

# Web Frameworks: Spring

An Introduction

# Introduction

- Web.xml routes requests to the individual servlet's doGet or doPost methods
- doGet(...)
  - //extract parameters from request

```
@WebServlet("/SelectCoffeeMVC.do")
public class CoffeeSelectMVC extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

        String color = request.getParameter("color");
        String addOn=request.getParameter("addOns");
        if(!color.equals("") && !addOn.equals("")) {
            Coffee c=new Coffee(color, addOn);

            Cookie ck1;  HttpSession session=request.getSession();

            CoffeeExpert ce = new CoffeeExpert();
            String result="";

            try{
                Connection con=(Connection)getServletContext().getAttribute("key2");
                result = ce.getBrands(c,con);
            }catch(Exception e){ System.out.println(e);}

            request.setAttribute("brands", result);
            RequestDispatcher view = request.getRequestDispatcher("result.jsp");
            view.forward(request, response);

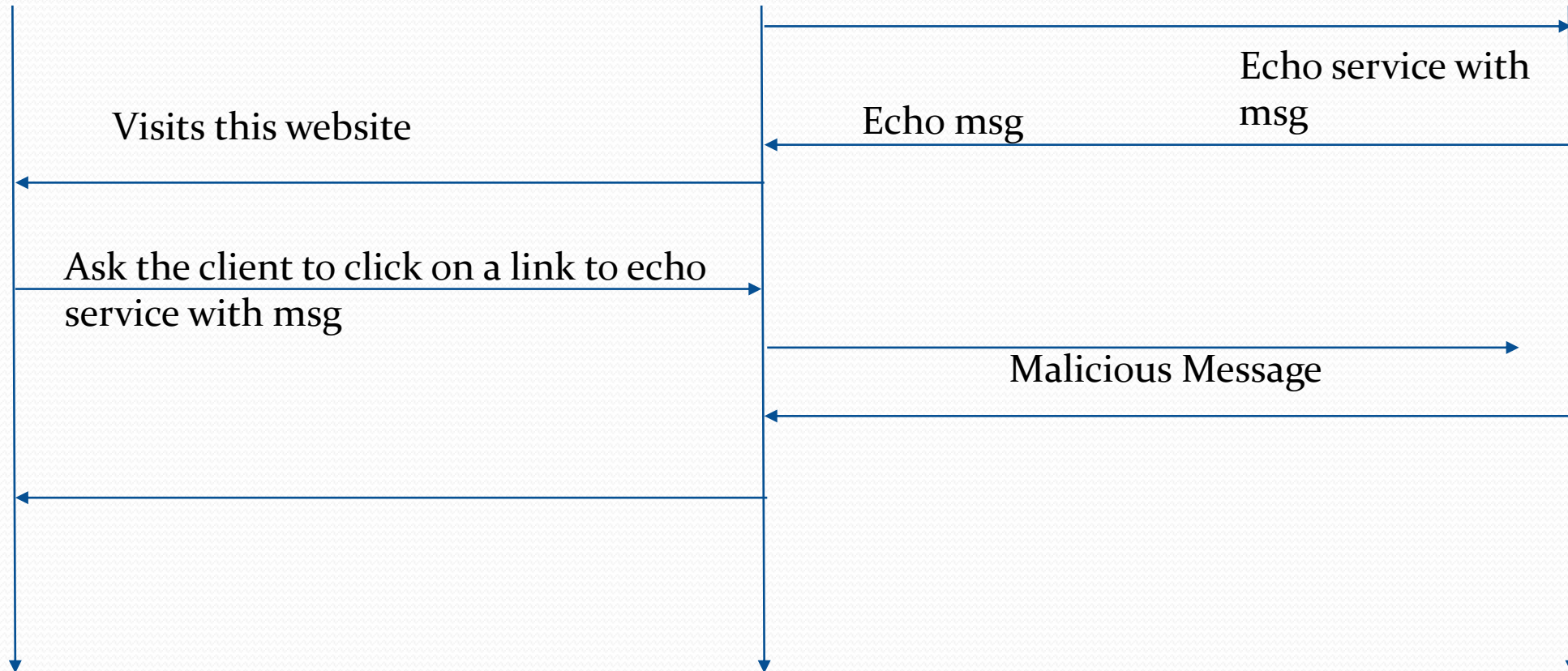
        }
    }
}
```

