Service layer vs DAO — Why both?

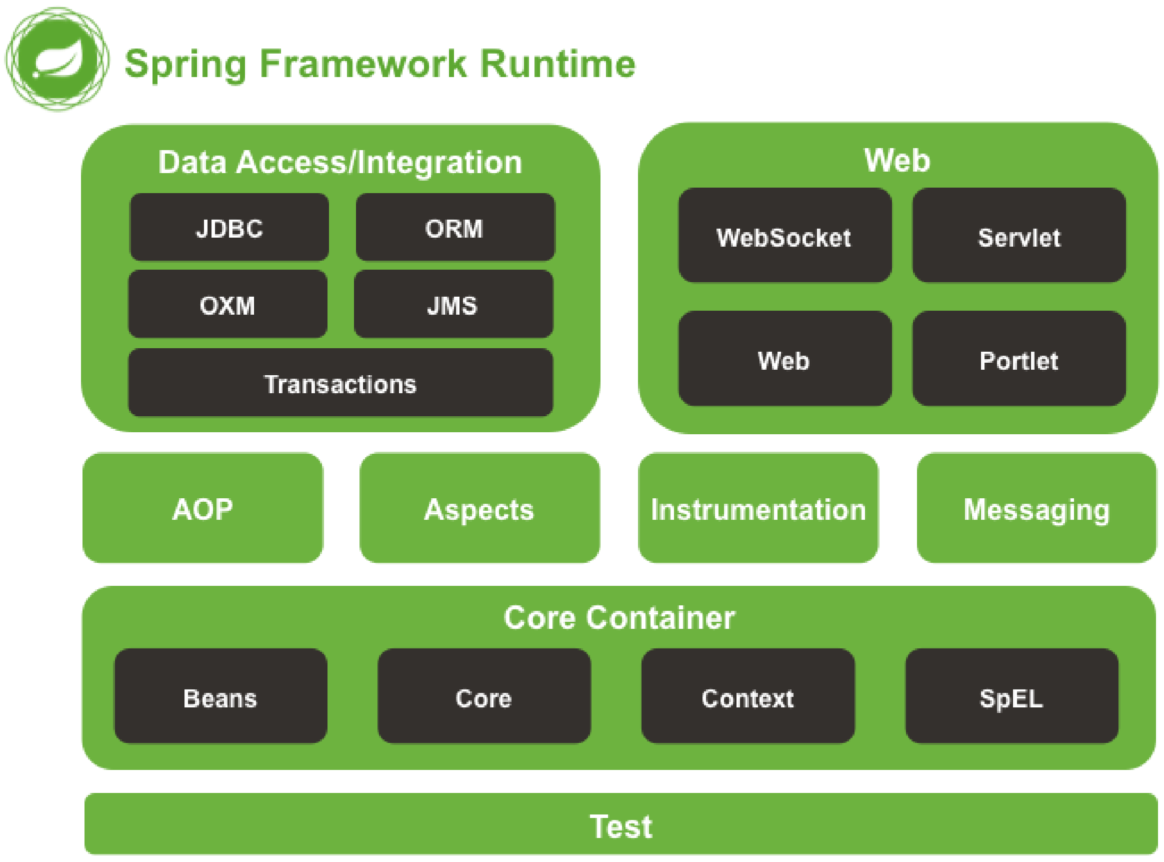
Generally the DAO is as light as possible and exists solely to provide a connection to the DB, sometimes abstracted so different DB backends can be used.

The service layer is there to provide logic to operate on the data sent to and from the DAO and the client. Very often these 2 pieces will be bundled together into the same module, and occasionally into the same code, but you'll still see them as distinct logical entities.
Another reason is security - If you provide a service layer that has no relation to the DB, then is it more difficult to gain access to the DB from the client except through the service. If the DB cannot be accessed directly from the client (and there is no trivial DAO module acting as the service) then all an attacker who has taken over the client can do is attempt to hack the service layer as well before he gets all but the most sanitised access to your data.

Dependency injection

Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the Service Locator pattern.

