Introducción a Spring Web y Maven

- Usamos `mvn archetype:generate` para generar el pom padre. Usamos para esto el arquetipo `org.codehaus.mojo.archetypes:pom-root`

- Entramos al directorio del proyecto y usamos `mvn archetype:generate` para generar el módulo de webapp usando el arquetipo `org.apache.maven.archetypes:maven-archetype-webapp`

- En el pom padre agregamos el plugin de Eclipse:

```
  <properties>
    <maven-eclipse-plugin.version>2.10</maven-eclipse-plu
gin.version>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-eclipse-plugin</artifactId>
        <version>${maven-eclipse-plugin.version}</version
>
```

```
      </plugin>
    </plugins>
  </build>
```

- Lo ejecutamos usando `mvn eclipse:eclipse`
- Importamos el proyecto en Eclipse. `File -> Import... -> Existing Project into workspace...`
- Agregamos la dependencia de spring:
    - Sumamos la property con la versión de spring a usar en el pom padre:

```
<org.springframework.version>4.2.5.RELEASE</org.springframework.version>
```

- Sumamos la config de dependencias en el pom padre

```
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${org.springframework.version}</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

- Hacemos `mvn eclipse:eclipse`, pero vemos que no cambió nada. `dependencyManagement` configura dependencias, pero

no las aplica en ningún módulo.

- Ahora sí, agregar dependencia de spring en pom hijo:

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
    </dependency>
```

Notese no hace falta definir la versión, esto ya está configurado en el pom padre.

- Finalmente hacemos `mvn eclipse:eclipse`, refrescamos el proyecto en Eclipse y vemos la dependencia fué agregada; y aparecen otros jars, que son dependencias transitivas de `spring-webmvc`.

- Configuramos `web.xml`. Por defecto viene configurado para usar la versión 2.3 (con DTD), pero nosotros queremos usar la versión 2.4 con XML Schema. Además, definimos el servlet de Spring que implementa el patrón `Front Controller`. El `web.xml` debe quedar así:

```
<web-app id="PAW" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>PAW test application</display-name>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.Dispat
cherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

- Agregamos nuestro primer controller, el `HelloWorldController`

```
@Controller
public class HelloWorldController {

    @RequestMapping("/")
    public ModelAndView helloWorld() {
        final ModelAndView mav = new ModelAndView("index"
);
        mav.addObject("greeting", "PAW");
        return mav;
```

```
    }
}
```

- Intentamos entrar al código de `ModelAndView`, ver que no hay sources ni javadoc (salvo quizás un decompilado automáticamente).

- Configurar eclipse plugin para que baje sources y javadoc. Dentro del bloque `<plugin>` de plugin de eclipse incluir:

```
        <configuration>
          <downloadSources>true</downloadSources>
          <downloadJavadocs>true</downloadJavadocs>
        </configuration>
```

Para luego volver a hacer `mvn eclipse:eclipse`

- Refrescar el proyecto en Eclipse e intentar entrar al código de `ModelAndView` de nuevo, ver que ahora sí hay sources y javadoc

- Instalar desde el Marketplace de Eclipse el plugin `Run Jetty, Run`, y configurarlo usando la versión 8.x del mismo para correr nuestra webapp. Ver que da un 404.

- Necesitamos indicarle a Spring como encontrar nuestras clases. En el `web.xml` sumar al `<servlet>` lo siguiente:

```
    <init-param>
      <param-name>contextClass</param-name>
      <param-value>
        org.springframework.web.context.support.Annotatio
nConfigWebApplicationContext
      </param-value>
    </init-param>
```

Y dentro del nodo `<web-app>` sumar:

```
  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationC
onfigWebApplicationContext
    </param-value>
  </context-param>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      ar.edu.itba.paw.webapp.config.WebConfig,
    </param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListen
```

```
er
    </listener-class>
  </listener>
```

Esto configura el listener de contexto que va a inicializar todo el contexto de Spring al deployarse la app, y lo destruye ordenadamente al darse de baja. Además, se le indica que esta configuración del contexto va a ser basada en annotations, y que la config está en la clase `ar.edu.itba.paw.webapp.config.WebConfig`.

- Armar la clase `WebConfig`

```
@EnableWebMvc
@ComponentScan({ "ar.edu.itba.paw.webapp.controller" })
@Configuration
public class WebConfig {

}
```

- Correr Jetty. La app levanta, pero reporta un error al querer acceder a `http://localhost:8080/`. No hay vista definida, e interpreta el `helloworld` del `viewName` indicado al `ModelAndView` como un redirect. Necesitamos definir nosotros como queremos resolver vistas, para esto se debe definir un `ViewResolver`. Agregar al `WebConfig`:

```
    @Bean
    public ViewResolver viewResolver() {
```

```
        final InternalResourceViewResolver viewResolver =
 new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");

        return viewResolver;
    }
```

y crear la vista correspondiente en `src/main/webapp/WEB-INF/jsp/index.jsp`

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_
rt"%>

<html>
<body>
<h2>Hello ${greeting}!</h2>
</body>
</html>
```

- Levantar Jetty, ver que ahora todo funciona correctamente.

