Friday, March 30, 2018

Apuntes de Spring MVC

Spring Boot

Application Context

- Conjunto de beans ya funcionales
- Dependencias resueltas entre el bean y la dependencia
- ComponentScan
  - Busca en todas las clases si tienen una anotación. Permite identificar si existe un bean para la anotación
  - Se realiza desde un package inicial hacia adentro

@Component

- Generica
- Define un componente en el ApplicationContext

@Controller

- Es en base un@Component
- Componente destinado para atender peticiones web, como @RequestMapping
- Detecta peticiones web
- @RequestMapping(path=“/info”)
- @ReponseBody

@Service

- Es en base un @Component
- Componente con la lógica de negocio

@Repository

- Componente de acceso a base de datos

@Configuration

- Componente que participará en la construcción del application context

El entorno es quien resuelve las dependencias, no los componentes

- @Autowired
- Para que un componente tenga las referencias a las dependencias

Tipos de Inyección de Dependencias

- Field Injection
  - Inyectar una dependencia usando una variable en el component
- Constructor Injection
  - Inyectar usando un atributo en el constructor

Spring creará los componentes e iniciará las dependencias

PostContruct

- Cuando se ejecuta el constructor en el componente, aún no se tienen las dependencias correctamente inicializadas
- Si la tarea de inicialización necesita de esta dependencia, de esta variable, hay que designar a un método la anotación @PostConstruct
- Un método con @PostContruct es el lugar correcto para poder realizar una tarea de inicialización

Resumen

En esta sección hemos introducido las bases de Spring, fundamentales para cualquier aplicación.

Hemos visto como Spring ve nuestra aplicación como un conjunto de componentes que realizan el trabajo colaborando entre ellos. Estos componentes reciben el nombre de beans.

Cuando Spring se inicia, va construyendo e inicializando el conjunto de beans que formará nuestra aplicación en una estructura que recibe el nombre de application context.

Para crear esta estructura, Spring realiza dos pasos:

En primer lugar busca todas las clases de nuestra aplicación anotadas con @Component o alguno de sus subtipos e instancia un objeto de este tipo para construir un bean. Este paso recibe el nombre de component scan y se inicia a partir del package con el método main.

En segundo lugar, visita cada uno de los beans creados y busca anotaciones @Autowired que indiquen que este bean necesita inicializarse con referencias a otros beans. Esta técnica donde el entorno (Spring) resuelve las dependencias en lugar de ser cada clase que tienen el código para buscarlas recibe el nombre de dependency injection.

Con los beans ya creados e inicializados, si alguno de ellos tiene un método anotado con @PostConstruct, recibe una invocación para que tenga la oportunidad de completar la inicialización con código propio.

¿Qué es SpringBoot?

- Facilita la construcción de aplicaciones Spring, creación inicial, configuración y ejecución
- Con infraestructura profesional
- Permite crear aplicaciones independientes
- Define dependencias

Como se soluciona problema de Configuración

- Toma el punto de partida, con las anotaciones **@Component**

Como se soluciona problema de Dependencias

- Mantiene una lista de dependencias compatibles entres ellas
- También tiene dependencias starter, que son agrupaciones de dependencias
-

@Controller

- En esta sección introduciremos el componente principal de Spring MVC: el @Controller.
- Este elemento central es el que recibe las peticiones web del cliente y determina las acciones a realizar.
- Introduciremos los recursos más usados en las aplicaciones Spring MVC pero sin descuidar los fundamentos de HTTP más importantes.
- Es decir: el objetivo es salir de esta sección conociendo no solo como implementar un controller básico en Spring MVC sino también conociendo los detalles "de bajo nivel" que suelen ser obviados en cursos introductorios pero son necesarios para trabajar de forma solvente con estas tecnologías.
- @RequestMapping(path=“/info”)
- @ReponseBody
- Será directamente lo que se le devolverá al cliente que ha realizado la petición web a “/info”

Definir una ruta

```
@Controller
@RequestMapping(“/calc”)
public class CalcController{
	@RequestMapping(path=/“info”);
	@ResponseBody
	public String info(){
		return “hola”;
	}
}
```

Filtrar por método o verbo. En este caso acepta GET y POST

```
@RequestMapping(path=/“info”, method=RequestMethod.GET,RequestMethod.POST);
        @ResponseBody
        public String info(){
                return “hola”;
        }
}
```

Recibir parámetros vía URL, deben llamarse iguales a los que se reciben en la URL

```
@RequestMapping(path=/“info”, method=RequestMethod.GET,RequestMethod.POST);
        @ResponseBody
        public String info(int a, int b){
                return “hola”;
        }
}
```