# Injection Attack

Server 2

Client

Server 1



# Introduction

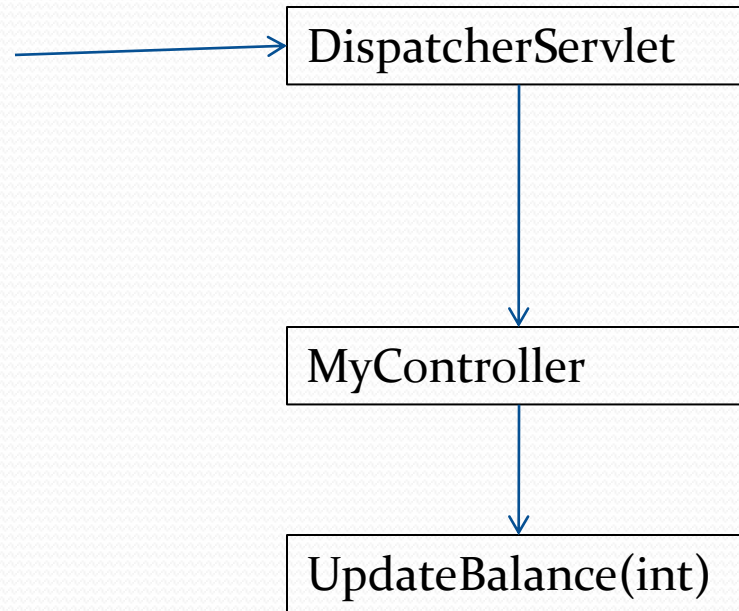
- Web.xml routes requests to the individual servlet's doGet or doPost methods
- doGet(...)
  - //extract parameters from request
  - //validation
  - //Construct objects with parameters
  - //do the processing

# Spring

- In Spring
  - A specialized servlet-DispatcherServlet
  - One or more controllers having simple methods to process HTTP requests
  - The DispatcherServlet routes requests to appropriate controller-individual methods of the controllers
  - DispatcherServlet extracts request parameters, performs data validation and marshalling
  - Provides an extra layer of routing over web.xml

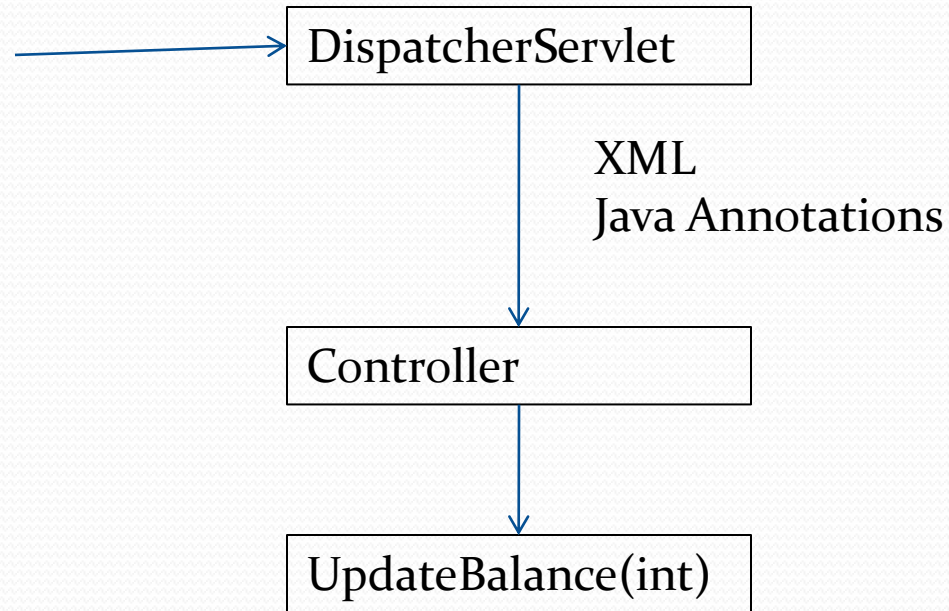
```
public class MyController {  
    Integer UpdateBalance(int) {  
        ...  
    return ...  
}
```

```
doGet(...)  
    //extract parameters from  
    request  
    //validation  
    //Construct objects with  
    parameters  
    //do the processing
```



