Prototipo de Proyecto de Desarrollo de Aplicaciones Web: Guild Manager

Martín Rubio Fernández

CONTROL DE VERSIONES Y DISTRIBUCIÓN
"*Guild Manager*"

NOMBRE DEL DOCUMENTO: Documento Funcional.
VERSIÓN: 7.0.
ELABORADO POR: Martín Rubio Fernández.
FECHA: 14 de noviembre de 2016.

Versión	Causa de la nueva versión	Fecha
1.0	Versión inicial	18.09.16
2.0	Construcción y presentación del prototipo HTML.	25.09.16
3.0	Implementación y presentación de las hojas CSS. Modificación en la arquitectura de ficheros debido a la misma.	09.10.16
4.0	Aplicación de dinamismo mediante manejo del árbol DOM y las librerías de js, jQuery y AngularJS. Modificación en la arquitectura de ficheros debido a la misma.	23.10.16
5.0	Añadido de los formularios de encuesta, login y registro. Modificación en la arquitectura de ficheros debido a la misma.	29.10.16
6.0	Manejado de formularios mediante servlets de la encuesta a usuarios. Modificación de la arquitectura de ficheros para el empaqueta en war.	6.11.16
7.0	MAnejo formulario registros y modificación de la arquitectura en respuesta a ello.	14.11.16

1. Ámbito y alcance del proyecto

1.1. Descripción General:

Objetivo principal: El objetivo principal de este proyecto es el de emplear la api de acceso a datos de Guild Wars 2(un mmorpg), para proporcionar a jugadores un medio de comunicación entre ellos y una herramienta de visualización y organización de la información diferente a la que proporcionan los distintos paneles in-game. La meta principal es la de dar soporte a las guilds(asociaciones grandes de jugadores) a la hora de organizarse, comunicarse y administrarse.

Descripción: Para ello, empleando la api de acceso proporcionada por el juego, pretendo almacenar información de jugadores, sus personajes, y las guilds(organizaciones) a las que pertenecen, para, de esta forma poder proporcionar distintas herramientas de organización.
La api permite dos tipos de acceso, público y privado. El acceso público permite la obtención de datos de carácter global y accesible por todos los jugadores dentro del juego(nombres de jugador, objetos, mapas,etc..). El acceso privado permite la lectura de datos privados de las cuentas. Será en este último en el que se basará la aplicación.
El acceso privado a la api permite el acceso a varios ítems propio de cada cuenta. Los relevantes para este proyecto son la lista de guilds a las que pertenece y personajes de la cuenta, el inventario(objetos en su posesión), el equipo(armas, armaduras… que lleva cada personaje) y sus habilidades.
Este acceso se lleva a cabo mediante una API-KEY, que cada usuario del juego puede obtener en el panel de control de la web oficial del juego, y permite los distintos permisos que se le dan a dicha KEY(la propia api permite conocer los permisos de cada key).Por lo tanto, se busca enlazar en una base de datos propia cuentas de usuario de nuestra aplicación con una API-KEY que nos de acceso a sus datos en los servidores del Guild Wars 2.

Puntos básicos: Las herramientas fundamentales que este servicio proporcionará serán los siguientes:

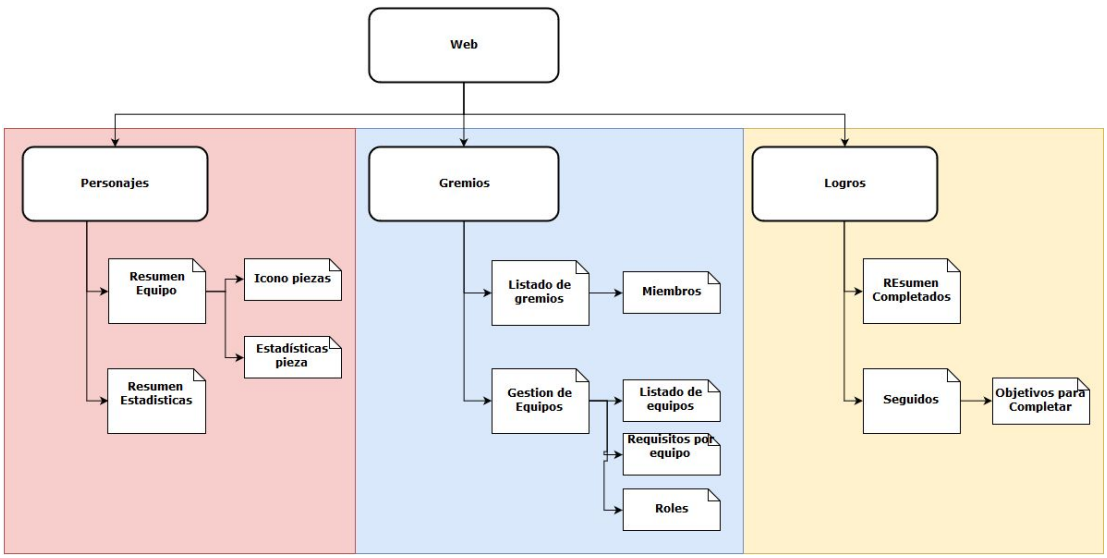
- Panel de personajes: Un resumen básico de cada personaje de la cuenta que muestre el equipo así comas las estadísticas tanto de cada pieza como las totales de cada personaje.
- Panel de Guilds: Para cada guild a la que la cuenta pertenezca:

- Lista de miembros: Mostrando un resumen similar al mostrado in-game, pero empleando el nombre de cuenta y no de personaje o bien el nombre de usuario empleado en la web.
- Administración de equipos: Permitirá al dueño y miembros permitidos crear y administrar equipos. Consistirá en organizar a los distintos miembros en listas para distintos contenidos del juego que el líder del gremio considere(pvp, pve endgame…). Dentro de los grupos, se establecerán roles, y para cada rol y/o clase de personaje los administradores podrán establecer requisitos(estadísticas, mínimas, equipo necesario/recomendado...etc).
- Administración de banco y mejoras del gremio: Muestra el contenido del banco del gremio así como indicar las mejoras que se persiguen en el momento y que materiales se necesitan, de forma que los miembros del gremio sepan en qué enfocar sus esfuerzos dentro del juego.

- Panel de logros: Mostrará un resumen de los logros del usuarios y le permitirá marcar cualquier logro no completado como seguido. Los logros seguidos aparecerán junto con el resumen indicando que falta para completarlos.

2. Contenido

2.1. Arquitectura de la información:



3. Especificaciones del Sistema

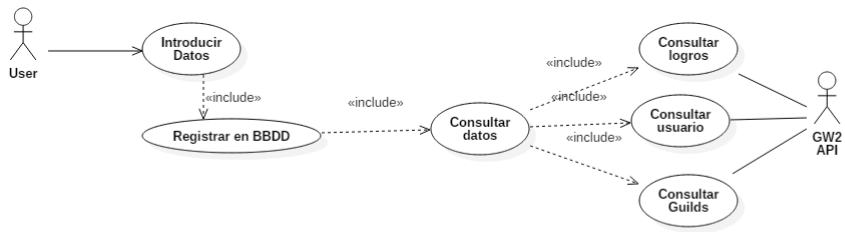
3.1. Requisitos:

Como se indica en la descripción inicial, una vez finalizado el proyecto el usuario deberá ser capaz de realizar las siguientes operaciones:

- Consultar el estado de sus distintos personajes
- Consultar información de su gremio:
  - Lista de miembros.
  - Lista de equipos y requisitos de los mismos. Así como crear o ingresar en uno.
  - Estado del banco del gremio y de las mejoras desarrolladas, en desarrollo y en proyección.
- Consultar información sobre sus logros:
  - Lista de logros completados.
  - Marcar logros como seguidos.

3.2. Diagrama de casos de uso

Registro de nuevo usuario:



Identificación de usuario existente:

3.3. Descripción de casos de uso

Ambos casos de uso destacados representan un sistema de manejo de la información en dos partes. La primera parte sería la de lectura o escritura en la base de datos de nuestra web para comprobar la existencia del usuario(o crearlo) y el segundo y más relevante es el de la actualización de nuestra base de datos con los datos proporcionados por la api. De esta forma, cada vez que un usuario entre al servicio dispondrá de los datos actualizados y disponibles, en lugar de tener que realizar una llamada al servidor de gw2 cada vez que desee realizar una acción.

4. Modelo de datos del Sistema

4.1. Modelo lógico de datos

El sistema consistirá en mantener una base de datos propia para almacenar dos tipos de datos:

- Aquellos obtenidos a través de la API: Informacion de personajes, cuentas y gremios.
- Aquellos generados por los servicios de la plataforma: Información de usuario, APIKEYS, equipos, objetivos de gremio, etc...

