

Introducción parcial a Wicket

Autor: Carlos Lombardi

En este documento vamos a describir algunos conceptos y facetas del desarrollo de aplicaciones Web usando Wicket.

ATENCIÓN

Varias de las nociones que se describen, y de los comentarios, aplican no solamente a aplicaciones construidas usando Wicket, sino a cualquier aplicación Web, y más en general a cualquier aplicación con UI.

Nos vamos a apoyar en una aplicación ejemplo sobre organización de eventos musicales, a la que llamamos **Monsters of Rock**.

El código está disponible como un proyecto GitHub, en <https://github.com/obj2-material/monsters-wicket-to-see>.

Ahí mismo en la carpeta txt, hay un enunciado que describe el dominio.

El proyecto tiene dos source folders: src/domain tiene la implementación del dominio, src/web tiene la implementación de la aplicación Web.

La idea es ir leyendo el documento **mientras** se mira el código.

Aclaración importante: este documento debe ser leído *después* del “Hola Mundo”. O sea, estamos suponiendo que el lector ya sabe crear y configurar una aplicación Wicket.

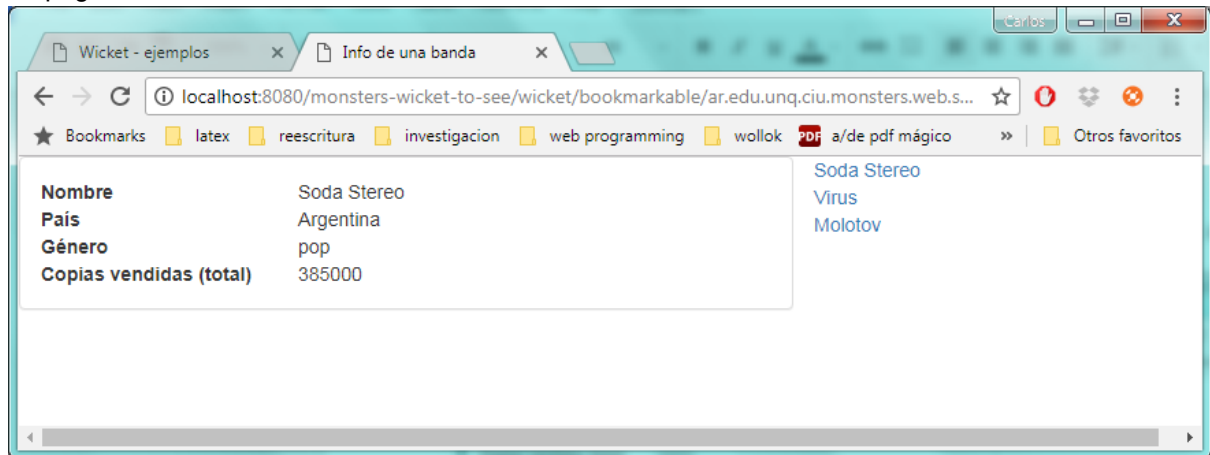
Otras aclaraciones:

- Las páginas están armadas usando Bootstrap, www.getbootstrap.com, una biblioteca de CSS que hace más fácil dar layout por filas y columnas, define defaults lindos para los tipos de letra y colores, y tiene opciones que quedan bien gráficamente para tablas y forms. No es necesario usar Bootstrap en las aplicaciones que se armen con Wicket, de hecho Wicket no necesita enterarse de cómo se maneja el CSS.

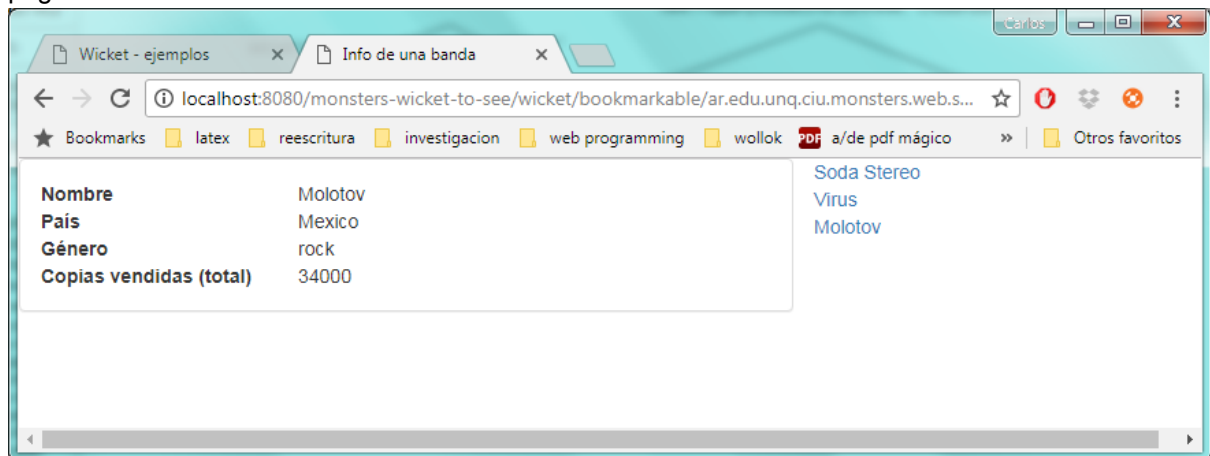
Una página - conceptos iniciales

Estudiemos qué hay que hacer para armar una página que muestre la información de una banda, permitiendo elegir entre dos de las bandas que maneja la aplicación.

La página arranca mostrando la información de Soda Stereo:



A la derecha hay dos links, que son los que permiten cambiar de banda. Si se elige Molotov, la página se ve así:



El **código** de esta página está en el package `ar.edu.unq.ciu.monsters.web.simpleBandPage`. Desde la página inicial, se accede con la opción **Página simple - datos de una banda**.

Mirando el código que implementa esta página podemos revisar varias ideas básicas de Wicket.

Componentes estáticos y dinámicos de una página

En la página que estamos estudiando, esta es la descripción que aparece si elegimos la banda "Virus" en los links de la derecha

Nombre	Virus
País	Argentina
Género	pop
Copias vendidas (total)	260000

Si cambiamos a "Soda Stereo", esta parte de la página cambia a

Nombre	Soda Stereo
País	Argentina
Género	pop
Copias vendidas (total)	385000

La descripción de cada campo, o sea “Nombre”, “País”, “Género”, “Copias vendidas (total)”, no cambian al elegir una banda u otra. Más en general, estas descripciones son **fijas**, no las afecta ninguna interacción posible con el usuario.

A los elementos fijos de una página los vamos a llamar **componentes estáticos**.

Al contrario, los datos correspondientes a la banda elegida (“Virus”, “Argentina”, “pop”, 260000) sí se ven afectados por la acción que puede tomar el usuario, que es elegir a una banda.

A los elementos de una página que no son fijos los vamos a llamar **componentes dinámicos**.

Tres componentes

Esta página admite una forma de interacción con el usuario, que está dada por los links de la derecha.

Para cualquier página que permita interacción, vamos a definir tres componentes:

1. Un **archivo .html** que describe el formato general de la página y **las partes estáticas**.
2. Una clase que define **las partes dinámicas** de la página. En principio, esta clase va a heredar de `org.apache.wicket.markup.html.WebPage`.
3. Una clase de **controlador**, que maneja la interacción. En el ejemplo, va a recordar cuál es la banda que eligió el usuario.

Si una página no admite **ninguna** interacción con algo de complejidad, entonces no hace falta el controlador. Pero por lo general, vamos a tener los tres componentes.

Para esta página, en el package encontramos:

1. El archivo `SimpleBandPage.html`.
2. La clase `SimpleBandPage`, esta define la parte dinámica de la página.
3. La clase `SimpleBandPageController`, el controlador.

El .html - referencias a componentes dinámicos

El archivo .html es ... un texto en lenguaje HTML, que incluye todos los elementos de una página Web: la división en head y body, todos los tags incluyendo `div` y `p`, forms, CSS, scripts¹, etc.

