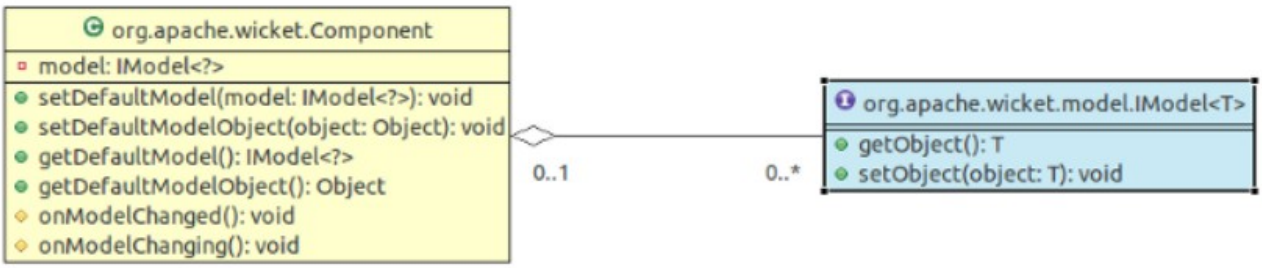
Apache Wicket
Modelos y formularios

En wicket el concepto de "modelo" es probablemente la característica más importante del framework y está estrictamente relacionado con el uso de componentes.

¿Qué es un modelo?
Un modelo es básicamente una *"facade interface"* la cuál permite a los componentes acceder y modificar sus datos sin conocer detalle de como los mismos son administrados o persistidos.
Cada componente tiene como máximo un modelo relacionado, mientras un modelo puede estar compartido por varios componentes. En Wicket un modelo es una implementación de la interface: *org.apache.wicket.model.IModel*.



La interface IModel define solo los métodos necesarios para obtener *(get)* y asignar *(set)* un objeto *(data object)*, desacoplando a los componentes de los detalles acerca de la estrategia de persistencia adoptada por el *data object*.

Veamos un ejemplo trabajando con el componente *Label* usando su constructor, el cuál recibe dos parámetros *String*:

```
add ( new Label ( "message", "Hola Lamadrid!!!" ) )
```

Este constructor internamente construye un modelo el cuál realiza un *wrap* sobre el segundo parámetro, a continuación vemos el código del constructor del componente *Label*:

```
public Label ( final String id, String label )  {
    this ( id, new Model<String> ( label ) );
}
```

La clase: *org.apache.wicket.model.Model* es una implementación de la interface: *IModel*, esta puede *wrapear* cualquier objeto que implemente la interface: *Serializable*. La razón de esta restricción es que el modelo se almacena en la *web session*.
Nota: Ver *detaching capability* para trabajar con objetos no serializables como data object.

Es bastante claro que si nuestro Label mostrará un texto estático no tiene mucho sentido definir un modelo, pero seguramente muchas veces mostraremos en el mismo textos dinámicos, proveídos por el usuario, leídos desde la base de datos, etc.; los modelos de Wicket están diseñados para abarcar estos casos.
Imaginemos que necesitamos mostrar la hora actual cada vez que la página es renderizada. Podemos implementar un modelo que retorne una nueva instancia de Date cuando se llama al método: *getObject()* del IModel.

```
IModel timeStampModel = new Model<String> () {
      @Override
      public String getObject ()  {
            return new Date().toString();
      }
};
add(new Label("timeStamp", timeStampModel));
```

Modelos y JavaBeans
Uno de los principales logros de Wicket es el uso de JavaBeans y POJO como *data model*, para hacer esta tarea lo más fácil posible, Wicket ofrede dos clases *(model classes)*:
org.apache.wicket.model.PropertyModel
org.apache.wicket.model.CompoundPropertyModel

PropertyModel
Digamos que queremos mostrar el atributo nombre de una persona con un label, perfectamente podemos utilizar la clase Model como vimos anteriormente:

```
Person persona = new Person();
//set data persona
Label label = new Label("name", new Model(persona.getNombre()));
```

Sin embargo esta solución tiene un gran inconveniente, el texto mostrado en el Label será estático y si nosotros cambiamos el valor del atributo, el Label no actualizará su contenido. Entonces para mostrar siempre el valor actual del atributo nombre deberíamos usar la clase: *org.apache.wicket.model.PropertyModel*

```
Person persona = new Person();
//set data persona
Label label = new Label("name", new PropertyModel(persona, "nombre"));
```

PropertyModel tiene un constructor con dos parámetros: el modelo (model object), persona en nuestro caso y el nombre del atributo que queremos acceder, este último parámetro se llama: *property expression*, las mismas soportan la notación *dotted*, ejemplo: "otraPersona.nombre", también se puede acceder a Arrays o List mediante su índice: "hijos.0.nombre" y Map.

CompoundPropertyModel y herencia de modelo

La clase: *org.apache.wicket.model.CompoundPropertyModel* es un caso particular de *model*, el cuál se usa en conjunto con otra característica de Wicket llamada *model inheritance*. Con esta característica, cuando un componente necesita usar un modelo pero no se le ha sido asignado ninguno, dicho componente buscará a través de toda la jerarquía de contenedores hasta dar con un contenedor que tenga asignado un modelo, es decir, dicho componente trabajará con un modelo heredado de su contenedor padre.
Por ejemplo:

```
//set CompoundPropertyModel como modelo para el contenedor del Label
setDefaultModel(new CompoundPropertyModel(persona));
Label label = new Label("nombre");
add(label);
```

mirando el ejemplo notaremos que el id del Label es igual al property expression, esto nos puede ahorrar mucho código, sin embargo también es posible usar un id diferente al property expression, para ello invocamos el método bind() del CompoundPropertyModel con la property expression asociando la misma con el component id, por ejemplo:

```
Person person = new Persona("Mariano", "Moreno");
CompoundPropertyModel compoundModel = new CompoundPropertyModel(persona);
setDefaultModel(compoundModel );
add(new Label("xyz", compoundModel.bind("nombre")));
```

CompoundPropertyModel son muy utilizados en combinación con forms, como veremos a continuación.

