**ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.**

**GRADO EN INGENIERÍA INFORMATÍCA EN**

**TECNOLOGIAS DE LA INFORMACIÓN**

**ÁREA DE INGENIERÍA TELEMÁTICA**

**TRABAJO FIN DE GRADO Nº 11111111111**

***“****Aplicación web para la gestión de un repositorio****”***

|  |  |
| --- | --- |
| **Autor:** | **Tutor:** |
| **Raúl García Fernández** | **Raquel Blanco Aguirre** |
|  |  |

**Junio de 2017**

Diseño Del Sistema

Contenido

[Contenido 2](#_Toc487130080)

[Ilustraciones 3](#_Toc487130081)

[Tablas 4](#_Toc487130082)

[1. Arquitectura del sistema 5](#_Toc487130083)

[1.1 Introducción al sistema: 5](#_Toc487130084)

[1.2 Arquitectura Modelo-vista-controlador 6](#_Toc487130085)

[1.3 Arquitectura REST: 7](#_Toc487130086)

[2. Diseño de datos: 8](#_Toc487130087)

[2.1 Introducción 8](#_Toc487130088)

[2.2 Nodos de la base de datos 8](#_Toc487130089)

[2.2.1 MD-Nodos:UserLogin: 8](#_Toc487130091)

[2.2.2 MD-Entidades: Person 9](#_Toc487130092)

[2.2.3 MD-Nodos: Project 10](#_Toc487130093)

[2.2.4 MD-Nodos: Group 11](#_Toc487130094)

[2.2.5 MD-Nodos: Ejecuciones 12](#_Toc487130095)

[2.2.6 MD – Relaciones de la base de datos 13](#_Toc487130106)

[2.2.7 MD – Estructura general de la base de datos 15](#_Toc487130107)

[2.3 Diseño de sistema: Subsistema repositorio 15](#_Toc487130108)

[2.4 Capa REST 16](#_Toc487130109)

[2.5 Capa Negocio 26](#_Toc487130110)

[2.6 Capa de datos 27](#_Toc487130111)

[2.7 Capa ejecución 28](#_Toc487130112)

[2.7.1 Sincronización de proyectos: 28](#_Toc487130113)

[2.7.2 Ejecución de proyectos 28](#_Toc487130114)

[2.7.3 Respuestas de ejecuciones 29](#_Toc487130115)

[3. Diseño de sistema: Subsistema aplicación web 30](#_Toc487130116)

[3.1 Lado servidor 30](#_Toc487130117)

[3.1.1 Gestión de recursos web: 30](#_Toc487130118)

[3.1.2 Creación de cuentas: 30](#_Toc487130119)

[3.2 Lado cliente 31](#_Toc487130120)

[3.2.1 Angular.js: El MVC en el cliente 31](#_Toc487130121)

# Ilustraciones

[Ilustración 2‑1 (Esquema general aplicación) 4](file:///C:\Users\rulo-\Desktop\UniApi\documentación\individual\DiesñoDelSistema(CasiArreglado).docx#_Toc481603521)

[Ilustración 2‑2 (Arquitectura MVC) 5](file:///C:\Users\rulo-\Desktop\UniApi\documentación\individual\DiesñoDelSistema(CasiArreglado).docx#_Toc481603522)

[Ilustración 2‑3 (Arquitectura-REST) 6](#_Toc481603523)

[Ilustración 3‑1 (neo4j: Ejemplo básico estructura) 14](#_Toc481603524)

[Ilustración 3‑2 (Neo4j: Ejemplo entidad-relación) 15](#_Toc481603525)

[Ilustración 3‑3 (Estructura general de datos) 17](#_Toc481603526)

[Ilustración 3.4‑1 (Logo Spring) 19](#_Toc481603527)

[Ilustración 3.4‑2 Esquema Servicios Negocio 21](#_Toc481603528)

[Ilustración 3.4‑3 Esquema DAO Entidades 22](#_Toc481603529)

[Ilustración 3.4‑4 Esquema DAO Relaciones 23](#_Toc481603530)

[Ilustración 4‑1 Arquitectura Angular.js MVC 29](#_Toc481603531)

[Ilustración 4‑2 Arquitectura básica de app cliente Angular.js 30](#_Toc481603532)

# Tablas

[Tabla 3‑1 (UserLogin) 8](#_Toc481603547)

[Tabla 3‑2 (Person) 9](#_Toc481603548)

[Tabla 3‑3 (Project) 11](#_Toc481603549)

[Tabla 3‑4 (Group) 12](#_Toc481603550)

[Tabla 3‑5 (Neo4j: conjunto de entidades) 16](#_Toc481603551)

# Arquitectura del sistema

## Introducción al sistema:

El sistema estará dividido en dos grandes subsistemas, que a su vez están divididos en capas independientes con el objetivo de buscar el mínino acoplamiento para futuros cambios. Cada subsistema utilizara una arquitectura diferente para sus objetivos principales y conviene diferenciarlos:

* **Aplicación web**: Subsistema cuyo principal objetivo es la comunicación del usuario con el repositorio. Prima el entender y proporcionar servicios al usuario mediante una arquitectura **modelo-vista-controlador** (MVC).
* **Repositorio**: Subsistema cuyo principal objetivo es la realización de actividades o tareas y el almacenado de datos. Prima la realización de actividades de gestión frente a la facilidad de comunicación con el usuario por parte del sistema. Para poder realizar tareas con el repositorio es necesario tener conocimientos en su funcionamiento y en su forma de comunicarse. El repositorio está basado en una arquitectura REST.