Para ello se emplea una base de datos relacional en la que se mezclaran ambos datos en las distintas entidades necesarias, siguiendo el modelo empleado por la api a la hora de proporcionar la información, y extendiendolo para permitir almacenar los datos necesarios para proporcionar el servicio deseado.

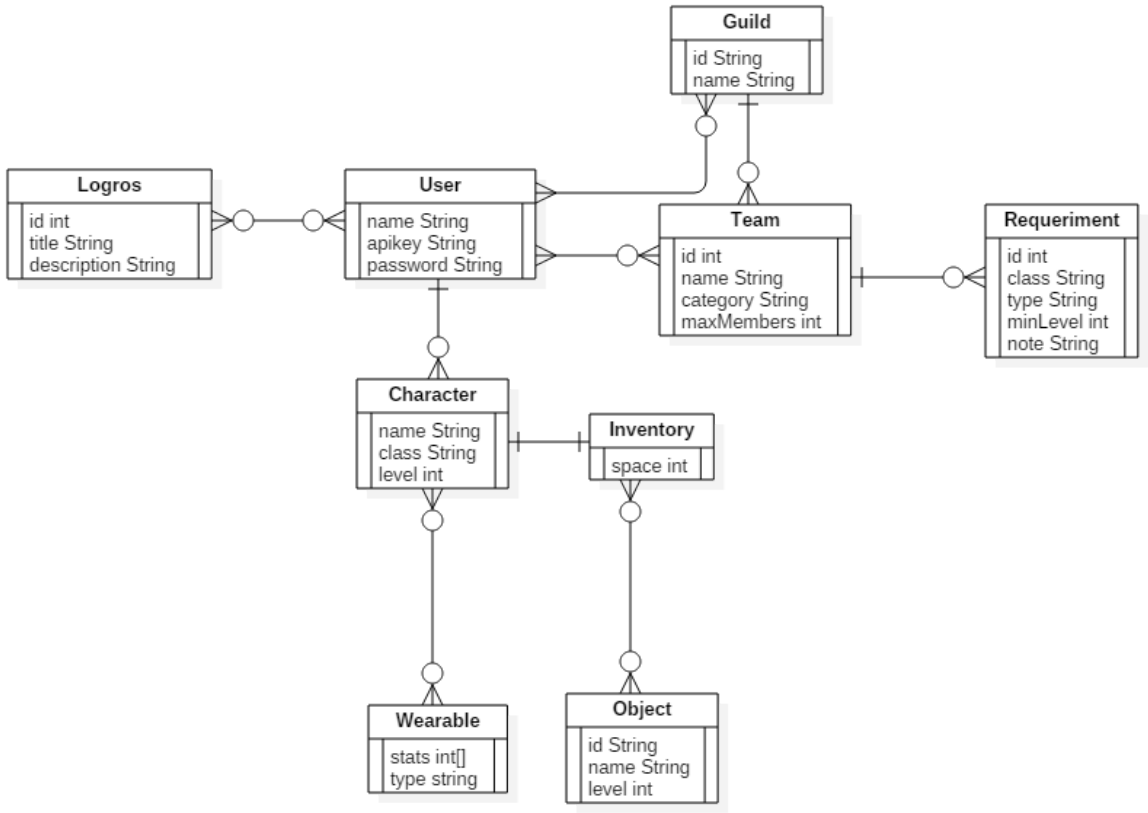
4.1.1. Definición de entidades/claves

Las entidades principales serán:

- **User**: Almacena credenciales y la api key de cada cuenta.
- **Character**: Nombre, nivel y clase de cada personaje.

- **Guild**: Id y nombre de cada gremio.
- **Team**: Nombre, categoría ,y máximo número de miembros(dado por la categoría) de un grupo.
- **Requisitos**: Clase a la que se aplica, tipo de requisito,y según esté, campos sobre el mismo.(nivel mínimo, anotaciones, etc).
- **Logros**: Id, nombre y descripción de los logros.
- **Objetos**: Super tipo. Guarda nombre y nivel.
- **Equipable**: Hijo de objeto. Objeto equipable por el personaje. Guarda estadísticas del mismo, y tipo de equipo.

4.1.2. Modelo entidad-relación:

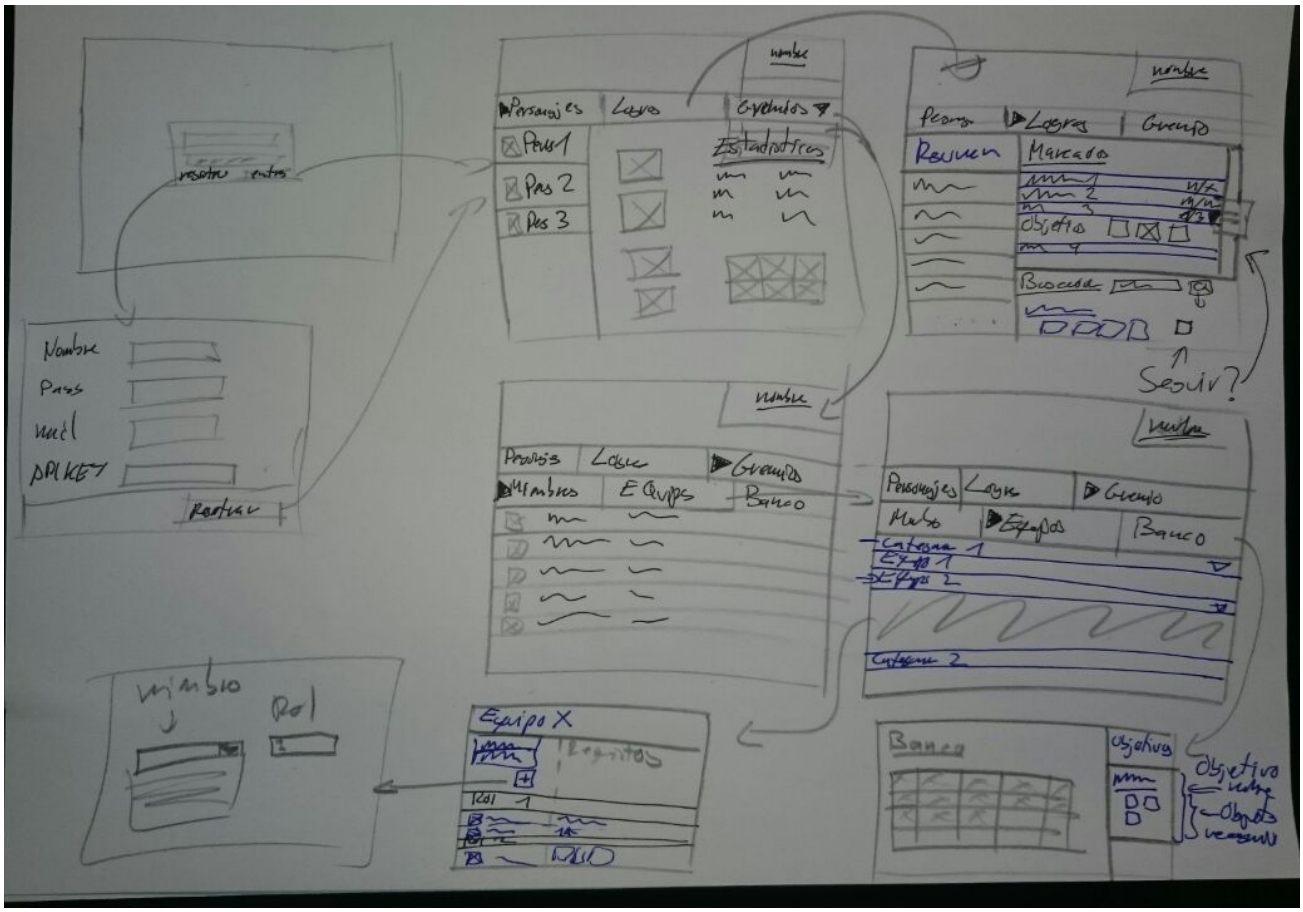


5. Interfaces de Usuario

5.1. Principios generales de la interfaz

La web se estructurará como un menú horizontal con un ítem por cada una de las secciones(personajes, logros y gremios), que permitirán la navegación a submenús de cada herramienta dentro de las secciones.

