Documentación Técnica

WebPerfilDev - Aplicación de Gestión de Perfil Profesional

Yorman Alexis Cortes Echeverri

Diciembre 2025

Índice

1. Explicación del flujo MVC de la aplicación

La aplicación **WebPerfilDev** está construida usando el patrón de arquitectura **Modelo–Vista–Controlador (MVC)**. Este patrón ayuda a separar la lógica de negocio, la presentación y el control del flujo de la aplicación. Esto es especialmente útil, incluso para un desarrollador junior, porque obliga a organizar el código por responsabilidades.

1.1. Componentes principales

En este proyecto, los elementos del patrón MVC se pueden identificar así:

- **Modelo (Model):**
  - `Usuario`: representa la información principal del perfil (nombre completo, descripción personal, correo, teléfono, años de experiencia y URL de la foto de perfil).
  - `Competencia`: representa una capacidad técnica del usuario (id, nombre de la tecnología, nivel de dominio y color asociado).
  - `AdministradorDatos`: es una clase de apoyo que se encarga de leer y escribir los datos en archivos JSON (`usuarioBase.json` y `competenciasBase.json`). Desde el punto de vista del patrón, funciona como una pequeña "capa de persistencia".
- **Controlador (Controller):**
  - `ControladorPrincipal` (`@WebServlet(/")`): atiende las peticiones a la raíz de la aplicación. Para las peticiones GET carga los datos del usuario y sus competencias y los pasa a la vista. Para las peticiones POST procesa la actualización del perfil.
  - `ControladorCompetencias` (`@WebServlet(/controlador-competencias")`): atiende las peticiones relacionadas con las capacidades técnicas (agregar, editar y eliminar competencias).
  - `InicializadorAplicacion` (`@WebListener`): se ejecuta cuando el servidor inicia la aplicación. Inicializa las rutas de los archivos JSON y crea los archivos con datos por defecto si aún no existen.
- **Vista (View):**
  - `index.jsp`: es la página principal que ve el usuario. Muestra el perfil (nombre, descripción, foto, datos de contacto) y las tarjetas de capacidades técnicas.
  - `modales.jsp`: contiene los formularios en ventanas modales (Bootstrap) para editar el perfil, agregar capacidades y editar capacidades existentes.
  - `JavaScript inline`: en el propio `index.jsp` hay código JavaScript que se encarga de abrir los modales, capturar los formularios y enviar las peticiones *fetch* al backend.

Aunque el autor del proyecto tiene conocimientos básicos de programación, el patrón MVC aparece de forma clara: el modelo no sabe nada de la interfaz, la vista no sabe cómo se guardan los datos y los controladores hacen de puente entre ambos.

1.2. Flujo MVC: carga inicial (GET /)

Cuando un usuario abre la URL de la aplicación, por ejemplo:

`http://localhost:8085/WebPerfilDev/`

se sigue el siguiente flujo:

1. El navegador envía una petición HTTP `GET` al servidor Tomcat.
2. Tomcat detecta que la ruta `/` está asociada al servlet `ControladorPrincipal`.
3. El método `doGet` de `ControladorPrincipal` se ejecuta y:
   a) Llama a `AdministradorDatos.cargarDatosUsuario()`, que lee el archivo `usuarioBase.jso` y construye un objeto `Usuario`.
   b) Llama a `AdministradorDatos.cargarTodasLasCompetencias()`, que lee `competenciasBase` y construye una lista (o arreglo) de objetos `Competencia`.
   c) Coloca estos objetos como atributos en el `request` (por ejemplo, `usuarioEnSesion` y `competenciasDelUsuario`).
   d) Hace un `forward` hacia `/WEB-INF/views/index.jsp`.
4. El archivo `index.jsp` recibe el `request`, extrae el usuario y las competencias y genera el HTML que se enviará al navegador.
5. El navegador muestra la página con el perfil y las tarjetas de capacidades.

Aquí se ve claramente el patrón MVC:

- El **Modelo** (clases y JSON) contiene los datos.
- El **Controlador** (servlet) prepara esos datos.
- La **Vista** (JSP) se encarga de mostrarlos de forma amigable.

