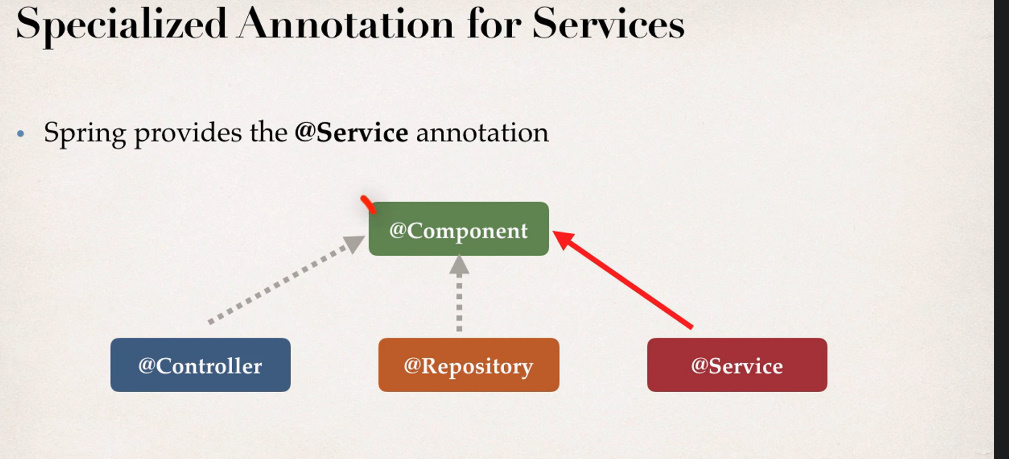
De ex, putem sa avem nevoie nu doar de datele clientului, dar de mai multe date din baza de date, ca produsele cumparate, licentele etc.



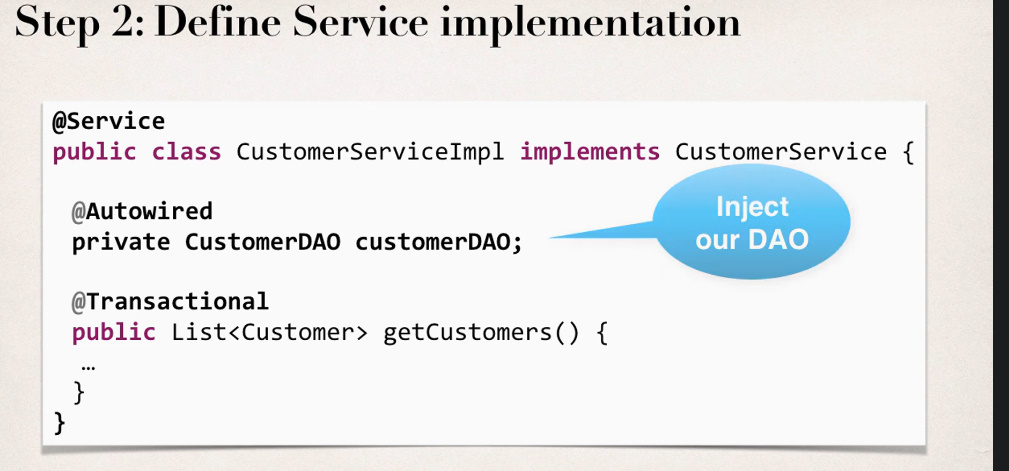
Scopul la Service Layer este de a extrage toate aceste date si de a le pune impreuna intr-un view, ca Controllerul sa il poata folosi.

**@Service**

* Pe langa @Controller si @Repository, care sunt de tip @Component, mai exista si @Service, care tot e de tip @Component.
* **Service e folosit pentru a utiliza mai multe DAO si a efectua tranzactii intre ele**