5.2. Prototipo Manual(escaneado)

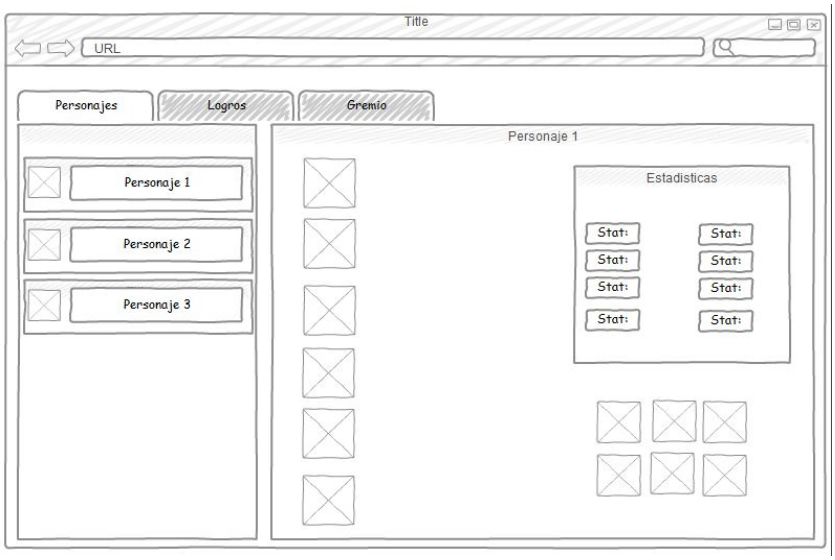


5.3. Prototipo interfaz

5.3.1. Descripción de las secciones

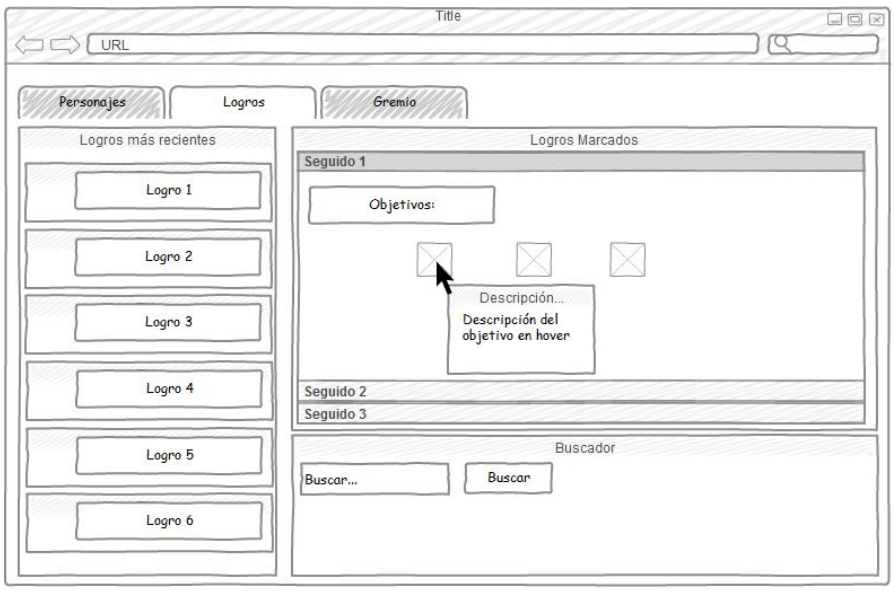
5.3.1.1. Panel de personajes:

Formado por una lista lateral con todos los personajes de la cuenta que permite seleccionar uno para mostrar su resumen en el espacio restante a la derecha. El resumen muestra el equipo del personaje seleccionado y las estadísticas totales que este le proporciona.



5.3.1.2. Panel de logros:

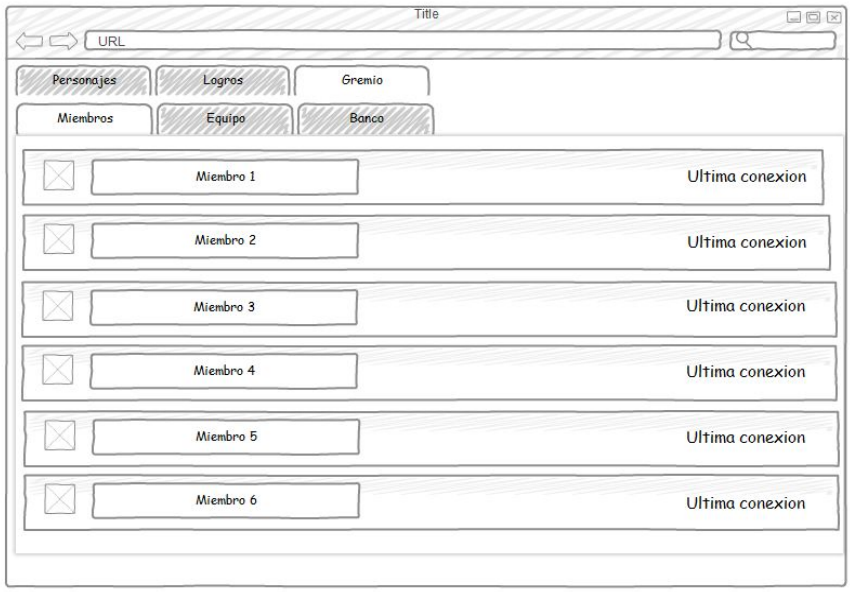
Proporciona un resumen en forma de lista en el lateral izquierda con los nombres de los logros más recientes. Un panel que muestra los logros marcados en forma de lista. Los ítems de la lista son desplegables, de forma que se puedan mostrar los objetivos pendientes. En la parte inferior se encuentra un buscador de logros que permitirá localizar los logros que sea pueda desear marcar como seguidos.



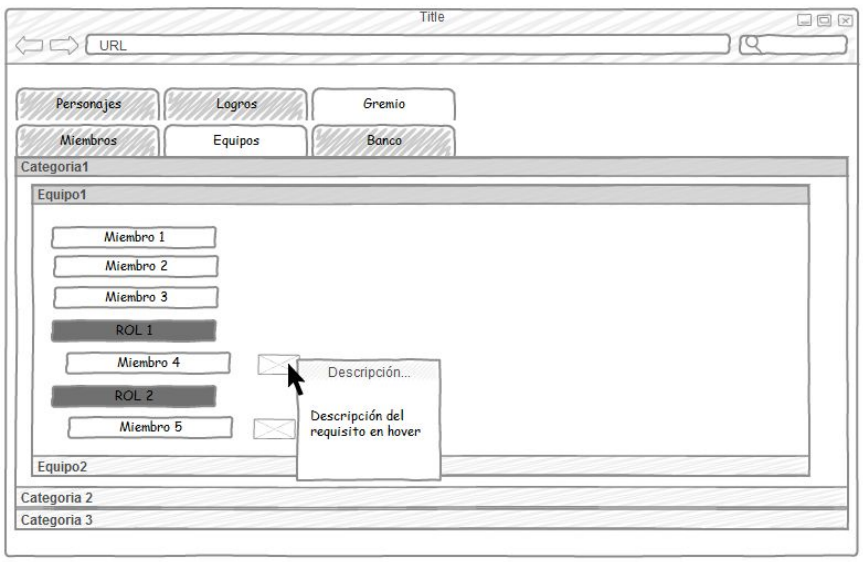
5.3.1.3. Panel de gremio:

En este caso el ítem del menú despliega una lista con los gremios a los que pertenece la cuenta para acceder al submenú formado por los ítems miembros, equipos y banco, y una vez seccionado se accede a la primera pestaña(miembros).

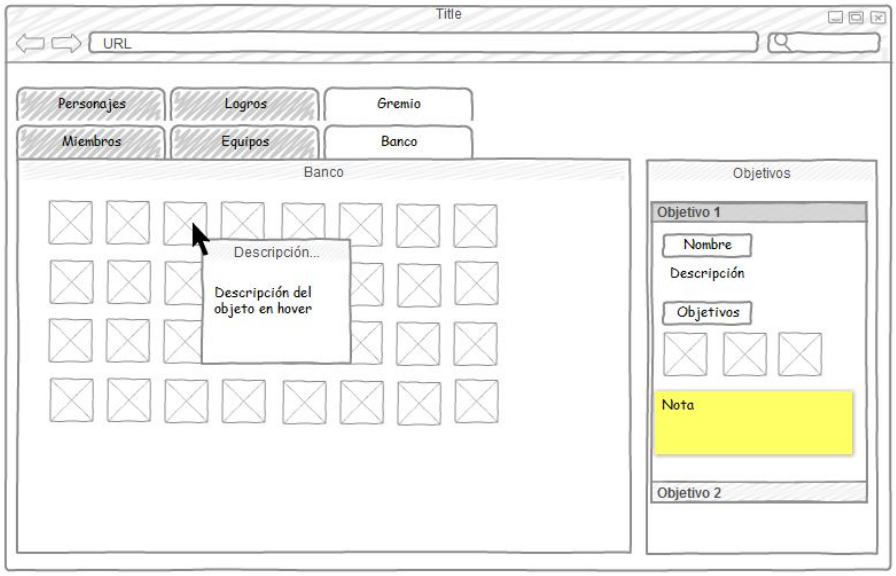
Miembros: Resumen en forma de lista de los usuarios miembros de ese gremio. Muestra nombre, icono y última conexión.