Dentro del head encontramos esta línea

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-
BVYiISIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
```

Esto incluye Bootstrap dentro de la página. Recordemos que Bootstrap es una biblioteca de CSS.

Veamos el código HTML para mostrar el país de la banda

```
<div class="row">
  <div class="col-sm-4" style="font-weight: bold">
    País
  </div>
  <div class="col-sm-8">
```

¹ En particular soporta scripts en Javascript, googlear “script Javascript HTML page” y “DOM HTML page Javascript”. Wicket incluye soporte específico para la interacción con Javascript, ver capítulos 16 y 17 de la Reference Guide.

```
        <div wicket:id="pais"></div>
    </div>
</div> <!-- /.row -->
```

Los div permiten aplicar layout. Las clases CSS row, col-sm-4 y col-sm-8 vienen con Bootstrap. La clase row marca una fila (en este caso dentro del recuadro donde aparecen los datos de la banda elegida, este recuadro se define con las clases panel y panel-body que también define Bootstrap. Esta biblioteca divide el espacio en 12 columnas, las clases col-sm-4 y col-sm-8 indican que el label “País” y el nombre del país toman 4 y 8 columnas respectivamente.

Notar que se le puede agregar CSS a un tag, igual que en HTML. Así se logra que la palabra “País” esté en negrita.

Observamos que la palabra “País” está puesta fija en el HTML, no hace ninguna referencia a Wicket. Esto la define como componente estático.

Por otra parte, el nombre del país está definido como un div que tiene un atributo wicket:id. Esto marca que es un componente dinámico, que va a estar manejado desde la clase Java. El valor del atributo wicket:id, en este caso pais, es la referencia que permite definir este componente en Java.

La clase Java para definir la página

La clase que describe la página en Java es SimpleBandPage .

Hereda de org.apache.wicket.markup.html.WebPage, esa es la forma de definirla como página (al menos en principio). De WebPage se habla en la sección 4.3 de la referencia de Wicket.

Lo que espera Wicket es que agreguemos la definición de los componentes dinámicos en un constructor. En principio, el constructor es sin parámetros.

Como esta definición puede ser larga, conviene organizarla en subtareas, para cada subtarea definimos un método interno.

Por eso el constructor de SimpleBandPage tiene esta forma:

```
public SimpleBandPage() {
    this.controller.setChosenBand(
        MonstersStore.store().getBandaLlamada("Soda Stereo")
    );

    this.fillBandFile();
    this.fillBandSelectionLinks();
}
```

El método fillBandFile() se va a ocupar de los datos de la banda, el fillBandSelectionLinks() de los links para elegir banda.

Viendo la definición del país dentro de fillBandFile() y la parte correspondiente del HTML

```
// en Java
this.add(new Label(
    "pais", // parámetro id
    new PropertyModel<>(this.controller, "chosenBand.pais.nombre")
));

// en el HTML
<div class="col-sm-8">
    <div wicket:id="pais"></div>
</div>
```

podemos aprender varias cosas que nos van a servir para armar cualquier página Wicket:

1. Wicket incluye clases para cada tipo de componente que vayamos a manejar. Acá se está usando un Label, que es el componente más sencillo: simplemente representa un String. Los x se describen en la sección 4.3 de la referencia de Wicket.
2. Para agregar un componente dinámico se le envía el mensaje add al contenedor. Por ahora, el contenedor es la misma página, por eso this.add(...) . Hay casos en que el contenedor es otro componente, p.ej.
 - se puede definir una estructura de paneles
 - las columnas de una tabla se definen como componentes cuyo contenedor es el componente que representa la tabla.
3. El archivo .html y la clase que representa la página se relacionan haciendo que coincidan
 - el valor del atributo **wicket:id** en el tag HTML y
 - el parámetro **id** del constructor del componente Wicket.

En este caso, el valor es **"pais"** .

Modelos Wicket.

El segundo parámetro en el constructor del Label le indica qué tiene que mostrar, estableciendo el **vínculo entre vista y modelo** .

Para que realmente sea dinámico (o sea, que no sea un valor fijo) hay que pasar un objeto aparte, al que Wicket llama *Model*. Hay varias variantes de Model, la más sencilla es la que usamos acá, la clase org.apache.wicket.model.PropertyModel que viene con Wicket.

Un PropertyModel se define muy fácil: a qué objeto preguntarle el valor que hay que mostrar, qué mensaje le tengo que enviar.

El mensaje acepta notación de puntos como vimos en Arena. En este caso al controller le pide getChosenBand(), a lo que devuelve eso (que es una Banda) le pide getPais(), a lo que devuelve eso (que es un Pais) le pide getNombre(), eso es lo que muestra.

Se habla de PropertyModel en la sección 11.2 de la referencia de Wicket.

En Arena, la definición del mismo componente sería así:

```
new Label(panel).bindValueToProperty("chosenBand.pais.nombre");
```

suponiendo que el modelo del panel es el controller. Si no,

```
new Label(panel).bindValue(
    new ObservableProperty<>(this.controller, "chosenBand.pais.nombre")
);
```

Se puede hacer esta analogía:

1. El segundo parámetro del constructor del componente Wicket tiene la misma función que el método `bindValue` del Arena: **establecer el vínculo entre vista y modelo**, para el valor del componente.
2. El `PropertyModel` de Wicket corresponde al `ObservableProperty` de Arena. Arena provee la forma abreviada `bindValueToProperty` porque maneja el concepto de modelo de un panel, y por lo tanto puede suponer a qué objeto le tiene que pedir el valor de la property.

Más sobre `PropertyModel` y los componentes Wicket

Sigamos con un comentario sobre `PropertyModel` y otro sobre los componentes Wicket.

`PropertyModel` - OJO con cómo se configuran

Transcribimos la definición del `PropertyModel` para el país

```
new PropertyModel<>(this.controller, "chosenBand.pais.nombre")
```

El constructor de `PropertyModel` tiene dos parámetros, un objeto y un nombre de property que puede tener puntos. Para que nuestros `PropertyModel` anden bien, hay que tener en cuenta lo siguiente:

el objeto se le setea al `PropertyModel` en la creación, y después no cambia más aunque la expresión que usamos para definir qué objeto es cambie.

Si en este caso ponemos

```
new PropertyModel<>(this.controller.getChosenBand(), "pais.nombre")
```

la expresión

```
this.controller.getChosenBand()
```

se evalúa una sola vez, al crearse la instancia de `SimpleBandPage`. Entonces, por más que cambiemos la banda elegida con los links de la derecha, este `PropertyModel` va a estar vinculado siempre con la misma banda, y por lo tanto no va a cambiar el país.

En la forma en que lo definimos, lo que se ejecuta al principio es solamente

```
this.controller
```

Esto está bien porque el controller no cambia. La banda elegida, que sí cambia, es parte de la property, eso sí se evalúa cada vez.

Componentes Wicket - su efecto

Los componentes Wicket agregan (una palabra adecuada para este caso es *inyectan*) texto en la página HTML que Wicket envía como respuesta a un pedido.

Para ver en concreto el efecto del `Label`, observamos el código fuente de la página al levantarla desde un browser.

```
<div class="col-sm-8">
  <div wicket:id="pais">Argentina</div>
</div>
```

El efecto del componente `Label` es poner el valor de su modelo (en el ejemplo, el `PropertyModel` con "chosenBand.pais.nombre") como valor del tag `div`.

Links y clases anónimas

OJO una cosa con los links de la derecha. Lo que esperamos que hagan es que cambie la banda elegida y se vuelva a mostrar *la misma* pantalla. En general, en los links el href lo vamos a manejar desde Wicket, no desde el HTML. Queda así (va el de Soda Stereo de ejemplo)

```
<div class="row">
    <a href="#" wicket:id="chooseSoda">Soda Stereo</a>
</div>
```

El href="#" indica que no estamos indicando la URL destino desde HTML. El wicket:id="chooseSoda" es la referencia para vincular el tag con un componente Wicket, como hicimos con el Label.

Veamos la definición en la clase SimpleBandPage .

