Decir a spring los objetos dependientes y q dependen. puede ser con xml o con anotaciones o como javaconfig, qe es la más usada

Declarar todos los objetos como beans. Para que se encuentren en el contenedor y los gestione Spring

Por defecto spring las mete en contenedor con el nombre de la clase con minúscula, se puede cambiar depues del Component

Con el @Resource agregamos la dependencia de que clase es. Es una inyección esplicita

La carpeta config debe llevar el xml con los packages q quieres incluir. Y esa carpeta config, en el buildpath hay que agregarlo con Add Class

En apps web no hace falta lanzar spring, lo haces desde configuración. En los de escritorio si

Se estarán creando tantos objetos como **Components** haya. Se pueden marcar como **@lazy** para que no lo instancie de primeras al arrancar y no use memoria. Los instancia cuando se usen directamente

Se puede poner en el config para que todos por defecto arranquen como lazy y no tener q ponerlo en todos

En vez de resource, se puede usar **Autowired** (tú te lo guisas tú te lo comes). Buscaría el tipo por el mismo. Pero como tenemos una interfaz, no sabe cuál de las 3 pillar.

Cuando tiene varios constructores la clase, hay que ponérselo al constructor que quieres por defecto. El objeto que reciba en el constructor lo cogerá del contenedor

Lo mismo con los setter, el objeto q reciban será el q tenga spring en el contenedor

Se puede hacer para cualquier método en realidad, se invoca una vez se ha instanciado el objeto. Después del constructor se ejecutaría al arrancar

Si lo hacemos para un atributo de array, meterá todas las clases q implemeten la interfaz

En un Map la clave q mete es el name que le has puesto al Component.

Puedes definir una de las clases como **@Primary** y es la de defecto que cogerá.

Docu:

<https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/>

El XML se puede sustituir por clases de configuración: @Configuration

Cualquier método con @Bean, se lanza al principio y el objeto que devuelve ese método lo mete en el contenedor

*@Configuration*

**public** **class** MovieConfiguration {

*@Bean*

**@Primary**

**public** MovieCatalog firstMovieCatalog() { ... }

*@Bean*

**public** MovieCatalog secondMovieCatalog() { ... }

*// ...*

}

**@PostConstruct and @PreDestroy**

¿Ejecuta antes de eliminar el objeto de memoria?

**Ambitos de vida, scopes**

En escritorio hay dos, el singleton (una única copia del vean, compartida).

O prototype (se creará una nueva copia cada vez q se inyecte)

En WEB tiene todos los demás

| **Scope** | **Description** |
| --- | --- |
| [singleton](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton) | (Default) Scopes a single bean definition to a single object instance per Spring IoC container. |
| [prototype](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-request) | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| [session](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-session) | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| [globalSession](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-global-session) | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a Portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |
| [application](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/beans.html#beans-factory-scopes-application) | Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext. |
| [websocket](https://docs.spring.io/autorepo/docs/spring-framework/4.3.7.RELEASE/spring-framework-reference/html/websocket.html#websocket-stomp-websocket-scope) | Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext. |

**Día 2**

@Component puede ser:

* Controller (atiende a peticiones http)
* Service (capa de servicio o de negocio). Así sabe q pude iniciar un contexto transaccional, commit, rollback…
* Repository: DAO. Excepciones chequeadas y se convierte en no chequeadas.
  + Capturar en try y relanzar una propia o una que no se requiere capturar

Capas MVC

Controller: Centralizar la recepción de todas las peticiones. Seguridad, validación, control de usuarios, conversión

Vista: la representación, html, jsp..

Modelo: datos que van a ir a memoria Objetos con datos

Service/negocio:

DAO (modelo extendido)

**Spring y DB**

JdbcTemplate: recubre objetos de JDBC

Un DataSource: representación de la DB. Ecanpsulado por el template. Si lanza una excepción chequeada, el de arriba la transforma en no chequeada.

**Día 3**

Perfiles: tener una clase de conf diferente para cada tipo de perfil.

Puede haber uno de desarrollo, que instancie los mocks, otro de producción para los jdbc por ej…

Se pone el **@Profile(“nombre”)** para la clase o a nivel de método.