Equipos: Lista desplegable de categorías que contienen otra lista desplegable por cada equipo. Cada ítem del equipo es un miembro del gremio y, opcionalmente, los requisitos que le falta por cumplir, además de una primera sección que permitirá a los miembros autorizados a añadir miembros y requisitos.



Banco: Resumen en forma de cuadrícula con los objetos en el banco del gremio, y una sección derecha con los objetivos del

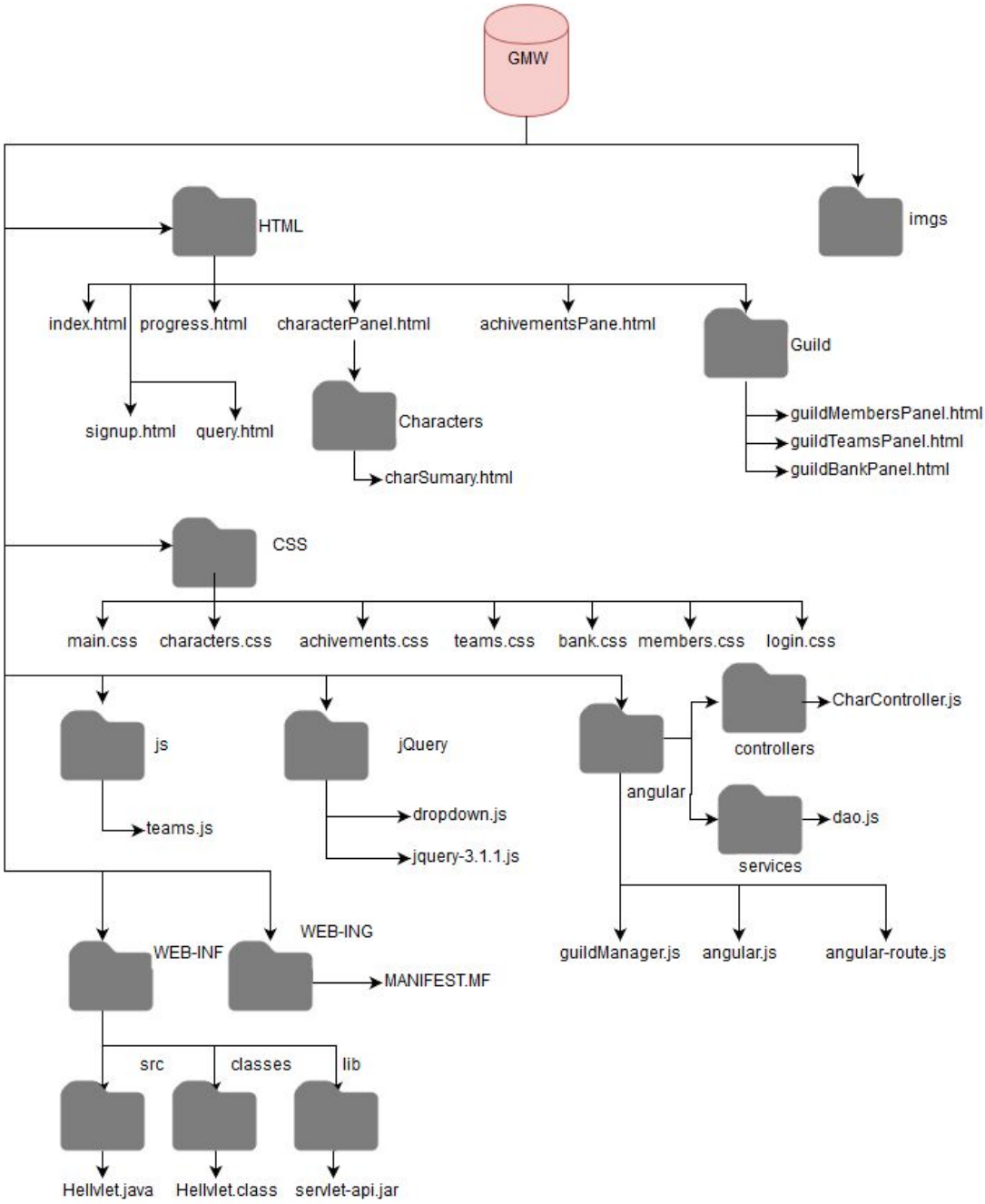


gremio y los objetos que necesitara para complirlos.

6. Modelo de despliegue

6.1. Estructura de ficheros

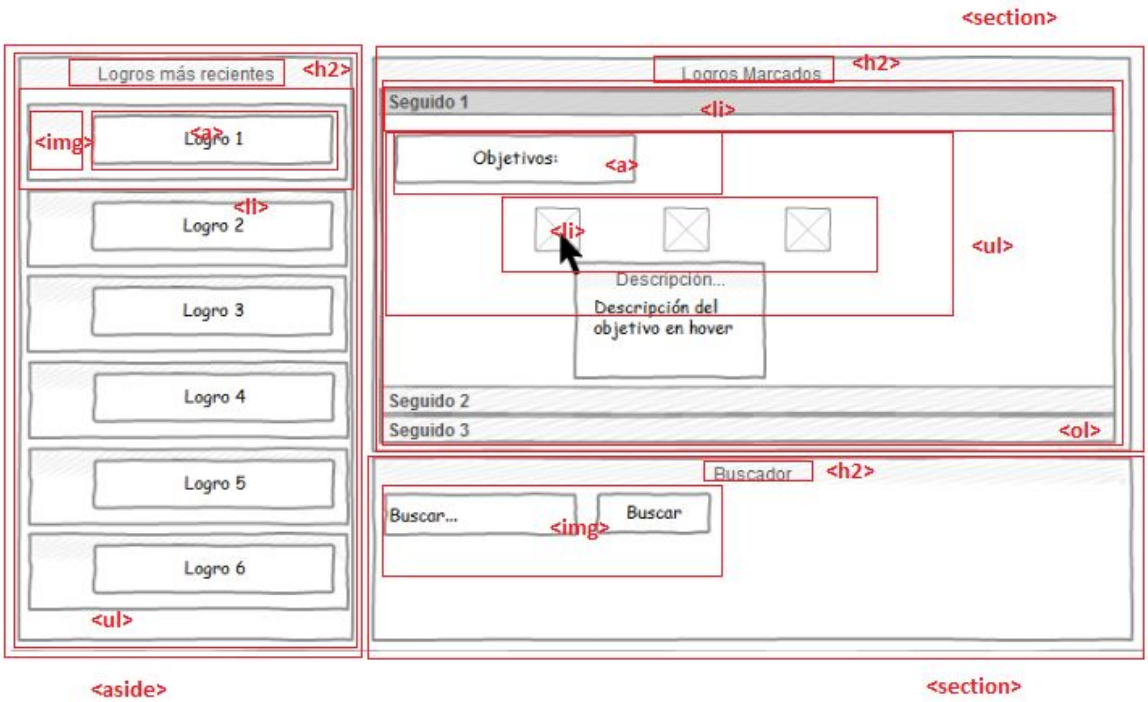
A la hora de estructurar los ficheros se separará los ficheros fuente html, las hojas de estilo css y los recursos. Los recursos en este caso son todos imágenes y se decidió separarlos ya que en un futuro podrán ser empleadas por otras partes de la aplicación.