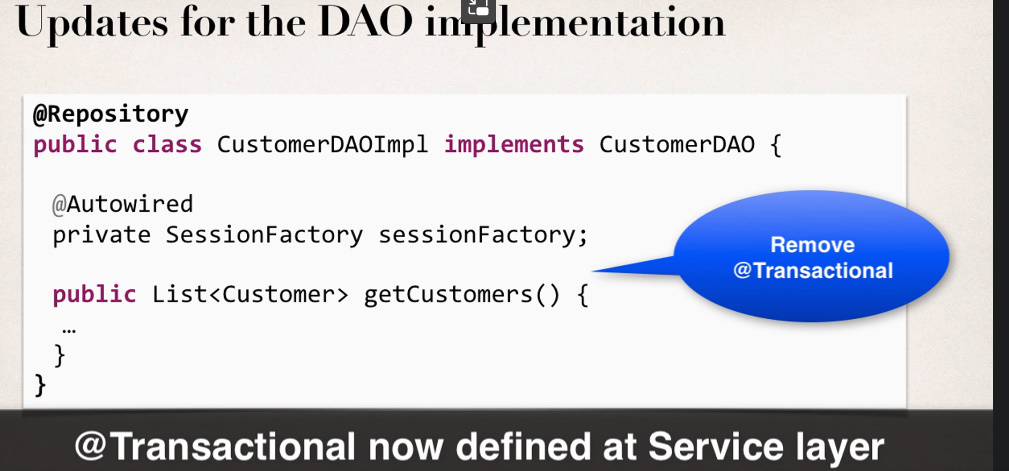


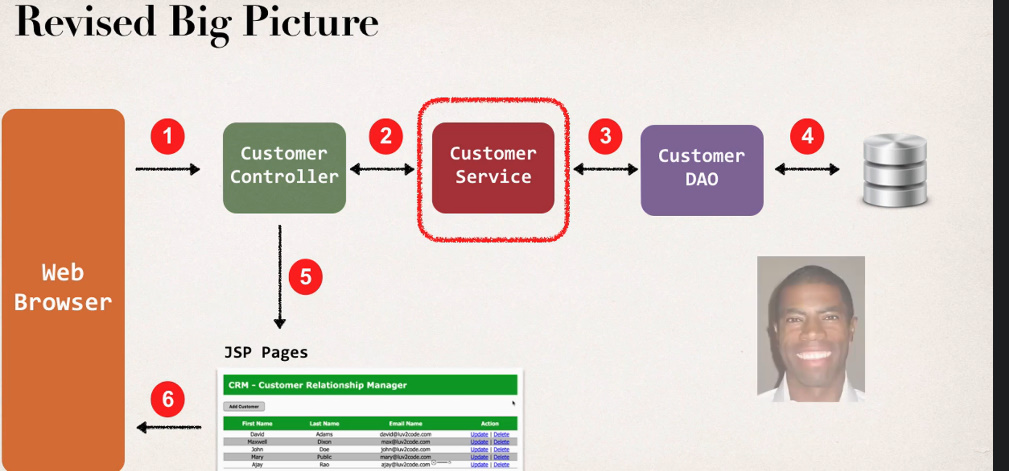
* Anotatia face ca Spring sa inregistreze automat Service ca bean, datorita la component scanning
* Service este o **interfata** deobicei.
* Trebuie sa injectam in Service DAO



CustomerDAO tot are o metoda getCustomers() cu @Transactional, dar si Service are, caci el va fi cel care va folosi metoda cu acest nume din customerDAO, si pe aceasta care e a lui, pentru a mai face ceva. Deci, **eliminam @Transactional din CustomerDAOImpl**, si lasam Service sa se ocupe de tranzactie, caci metoda din CustomerDAOImpl va folosi tranzactia creata de Service in metoda.

Asta si e rolul la Service, el sa genereze Tranzactia si DAO sa ruleze in contextul lui.





CustomerService:

import java.util.List;  
  
public interface CustomerService {  
 public List<Customer> getCustomers();  
}

Implementarea ei

@Service  
public class CustomerServiceImpl implements CustomerService{  
  
 @Autowired  
 private CustomerDAO customerDAO;  
  
 @Override  
 @Transactional  
 public List<Customer> getCustomers() {  
 return customerDAO.getCustomers();  
 }  
}

Insa trebuie sa modificam Controllerul, asa cum vom folosi deja Service, nu DAO

@Controller  
@RequestMapping("/customer")  
public class CustomerController {  
  
 @Autowired  
 private CustomerService customerService;  
  
 @GetMapping("/list")  
 public String listCustomers(Model model){  
List<Customer> customerList = customerService.getCustomers();  
model.addAttribute("customers",customerList);  
 return "list-customers";  
 }  
}

**De ce folosim Service?**

* Service ne permite sa lucram cu mai multe DAO intr-o singura tranzactie.
* Un DAO ar putea avea mai multe metode, si unele ar putea trebui rulate impreuna mereu. In DAO ele ar fi in tranzactii aparte, dar daca una esueaza, cealalta nu va esua. De ex, fie ca avem:

BankDAO

transferMoney();

updateDeposit();

Acum, cand transferam bani, trebuie sa actualizam depozitul, insa daca de ex updateDeposit va esua, nu trebuie ca transferul sa se faca, insa daca am lucra cu DAO, metodele au tranzactie separata, si esuarea la una nu va duce la oprirerea commitului la alta. Aici Service creaza o tranzactie pentru ambele metode, caci doar le vom folosi intr-o metoda de la Service ce are o tranzactie doar.

* Un Service ne permite chiar si sa lucram cu DAO din difertie baze de date, cum ar fi de ex Customers si Employees. Chiar daca am vrea pur si simplu sa le afisam, anume pe ambele, daca apare eroare la afisarea unuia, nu se mai afiseaza celalalt
* Cu Service cream mult mai putine tranzactii

**redirect**

* Daca dorim ca metoda noastra de Request sa nu trimita o pagina JSP, ci un alt request, atunci la return pur si simplu scriem “redirect:/../”;

De ex:

@PostMapping("/saveCustomer")  
public String saveCustomer(@ModelAttribute("customer")Customer customer){  
 return "redirect:/customer/list";  
}

**Cum ramane cu Session**

* Service nu creaza obiectele Session, ele se creaza in DAO, insa metode ce are @Transactional se va asigura ca toate DAO executate din ea sa aiba aceeasi sesiune. Adica beanFactory va returna pentru toate aceeasi sesiune, exact asa cum creaza un trasnaction pentru toate.
* session.getCurrentSession() nu spune ca se creaza o sesiune noua, ci se obtine cea curenta!