```
this.add(new Link<String>("chooseSoda") {
    private static final long serialVersionUID = -7850955536756349914L;

    @Override
    public void onClick() {
        SimpleBandPage.this.controller.setChosenBand(
            MonstersStore.store().getBandaLlamada("Soda Stereo")
        );
    }
});
```

Acá se está definiendo una *clase anónima*. La clase Link (en rigor, org.apache.wicket.markup.html.link.Link) que viene con Wicket es abstracta, la definición tiene esta pinta

```
public abstract class Link {
    // ... cosas
    public abstract void onClick();
    // ... más cosas
}
```

Por lo tanto, esto

```
this.add(new Link<String>("chooseSoda"));
```

no compila, Eclipse aclara "Cannot instantiate the type Link<String>".

La notación

```
new Link<String>("chooseSoda") {
    // ...cosas...
});
```

crea una clase nueva que es subclase de Link, y una instancia de esa clase nueva. Esta nueva clase no tiene nombre, se dice que estamos creando una **clase anónima**.

Para poder crear una instancia, esta clase anónima debe definir el método onClick() que la clase Link deja abstracto.

Esta clase anónima, también es interna (*inner class*) a la definición de la clase SimpleBandPage (que es la *outer class* en este caso). En el onClick() hay que enviarle un mensaje al controller que es un atributo ... de la página, no del link. Java permite acceder a la instancia de la outer class con la notación

NombreDeOuterClass.this

Por eso el método onClick() dice

```
public void onClick() {
    SimpleBandPage.this.controller.setChosenBand(
```

```

        MonstersStore.store().getBandaLlamada("Soda Stereo")
    );
}

```

Si dijera

```

public void onClick() {
    this.controller.setChosenBand(
        MonstersStore.store().getBandaLlamada("Soda Stereo")
    );
}

```

(o sea, si le sacamos el **SimpleBandPage** al this) no compilaría, porque this “solo” es el Link, que no tiene un atributo controller .

Volvamos ahora al tema del Link. En el código del método x ponemos lo que queremos que pase cuando el usuario pulse el link, en este caso, avisarle al controlador que cambió la banda elegida. Si no se indica un cambio de página, después de los cambios la página se vuelve a generar, volviendo a evaluar los PropertyModel de cada Label.

Los links se describen en la sección 4.4 de la referencia de Wicket. La sección 10, que no leí, parece que también habla sobre links.

Tres comentarios finales

Qué pasa cuando se pulsa un link

No hay que olvidar que en una aplicación Web tenemos un servidor y un cliente (que es el browser).

Todo el código Java se ejecuta en el servidor. Por otro lado, las interacciones del usuario se hacen en el cliente.

Esta es una descripción sucinta de lo que pasa desde que el usuario pulsa el link “Molotov” hasta que ve en su browser los datos de Molotov.

Paso	En el servidor	Comunicación	En el cliente
1			El usuario pulsa el link “Molotov”.
2		Señal de “se pulsó el link Molotov”.	
3	Se ejecuta el código del método onClick() del objeto Link vinculado al link HTML.		
4	Se recalcula el HTML, con los datos de Molotov porque es la banda elegida del controller. Se evalúan todos los PropertyModel.		
5		El nuevo HTML generado <i>viaja al browser</i> .	
6			Se muestra el nuevo HTML.

Wicket hace la magia que permite que pase todo esto. Hay información sobre este tema (que no leí) en el capítulo 9 de la referencia.

Esto es importante tenerlo en cuenta por dos cosas

1. Sí, para muchas cosas que hace el usuario hay una ida al servidor y vuelta al browser. Si esto pasa muy seguido, se resiente la velocidad. Para evitar esto, hay tecnologías y técnicas

que permiten resolver interacciones con *código que se ejecuta dentro del browser*, de forma tal que el servidor no se entera. Esto se suele hacer usando Javascript.

2. A veces el server no tiene la información que se necesita. P.ej. si una página tiene varios forms y se submitea uno, solamente la info relacionada **con ese form** viaja al server, el resto no. Por eso algunas cosas que uno piensa que están actualizadas, en realidad no lo están. Análogamente, hay situaciones en las que no se actualizan partes de una página que uno espera, porque no hubo ningún evento que forzara ir al server y volver.

Serialización

Pensemos qué pasa en el server una vez que generó el HTML que le llegó al browser. Cuando el usuario realiza otra acción, que puede ser minutos después, el server se tiene que acordar de la info necesaria para reconstruir “qué estaba viendo” el usuario. En general, esta info incluye el estado del controlador.

Recordemos también que un servidor puede estar sirviendo la misma página a varios clientes, cada página va a tener su controlador. Cuando un usuario realiza una acción, el servidor tiene que saber de qué usuario se trata. Técnicamente, se dice que el servidor mantiene el estado de la **sesión** de cada usuario conectado. De esto se habla (creo) en el capítulo 9 de la referencia de Wicket.

Todo esto no es automático. La magia de Wicket es que maneja estas cuestiones, de forma tal que podemos escribir una aplicación Web con un código bastante parecido al de Arena.

Un servidor Wicket podría necesitar mucha memoria para recordar toda la información de sus sesiones. Por esto podría decidir “bajar” parte de esta info a un disco u otro dispositivo.

Para poder hacer estos manejos, Wicket necesita que cualquier objeto que eventualmente tuviera que recordar implemente una interface que se llama Serializable.

Por eso, las páginas, los elementos gráficos, los controladores, y también los objetos de dominio, tienen que implementar Serializable para que Wicket funcione correctamente.

A su vez, conviene que cada clase Serializable tenga su

```
private static final long serialVersionUID = ...un número feo ...;
```

Por suerte, Eclipse genera esta línea por nosotros. Cuando creamos una clase que es serializable (incluyendo las clases anónimas), marca un warning que es la falta del x. Le damos Ctrl-1 a la línea, elegimos la opción “Add generated serial version ID”, listo. Yo suelo borrar las líneas de comentario que pone, dejo el x pegado a la definición de la clase. P.ej.