7. Desarrollo del Sistema

7.1. HTML

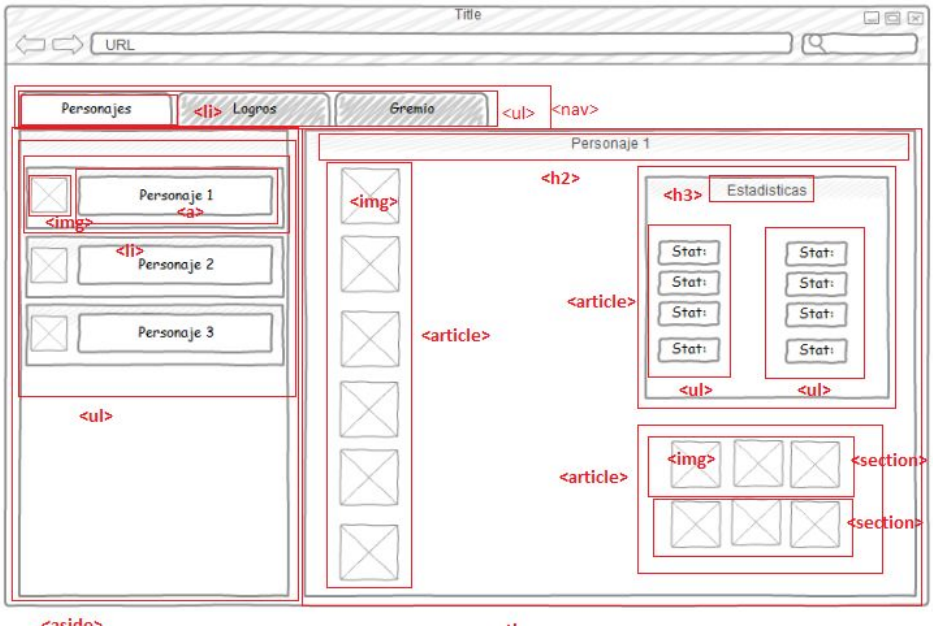
- Archivos principales:
  - *index.html*: Está formado por una imagen que representa un formulario de log y dos botones(en este caso enlaces <a>) para



registrarse y para entrar, que más adelante serán implementados.
  - *progress.html*: Una página a la que será redirigida toda acción aún no definida(normalmente debido a que implica alguna otra tecnología). Formada únicamente por una imagen.
  - *characterPanel.html*: Funcionara como página inicial una vez iniciada sesión, y está formada por un menú de navegación entre las secciones, y una lista de personajes.Esta lista de personajes redirecciona a una plantilla llamada *charSummary.html*, que se completara dinámicamente con los datos de los personajes.

Todas las páginas comparten la misma estructura como marco. Formada por un navegador superior horizontal y una serie de secciones determinadas para cada funcionalidad. El caso de *char1.html* es un buen ejemplo de ello.

Un <nav> para la barra de navegación. Un <aside> y un <section> dividido en en tres <article>.



El resto de páginas siguen la misma estructura básica, pero con diferentes secciones.

- *achivementsPanel.html*: Está formada por tres secciones, el resumen de los logros más recientes, formado por una lista con una imagen y un enlace a la wiki de cada logro, la sección principal de los logros seguidos, formado por otra lista que será implementada en forma de acordeón o lista desplegable con una sublista de los objetivos de cada logro, y un buscador de logros que permitirá la búsqueda de logros para seguirlos, por el momento representado por una imagen.

- *Carpeta Guild*: Dado que la sección de gremios está muy segmentada opte por englobar todas sus partes en una carpeta independiente formada por:
  - *guildMemberPanel.html*: Página que funcionara de inicio del panel de gremios. Es una lista simple formada por los miembros del gremio.(nombre, puntos de logros, y última conexión).
  - *guildTeamsPanel.html*: Formado por una serie de listas de categorías. Cada lista está formada por una lista de equipos, y cada equipo por una lista de miembros(organizados o no en listas de roles). Cada lista a excepción de la de categorías tiene siempre una entrada extra para administradores para

añadir items. Se busca que la versión final de esta sección se comporte como una lista extensible.

- *guildBankPanel.html*: Formado por dos secciones. Una formada por una tabla que representa cada casilla del banco del gremio, que se rellenara de manera dinámica en el futuro. Y otra por una listas de objetivos que el gremio persigue y los objetos necesarios que requiere dicho objetivo.

7.2. CSS

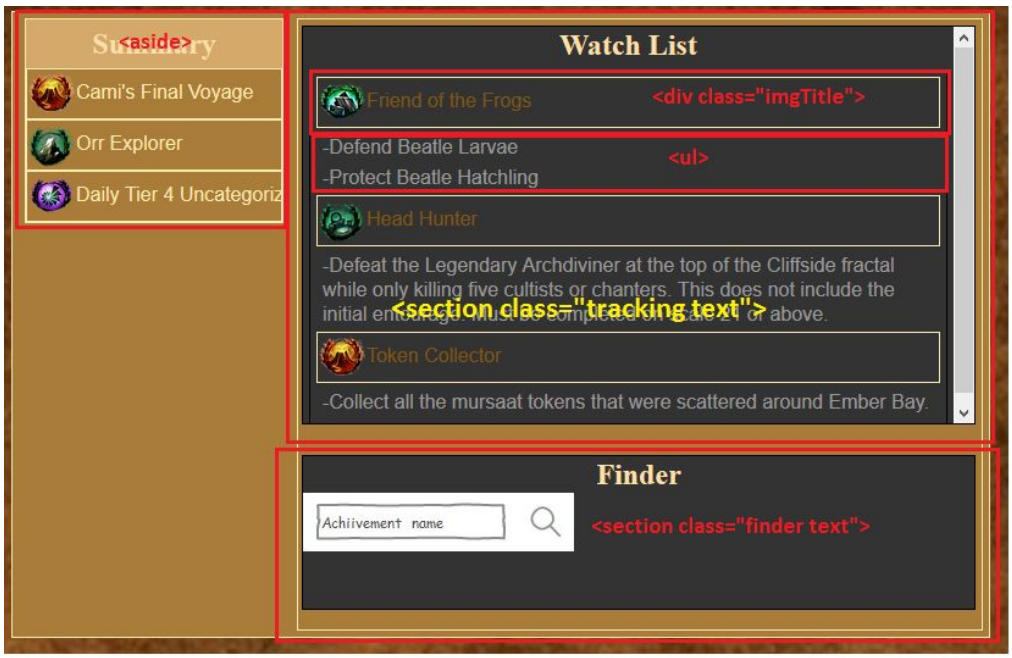
Se define una hoja de estilo principal que se aplicará a todas las páginas.

- *main.css*: Define reglas generales que se aplican a la totalidad del sitio. fuentes, colores, centrado del cuerpo, colocación de las partes comunes(nav, aside…), mecanismo de dropdown del menú de guilds…

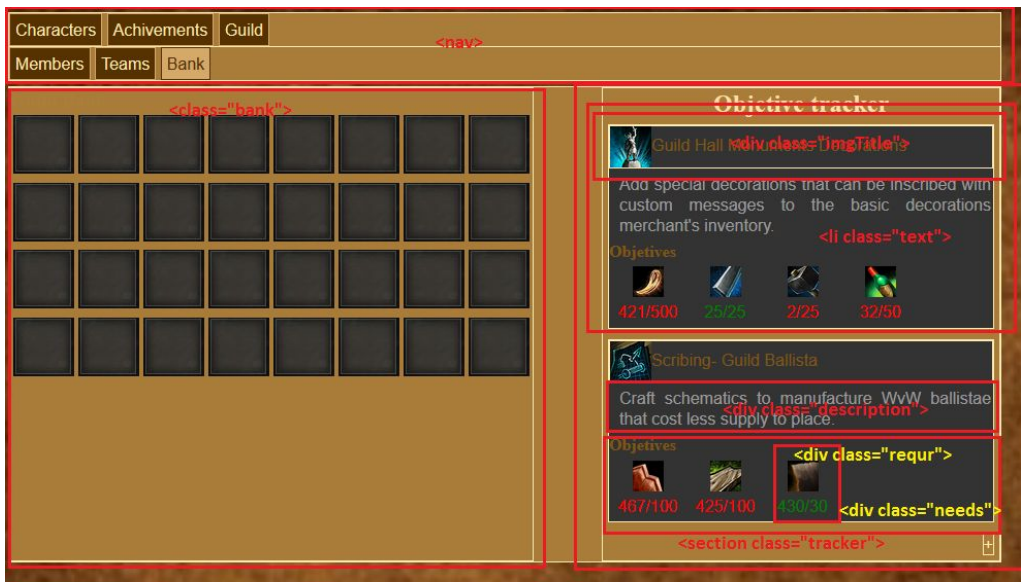
Se definen hojas individuales para cada página con el objetivo de alcanzar en diseño de colocación y estilo presentados con anterioridad. A continuación se listan y se muestra la aplicación de aquellas más relevantes:

- *characters.css*: Se aplica sobre la página de selección de personaje(*charPanel.html*), y cada una de las páginas de resumen de personaje(*char1.html* y *char2.html*). Principalmente define la colocación de los distintos elementos del resumen mediante flotación y márgenes. El resultado es el siguiente:

- *achivements.css*: Se aplica sobre la página de logros(*achivements.html*). Define la colocación de las dos partes que forman dicha página, el *tracker* y el *finder*. Además de matizar ciertas reglas para las fuentes y colores de las descripciones de los logros.El resultado es el siguiente:



- *members.css*, *teams.css* y *bank.css*: Se aplican, respectivamente a los paneles de miembros, equipos y banco de los gremios, y de la misma forma que los anteriores, definen la posición de los distintos elementos que conforman el contenido de cada sección. A continuación se muestra el panel de banco:



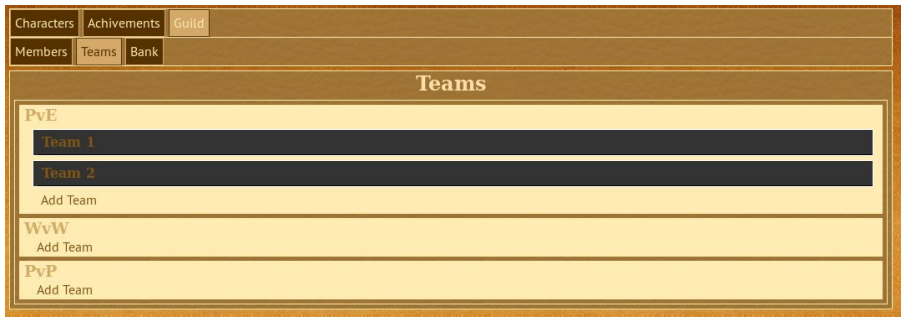
7.3. JavaScript: Manipulación DOM

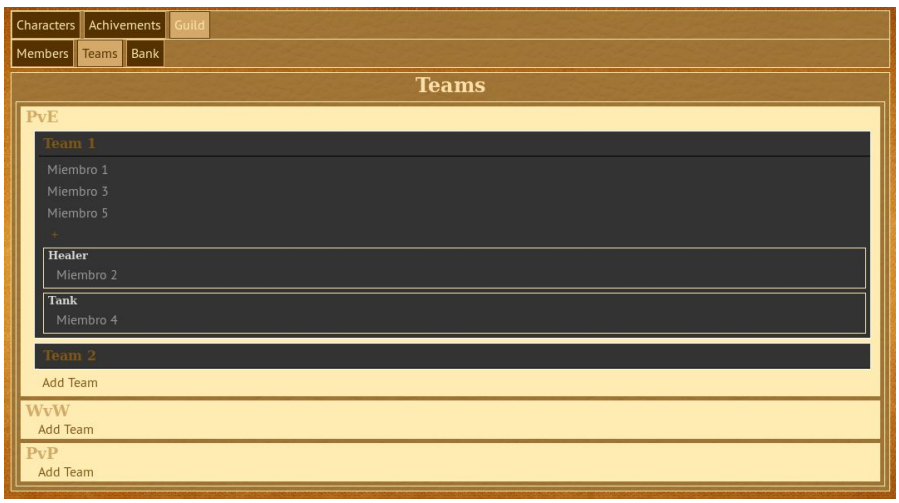
Las funciones de manipulación del árbol DOM se almacenan en la carpeta "*js*", separados en ficheros js para cada página que pueda necesitarlas

En el fichero *teams.js* se define la función *showTeamsOnclick(id)*, que recoge el id del un elemento a mostrar o ocultar para crear una lista de equipos desplegable. El id indica cuál de los equipos será el sujeto de la función. Esta función se definirá como el comportamiento ante el evento de click de los títulos de cada equipo.

```
function showTeamOnClick(id){
  var parent = document.getElementById(id)
  parent.getElementsByTagName('ul')[0].classList.toggle("show");
}
```

La función del evento busca el elemento en el documento con el id indicado, y al encontrarlo, le añade la clase '*show*', si no la tiene, o se la elimina si es así. Dicha clase sobre escribe el parámetro *display* de los elementos su interior a *block*, haciéndolos visibles. El resultado es el siguiente:





7.4. jQuery

Las funciones para la implementación de dinamismo mediante jQuery se almacenan en la carpeta "*jQuery*", separando cada efecto en un fichero diferenciado.

En el fichero *dropdown.js* se define el comportamiento ante hover de los elementos del menú de navegación con subentradas.

```
$(document).ready(function(){

  $('.dropContent').hide().removeClass('.dropContent');

  $('.dropBtn').hover(
    function() {
      console.log("Down");
      $('ul', this).stop().slideDown(150);
    },
    function () {
      $('ul', this).stop().slideUp(250);
    }
  );

});
```

Se definen dos clases *.dropContent*(contenido que se desplegará), y *.drop Btn*(entrada que desplegará el submenú). Esta función, tras determinar que el documento es cargado, busca los elementos con la clase *.dropBtn*, y le asigna el comportamiento antes el evento hover mediante dos funciones, hoverIn y hoverLeave. Las funciones detienen cualquier posible animación que estén realizando e inician un slide.

El resultado es el siguiente:

7.5. AngularJS

Los archivos que emplean Angular Js se almacenan en la carpeta angular. En esta carpeta se encuentra el script principal de la aplicación, *guilManager.js*, única aperta para los controladores, y otro para los servicios.

El objetivo de esta aplicación es la de realizar una petición http a los servidores de Guild Wars 2 para obtener los personajes vinculados a la cuenta del usuario y rellenar con ellos la lista lateral. Para ello defino un servicio de acceso a datos, *dao.js* y un controlador que emplea dicho servicio, *CharController.js*.

```
function dao($http, $q){

  var url = 'http://api.guildwars2.com/v2/characters';
  this.getCredentials = function() {
    return $http.get('json/credentials.json').then(function(response) {
      return $q.all(response.data);
  });

  }

  function getCharsByName(name,KEY) {
    return $http.get(url+'/'+name+'?access_token='+KEY).then(function(response){
      return $q.all(response.data);
    });
  };

  this.getChars = function(KEY){
    console.log(KEY);
    return $http.get(url+'?access_token='+KEY).then(function(response){
      return $q.all(response.data.map(function(x){
          return getCharsByName(x, KEY);
        }
      ));
    });
  };

};

dao.$inject = ['$http', '$q', '$rootScope'];
```

dao.js: Define tres funciones:

Emplean el servicio $q para, utilizando a su vez el servicio $http, realizar llamadas http mediante promesas, es decir, esperando a que termine la anterior, antes de realizar la siguiente. *getCharByName(name)* solicita información de un personaje concreto y recibe un json con la información, y *getchars()*, solicita todos los personajes de la cuenta y llama a la función anterior con cada uno de ellos, para ir introduciendo esa información en un json global que devolverá como salida. También se define una función para leer las credenciales desde un fichero local json, evitando el tener algún dato de acceso escrito en el código.

Define una función más para la consulta de ítems por id, que será empleada posteriormente:

```
this.getItemById = function(id){
  return $http.get(items+'/'+id).then(function(response){
      return $q.all(response.data);
  });
};
```

CharController.js: Empleando *dao.js*, define varias variables en $scope para su intercambio con la vista, una función para seleccionar el personaje según su posición en el menú y otra función de acceso a credenciales para obtener la APIKEY antes de consultar los personajes.

```
function CharController($scope, $http, dao) {

  var url = 'http://api.guildwars2.com/v2/characters';
  $scope.chars = [];
  $scope.creds = [];
  dao.getAPIKEY().then( function(res){
      $scope.creds = res;
      dao.getChars($scope.creds.APIKEY).then( function(chrs){
          $scope.chars = chrs;
      });
  });
```

La función de selección de personaje además, para cada ítem del personaje consulta sus estadisticas y las almacena en un mapa según estadística que posteriormente el panel *charsummary.html*, podrá leer del scope para completar los datos:

```
$scope.selectChar = function selectChar(index){
  $scope.selectedChar = index;
  $scope.stats = new Map();

  //Armor
  for( i = 2; i<=7; i++){
    var itemId = $scope.chars[$scope.selectedChar].equipment[i].id;

    dao.getItemById(itemId).then( function( item ) {
      var attributes = item.details.infix_upgrade.attributes;
      for(i in attributes){
        var value = attributes[i].modifier;
        var key = attributes[i].attribute;
        if( $scope.stats.has(key) ){
          value = $scope.stats.get(key) + value;
        }
        $scope.stats.set(key,  value);
      }
    });
  }
```

```
><span class="key">Power:</span><span class="valor"> {{ 1000 + stats.get('Power') }}</span></div>

ng"/><span class="key">Precision:</span><span class="valor">{{ 0 + stats.get('Precision') }}</span></div>

g"/><span class="key">Ferocity:</span><span class="valor">{{ 0 + stats.get('CritDamage') }}</span></div>

ng"/><span class="key">Toughness:</span><span class="valor">{{ 0 + stats.get('Toughness') }}</span></div>

g"/><span class="key">Vitality:</span><span class="valor">{{ 1000 + stats.get('Vitality') }}</span></div>
```

guidManager.js: Determina el uso del servicio *dao.js* y el controllador *Charcontroller.js*. Además indica el enrutado para el click sobre los personajes de la lista.

```
var app = angular.module("characters", ["ngRoute"]);

app.service('dao', dao);
app.controller('CharController', CharController);
app.config(function($routeProvider) {
    $routeProvider.when("/char1", {
        templateUrl : "characters/charSummary.html",
        controller : "CharController"
    });
});
```

Vista: Estos tres archivos emplean el archivo *characterPanel.html* como vista, para ello se define el *CharController*, como ngcontroller de la lista lateral y se añadirá un <li> por cada personaje obtenido mediante la directiva ng-repeat.