1.3. Flujo MVC: agregar capacidad (POST /controlador-competencias)

Cuando el usuario quiere agregar una nueva capacidad técnica, por ejemplo "React" o "Docker", el flujo es el siguiente:

1. El usuario hace clic en el botón *"Agregar Capacidad"*. El JavaScript abre un modal con un formulario.
2. El usuario completa los campos (nombre de la tecnología, nivel de dominio, color) y pulsa "Agregar".
3. El JavaScript captura el evento del formulario, evita que la página se recargue automáticamente y construye un objeto con los datos. Luego envía una petición HTTP `POST` a la ruta:

   `/WebPerfilDev/controlador-competencias`

   incluyendo en el cuerpo de la petición:

- `accion = ‘‘agregar’’`
- `nombreTecnologia`
- `nivelTecnologia`
- `colorTecnologia`

4. El servidor Tomcat redirige esta petición al servlet `ControladorCompetencias`, que ejecuta su método `doPost`.

5. Dentro de `doPost`, el controlador revisa el parámetro `accion`. Como vale `‘‘agregar’’`, llama a un método privado (por ejemplo `agregarCapacidad`):

   a) Lee todas las competencias existentes usando `AdministradorDatos.cargarTodasLasCompete`

   b) Calcula un nuevo identificador para la capacidad (por ejemplo, el máximo ID existente + 1).

   c) Crea un nuevo objeto `Competencia` con los datos del formulario.

   d) Añade esa competencia a la lista y llama a `AdministradorDatos.guardarCompetencias(lis` para guardar todo en `competenciasBase.json`.

6. El servlet responde con un texto sencillo `‘‘OK’’` si todo salió bien.

7. El JavaScript en el navegador, al recibir `‘‘OK’’`, muestra un mensaje al usuario (por ejemplo, un `alert`) y recarga la página para que aparezca la nueva tarjeta en la vista.

Aunque la lógica de programación es básica, se respeta el patrón MVC:

- La vista (JSP + JS) solo arma y envía los datos.
- El controlador decide qué hacer según la `accion`.
- El modelo y `AdministradorDatos` se ocupan de leer y escribir los archivos JSON.

1.4. Flujo MVC: editar perfil (POST /)

El flujo para editar el perfil del usuario es muy similar:

1. El usuario hace clic en el botón *“Editar Perfil”*. El JavaScript abre un modal rellenado con los datos actuales del usuario.

2. El usuario modifica algunos campos (por ejemplo el teléfono, la experiencia o la URL de la foto) y pulsa “Guardar”.

3. El JavaScript captura el formulario, construye un objeto con los campos y envía una petición HTTP `POST` a la ruta:

   `/WebPerfilDev/`

   con los parámetros:

   - `accion = ‘‘actualizarPerfil’’`

- `nombre`
- `descripcion`
- `email`
- `telefono`
- `experiencia`
- `foto`

4. El servlet `ControladorPrincipal` recibe la petición en su método `doPost`.
5. El controlador verifica que `accion` es ‘‘`actualizarPerfil`’’, crea un nuevo objeto `Usuario` con todos los datos recibidos y llama a `AdministradorDatos.guardarDatosUsuario(usu`
6. `AdministradorDatos` escribe el archivo `usuarioBase.json` con los nuevos valores.
7. El servlet responde con ‘‘`OK`’’ y el JavaScript recarga la página para mostrar el perfil actualizado.

1.5. Persistencia de datos con JSON

Finalmente, la **persistencia** se realiza mediante dos archivos JSON que viven en el directorio `data` generado dentro de `target/WebPerfilDev-1.0-SNAPSHOT/`:

- `usuarioBase.json`: guarda un único objeto con los datos del usuario (nombre, descripción, contacto, experiencia y URL de la foto).
- `competenciasBase.json`: guarda un arreglo (lista) de objetos, donde cada objeto representa una capacidad técnica.

La clase `AdministradorDatos` se encarga de todo lo que tiene que ver con estos archivos: crear archivos por defecto, leerlos al iniciar la aplicación y escribirlos cuando el usuario hace cambios desde la interfaz.

Aunque el desarrollador tiene una lógica de programación básica, este mecanismo le permite trabajar con datos persistentes sin necesidad de utilizar una base de datos compleja.

2. Instrucciones de ejecución

En esta sección se describen, paso a paso, las instrucciones necesarias para ejecutar la aplicación **WebPerfilDev**. El proyecto fue desarrollado por un estudiante con conocimientos básicos de programación, por lo que los pasos están explicados de manera sencilla.

2.1. Requisitos previos

Antes de ejecutar el proyecto, se deben cumplir los siguientes requisitos:

- **Java Development Kit (JDK) 21**: instalado y configurada la variable de entorno `JAVA_HOME`.
- **Apache NetBeans IDE 25**: entorno de desarrollo utilizado para abrir y ejecutar el proyecto.
- **Apache Tomcat 10.1.49**: servidor de aplicaciones donde se despliega la aplicación web.
- **Maven 3.x**: viene integrado en NetBeans y se usa para compilar y empacar el proyecto (tipo `war`).