```
public class SimpleBandPage extends WebPage {  
    private static final long serialVersionUID = -5617749468806067306L;  
    // ... etc ...  
}
```

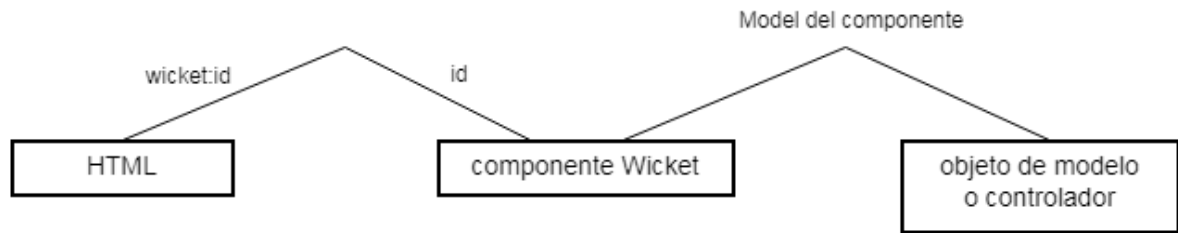
Dos niveles de vinculación = binding

Repasemos: para una página, definimos tres elementos: página HTML, clase Java de vista (la WebPage), clase Java de controlador (que es el modelo de la vista).

Esta estructura define dos niveles de binding:

1. además del vínculo entre vista y controlador que es como en Arena, del que se encargan los Model (en el ejemplo, PropertyModel),
2. también hay que vincular tags HTML con elementos gráficos de la vista Java. Esto lo hacemos haciendo que coincidan wicket:id del tag con id del elemento Java.

Algo así:



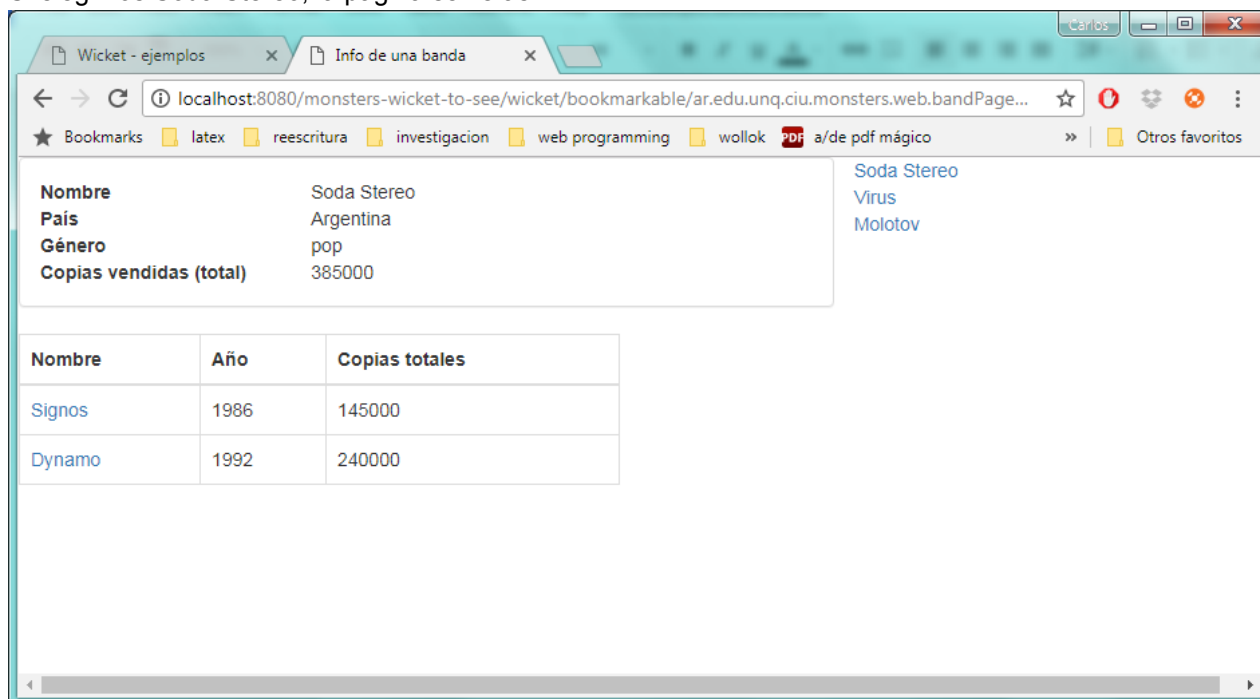
Para el país de la banda tenemos específicamente



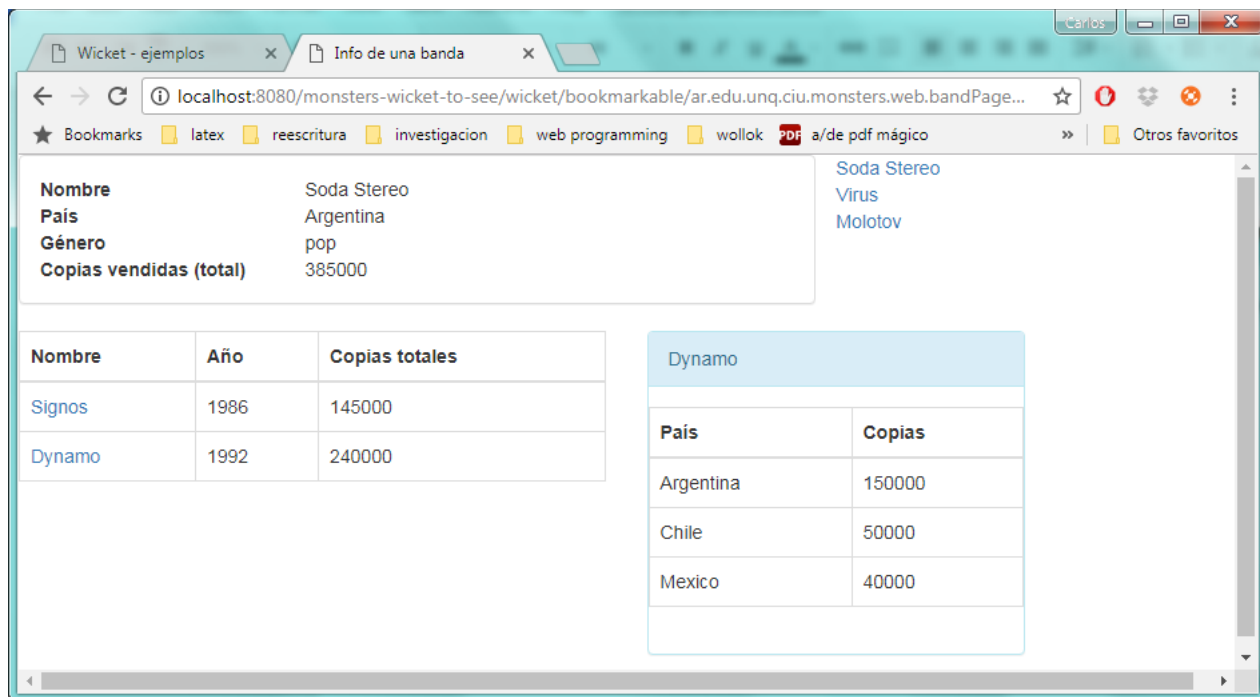
Una página más cargada - tablas, links dinámicos, panel

Estudiemos ahora una página que agrega datos sobre los **discos** de la banda elegida.

Si elegimos Soda Stereo, la página se ve así:



Epa, el nombre de cada disco es *un link*. Al elegir p.ej. Dynamo, nos aparece a la derecha información sobre ese disco.



El **código** está en el package `ar.edu.unq.ciu.monsters.web.bandPageWithAlbums`.

Desde la página inicial, se accede con la opción **Página más completa - banda y discos**.

Tablas

La tabla de discos de una banda, en el HTML se describe como un tag table, con su fila de títulos, y después una sola fila de datos. Esta fila tiene un atributo wicket:id que la relaciona con el componente gráfico en Java que representa la tabla. A su vez, cada <td> también lleva su wicket:id, que lo relaciona con una columna definida en la clase Java.

El HTML queda así:

```
<table class="table table-bordered">
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Año</th>
      <th>Copias totales</th>
    </tr>
  </thead>
  <tbody>
    <tr wicket:id="discos">
      <td><a href="#" wicket:id="nombre"></a></td>
      <td><span wicket:id="anio"></span></td>
      <td><span wicket:id="copiasVendidas"></span></td>
    </tr>
  </tbody>
</table>
```

La definición en la clase BandPageWithAlbums es esta:

```
this.add(new ListView<Disco>(
    "discos",
    new PropertyModel<>(this.controller, "chosenBand.discos")
) {
    private static final long serialVersionUID = -4547597546545617797L;

    @Override
    protected void populateItem(ListItem<Disco> panel) {
        Disco elDisco = panel.getModelObject();
        // ... definicion del Link para el nombre ...
        panel.add(nombreLink);

        panel.add(new Label("anio", new PropertyModel<>(elDisco, "anio")));
        panel.add(new Label(
            "copiasVendidas",
            new PropertyModel<>(elDisco, "totalCopiasVendidas")
        ));
    }
});
```

El componente que provee Wicket para manejar tablas es la ListView, que se describe en la sección 13.2 del manual de referencia. El constructor toma como parámetros el id que engancha con el HTML, y el modelo del que toma la lista de items (lo que en Arena son los métodos bindItems o bindItemsToProperty).

La clase ListView es abstracta, acá estamos creando una subclase anónima a la que le definimos el método que le falta, que es populateItem. Este método recibe un panel (o sea, un componente que puede contener otros componentes) que ya tiene como modelo el elemento de la lista que hay que desplegar (en este caso, el disco). Para pedirle el objeto a mostrar (o sea el Disco) al panel se le envía el mensaje getModelObject().

Las columnas dinámicas de la fila correspondiente al disco van a ser los componentes que se le agreguen al panel.

En este ejemplo tenemos tres columnas: una es un Link (del que hablamos en el próximo párrafo), y dos son Labels como los que vimos antes. Observar que en lugar de

```
this.add(...)
```

dice

```
panel.add(...)
```

para que los componentes se agreguen a la tabla, no a la página directamente.

Links dinámicos

Ahora veamos cómo está definido el link que despliega la información sobre un disco.