En este bucle se emplea la variable chars del scope para conocer el nombre de cada personaje y crear un id para cada enlace, que se emplea en el click para enrutar al panel de resumen del personaje adecuado. El enrutado colocara el contenido en el div correspondiente mediante la clase *ng-view*, y dado que está dentro del mismo control, podrá acceder a la información del personaje seleccionado, y cambiar la información en base a ello. También se emplean los filtros de angular y la directiva ng-src para definir la ruta de la imagen a emplear para cada entrada según la profesión del personaje.

```
<section class="main" ng-controller="CharController">
  <aside><!-- List of characters  (D)-->
    <h2 >Characters</h2>
    <ul >
      <li class="menuitem" ng-repeat="char in chars" ng-click="selectChar($index)">
        <img width="40px"  ng-src="imgs/profesions/{{ char.profession | lowercase }}.png" >
        <a id="{{ char.name }}"  href="#char1">{{ char.name }}</a>
      </li>
    </ul>
  </aside>

  <div class="content ng-view"></div>
</section>
```

El resultado es el siguiente:

7.6. Formularios:

Se definen dos formularios para GMW, un registro de usuario y una encuesta sobre su experiencia en el juego y en la página.

7.6.1. **Registro**: Se añade mediante el archivo *signup.html*, y permitirá la creación de una cuenta de usuario, mediante un mail, un nombre, una contraseña y una API-KEY optativa.
Los campos obligatorios van marcados con el atributo required en el

html, por lo que su validación es innecesaria. No obstante, los campos de contraseña y validación de la misma emplean una validación por funciones javascript. La contraseña pasa tres filtros de distinto rigor, e indica el usuario, mediante un código de colores la robustez de su contraseña.

Dichos filtros se aplican mediante expresiones retgales y son los siguientes:

- **Mínimo**: Longitud superior a 6 caracteres.
- **Medio**: Cualquier combinación de elementos de dos de los siguientes grupos: mayusculas, minúsculas, numeros o simbolos.
- **Alto**: combinación de elementos de tres de los grupos anteriores.

```
<script>
    function psswdChange(paswdInput){
      var max = new RegExp("^(?=.{8,})(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).*$", "g");
      var med = new RegExp("^(?=.{7,})(((?=.*[A-Z])(?=.*[a-z]))|((?=.*[A-Z])(?=.*[0-9]))|((?=.*[a-z])(?=.*[0-9]))).*$", "g");
      var min = new RegExp("(?=.{6,}).*", "g");

      var pssswordLevel = document.getElementById('pssswordLevel');
      if (paswdInput.value != ""){
        pssswordLevel.style.display = 'flex';
      }else {
        pssswordLevel.style.display = 'none';
      }

      var pl = document.getElementById('pl');
      var pl2 = document.getElementById('pl2');
      var pl3 = document.getElementById('pl3');

      if (min.test(paswdInput.value)){
        pl.style.display = 'flex';
      }else{
        pl.style.display = 'none';
      }
      if(med.test(paswdInput.value)){
        pl2.style.display = 'flex';
      }else{
        pl2.style.display = 'none';
      }
      if(max.test(paswdInput.value)){
        pl3.style.display = 'flex';
      }else{
        pl3.style.display = 'none';
      }
    }
```

Comprueba si alguno de los filtros se cumple, y si es así, hace visible el elemento del color correspondiente.

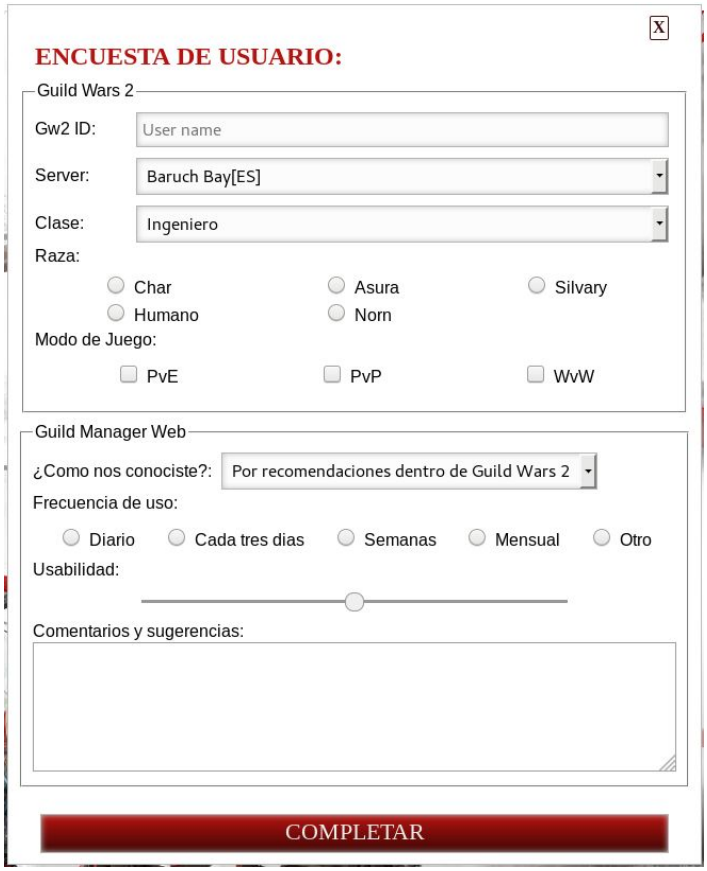
Además, antes del envío del formulario, se comprueba si las contraseñas son iguales con otra función indicada en el atributo *onsubmit*:

```
function checkEqualPass(){
    var psswd = document.forms[0]["password"];
    var rPsswd = document.forms[0]["rPassword"];
    rPsswd.style.border = "1px solid #b5b5b5"
    if ( psswd.value !== rPsswd.value){
        rPsswd.style.border = "1px solid red"
        //alert(psswd.value+" "+rPsswd.value);
        return false;
    }
}
```

El resto de campos son validados directamente empleando html5, a excepción del campo de API-KEY, que emplea una expresión regular que exige el formato apropiado:

```
<form name="signUp" method="post" onsubmit="return checkEqualPass()" action="complete.html">
  <h2>Registrarse</h2>
  <p class="info"> Los campos marcados con un * son obligatorios</p>
  <input name="email" placeholder="Correo electronico*" autocomplete="on" type="email" required="required"/>
  <input name="name" placeholder="Nombre de usuario*"  type="text" required="required"/>
  <div id="pssswordLevel"><div id="pl"></div><div id="pl2"></div><div id="pl3"></div></div>
  <input name="password" placeholder="Contraseña*" type="password" required="required" oninput="psswdChange(this)"/>
  <input name="rPassword" placeholder="Repite la contraseña*" type="password" required="required" oninput="checkEqualPass()"/>
  <input name="apikey" placeholder="Api Key" type="text" pattern="[A-Z0-9]{8}(-[A-Z0-9]{4}){3}-[A-Z0-9]{20}(-[A-Z0-9]{4}){3}-[A-Z0-9]{12}"/>
  <label><input name="subs" type="checkbox"/>Deseo subscrbrime al boletin de noticias</label>
  <input id="signup" name="submit" type="submit" value="REGISTRARSE"/>
</form>
```

7.6.2. **Encuesta**: Se añade mediante el archivo *query.html*, y permitirá consultar información sobre el tipo de usuario que emplea el portal. Recoge información tanto de sus preferencias dentro del juego, como de su uso de la propia aplicación.