Opcionales y cambiar de nombre al parámetro

```
@RequestMapping(path=/“info”,
method={RequestMethod.GET,RequestMethod.POST});
        @ResponseBody
        public String suma(int a, @RequestParam(
                        name=“varb”,
                        required=false,
                        defaultValue=“0”) int b){
                return a+b;
        }
}
```

Otra forma de recibir parámetros desde la URL, tipo REST

En el request mapping definir que parte de la URL será la variable

```
@RequestMapping(path=/“info/{varoriginal}”);
@ResponseBody
public String recibir(@PathVariable int varoriginal){
    return “hola “ + varoriginal;
}
}
```

Recibir parámetros POST

```
@RequestMapping(path=/“iniciar”);
@ResponseBody
public String recibir(@RequestParam(“usuario”) String usuario){
    return “hola “ + usuario;
}
}
```

Content Negotiation

- La respuesta del servidor debe incluir el formato para que el cliente lo entienda
- REST
- Petición GET /persona
  - Identifica un recurso en el servidor
  - Van cabeceras:
    - Accept: Conjunto de formatos que entiende el cliente y el servidor deberá responder según esa petición
- Debe responder un MediaType, como JSON
- El servidor incluye una cabecera con el tipo de formato, **Content-Type**
- **Un recurso puede ser transmitido usando distintos MediaType**

Message Converters

- HttpMessageConverter
  - Sabe como convertir objetos en RespondeBody
  - Los MessageConverter se registra automáticamente, usando librerías Jackson.
  - Jackson: Java a JSON y viceversa
  - Todo esto es realizado usando Spring Boot Starter Web
  - Para convertir a XML, hay que agregar librería jackson-dataformat-xml a la dependencia del pom.xml
  - Así se podrá usar XML como MediaType
  - Se puede ocupar una librería nativa para convertir XML, JAXB (Incluida en el JDK)
    - Para esto hay que usar la anotación @XMLRootElement

ContentNegotiation View Resolver

- Si tiene las librerías registradas, va a responder en lo pedido
- Spring si a la petición se le agrega .JSON retornará en JSON. Si es en .XML retornará en XML
- Para determinar los tipos permitidos de data type de respuesta
  - En el @RequestMapping(produces = {MediaType.Application_XML_Value})

Controlando la conversión a JSON

- Por ejemplo si queremos cambiar un atributo, hay que ocupar una anotación para especificar que es lo que irá en el JSON final generado
- Esto se hace en la clase del objeto a representar via JSON
  - @JsonProperty(“localidad”)
  - private String municipio
  - @JsonIgnore
    - Así no sale en el JSON Correspondiente

Excepciones

- Hay que tratar los errores que se pueden producir para capturar y tratar los errores
- ResponseEntity<T>
  - Si se quiere controlar aparte del Body, en vez de devolver solo un entero o string, se puede usar ResponseEntity
  - Permite acceder a todas las partes que contienen la respuesta
    - Código de estado
    - Cabecera
    - Estado
    - Body
-

Views

- Es necesario un mecanismos de plantillas
- No ocupar o eliminar @ResponseBody en los controllers
  - Esto lo que hace es que lo muestra al cliente es exactamente lo devuelve el método
- Haciendo esto, la respuesta del método no es el objeto que se envía al cliente, sino que un nombre de la visa que ocupa para la vista
- Tecnologías para crear vistas
  - JSP
    - No funciona en todas las funcionalidades de ejecución que existen
  - Velocity
  - FreeMarker
  - Thymeleaf
    - Este se ocupará
    - Es automático
  - Con springboot es muy sencillo agregar la dependencia en el pom.xml

Formas de renderar vistas

Devolver el nombre explícitamente

```
@RequestMapping(path="/lista")
public String estado(){
        return "lista";
        //esto se transforma en el nombre lógico se convierte en "/templates/lista.html"
}
```

Que lo retorne automáticamente según el nombre del path

```
@RequestMapping(path="/defecto")
public void resolutionPorDefecto(){
        // se devuelve automáticamente al nombre "defecto" o sea "/templates/defecto.html"
}
```

Models

- Hace un papel restringido en SpringMVC
- Comunica Controller y el View
- Es como un map
- model.addAttribute("nombre", "lista de nombres");
- Spring MVC pasa automáticamente un model si lo declaramos como parámetro
- Todo lo que está en este model, estará disponible para la vista
- En la vista, para poder ocupar los atributos de un modelo
  - <h2 th:text="${nombre}"></h2>

Thymeleaf

- Revisar ViewModel

Redirect

Internacionalización

Seguridad

Autenticación

- Identificar al usuario
- Conjunto de usuarios y contraseñas
- Si falla, lanzar una respuesta HTTP: **Error 401**

Autorización

- ¿Tiene el usuario permitido realizar esta acción, acceder a este recurso?
- Solo los usuarios con un perfil, pueden acceder al recurso específico
- Si falla, lanzar una respuesta HTTP: **Error 403**