# Spring

Spring Controllers are plain java objects  
No special interfaces to be implemented or classes to be inherited



- ☐ Routing is possible based on Path like servlets
- ☐ Request parameters using annotations
- ☐ Data validation is taken care of



## Routing through DispatcherServlet

```
public class ContactController {  
  
    public Contacts getContacts() {  
        //retrieve contacts  
        Contacts c=...  
        ...  
        return c;  
    }  
}
```

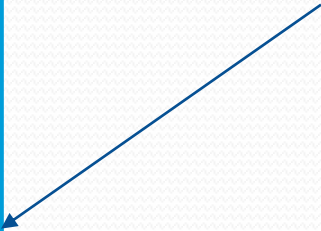
A Simple java class-no  
framework code



# Routing through DispatcherServlet

```
public class ContactController {  
    public Contacts getContacts() {  
        //retrieve contacts  
        Contacts c=...  
        ...  
        return c;  
    }  
  
    public void friends(){  
        ...  
    }  
}
```

A Simple java class-no  
framework code



# Introduction

- In EJB *public class HelloWorldBean implements SessionBean {*
- Spring avoids (as much as possible) littering your application code with its API
- Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class
- Instead, the classes in a Spring-based application often have no indication that they're being used by Spring
- Spring has enabled the return of the plain old Java object (POJO) to enterprise development

## Mapping Request parameters to method parameters

@Controller

```
public class ContactController {
```

@RequestMapping("/search")

```
    public Contacts searchContacts(
```

```
        String SearchStr) {
```

```
        //retrieve contacts
```

```
        Contacts c=...
```

```
        ...
```

```
        return c;
```

```
    }
```

Retrieves request parameters and performs basic data validation so that value of *searchstr* can be mapped to *SearchStr*

## Mapping Request parameters to method parameters

**@Controller**

```
public class ContactController {
```

```
    @RequestMapping(value={"/search"}, method = RequestMethod.GET)
```

```
    public Contacts searchContacts(
```

```
        @RequestParam searchstr String SearchStr) {
```

```
        //retrieve contacts
```

```
        Contacts c=...
```

```
        ...
```

```
        return c;
```

```
    }
```


# Mapping Requests

```
@RestController
@RequestMapping(value = "/demo")
public class LoginController {

    @RequestMapping(value = "/login")
    public String sayHello1() {
        return "Hello World ";
    }

    @RequestMapping(value = "/dummy")
    public String sayHello() {
        return "Hello World dummy";
    }

}
```

- 
- No need to worry about
    - how that request got to the server,
    - what format it got there in,
    - how all the data got extracted from it.
  - It simplifies the methods and write cleaner, simpler methods, by using request parameters in the request mapping to extract that data and pass it into the method

@Controller

```
public class ContactController {
```

@RequestMapping("/search/{str}")

```
    public Contacts searchContacts(  
        Search s) {
```

```
        //retrieve contacts
```

```
        Contacts c=...
```

```
        ...
```

```
        return c;
```

```
    }
```

Path variable provides a nicer way of parsing the request parameters rather than  
?<key>=value



@Controller

```
public class ContactController {
```

```
@RequestMapping("/search/")
```

```
    public Contacts searchContacts(
```

```
        Search s) {
```

```
        //retrieve contacts
```

```
        Contacts c=...
```

```
        ...
```

```
        return c;
```

```
}
```

```
public class Search {
```

```
    private String fname;
```

```
    private String lname;
```

```
    public String
```

```
    getFname() {..}
```

```
    public setFname(String  
        name) {..}
```

```
    ...}
```

Automatic data marshalling  
through HTTP message  
converters



## Import Getting Started Content



Type pattern to match

Messaging Stomp Websocket  
Multi Module  
Producing Web Service  
Reactive Rest Service  
Relational Data Access  
Rest Hateoas  
Rest Service  
Rest Service Cors

Building a RESTful Web Service :: Learn how to create a RESTful web service with Spring.