2.2. Abrir el proyecto en NetBeans

1. Abrir **NetBeans**.
2. Ir al menú **File** → **Open Project**.
3. Navegar hasta la carpeta donde se encuentra el proyecto, por ejemplo:

```
C:\Users\...\Tecnicas\HTML\Web-App\WebPerfilDev
```

4. Seleccionar la carpeta `WebPerfilDev` y hacer clic en **Open Project**.

2.3. Configurar Apache Tomcat en NetBeans

1. En NetBeans, ir al menú **Tools** → **Servers**.
2. Hacer clic en **Add Server**.
3. Seleccionar **Apache Tomcat or TomEE** y hacer clic en **Next**.
4. En *Server Location*, indicar la ruta donde está instalado Tomcat, por ejemplo:

```
C:\Program Files\apache-tomcat-10.1.49
```

5. Configurar el usuario y la contraseña de administración (si es necesario).
6. Hacer clic en **Finish**.

2.4. Verificar el archivo pom.xml

El proyecto utiliza **Maven** para administrar dependencias y construir el archivo `.war`. Es importante que el archivo `pom.xml` contenga, al menos, las siguientes dependencias:

- **Jakarta Servlet API** (scope `provided`).
- **Gson** (para facilitar el manejo de JSON, si se usa).

Por ejemplo (fragmento ilustrativo):

```
<dependencies>
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.10.1</version>
    </dependency>
</dependencies>
```

En el caso de este proyecto, gran parte del manejo de JSON se realiza manualmente, pero la estructura con Maven ya está preparada.

2.5. Limpiar y construir el proyecto

1. En el panel de proyectos de NetBeans, hacer clic derecho sobre el proyecto `WebPerfilDev`.
2. Seleccionar la opción **Clean**. Esto borrará los archivos compilados previos.
3. Esperar a que en la ventana de **Output** aparezca el mensaje `BUILD SUCCESS`.
4. Luego, nuevamente clic derecho sobre el proyecto y seleccionar **Build**.
5. Esperar a que se complete la compilación y vuelva a aparecer `BUILD SUCCESS`.

Si hubiese errores de sintaxis o de configuración, NetBeans los mostrará en la ventana **Output** y en el panel de errores.

2.6. Ejecutar la aplicación en Tomcat

Una vez que el proyecto compila correctamente:

1. Hacer clic derecho sobre el proyecto `WebPerfilDev`.
2. Seleccionar la opción **Run**.
3. NetBeans se encargará de:

- Empaquetar el proyecto como `.war`.
- Desplegarlo en el servidor Apache Tomcat configurado.
- Iniciar el servidor si aún no estaba iniciado.
- Abrir el navegador con la URL de la aplicación.

4. La URL típica de acceso será algo similar a:

   `http://localhost:8085/WebPerfilDev/`

   (el puerto puede variar según la configuración de Tomcat).

2.7. Detener la aplicación

Para detener la ejecución de la aplicación:

1. En la ventana **Output** de NetBeans, localizar la salida del servidor Apache Tomcat.
2. Hacer clic en el botón rojo de **Stop** (cuadrado).
3. Alternativamente, se puede ir al panel de **Services**, ubicar el servidor Tomcat y detenerlo desde allí.

2.8. Estructura relevante del proyecto

A nivel de ejecución, es útil conocer algunos directorios importantes:

- `src/main/java/controllers`: contiene los servlets y el listener.
- `src/main/java/models`: contiene las clases del modelo y la clase `AdministradorDatos`.
- `src/main/webapp/WEB-INF/views`: contiene los JSP de la vista (`index.jsp` y `modales.jsp`).
- `target/WebPerfilDev-1.0-SNAPSHOT/data`: contiene los archivos JSON que guardan el perfil y las competencias.

Esta estructura permite que, aunque el programador tenga lógica y conocimientos básicos, la aplicación siga un orden razonable y sea mantenible.

3. Pruebas básicas del sistema

Dado que el proyecto fue desarrollado por un estudiante con experiencia limitada, es importante definir un conjunto de **pruebas básicas** que permitan verificar que la aplicación funciona correctamente desde el punto de vista del usuario final.

3.1. Prueba 1: Carga inicial de la aplicación

Objetivo: comprobar que la aplicación arranca correctamente y muestra el perfil y las capacidades iniciales.

Pasos:

1. Asegurarse de que el servidor Tomcat está en ejecución y que la aplicación ha sido desplegada.
2. Abrir un navegador web (por ejemplo, Edge o Chrome).
3. Acceder a la URL:

   `http://localhost:8085/WebPerfilDev/`

4. Observar que:
   - Se muestra el nombre completo del usuario (por ejemplo, *Yorman Alexis Cortes Echeverri*).
   - Se muestra una descripción personal, correo, teléfono y años de experiencia.
   - Se listan varias tarjetas de capacidades técnicas (por ejemplo, Java, JavaScript, HTML/CSS, etc.).

Resultado esperado:

- La página carga sin errores (no aparece pantalla en blanco ni mensajes de error del servidor).
- El usuario ve su perfil y las capacidades preconfiguradas.