```
final Link<String> nombreLink = new Link<String>("nombre") {  
    private static final long serialVersionUID = -5776431313490694323L;  
  
    @Override  
    public void onClick() {  
        BandPageWithAlbums.this.controller.setChosenAlbum(elDisco);  
    }  
};  
nombreLink.setBody(new PropertyModel<>(elDisco, "nombre"));
```

La acción consiste en elegir el disco *que corresponde a la fila* que se está desplegando, no es fijo. Por eso en el `onClick` se setea el disco que se obtiene del panel de la fila (ver parágrafo anterior). Además, se le puede setear un texto dinámicamente enviándole al componente el mensaje `setBody(...)` con un `Model` (en este caso un `PropertyModel`) como parámetro. En el ejemplo, con el `setBody` se está diciendo que el texto del link es el nombre del disco correspondiente.

Paneles e `isVisible`

En Wicket se pueden definir *paneles*. Como en Arena, un panel es un componente, que a su vez incluye otros componentes (o sea, la idea del patrón Composite). En Wicket, se puede definir un panel en una clase separada, con su HTML correspondiente. La clase que define al panel hereda de `org.apache.wicket.markup.html.panel.Panel`.

En esta página, es lo que hicimos para los datos del disco elegido, que se define en `AlbumPanel` (.java y .html).

En la página que incluye el panel, es fácil. En el HTML definimos un tag `div` con un `wicket:id`.

```
<div class="col-sm-4" wicket:id="infoDiscoElegido"></div>
```

En la clase agregamos una nueva instancia del panel como componente con el mismo id.

```
this.add(new AlbumPanel("infoDiscoElegido", this.controller));
```

Un detalle importante: le pasamos el controlador a la instancia de `AlbumPanel`, porque el panel va a compartir el controlador con la página.

El panel se define muy parecido a una página. Técnicamente, la única diferencia es que en el HTML se tiene en cuenta solamente la parte del `body` que está enmarcada con el tag `wicket:panel`. Se pueden agregar cosas al `head`, y también más cosas al `body`, pero no se tienen en cuenta para generar el HTML que viaja al browser. Ver `AlbumPanel.html`.

La subclase de `Panel` se define como una `WebPage`: se agregan componentes en el constructor.

El panel de disco elegido tiene una particularidad: se muestra solamente si hay un disco elegido; si no, no tiene que aparecer nada.

Eso se puede manejar con una propiedad de los paneles, y en general de cualquier componente Wicket, que se llama `isVisible`. Si se setea en `false`, el componente no se incluye al generar el HTML, esto se puede ver chusmeando el código fuente de la página en el browser.

En `AlbumPanel` redefinimos directamente el método `isVisible`, para que no hiciera falta setearlo en `true` o `false`.