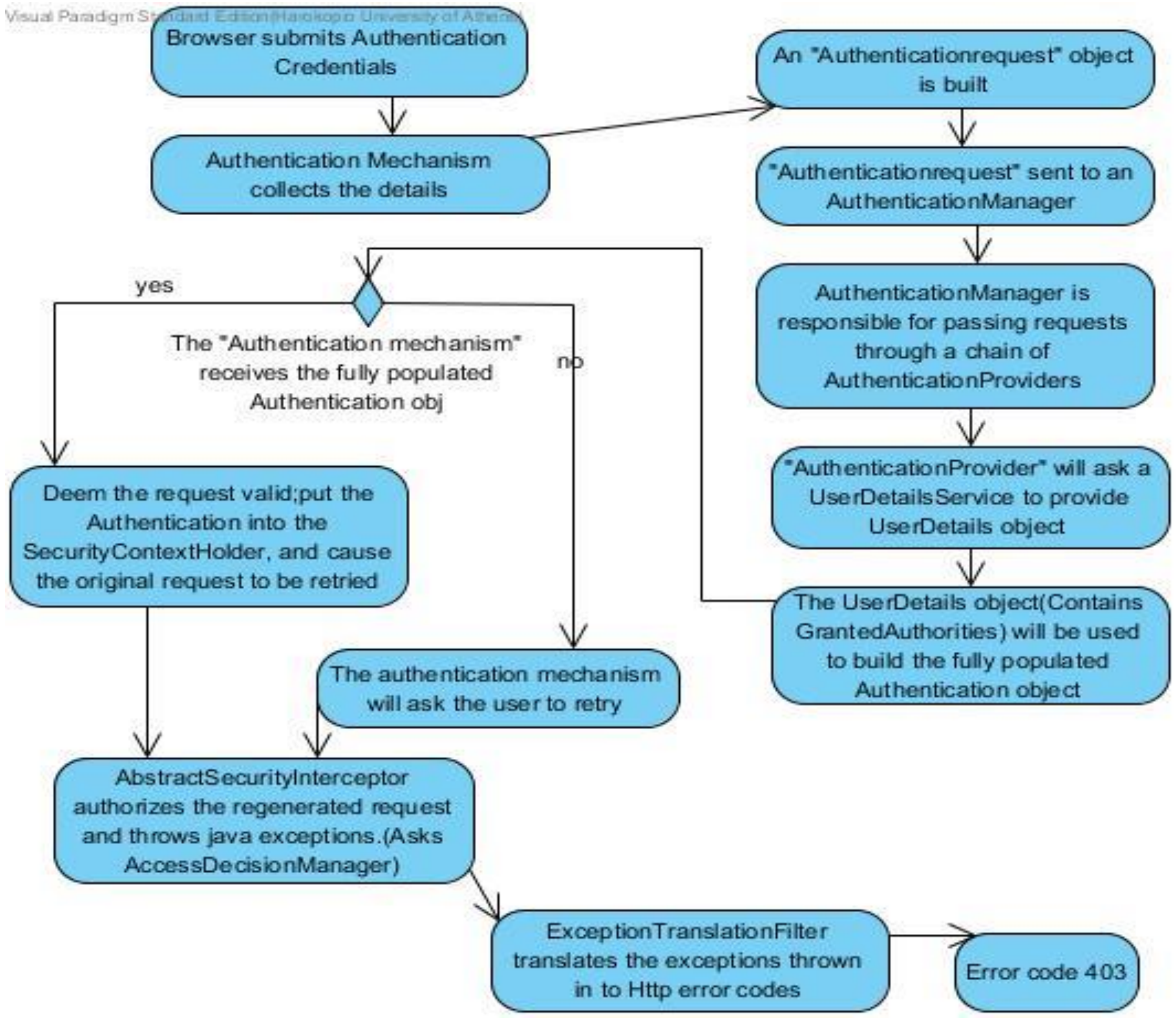
Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

Spring Security

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

The basic flow of an authentication request using the Spring Security system shows the different filters and how they interact from the initial browser request, to either a successful authentication or an HTTP 403 error.


Visual Paradigm Standard Edition(Harokopio University of Athens)
Browser submits Authentication Credentials
Authentication Mechanism collects the details
An "Authenticationrequest" object is built
"Authenticationrequest" sent to an AuthenticationManager
AuthenticationManager is responsible for passing requests through a chain of AuthenticationProviders
"AuthenticationProvider" will ask a UserDetailsService to provide UserDetails object
The UserDetails object(Contains GrantedAuthorities) will be used to build the fully populated Authentication object
yes
The "Authentication mechanism" receives the fully populated Authentication obj
no
Deem the request valid;put the Authentication into the SecurityContextHolder, and cause the original request to be retried
The authentication mechanism will ask the user to retry
AbstractSecurityInterceptor authorizes the regenerated request and throws java exceptions.(Asks AccessDecisionManager)
ExceptionTranslationFilter translates the exceptions thrown in to Http error codes
Error code 403

In this Project a custom UserDetailServiceImplementation is used to pass the available services of a role to the GrantedAuthorities.
The url paths are intercepted based on these services.
And finally a CustomSuccessAuthenticationHandler redirect the user based on his available services.