Dominio de Seguridad

- Un lugar en dónde se guardan los datos

Configuración de la aplicación, seguridad declarativa

- Agregar librerías de seguridad al pom.xml
  - spring-boot-starter-security
- Extender a WebSecurityConfigurerAdapter
  - Filtros con anotación @EnableWebSecurity
- @Autowired
  - public void configuracionGlobal(AuthenticationManagerBuilder auth) throws Exception {}
- Segurizar las URL
  - Permite configurar el objeto HttpSecurity indicando las URL que cierto usuario pueda acceder o no

```
void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
        .antMatchers(“/public.txt”).permitAll()
        .antMatchers(“/info.txt”).authenticated()
        .antMatchers(“/gestion/**”).hasAnyRole(“USER”,”ADMIN”)
        .antMatchers(“/admin/**”).hasAnyRole(“ADMIN”)
```

  -

- void configure(HttpSecurity http) throws Exception {

```
        http.authorizeRequests()
                and()
                .httpBasic();
        }
```

Seguridad Programática

- Acceso al usuario de la petición

- Necesitar saber los datos del usuario logueado conectado

```
@RequestMaping(path="/user.txt")
public String infoGestores(@AuthenticationPrincipal User user){
        return "Hola "+user.getUserName();
}
```

- Acceso a los permisos del usuario

```
@RequestMaping(path="/gestion/user.txt")
public String infoGestores(@AuthenticationPrincipal User user, Authentication
authentication){
        String msg ="Hola "+user.getUsername() + ", estas en la parte de gestión.
Perfiles :\n";
        for (GrantedAuthority o: authentication.getAuthorities()){
                msg +=o.getAuthority() + "\n";
        }
        return msg;
}
```

Packaging

Comandos

- Más acceso a los parámetros
- Si se produce un error
- Posibilidad de automatizar tareas

Construcción y Ejecución con Maven

- mvnw
  - Permite ejecutar script maven en un lugar dónde no está instalado maven
  - Si no lo está, instalará maven
  - Comando: ./mvnw package
  - Baja maven y empaca el proyecto
- Ejecutar un .jar
  - Ejecutable
  - Comando: java-jar /target/compilado.jar
- Run usando maven
  - ./mvnw spring-boot:run

- Limpiar
  - ./mvnw clean
  - Útil para asegurarse de que la construcción es totalmente nueva, sin compilados anteriores
  - El producto generado será a partir del código fuente tal como está
- Empaquetar
  - -/mvnw package
- Varios códigos
  - ./mvnw clean package

---

pom.xml

- Las acciones de ./mvnw provienen desde el pom.xml
- Nombre del .jar final en la carpeta target
- Editar pom.xml
- agregar
  - <build><finalName>nombrefinal</finalName>

Proceso de Construcción

- pom.xml
- fat jars
  - Contiene todas las dependencias
- SpringBoot al construir, incluye todas las dependencias por defecto, de un archivo jar dentro de otros jar
- Además incluye un mecanismos para que las dependencias estén dentro del jar
- manifest.mf contiene la especificación de la clase principal de SpringBoot

---

Creación de WAR

- Usado para cuando el servidor web ocupa un contenedor
- Hay que hacer algunos cambios en la construcción

- Cambiar el pom.xml
  - Agregar finalName a pom.xml
  - <build>
    - <finalName>app</finalName>
  - </build>
  - Cambiar el formato de jar a war
  - <packaging>war</packaging>
- El war incluye embedded Tomcat
- Pero no es necesario porque el servidor web de producción ya es un Tomcat
- Hay que sacarlo
- <dependency>
  - <artifact>spring-boot-starter-tomcat</artifact>
  - <scope>provided</scope>
- </dependency>
- Las dependencias con scope provided, son libraries que se necesita para ejecutar y compilar
- Son librerías que se encuentran en el entorno de ejecución, por ende, al sacar el war no hay que incluir el tomcat de spring boot
- En la clase con el Main, hacer que extienda SpringBootServletInitializer y agregar el war a la lista de dependencias, para que todos los componentes sean inicializados
- Algo así será el archivo principal de la aplicación

```
@SpringBootApplication
public class WarApplication extends SpringBootServletInitializer{
	@Override
	protected SpringApplicationBuilder configure(SpringApplicationBuilder application){
		return application.sources(WarApplication.class);
	}
	public static void main(String[] args) throws Exception {
```

```
        SpringApplication.run (WarApplication.class,args)
    }
}
```

Despliegue

- Copiar el .war a la carpeta tomcats/webapps
- Para acceder a la aplicación, agregar el nombre de la aplicación o **finalName**
- http://localhost:8080/**app**/miruta
-