Solicita datos de carácter informativo, y en ningún momento se exige el completado de esta encuesta, por lo que todos los campos se consideran optativos.

7.7. Servlet:

7.7.1. Manejo de la encuesta

Con el objetivo de manejar los datos que el formulario rellenen en la encuesta presentada anteriormente se emplea servlet corriendo sobre un servidor tomcat. Dicho servlet está compuesto por:

- Todos los archivos presentados anteriormente.
- El fichero "*Hellvlet.java"* que define el comportamiento a seguir por el servidor al recibir el envio de la encuesta.

En el código se añade las lineas de cabeza necesarias para cuplir el formato html, meta información y la llamada al css. Se definen las variables que almacenarán los distintos campos del formulario y se inicializan con el valor introducido.

```
out.println("<!DOCTYPE html>");
out.println("<html>");
out.println("<head>");
out.println("<meta charset=\"UTF-8\">");
out.println("<meta name=\"author\" content=\"Martin Rubio\">");
out.println("<link type=\"text/css\" rel=\"stylesheet\" href=\"css/login.css\"/>");

out.println("<title>Confirmacion de envio</title>");
out.println("</head>");
out.println("<body>");
out.println("<div class=\"results\">");

String ID = request.getParameter("ID");
String server = request.getParameter("server");
String profession = request.getParameter("profession");
String race = request.getParameter("race");
String[] gamemodes = request.getParameterValues("gamemode");
String known = request.getParameter("known");
String periody = request.getParameter("periody");
String usability = request.getParameter("usability");
String comments = request.getParameter("comments");
```

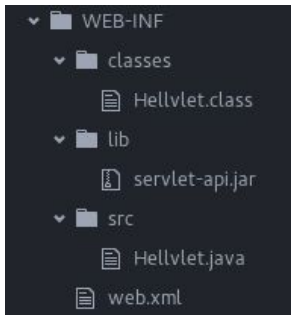
Muestra por pantalla los datos introducidos y un enlace de vuelta a la página principal, comprobando si se envían o no los campos optativos para evitar mostrar "null"s.

- La librería “*servlet-api.jar*” necesaria para la compilación y ejecución del servlet java.
- El archivo *web.xml*, que define el el servlet y la ruta al mismo siguiendo un formato concreto xml, para poder referenciar al servlet desde el html.

```
<web-app>

  <servlet>
    <servlet-name>Hellvlet</servlet-name>
    <servlet-class>Hellvlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Hellvlet</servlet-name>
    <url-pattern>/Hellvlet</url-pattern>
  </servlet-mapping>

</web-app>
```

Los archivo añadidos para la implementación del servlet se agrupan todos el la carpeta “WEB-INF”. Además, para el empaquetado, se añade la carpeta “META-INF”, que guarda información sobre el paquete y el formato de empaquetado como el manifesto.

La estructura de la carpeta WEB-INF añadida al árbol de ficheros el la siguiente:

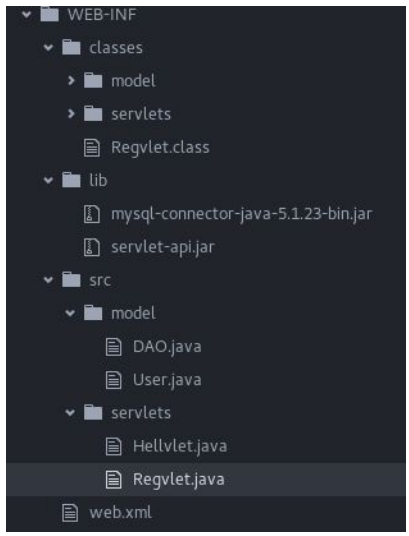


–

7.7.2. Manejo del registro

Para la implementación del registro de usuarios del sistema, se instala en el servidor una base de datos mysql con una tabla usuarios con el formato indicado.

El empaquetado del código cambia para adaptarse a esta funcionalidad siguiendo la siguiente estructura:



Además se altera el fichero *web.xml* tanto para actualizar la localización de las clases, como para añadir un atributo de contexto web master que se mostrará en el pie.

```
<web-app>

  <servlet>
    <servlet-name>Hellvlet</servlet-name>
    <servlet-class>servlets.Hellvlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Hellvlet</servlet-name>
    <url-pattern>/Hellvlet</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>Regvlet</servlet-name>
    <servlet-class>servlets.Regvlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Regvlet</servlet-name>
    <url-pattern>/Regvlet</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>webMaster</param-name>
    <param-value>Martin Rubio Fernández</param-value>
  </context-param>

</web-app>
```

Para el manejo de peticiones de registro se define un nuevo servlet llamado *Regvlet.java*.

Para el manejo de los datos se crean dos clases definidas en el paquete *model*,

User.java: Clase para el manejo de usuarios en java con un tatibuto por cada campo de la tabla usuarios de la base de datos.

```java
public class User{
  private String DNI;
  private String nombre;
  private String password;
  private String apellidos;
  private String email;


  public User(String DNI, String nombre, String password, String apellidos, String email) {
    this.DNI = DNI;
    this.nombre = nombre;
    this.password = password;
    this.apellidos = apellidos;
    this.email = email;
  }

  @Override
  public String toString(){
    return DNI+": "+nombre+", "+apellidos+", "+password+", "+apellidos+", "+email+".";
  }
```

DAO.java: Clase de manejo de la base de datos con un métodos de consulta de los usuarios, generica y por DNI, comprubacion de existencia de usuario, y insercion de usuario.

```java
public DAO(String user, String password) throws ClassNotFoundException, SQLException{
  Class.forName(JDBC_DRIVER);
  connection = DriverManager.getConnection(DB_URL , user, password);
}

public ArrayList<User> getUsers() throws SQLException{
  Statement statement = null;
  ResultSet resultSet = null;

  ArrayList<User> users = new ArrayList<User>();
  String[] Usuarios;
  statement = connection.createStatement();
  resultSet = statement.executeQuery("select dni, password, nombre, apellidos, email from usuarios");

  while (resultSet.next()) {
    String dni = resultSet.getString("dni");
    String nombre = resultSet.getString("nombre");
    String password = resultSet.getString("password");
    String apellidos = resultSet.getString("apellidos");
    String email = resultSet.getString("email");
    User user = new User(dni, nombre, password, apellidos, email);
    users.add(user);
  }
  return users;
}
```

```java
public User getUserByDNI(String DNI) {
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  User user = null;
  String prep = "select * from usuarios where DNI like ?";
  try{
    statement = connection.prepareStatement(prep);
    statement.setString(1, DNI);
    resultSet = statement.executeQuery();
    resultSet.first();
    user = new User(resultSet.getString("dni") , resultSet.getString("nombre"),
    resultSet.getString("password"), resultSet.getString("apellidos"), resultSet.getString("email"));
    System.out.println(user+" "+resultSet);

  }catch(SQLException ex){
    System.out.println(ex +"\n--No se ha podido establecer el statement: "+ statement);

  }
  return user;
}
```

```java
public boolean isRegistered(User user) throws SQLException{
  if( getUserByDNI(user.getDNI()) != null){
    return true;
  }
  return false;
}

public void insertUser(User user) throws SQLException{
    PreparedStatement insert = null;
    String insertString = "INSERT INTO usuarios VALUES (?, ?, ? ,?, ?)";
    insert = connection.prepareStatement(insertString);
    insert.setString(1, user.getDNI());
    insert.setString(2, user.getNombre());
    insert.setString(3, user.getPassword());
    insert.setString(4, user.getApellidos());
    insert.setString(5, user.getEmail());

    System.out.println(insert);
    insert.executeUpdate();
}
```

Los campos se validan mediante patterns HTML5 y javascript, por lo que al recibir la petición, el servlet se inicializa abriendo una conexión y comprueba si existe el usuarios.

```
String DNI = request.getParameter("DNI");
String name = request.getParameter("name");
String password = request.getParameter("password");
String apellidos = request.getParameter("apellidos");
String email = request.getParameter("email");

String msg ="";
User user = new User(DNI, name, password, apellidos, email);
```

Si existe emite un error:

```
if(dao.isRegistered(user)){
  msg = "Usuarios ya registrado.";
  out.println(format(msg));
```

Si no lo registra y muestra una notificación:

```
}else{
  System.out.println(user);
  dao.insertUser(user);
  user = dao.getUserByDNI(user.getDNI());
  msg = "Usuario con DNI: "+user.getDNI()+" registrado";
  out.println(format(msg));
  out.println(format(user.toString()));
}
```


Usuario con DNI: 35488513Y registrado.
Atras
Web Master: Martín Rubio Fernández