Build Type

☒ Maven

☐ Gradle

☐ General

Code Sets

☒ initial

☒ complete

Home Page

<https://spring.io/guides/gs/rest-service>

☒ Open



Finish

Cancel

# Key Features of Spring

- Inversion of Control
  - Inversion of Control (or IoC) covers a broad range of techniques that allow an object to become a passive participant in the system
  - IoC is a software engineering principle that transfers control over objects or parts of a system to a container.
  - It is most commonly used in the context of object-oriented programming.
  - The IoC allows the framework to take control of the program execution flow and sends calls to the written code.
  - The benefits of using IoC would be:
    - easier transition between different implementations,
    - greater modularity of the program,
    - easier testing of the program by isolating its components.
- Dependency Injection
- Aspect Oriented Programming

# Inversion of Control

```
public class BankAccount {  
    public void transfer(BigDecimal amount, BankAccount recipient) {  
  
        recipient.deposit(this.withdraw(amount));  
    }  
  
    public void closeOut() {  
  
        this.open = false;  
    }  
  
    public void changeRates(BigDecimal newRate) {  
  
        this.rate = newRate;  
    }  
}
```

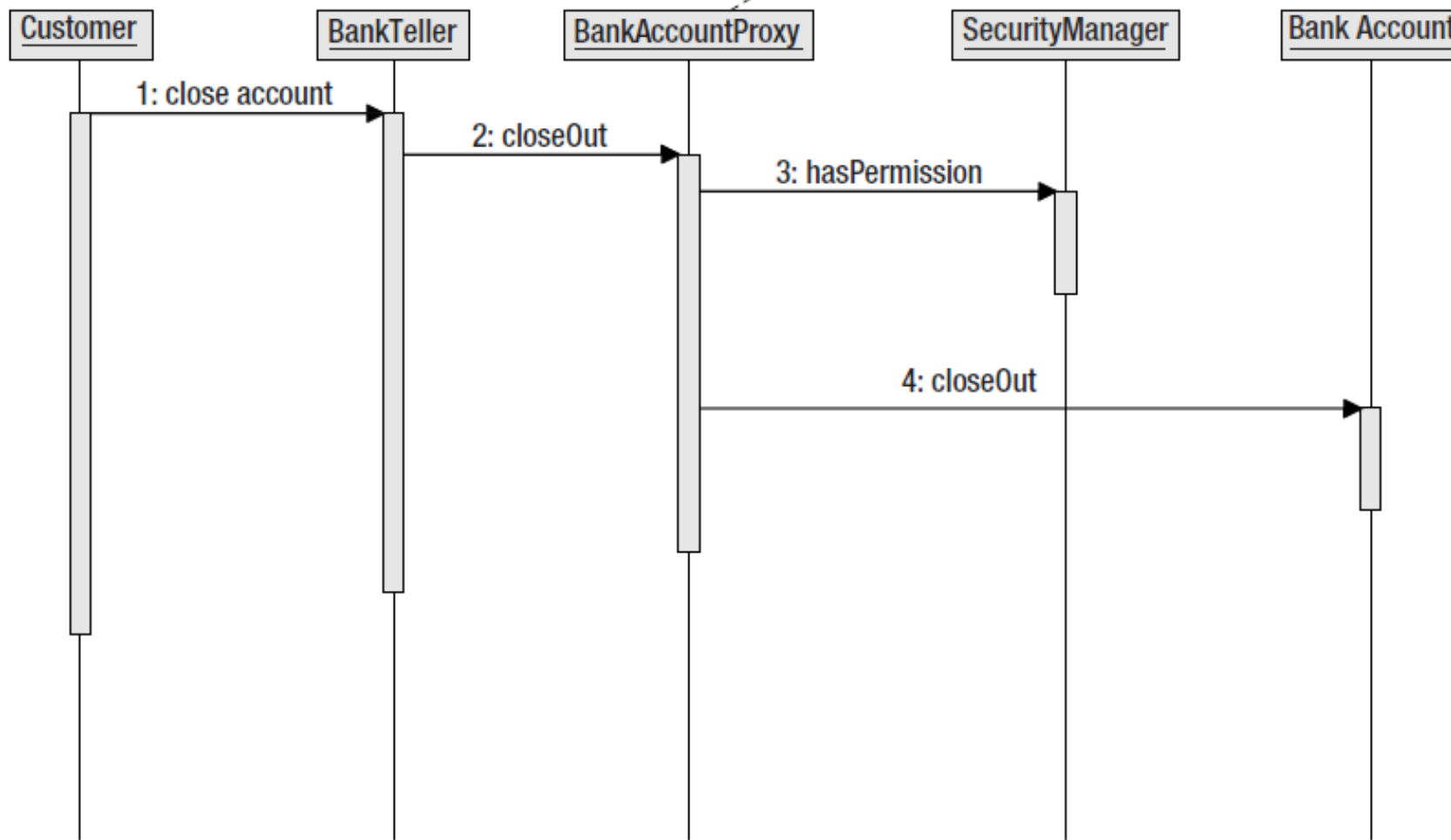
# IoC Enabled code

```
public class BankAccount {  
    public void transfer(BigDecimal amount, BankAccount recipient) {  
        recipient.deposit(this.withdraw(amount));  
    }  
  
    public void closeOut() {  
        this.open = false;  
    }  
  
    public void changeRates(BigDecimal newRate) {  
        this.rate = newRate;  
    }  
}
```