Al recoger el contexto le haces un setProfile para poner cuál es el activo en ese momento, y luego le registras todos los .class de configuración que tengas. Se usará el que le hayas activado.

Transacciones: **@EnableTransactionManagement**

Si marco un método como transactional o si marco una clase, es para todos sus métodos

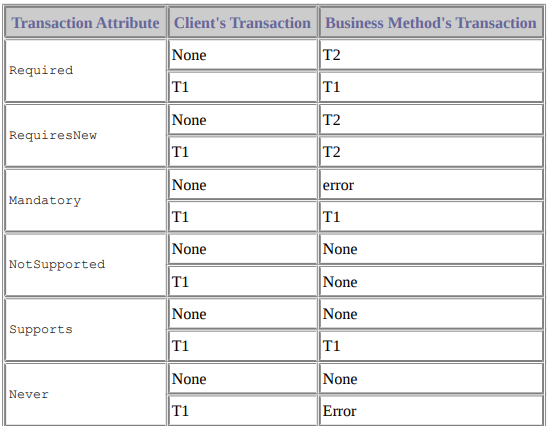
Spring mira el atributo de propragación. Si no pones nada, es required.

Si marco algo como transaction tengo que enchufar un Bean al contenedor de tipo DataSourceTransactionManager por ejemplo, para saber gestionarlas.

@Transactional(propagation=Propagation.REQUIRES\_NEW)

// cuando se invoque, inicia siempre una nueva transacción, haya una iniciada o no

Si lo pongo a nivel de clase ya es otro tema. Puede haber transacciones anidadas, desde arriba la tengo a nivel de clase, pero luego dentro a nivel de método



Puede haber transacciones que se abren y cierran dentro de otra superior iniciada, por ejemplo. En el NegocioDAO lo pongo a nivel de clase, y abre una siempre q se llame al método de insertEmpleados, pero luego dentro de cada insert he puesto otra, que se cierra después de insertar cada uno. Con lo que entran los datos. Hasta que uno falla y sube la excepción a la de arriba y cierra con rollback y no entran más.

Required es la más usada, si no existe una transacción previa, se crea, si no la usa.

New siempre se crea una nueva

**Dao con ORM Hibernate**

Seguir pensando en objetos invocando métodos y él traduce internamente a tipos de DB.

Se puede configurar por XML o anotaciones.

Marcar clases del modelo como @Entity, y le agregamos @Table para que sepa unir.

Con @Id en el atributo le indicamos la pK

Si no quiero mapear algún atributo le pongo @Transit

Si no coinciden los nombres de los atributos con los de las columnas tengo que ponerle @Column con el nombre asociado.

**Día 4**

**Aplicación WEB Tomcat**

Programación orientada a Aspectos

Mediante proxys intermedios para los beans que son los que aplican transaccionalidad por ejemplo.

* Proxy dinámico: Si invicamos los this no obligamos a pasar por el proxy y no activa los aspectos o transacciones asociados. Es la de defecto. Solo funciona sobre métodos públicos y declarados a partir de una interfaz, porq el proxy se crea a partit de la interfaz
* CGLib: por configuración. SI el objeto no implementa la interfaz Spring usa estos. Proxy estático
* AspectJ Weaving (LTW, CTW): no se tiene el problema de las autoinvocaciones del priero.

Normalmente:

Cliente hace petición a servlet, la lleva a la capa de negocio, devuelve lo que sea l servlet y este con jsp pinta la vuelta.

Con Spring tenemos un Controller base (FrontController), q es servlet, delegando en controladores secundarios.

Creamos un controlador principal que va a recibir la petición en una ruta de request

@Controller

@RequestMapping("/hola")

Con el GetMapping cuando reciba petición GET se lanzará ese método

Resolutor de vistas; en el estándar de Spring el string q devolvemos del método anotado es el nombre de la página JSP q se va a encargar de la petición. El nombre del fichero jsp tiene que ser el mismo que el String que estoy devolviendo.

Defino varios contextos, puedo tener varias apps spring en una misma. Los beans definidos en cada contexto son propios, no se comparten entre ellos.

getRootConfigClasses…

petición en localhost con “empleadosbme” y lueo puedo acceder a los .do que son los servidos por el controller como hola.do

localhost8080/empleadosbme/hola.do

epedraza@mazarredos.com