# Obstáculos Técnicos Encontrados

# En Tempore

## Spring y GWT

En la propuesta para iniciar el trabajo profesional se había pensado el uso del framework Spring para integrarlo con GWT, de manera de usar el patrón MVC.

Investigando un poco más la herramienta GWT y comenzando el desarrollo del mismo nos dimos cuenta que no se requería el uso del patrón MVC por parte del framework Srping, ya que el propio GWT utiliza el patrón MVC con una muy fácil implementación.

## Arquitectura y encapsulamiento aplicaciones GWT

Se requería separar en diferentes proyectos la aplicación GWT de manera que sea más fácil de mantener y trabajar entre varias personas. Se analizo y se llego a la conclusión de separar entre los servicios que acceden a BBDD (DAOs y Entities), la vista de la aplicación, los servicios requeridos por la vista para funcionar y los componentes requeridos para la integración entre la vista y servicios (DTOs).

El problema que se presentó es que el IDE de Eclipse no permitía la integración de proyectos de la manera convencional de una aplicación JAVA o J2EE.GWT buscaba su código en un único proyecto.

De esta manera se encontró una opcion de emular el código en un solo proyecto y que GWT no se diera cuenta de la separación de los mismos. La solución fue la de cambiar la “Importación” por el uso de “links” entre los proyectos a unir. El link proporciona la ventaja de que un proyecto que linkea, ve el código fuente del otro como si fuera propio.

## Servicios de GWT y SmartGWT

Se eligió GWT como leguaje para el desarrollo de la aplicación Tempore, con el objetivo de facilitar el diseño (Look and Feel) y la velocidad dela aplicación ya que termina trabajando con JavaScript.

Luego se encontró el framework SmartGWT que trabaja en base a GWT. Este mejoraba aun más el diseño y el desarrollo del mismo.

Hubo dos grandes obstáculos sobre la elección GWT y SmartGWT.

El primero y totalmente esperado era el desconocimiento del lenguaje. Aunque ayudo bastante que sea java, GWT tiene varias particularidades de las que hay que acostumbrarse. Por ejemplo, el código es recompilado cada vez que se quiere desplegar la aplicación, debido a que es javascript lo se ejecuta al final. También el desarrollo del frontend era lento ya que no se conocía ningún componente de la vista y había que investigar todas las propiedades del mismo.

El segundo obstáculo era en el desarrollo de los servicios, ya que al desarrollar el frontend debía pensarse como un desarrollo de una aplicación java de escritorio (swing), con eventos y acciones asincrónicas que hacen difícil la sincronización de componentes.

Para el primer problema, era cuestión de usar el lenguaje y los componentes. De a poco se comenzó a entender como funcionaban e interactuaban entre ellos.

Para el segundo problema se había solucionado llamando los servicios de manera anidada. Llamar al primer servicio requerido y en la respuesta que daba invocar al servicio que dependía de este y así sucesivamente. Esta solución no era prolija cuando se trataba de sincronizar paneles diferentes ya que eran paquetes de códigos diferentes. La solución para esto fue la de utilizar el patrón *Observer*, que avisaba a los observadores cuando debían actualizarse, sincronizando de esta manera los servicios.

## Framework Hibernate y Jetty:

Se eligió el framework Hibernate como ORM para implementar los servicios que acceden a la BBDD. Se implementaron los servicios y entidades de la BBDD, pero cuando se intento realizar la integración con GWT no funcionaba. Diciendo que una clase requerida por Hibernate no existía.

Luego de un par de semanas de revisar el código, ver foros de Hibernate y GWT. Encontramos que el problema residia en el conflicto de una librería que traía el servidor de aplicaciones que traía por defecto el GWT (Jetty). La librería que utilizaba el servidor Jetty era mas antigua y al parecer cuando realizábamos la integración tomaba primero este jar antes que las librerías de hibernate.