- You can add the authorization mechanism into the execution path with a type of IoC implementation called *aspect-oriented programming (AOP)*

# AOP

- Aspects are concerns of the application that apply themselves across the entire system
- The SecurityManager is one example of a system-wide aspect, as its hasPermission methods are used by many methods
- Other typical aspects include logging, and auditing of events
- While we may have removed calls to the SecurityManager from the BankAccount, the deleted code will still be executed in the AOP framework either at compile time or at runtime
- Runtime-proxy based AOP (simple)
- Compile time- weaving (stronger than proxies)
- While we may have removed calls to the SecurityManager from the BankAccount, the deleted code will still be executed in the AOP framework.
- The beauty of this technique is that both the domain model (the BankAccount) and any client of the code are unaware of this enhancement to the code.



Inversion of Control is the broad concept of giving control back to the framework  
This control can be control over creating new objects, control over transactions, or control over the security implementation

```
public interface PriceMatrix {  
    public BigDecimal lookupPrice(Item item);  
}
```

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix = new PriceMatrixImpl();  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```



# Problems

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix = new PriceMatrixImpl();  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

- Every instance of CashRegisterImpl has a separate instance of PriceMatrixImpl
  - With heavy services (those that are remote or those that require connections to external resources such as databases) it is preferable to share a single instance across multiple clients
- The CashRegisterImpl now has concrete knowledge of the implementation of PriceMatrix
  - CashRegisterImpl has tightly coupled itself to the concrete implementation class
- One of the most important tenets of writing unit tests is to divorce them from any environment requirements
- The unit test itself should run without connecting to outside resources

Don't ask for  
the resource;  
I'll give it to  
you

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix = new PriceMatrixImpl();  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix;  
  
    public setPriceMatrix(PriceMatrix priceMatrix) {  
        this.priceMatrix = priceMatrix;  
    }  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

# Dependency Injection

- Dependency Injection is a technique to wire an application together without any participation by the code that requires the dependency.
- The client usually exposes setter methods so that the framework may inject any needed dependencies.
- By moving the dependency out of the client object, it is no longer solely owned by CashRegisterImpl, and can now easily be shared among all classes.
- The client also becomes much more testable.
- The client has no environment-specific code to tie it to a particular framework.
- DispatcherServlets route to Controllers and controllers depend on certain objects

# DI Example

- The IoC container is in charge of creating objects, connecting them, configuring them and managing their entire life cycle from creation to destruction
- Spring Container uses DI to manage the components that make the application.
- These objects are named *Spring beans*.
- In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- Spring IoC Container defines the rules by which beans work.
- Bean is pre-initialized through its dependencies.
- After that, bean enters the state of readiness to perform its own functions.
- Finally, the IoC Container destroys bean

## DI in Spring

- Beans are defined to be deployed in one of two modes:
  - singleton or
  - non-singleton.
- When a bean is a singleton, which is a default mode for bean's deployment, only one shared instance of the bean will be managed
  - all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned.
- The non-singleton, prototype mode of a bean deployment, results in the creation of a new bean instance every time a request for that specific bean occurs.