Wicket forms

Las aplicaciones web utilizan formularios HTML para recolectar información introducida por el usuario y enviarla al servidor.
Wicket provee la clase: *org.apache.wicket.markup.html.form.Form* para manejar los formularios web, a continuación vemos la definición del formulario:

HTML

```
<form wicket:id="form">
   <input type="submit" value="submit"/>
</form>
```

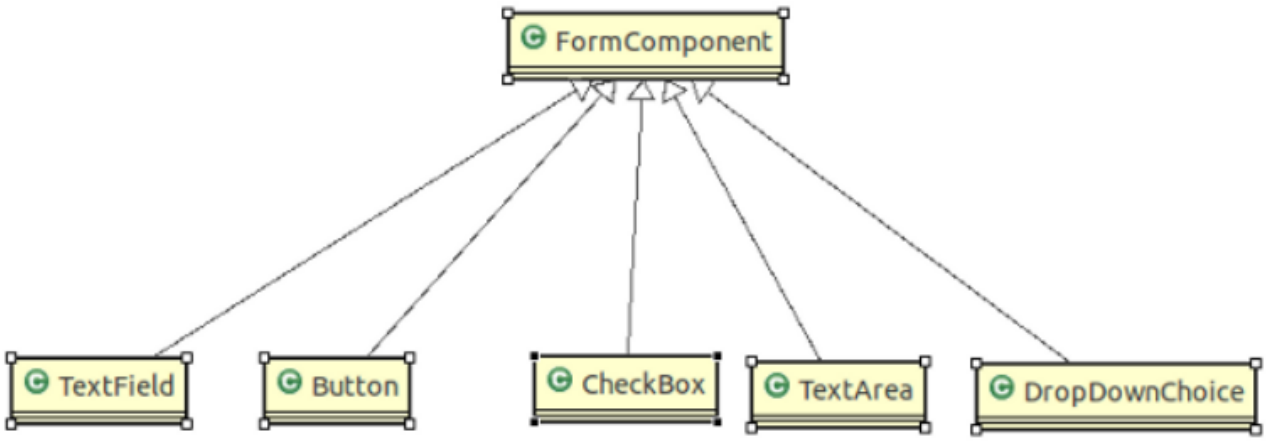
Java

```
Form form = new Form("form") {
   @Override
   protected void onSubmit()  {
      System.out.println("Form submitted.");
   }
};
add(form);
```

El método onSubmit se llama cuando el formulario ha sido submiteado, este debe ser sobre escrito para ejecutar la acción correspondiente.

Form y modelos

Un formulario debe contener campos de entrada, como lo son, campos de texto, check boxes, radio buttoms, text areas, etc. para interactuar con el usuario. Wicket provee una abstracción de todos estos elementos con el componente: *org.apache.wicket.markup.html.form.FormComponent*



A continuación veremos un breve ejemplo donde construiremos una simulación de pantalla de login, la misma estará compuesta por tres archivos: Login.html, Login.java y LoginForm.java; recordar que por el concepto de *markup* de Wicket los dos primeros archivos deben tener el mismo nombre *(case sensitive)*:

Login.html

```
<html>
  <head>
    <title>Login page</title>
  </head>
  <body>
    <form id="loginForm" method="get" wicket:id="loginForm">
      <fieldset>
        <legend style="color: #F90">Login</legend>
        <p wicket:id="loginStatus"></p>
        <span>Username: </span>
        <input wicket:id="username" type="text" id="username" />
        <br/>
        <span>Password: </span>
        <input wicket:id="password" type="password" id="password" />
        <p><input type="submit" name="Login" value="Login"/></p>
      </fieldset>
    </form>
  </body>
</html>
```

Nota: en rojo los id de los componentes que definiremos en la clase WebPage

Login.java

```
public class HomePage extends WebPage {
  public HomePage(final PageParameters parameters) {
    super(parameters);
    add(new LoginForm("loginForm"));
  }
}
```

como podemos ver, en nuestra página HomePage (WebPage) agregamos el componente LoginForm (Form), a dicho componente le asignamos el id: loginForm para luego identificarlo en el html como se mencionó anteriormente.

Ahora bien nos queda definir nuestro formulario:

LoginForm.java

```
public class LoginForm extends Form {

   private String username;
   private String password;
   private String loginStatus;

   public LoginForm(String id) {
      super(id);
      setDefaultModel(new CompoundPropertyModel(this));
      add(new TextField("username"));
      add(new PasswordTextField("password"));
      add(new Label("loginStatus"));
   }

   public final void onSubmit() {
      if(username.equals("test") && password.equals("test"))
         loginStatus = "Congratulations!";
      else
         loginStatus = "Wrong username or password !";
   }
}
```

como vemos en el código, en el formulario cargamos los componentes con sus respectivos id, que luego serán visualizados en el html, notar que utilizamos la clase: CompoundPropertyModel para definir el modelo, y dicho modelo es el mismo formulario, además vemos que el id coincide con la property expression ya que no utilizamos el *model bind.*

Fuentes:

Web Wicket
https://wicket.apache.org/

Wicket 7.x Reference Guide
https://ci.apache.org/projects/wicket/guide/7.x/single.pdf