La solución fue la de remplazar el servidor Web Jetty por el Apache Tomcat 7.0, que no tenia la librería en conflicto.

Se requirió tiempo para lograr configurar el Tomcat correctamente con GWT y en modo debug.

## Uso de DBUnit

Se deseaba mantener un testeo continuo a los DAOs desarrollados de manera que cualquier cambio en la BBDD podríamos detectar rápidamente el impacto en estos.

El problema que encontramos al intentar realizar este tipo de test continuo es que para probar servicios que tienen impacto en BBDD, es que necesitamos conocer que hay en la BBDD inicialmente para luego conocer el estado final con el que quedara para realizar la verificación del servicio.

Para este tipo de pruebas se nos sugirio el uso el framework DBUnit. Este framework se encargaba de poblar la BBDD a través de un XML y luego a través de otros XML verficiar el estado final de los mismos.

## Framework Dozer:

La aplicación utiliza la entities para el acceso a BBDD y DTO para la comunicación con la vista. Ambas capas tienen los mismos datos (mismo POJOS), por lo que se encontró muy tedioso la copia de todos los atributos cuando se trataba de pasar la información de la entities a los DTO.

Por este motivo se busco una herramienta que automatice la copia de entidades (mapper de Bean a Bean). Entre varias opciones se selecciona la opción de Dozer debido a la fácil implementación que tiene.

## Hibernate Session:

Se encontró problemas de sincronización de servicios DAOs. Cuando varios servicios se invocaban simultáneamente (buscar el detalle de las tareas) varios utilizaban la misma sesión y el primero de los servicios que terminaba de usarla la cerraba y quedaban todos los demás trabados con la sesión cerrada.

La solución fue la de crear una clase que sincronizaba el pedido de sesión de hibernate y que con cada solicitud se crea una nueva e independiente al resto.

## Framework para la generación de Reportes (GWT-Visaulization)

Los reportes que ofrecía el framework smartgwt eran pagos. Por lo que se decidió investigar otros frameworks gratuitos.

El problema es que los que se encontraron estaban todos desarrollados sobre GWT y no sobre SmartGWT. Este inconveniente trajo el problema que el reporte no podía ser adherido a los layout propios de SmartGwt.

La solución fue la de utilizar código JSNI (JavaScript Native Interface). JSNI es una manera de escribir en GWT código directamente en JavaScript para realizar la adaptación entre los dos componentes.

De esta manera para cada uno de los reportes se invoca a través de JNDI para que se acople al Layout de SmartGWT.

## Carga de imágenes en GWT y SmartGWT

La carga de imágenes en GWT es complicada, debido a que todos los servicios son asincrónicos e independientes. El problema que encontramos fue que teníamos el manejo de los usuarios y queríamos que estos puedan llegar a cargar imágenes a sus perfiles de manera que luego en la asignación pueda realizarse en forma mucho más amigable. Al intentar realizar esto con un form de Smartgwt nos dimos cuenta que este no soporta imágenes como ítems del form. Investigando encontramos una manera de resolver este problema y que era la misma manera en que lo resuelve aplicaciones como gmail u otras. Y es la de trabajar la imagen en forma independiente, cargar la imagen y subirla directamente a un servidor independiente del formulario. Luego cuando se completa el resto de la información del formulario se le asocia el ID de la imagen subida y se guarda como otro campo en la BBDD y luego con este ID se puede recuperar la imagen con un servicio simple (Servlet para visualizar imágenes)

## Manejo de ABM en GWT

Se entendió que se requería desarrollar un framework propio para el manejo de todos los servicios tipo ABM. Se desarrollo interfaces que implementan los servicios y los datasource requeridos para que sean integrados rápidamente con el framework GWT y los servicios de background con pocas líneas de código.

## Utilización del Tree en SmartGWT

Al trabajar con el framework propio de ABM, no se comportaba correctamente cuando trabajábamos con el componente Tree de manera que se volvió a trabajar con servicios en forma indirecta.