# DI example

- Within a Controller
- `@Autowired`
- `CarService service;`

```
public interface VehicleService {  
    public String transport();  
}
```

```
@Service  
public class CarService implements VehicleService {  
    @Override  
    public String transport() {  
        return "Car transport";  
    }  
}
```

# Managing Beans

Spring IoC container does the following

- to create a bean of *CarService*
- Assign the bean to the controller and add it to the *service* property
- To make an instance of any class that should be managed by Spring, it is necessary to add the annotation *@Component* over this particular class.
- This way Spring can manage this dependence, which means that Spring detects this as a bean, or an object managed by the Spring IoC Container.
- Spring *@Service* annotation is a specialization of *@Component* annotation. It is used to mark the class as a service provider
- The *@Autowired* annotation is injecting *CarService* object into the property named *service*.

```
public interface VehicleService {  
  
    public String transport();  
  
}
```

- Within a Controller
- @Autowired
- VehicleService service;
- When loaded, Spring will start looking for implementations of this interface, and since *CarService* class implements *VehicleService* interface, Spring will find this implementation and it will inject instance of appropriate class.

```
@Service  
public class CarService implements VehicleService {  
  
    @Override  
    public String transport() {  
        return "Car transport";  
    }  
  
}
```

```
@Service  
public class TruckService implements VehicleService {  
  
    @Override  
    public String transport() {  
        return "Truck transport";  
    }  
  
}
```



# Linking services

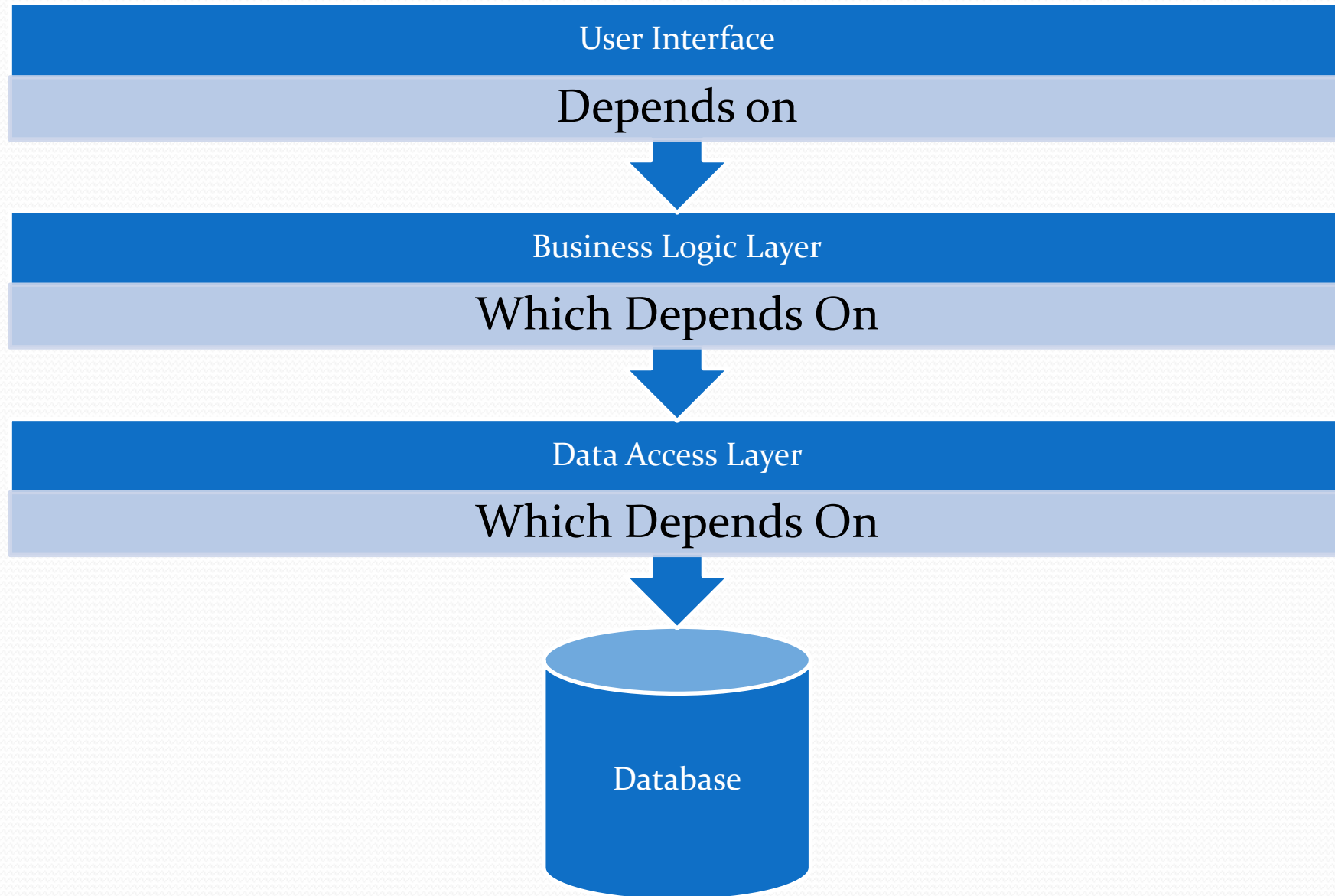
- First approach would be renaming a particular property to *carService*
  - `@Autowired`
  - `VehicleService carService;`
  - Spring detects that the name of the instance is *carService*, and that it is an implementation of *VehicleService*, so Spring is able to inject it
- Another approach in resolving this issue is adding `@Qualifier` annotation and passing the name of the bean that needs to be injected
  - `@Qualifier(value = "truckService")`
- DI can also be implemented through both constructor and set methods
- Performance cost of wiring beans is usually located in the start-up phase of application