3.2. Prueba 2: Edición del perfil de usuario

Objetivo: verificar que el usuario puede actualizar sus datos de perfil (nombre, descripción, contacto, etc.) y que los cambios se guardan.

Pasos:

1. En la parte superior de la página, hacer clic en el botón **"Editar Perfil"**.
2. Se abrirá un modal (ventana emergente) con un formulario que contiene los datos actuales del perfil.
3. Modificar algunos campos, por ejemplo:
   - Cambiar el número de teléfono.
   - Cambiar los años de experiencia.

- (Opcional) cambiar la URL de la foto de perfil.

4. Hacer clic en el botón **"Guardar Cambios"**.
5. Esperar a que aparezca un mensaje de confirmación (por ejemplo, un `alert` en el navegador).
6. Verificar que la página se recarga y que los nuevos datos se muestran en la sección de perfil.

Resultado esperado:

- El formulario no muestra errores de validación (todos los campos requeridos están llenos).
- Tras pulsar en "Guardar Cambios", los nuevos datos aparecen reflejados en la vista principal.
- Si se reinicia el servidor y se vuelve a entrar en la aplicación, los cambios se mantienen (gracias a que se guardan en `usuarioBase.json`).

3.3. Prueba 3: Agregar una nueva capacidad técnica

Objetivo: comprobar que el usuario puede añadir una nueva capacidad técnica desde la interfaz.

Pasos:

1. En la sección **"Mis Capacidades Técnicas"**, hacer clic en el botón **"Agregar Capacidad"**.
2. Se abrirá un modal con un formulario para introducir:
   - Nombre de la tecnología (por ejemplo, *React*).
   - Nivel de dominio (Básico, Intermedio, Avanzado o Experto).
   - Un color (seleccionado mediante un `input` de tipo color).
3. Completar los campos y hacer clic en **"Agregar"**.
4. Verificar que aparece un mensaje de éxito.
5. Comprobar que la página se recarga y ahora se ve una nueva tarjeta con la tecnología agregada.

Resultado esperado:

- Los campos del formulario se validan (por ejemplo, no se permite dejar el nombre vacío).
- La nueva capacidad técnica se añade a la lista visual.
- Si se revisa el archivo `competenciasBase.json`, debe aparecer un nuevo objeto con un identificador único.

3.4. Prueba 4: Edición de una capacidad existente

Objetivo: verificar que es posible modificar una capacidad técnica ya registrada.
Pasos:

1. En la lista de capacidades, buscar una tarjeta (por ejemplo, *Java*).
2. Hacer clic en el botón **"Editar"** de esa tarjeta.
3. En el modal que se abre, cambiar algún dato, como el nivel de dominio (por ejemplo, de *Intermedio* a *Avanzado*) o el color.
4. Hacer clic en **"Guardar Cambios"**.
5. Verificar que la tarjeta se actualiza en la página tras recargar.

Resultado esperado:

- El formulario se rellena con los valores actuales de la capacidad.
- Los cambios se reflejan visualmente en la tarjeta (nivel y color).
- En `competenciasBase.json`, el objeto correspondiente a esa capacidad muestra los nuevos datos.

3.5. Prueba 5: Eliminación de una capacidad

Objetivo: comprobar que el usuario puede eliminar una capacidad de la lista.
Pasos:

1. En la tarjeta de alguna capacidad (por ejemplo, *Bootstrap*), hacer clic en el botón **"Eliminar"**.
2. El navegador mostrará un cuadro de confirmación preguntando si se está seguro de eliminar.
3. Confirmar la eliminación.
4. Verificar que aparece un mensaje de éxito y que, tras recargar la página, la tarjeta ya no está.

Resultado esperado:

- La capacidad desaparece de la interfaz.
- El archivo `competenciasBase.json` ya no contiene el objeto con el identificador de la capacidad eliminada.

3.6. Prueba 6: Persistencia tras reinicio del servidor

Objetivo: validar que los datos se mantienen aunque el servidor Tomcat se detenga y se vuelva a iniciar.

Pasos:

1. Realizar varios cambios en la aplicación:
   - Editar el perfil del usuario.
   - Agregar una o dos capacidades nuevas.
   - Editar una capacidad existente.
2. Detener el servidor Tomcat desde NetBeans.
3. Esperar unos segundos y volver a ejecutar el proyecto (Run).
4. Acceder otra vez a la URL de la aplicación.
5. Verificar que todos los cambios realizados anteriormente siguen presentes.

Resultado esperado:

- Los datos se mantienen porque están guardados en los archivos JSON de la carpeta `data`.
- El comportamiento de la aplicación es consistente antes y después del reinicio.

Estas pruebas cubren el comportamiento básico y más importante de la aplicación, y son adecuadas para el nivel de un desarrollador junior que está reforzando sus conocimientos de programación web y del patrón MVC.