```
@Override
public boolean isVisible() {
    return super.isVisible() && this.controller.hasChosenAlbum();
}
```

Aclaración:

en una página se puede definir un panel aparte que se puede controlar como un componente (p.ej. usando `isVisible` para que aparezca cuando querremos) sin necesidad de definir una clase y `.html` aparte. Ver la sección 6.4 del manual de referencia, `WebMarkupContainer`.

En general, en el capítulo 6 de la referencia se describen otros atributos que se le pueden poner a los componentes Wicket, y otras formas (`Border`, `Fragment`) de definir componentes compuestos.

¿Cómo armar una página?

OK, ya entendí las ideas básicas. Ahora, me dicen de armar una página a mí, ¿cómo lo pienso, por dónde empiezo?

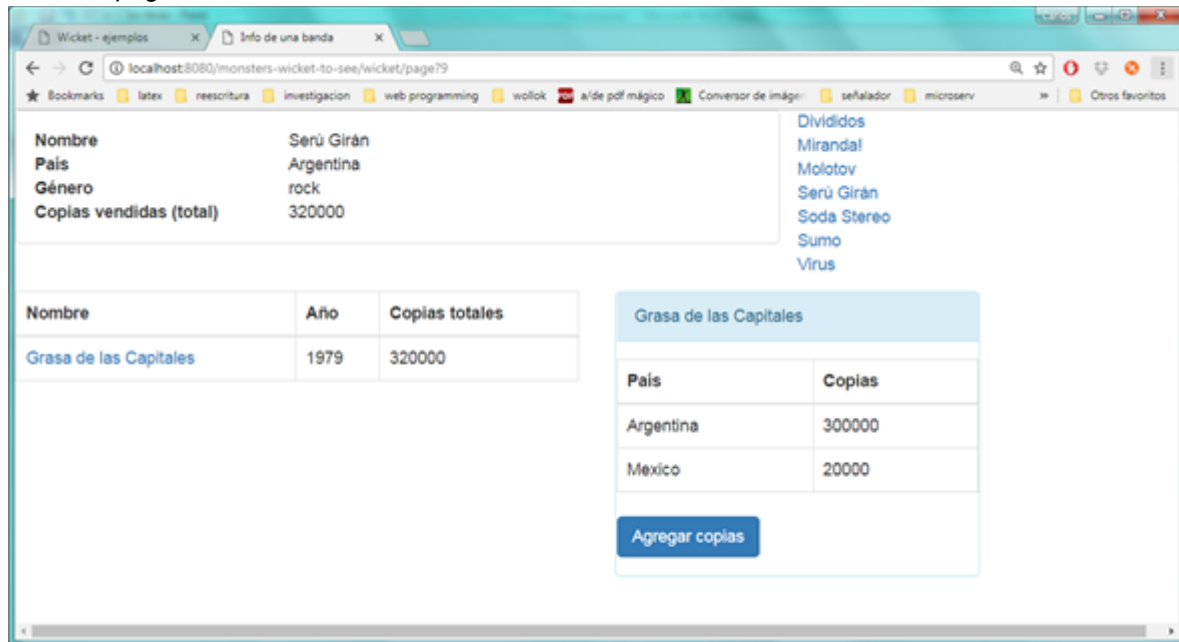
Hasta agarrarle la mano al esquema que propone Wicket (y este documento), creo que conviene ir despacito.

- Primero, dibujar en papel cómo queremos que quede la página, identificar la estructura de tags HTML, y qué partes son dinámicas.
- Con esto, empezar con el HTML. Armar bien los div que van a definir los paneles, los datos, las tablas, los links y botones, si es un form cómo van los componentes de ingreso (input text, checkbox, select).
Al armar el HTML, van a ir saltando cuáles de los tags corresponden a información dinámica, que debe ser suministrada por Wicket. A cada uno ir poniéndole el wicket:id.
Puede servir ir anotando en un papel los wicket:id, para no perdernos.
Acá vale usar pedazos de ejemplos que anden por ahí, adaptándolos a lo que se necesita para la pantalla que se diagramó.
- Después, armar en paralelo las clases Page y Controller. Definir bien con qué elemento de dominio se vincula (binding) cada componente Wicket, con eso va a quedar claro cómo armar los PropertyModel (u otros Model si hacen falta o convienen). Definir un controller que provea la información, recuerde las opciones elegidas y/o los valores ingresados por el usuario, y lance las acciones de dominio que se requieran desde la página.

Cargar info – form y algo de navegación

El siguiente paso es permitir al usuario **ingresar** información. En concreto, se incorpora un formulario que permite agregar copias a un disco, eligiendo país y cantidad de copias.

Ahora la página se ve así:



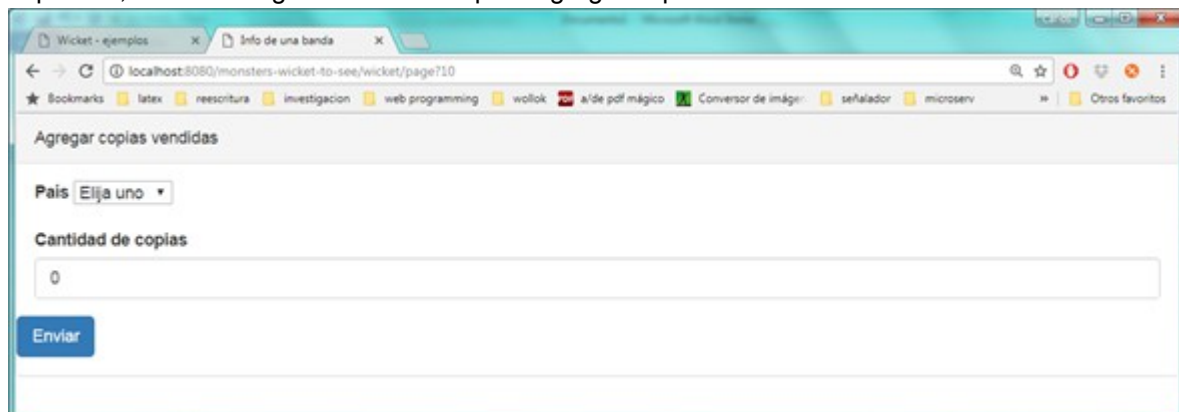
The screenshot shows a web browser window with two tabs: 'Wicket - ejemplos' and 'Info de una banda'. The address bar shows 'localhost:8080/monsters-wicket-to-see/wicket/page?9'. The page displays information for the band 'Serú Girán' from 'Argentina' in the 'rock' genre, with '320000' copies sold. A list of bands is shown on the right: Divididos, Miranda!, Molotov, Serú Girán, Soda Stereo, Sumo, and Virus. Below this, a table lists 'Grasa de las Capitales' from 1979 with 320000 copies. To the right, a form titled 'Grasa de las Capitales' has a table with 'País' and 'Copias' columns, showing 'Argentina' with 300000 and 'Mexico' with 20000. A blue button labeled 'Agregar copias' is at the bottom of the form.

Nombre	Año	Copias totales
Grasa de las Capitales	1979	320000

País	Copias
Argentina	300000
Mexico	20000

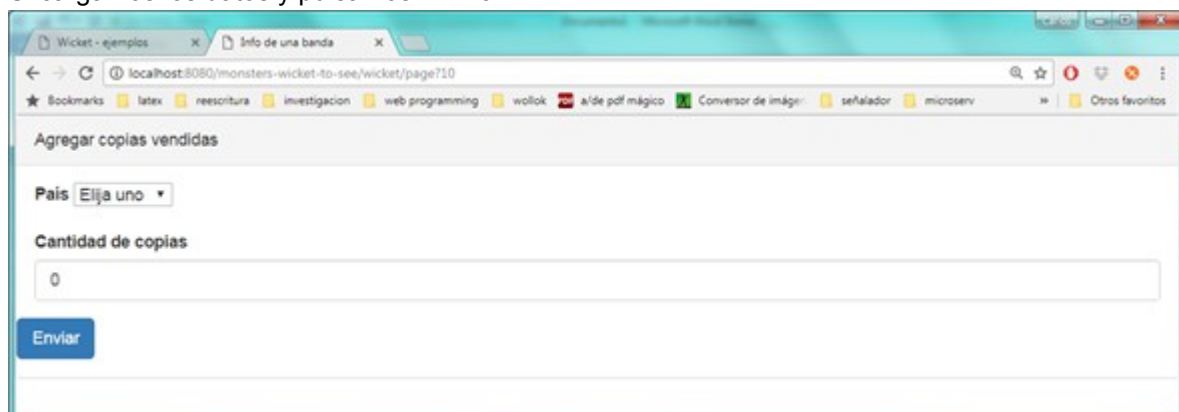
Agregar copias

Abajo a la derecha aparece un nuevo botón “Agregar copias”. Si lo pulsamos, saltamos a una página separada, donde se ingresan los datos para agregar copias.



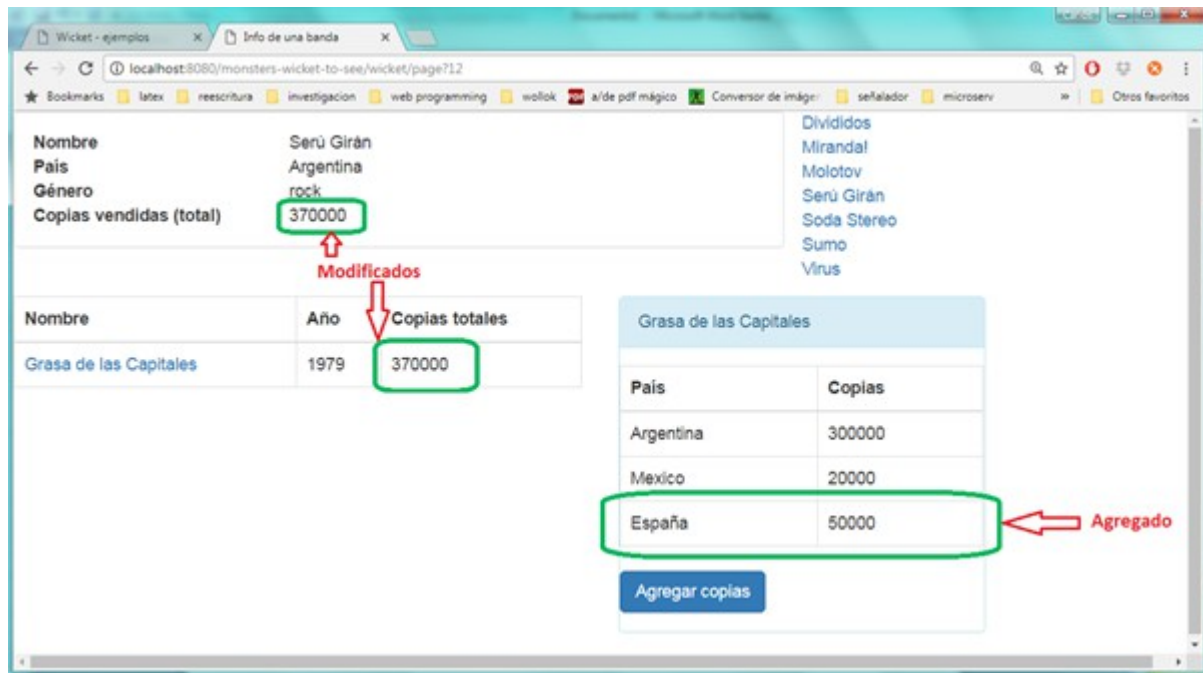
The screenshot shows a web browser window with the address bar displaying 'localhost:8080/monsters-wicket-to-see/wicket/page?10'. The page title is 'Agregar copias vendidas'. It features a dropdown menu for 'País' with the text 'Elija uno', a text input field for 'Cantidad de copias' with the value '0', and a blue button labeled 'Enviar'.

Si cargamos los datos y pulsamos “Enviar”



This screenshot is identical to the previous one, showing the 'Agregar copias vendidas' form with the 'País' dropdown set to 'Elija uno', the 'Cantidad de copias' input at '0', and the 'Enviar' button.

Volvemos a la página anterior, donde se ve el estado actualizado de la banda y el disco.



En esta página hay otro cambio respecto de la versión anterior: la lista de bandas tiene más elementos. Se cambió de tres links fijos a una lista variable, un link para cada banda. Esto se lo dejamos a ustedes para descubrir cómo está hecho, con un comentario que hacemos al final de la sección.

El **código** está en el package `ar.edu.unq.ciu.monsters.web.bandsDiscsCopies`. Desde la página inicial, se accede con la opción **Página con banda, discos, y carga de copias**.

Un poquito de navegación

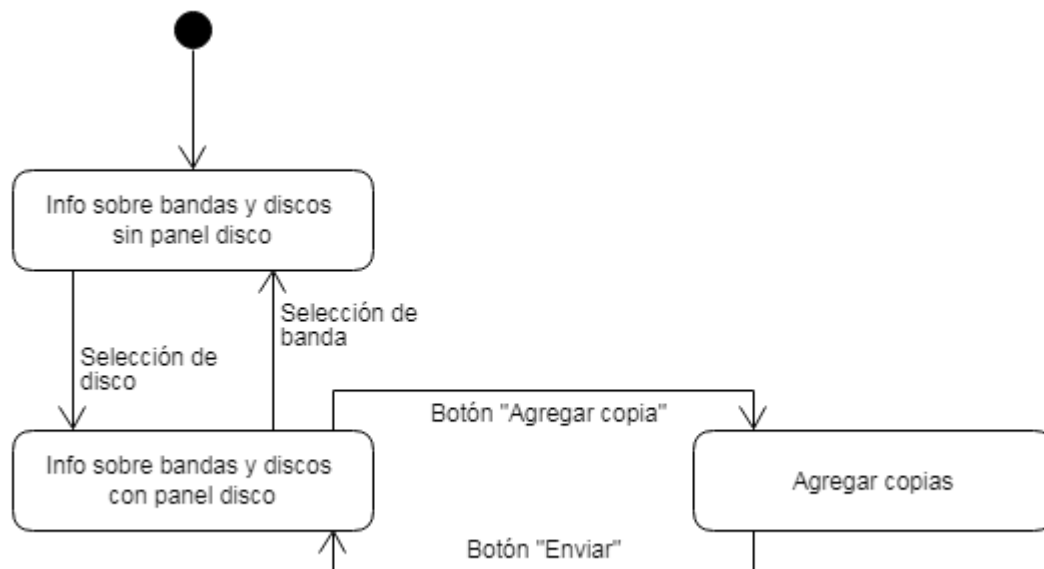
Ahora nuestra aplicación tiene dos páginas: la que muestra información de bandas y discos, y la que permite agregar copias. A su vez, la información sobre un disco se muestra en un panel aparte. Por lo tanto vamos a tener tres ternas `.html` / `clase Page` / `clase Controller`. Son estas

Info bandas y discos	Agregar copias	Info de un disco
<code>BandPageWithLinkToCopies.html</code>	<code>CopiesFormPage.html</code>	<code>AlbumPanelWithLink.html</code>
<code>BandPageWithLinkToCopies.java</code>	<code>CopiesFormPage.java</code>	<code>AlbumPanelWithLink.java</code>
<code>BandPageController.java</code>	<code>CopiesFormController.java</code>	<code>BandPageController.java</code>

El panel de info de un disco comparte el controlador con la página en la que se inserta, por eso le corresponde la misma clase de controlador. Esta tabla explica los ocho archivos, cinco `.java` y tres `.html`, que se encuentran en el package.

Cuando tenemos más de una página se incorpora el concepto de **navegación**², o sea cuáles son los eventos que provocan que se salte de una página a otra, y a qué página hay que ir en cada caso. Esto lo podemos mostrar armando un *diagrama de navegación*³. En este caso el diagrama de navegación es sencillo:

² También puede ser útil pensar en la navegación entre distintos estados de una misma página, si es compleja.
³ En rigor, lo que estamos armando para mostrar la navegación es un *diagrama de transición de estados*, o *state transition diagram* en inglés. Si googlean esto van a encontrar un montón de info.



El punto negro representa el comienzo de la navegación.

Para saltar de la página actual a otra, le enviamos a la `WebPage` de la página actual el mensaje `setResponsePage`, pasándole como parámetro un objeto que representa la página a la que queremos ir. Este mensaje se describe en la sección 4.4 del manual de Wicket.

Para implementar la navegación entre las dos páginas tenemos que tener en cuenta dos cosas

1. Al saltar a la página de agregar copias, hay que indicar para qué disco.
2. Al saltar de la página de agregar copias a la de info de banda y discos, queremos que queden seleccionados la misma banda y el mismo disco de antes.

Para lograr esto, en el constructor de `CopiesFormPage` vamos a pasar como parámetros la banda y el disco elegidos. A su vez, cuando esta página crea su controlador, le manda los mismos valores. O sea:

```

public class CopiesFormPage extends WebPage {
    private static final long serialVersionUID = 8263570456720739583L;
    CopiesFormController controller;

    public CopiesFormPage(Banda band, Disco album) {
        super();
        this.controller = new CopiesFormController(band, album);
        this.add(this.buildForm());
    }

    public Form<CopiasVendidas> buildForm() {
        // ya vamos a esto ...
    }
}
  
```

Obviamente, el controlador tiene dos atributos, uno para la banda y otro para el disco, que se setean en el constructor. Esto se puede ver en el código.

Pra cerrar esta parte, comentamos que Wicket maneja el botón de agregar copias exactamente igual que un link. O sea, al tag `<button>` en el HTML (este está en `AlbumPanelWithLink.html`)

```
<button type="button" class="btn btn-primary" wicket:id="addCopies">
  Agregar copias
</button>
```

le corresponde, en la clase gráfica (en este caso AlbumPanelWithLink.java) un componente Link

```
final Link<String> addCopiesButton = new Link<String>("addCopies") {
  private static final long serialVersionUID = 3138145803437736720L;

  @Override
  public void onClick() {
    this.setResponsePage(new CopiesFormPage(
      AlbumPanelWithLink.this.controller.getChosenBand(),
      AlbumPanelWithLink.this.controller.getChosenAlbum()
    ));
  }
};
```

Acá vemos cómo se usa setResponsePage, y cómo se crea la nueva página pasándole banda y disco elegidos por constructor⁴.

El form

Del lado del HTML se arma un form, donde cada input field va a tener un wicket:id . Le puse también id por las dudas, no sé si es necesario. En el select observar que no se le pusieron las opciones, eso lo va a generar Wicket. Queda así

```
<form wicket:id="addCopiesForm">
  <div class="form-group">
    <label for="country">País</label>
    <select id="country" wicket:id="country"></select>
  </div>
  <div class="form-group">
    <label for="copyCount">Cantidad de copias</label>
    <input type="text" class="form-control"
      id="copyCount" wicket:id="copyCount" placeholder="25000">
  </div>
  <div class="row">
    <input type="submit" class="btn btn-primary"></input>
  </div>
</form>
```

⁴ Obviamente que pasar los valores por constructor no es la única opción. También se puede crear la página antes, setearle los valores que necesita, y mandar la página ya configurada. Otra opción es crear el controlador por afuera, setearlo, y pasárselo armadito a la página. Esto va en gustos. En particular, si la configuración se hace compleja, tal vez ya no convenga pasar todos los valores por constructor.

Lo que nos queda por saber es cuáles son los componentes Wicket para (1) el form (2) un <select> (3) un <input>. Este es el código de la creación del form.

```
public Form<CopiesFormController> buildForm() {
    Form<CopiesFormController> newForm =
        new Form<CopiesFormController>("addCopiesForm")
    {
        private static final long serialVersionUID = 9176124418022888414L;

        @Override
        protected void onSubmit() {
            // acción de dominio
            CopiesFormPage.this.controller.doAddCopies();
            // navegación
            this.setResponsePage(new BandPageWithLinkToCopies(
                CopiesFormPage.this.controller.getBand(),
                CopiesFormPage.this.controller.getAlbum()
            ));
        }
    };

    newForm.add(new DropDownChoice<>(
        // id
        "country",
        // binding del valor, acá va a ir la opción seleccionada
        new PropertyModel<>(this.controller, "country"),
        // binding de la lista de items
        new PropertyModel<>(MonstersStore.store(), "paísesOrdenados"),
        // qué se muestra de cada item
        new ChoiceRenderer<>("nombre")
    ));

    newForm.add(new TextField<>(
        "copyCount",
        new PropertyModel<>(this.controller, "copiesToAdd")
    ));

    return newForm;
}
```

Vamos por partes

Select

Wicket define el componente `org.apache.wicket.markup.html.form.DropDownChoice`, específicamente para manejar campos select. Se describe en la sección 11.4 del manual de Wicket. Se puede configurar como se muestra en el ejemplo, un constructor con cuatro parámetros. Aparecen los dos bindings, el de valor y el de lista de ítems, igual que en Arena.

Input

Se manejan usando un `org.apache.wicket.markup.html.form.TextField`, que se configura igual que un Label. Este componente se describe en la sección 11.3.2 del manual de Wicket.

Form

Wicket incluye la clase `org.apache.wicket.markup.html.form.Form`, que es abstracta, hay que crear una subclase (en el ejemplo, esta subclase es anónima) en la que se define el método `onSubmit()`. En este caso, hay que hacer una acción de dominio, que es agregar las copias, y después saltar a la página de info de discos y bandas de acuerdo a lo que indica el diagrama de navegación.

La acción de dominio se la estamos delegando al controlador, más adelante volvemos a esto.

Cuando saltamos a la página de info de bandas y discos, tenemos que lograr que muestre la misma banda, y el mismo disco, que antes de entrar a la página de agregar copias. Para esto, creamos una nueva instancia pasándole la banda y el disco en el constructor. Veamos cómo queda `BandPageWithLinkToCopies` .