- Según la versión de Jetty usada, es posible obtener errores por clases faltantes, en dicho caso, en el pom padre agregar estas dependencias:

```
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>${servlet-api.version}</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>${jstl.version}</version>
      </dependency>
```

Donde las versiones a usarse son:

```
    <servlet-api.version>2.5</servlet-api.version>
    <jstl.version>1.2</jstl.version>
```

Y luego en `webapp/pom.xml` incluirlas:

```
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
```

```
    </dependency>
```

Correr `mvn eclipse:eclipse` y refrescar los proyectos.

- Usamos `mvn archetype:generate` 4 veces para generar los módulos de servicio, persistence, interfaces y modelos usando el arquetipo `org.apache.maven.archetypes:maven-archetype-quickstart`

- En el módulo de interfaces, agregamos la dependencia sobre los modelos de la siguiente forma:

```
    <dependency>
      <groupId>ar.edu.itba.paw</groupId>
      <artifactId>model</artifactId>
      <version>${parent.version}</version>
    </dependency>
```

- En el módulo de servicios, agregamos la dependencia sobre las interfaces analogamente.

- en webapp agregramos las dependencias sobre interfaces y servicios. La única salvedad, es que a los servicios los vamos a definir con scope runtime ya que queremos que estén en el war final, pero no sean considerados en el classpath al compilar as clases del módulo webapp:

```
<scope>runtime</scope>
```

- Hacemos `mvn eclipse:eclipse`, en Ecipse refrescamos todos los proyectos / importamos los que faltan.

- Creamos un modelo básico de `User` en el módulo de models.

- Creamos una interfaz de servicio de usuarios con un método `findById` en el módulo de interfaces; y un modelo `User` en model.

- Armamos una implementación concreta del servicio en el módulo de servicios.

- Vemos como en el `HelloWorldController` podemos escribir algo del estilo:

```
private final UserService us = new UserServiceImpl();
```

Eclipse va a compilar, ya que no entiende de scopes.

- Hacemos `mvn clean compile`, vemos que la compilación falla, porque `UserServiceImpl` está en el módulo de servicios, y al compilar el módulo webapp, este módulo no está disponible (scope runtime).

- Quitamos entonces la llamada al constructor y vamos a usar inyección de dependencias por Spring. Para esto, anotamos el atributo `us` del controller con `@Autowired`:

```
@Autowired
private UserService us;
```

y anotamos a la implementación del servicio `UserServiceImpl` con el estereotipo `@Service`. Para esto vemos que necesitamos agregar al pom del módulo de servicios la dependencia sobre `spring-context`, que también debe ser definida en el pom padre.

Además, editamos `WebConfig` para sumar al annotation `@ComponentScan` el paquete `"ar.edu.itba.paw.service"` para que encuentre nuestro servicio.

- Corremos la aplicación y vemos que la aplicación funciona, y una instancia de `UserServiceImpl` ha sido creada por Spring e inyectada en `HelloWorldController`

- Creamos una copia de `UserServiceImpl` llamada `AnotherUserServiceImpl`, también anotada con `@Service`. Vemos que al intentar correr ahora a aplicación falla con una excepción diciendo que no encuentra un único bean a ser inyectado.

** Se ve como usar `@Qualifier` para resolver el conflicto. Vemos que

hay nombres autogenerados (nombres de clase en camel case), pero es posible especificar nombres semánticos.

** Vemos que otra opción es usar `@Primary` en la implementación del `@Service` que querramos usar por defecto cuando no haya un `@Qualifier` que diga lo contrario.

- Creamos la interfaz `UserDao` en el módulo de interfaces con un único método `findById`, y una implementación concreta `UserDaoImpl` en el módulo de persistencia, anotando a esta con `@Repository`. Para esto incluimos la dependencia de `spring-context` sobre el módulo de persistencia.

- En `UserServiceImpl` agregamos un atributo con el DAO:

```
@Autowired
private UserDao userDao;
```

- Editamos `WebConfig` para sumar al annotation `@ComponentScan` el paquete `"ar.edu.itba.paw.persistence"`

- Corremos la aplicación y vemos que así como se crea un `UserServiceImpl` que se inyecta en `HelloWorldController`, se crea un `UserDaoImpl` y se inyecta en `UserServiceImpl`

- Modificamos la firma del metodo `helloWorld` en el

`HelloWorldController` para ilustrar el uso de `@QueryParam` para conseguir parámetros entrados por url, tanto opcionales como obligatorios, con y sin defaults. Vemos que Spring puede convertir entre tipos standard sólo.

- Hacemos lo mismo usando `@PathVariable` para obtener valores de placeholders en el path. No olvidar para esto tocar el path usado en `@RequestMapping` usando llaves, de la forma `@RequestMapping("/user/{userId}")