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix;  
  
    public setPriceMatrix(PriceMatrix priceMatrix) {  
        this.priceMatrix = priceMatrix;  
    }  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

# What is a “Dependency”?

- Some common dependencies include:
  - Application Layers
    - Data Access Layer & Databases
    - Business Layer
  - External services & Components
    - Web Services
    - Third Party Components

# Dependencies at a Very High Level



# Dependency Injection Pros & Cons

- Pros
  - Loosely Coupled
  - Increases Testability
  - Separates components cleanly
  - Allows for use of Inversion of Control Container
- Cons
  - Increases code complexity
  - Developers learning time increases
  - Can Complicate Debugging at First
  - Complicates following Code Flow

# Application Startup

- An application where configuration is stated
- Controllers
- POJOs
- Views

- `@SpringBootApplication`

- `public class ServingWebContentApplication {`

- `public static void main(String[] args) {`

- `SpringApplication.run(ServingWebContentApplication.class, args);`

- `}`

- `}`

Spring is going to look at our application and the configuration that we're expressing in it, and then automatically, in this case, create a web container put a dispatcher servlet in that web container, and then auto discover all of our controllers in our application

## @SpringBootApplication

- @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
- @ComponentScan: enable @Component scan on the package where the application is located
- @Configuration: allow to register beans in the context or import additional configuration classes


All of your application components (@Component, @Service, @Repository, @Controller etc.) are automatically registered as Spring Beans

## @EnableAutoConfiguration

- ❑ Spring Boot auto-configuration attempts to automatically configure our Spring application based on the jar dependencies Added by the programmer

Spring Boot adds @EnableWebMvc automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

- ❑ Auto-configuration is non-invasive.
- ❑ At any point, you can start to define your own configuration to replace specific parts of the auto-configuration.

- 
- these four annotations on this configuration class go and set up an entire web container
  - they create the `DispatcherServlet` that we need to route requests to our controllers.
  - They automatically scan the appropriate packages that we want, and discover our controllers,
  - they'll automatically configure our controllers with any dependencies that we want them to have.



# ApplicationContext

- If Dependency Injection is the core concept of Spring, then the `ApplicationContext` is its core object.
- The interface `org.springframework.context.ApplicationContext` represents the Spring IoC container
- It extends the `BeanFactory` interface, in addition to extending other interfaces to provide additional functionality in a more *application framework-oriented style*
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata
- Several implementations of the `ApplicationContext` interface are supplied out-of-the-box with Spring
  - such as `ContextLoader` that automatically instantiates an `ApplicationContext` as part of the normal startup process