```
public class BandPageWithLinkToCopies extends WebPage {
    private static final long serialVersionUID = -5617749468806067306L;

    private BandPageController controller = new BandPageController();

    // constructor sin parámetros
    public BandPageWithLinkToCopies() {
        this.controller.setChosenBand(
            MonstersStore.store().getBandaLlamada("Soda Stereo")
        );

        this.fillBandFile();
        this.fillBandSelectionLinks();
        this.fillAlbumList();
        this.add(new AlbumPanelWithLink("infoDiscoElegido", this.controller));
    }

    // constructor con dos parámetros
    public BandPageWithLinkToCopies(Banda band, Disco album) {
        // invoca al constructor sin parámetros ...
        this();
        // ... y después setea el controlador
        this.controller.setChosenBand(band);
        this.controller.setChosenAlbum(album);
    }

    // ... métodos ...
}
```

La clase `BandPageWithLinkToCopies` queda con dos constructores. Si se usa el constructor sin parámetros, entonces se elige Soda Stereo y no queda ningún disco elegido. Si se usa el constructor con dos parámetros, quedan elegidos banda y disco que vienen por parámetro.

Notar que el `this()` implica llamar al constructor sin parámetros, que genera todos los componentes Wicket. **Después** de crearse todos los componentes, se setean banda y disco.

Como los componentes se crearon haciendo binding, p.ej. (recordemos)

```
this.add(new Label(
    "nombre",
    new PropertyModel<>(this.controller, "chosenBand.nombre")
));
```

en el momento en que se arma el HTML que viaja al browser, se toman los valores de la banda y del disco que tenga seteado el controlador.

Se recomienda leer este párrafo hasta entenderlo, si es necesario varias veces y/o preguntándole a alguien.

En el manual, el manejo de forms se introduce en la sección 11.3. Más en general, la mayor parte de los capítulos 11 y 12 está relacionada con la definición y el manejo de forms.

Server y browser - ¿cómo es que anda el form?

Un form implica **dos interacciones** entre server y browser. La primera va *del server al browser*: el server genera una página HTML que incluye el form, y la envía al browser. La segunda va *del browser al server*: cuando el usuario submite el form, el browser empaqueta los valores ingresados y los manda al server.

Wicket interviene fuertemente en la segunda interacción, haciendo todo lo necesario para que desde el código de la WebPage no se note que pasó todo esto.

Lo que sigue es una descripción muy somera de los pasos de esta interacción en dos etapas.

Paso	En el servidor	Comunicación	En el cliente
1			El usuario pulsa el botón "Agregar copias"
2		Señal de "se pulsó el botón 'Agregar copias' ".	
3	Se ejecuta el código del método <code>onClick()</code> del objeto <code>Link</code> vinculado al botón HTML.		
4	Se genera el HTML de <code>CopiesFormPage</code> , que incluye al form. Se evalúan todos los <code>PropertyModel</code> , incluyendo los dos del <code>DropDownChoice</code> , el de valor y el de lista de ítems.		
5		El HTML generado viaja al browser.	
6			Se muestra la nueva página, que incluye el form.
7			El usuario carga los datos y pulsa el botón "Enviar", que es el submit del form.
8		Los datos del form submitido viajan al server.	
9	Wicket ejecuta los bindings correspondientes a los campos del form submitido, seteando los valores a partir de los datos del form recibidos desde el browser.		
10	Se ejecuta el método <code>onSubmit()</code> del componente <code>Form</code> que corresponde al form submitido.		

La intervención de Wicket en el procesamiento del form se describe en el paso 9. De esta forma, en el momento en el que se ejecuta el método `onSubmit()` del `Form`, los valores ingresados por el usuario ya están seteados en los objetos que hayan sido bindeados con los (componentes Wicket que corresponden a los) campos del form.

¿Por qué puede interesar saber esto, si justamente Wicket maneja todos los detalles para que no sea necesario pensar en la comunicación entre server y browser?

Menciono tres razones:

1. Si una página incluye más de un form, al submitirse uno de ellos, lo que viaja de browser a server es, **solamente, la información del form submitido**. Por lo tanto, los componentes Wicket bindeados a los otros forms no se van a actualizar.
2. La carga del form por parte del usuario ocurre completamente en el browser. El server ni se entera de lo que hace el usuario hasta que le da "submitir". Por eso, las validaciones que se

escriban en el código Java se van a ejecutar recién en el submit, no hay forma (al menos sencilla) de que se ejecuten antes.

Para hacer cosas mientras el usuario está editando, lo más usado (creo) es incluir *código Javascript que se ejecuta en el browser*.

Un ejemplo (no muy sencillo) se cuenta en el capítulo 17 del manual de Wicket.

3. Wicket hace un trabajo particularmente completo de manejar en forma transparente (o sea, sin que tengamos que pensar en ello) la comunicación entre server y browser. En otras tecnologías, hay que ser más consciente de esta cuestión, que es **el** problema que tienen en particular las aplicaciones Web.

Acción de dominio

Volvamos a la acción de dominio que hay que ejecutar cuando se hace submit del form.

En la clase que representa la página, en este caso x, delegamos inmediatamente en el controlador.

Recordemos:

```
Form<CopiesFormController> newForm =
    new Form<CopiesFormController>("addCopiesForm")
{
    @Override
    protected void onSubmit() {
        // acción de dominio
        CopiesFormPage.this.controller.doAddCopies();
        // ... navegación ...
    }
};
```

En general, delegar en el controlador es una buena idea, porque como el valor de cada campo se bindea con una propiedad del controlador, entonces cuando se ejecuta el `onSubmit()`, el controlador ya tiene seteados los valores ingresados por el usuario.

La acción de agregar copias tiene cierta inteligencia: si ya hay copias cargadas para el país elegido hay que sumar a la cantidad registrada hasta el momento, si no hay que agregar al país.

En general, cuando la acción es compleja, surge la pregunta de dónde poner el código correspondiente. En principio, hay que decidir si esta lógica va al controlador, o bien si la agregamos al modelo de dominio.

En este caso, se eligió agregar un método a la clase Disco, o sea, al modelo.

Moraleja: es absolutamente válido ajustar el modelo para que responda adecuadamente a las interacciones del usuario.

Lo que “no vale” es manejar en el modelo información específica de la interacción con el usuario. En este caso, estaría mal que el modelo se acordara de qué disco eligió el usuario para agregarle copias, eso es responsabilidad del controlador.

En aplicaciones más complejas, el tema de dónde manejar las acciones de negocio que son disparadas a partir de las interacciones con el usuario es clave para mantener un buen diseño. Esto lo vamos a ir trabajando. Por ahora, alcanza con:

- que la vista (en este caso, la clase Page) delegue en el controlador
 - entender que si la acción de dominio es compleja, debemos decidir si la manejamos en el controlador, o si agregamos la inteligencia necesaria al dominio.
- También hay una opción intermedia: manejar la inteligencia en el controlador pero agregando métodos a las clases del modelo que hagan parte del trabajo.

Lista dinámica de links

Terminamos con un comentario acerca de la lista de links que aparece arriba a la izquierda en la página de bandas y discos. Ahora muestra todas las bandas que conoce el MonsterStore, no son links fijos como en las versiones anteriores.

En el .java, esta lista se maneja con un ListView, pero en el .html no hay un <table>, sino simplemente un <div>. Esto demuestra que el ListView se puede usar para armar una estructura de repetición para cualesquiera tags HTML, no está limitado a <table>, y .

Los detalles véanlos ustedes en el código.