Ilustración ‑ (Esquema general aplicación)

## Arquitectura Modelo-vista-controlador

El modelo vista controlador es una arquitectura muy extendida en las aplicaciones web. Esta arquitectura se basa en un modelo en capas que facilita la interacción con el usuario. El modelo MVC genera para cada interacción de un usuario, una vista personalizada para él. Sus componentes principales son:

* **Modelo**: Parte lógica de la arquitectura qué gestiona el acceso a los datos y los maneja para envolverlos en condiciones correctas para el usuario y las capas superiores.
* **Controlador**: Es la capa intermedia entre la vista y el modelo. Esta capa analiza las interacciones entre el usuario y el sistema mediante eventos e invoca peticiones al modelo para conseguir datos del sistema y posteriormente realizar una vista personalizada con esos datos.
* **Vista**: Representación gráfica para los usuarios y proporciona las herramientas o medios qué generan eventos que avisaran al controlador para que los gestione.

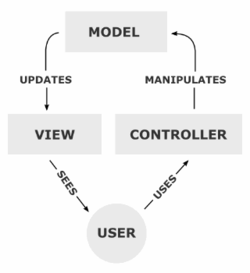


Ilustración ‑ (Arquitectura MVC)

## Arquitectura REST:

La Arquitectura REST es una tecnología HTTP, usada para el transporte de información. Esta arquitectura facilita la creación y gestión de una comunicación servidor-cliente genérica.

Esta arquitectura se basa en un sistema de Petición-Respuesta asíncrona sin estado, es decir: el cliente solicita algo y se le responde cuando sea posible, además, como es sin estado, cada petición es independiente y no se guardan datos de anteriores de las comunicaciones. Los elementos principales de la arquitectura REST son:

* **Recursos**: Un recurso es una entidad (Objeto o conjunto de atributos) definido por una identidad única (ID, Código o Hash). En REST los recursos son enviados, recibidos, manipulados, borrados... Cuando solicitamos cualquier servicio REST recibiremos como respuesta un Recurso, qué a su vez puede contener varios recursos a la vez.
* **Acciones**: Las acciones son una derivación de los recursos, es la abstracción en recursos de una acción. Por ejemplo, si se desea que el servidor realice la acción “correr”, se enviará un recurso “anda” con un módulo de dirección y una velocidad. El servidor REST analizará el recurso, entenderá la orden y realizará la acción devolviendo un recurso indicando el estado, por ejemplo, la posición actual.
* **Métodos prefijados**: Los servidores REST proveen de un conjunto de métodos prefijados para realizar sobre los recursos. Cuando se llama a un recurso siempre se tendrá una lista de acciones posibles (GET, PUT, POST, DELETE…)

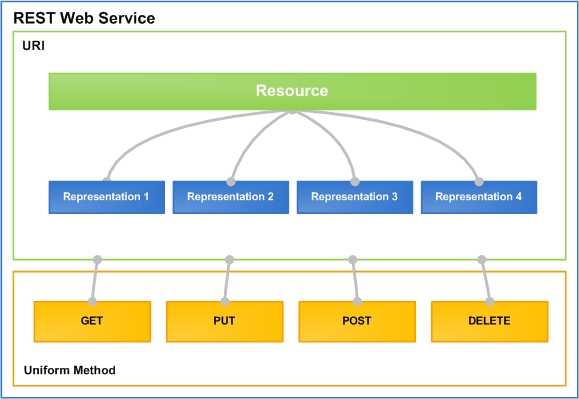


Ilustración ‑ (Arquitectura-REST)

# Diseño de datos:

## Introducción

El diseño de datos proporcionará la información e indicaciones para implementar el modelo de datos del sistema. Esto modelo será permanente y será el utilizado por la aplicación. Para el diseño del modelo físico de datos se utilizará una base de datos orientada a grafos, en concreto **Neo4j.**

Neo4j almacena la información en nodos y relaciones dirigidas entre ellos. Tanto nodos como relaciones pueden contener propiedades que los definen. La siguiente figura ilustra el modelo de datos manejado por Neo4j.



## Nodos de la base de datos

Los nodos de la base de datos representarán a las distintas entidades del modelo de dominio. Estos nodos, junto con sus propiedades, son descritos a continuación.

### MD-Nodos:UserLogin:

Su objetivo es Identificar al usuario del sistema y facilitar información para su autenticación

Tabla ‑ (UserLogin)

|  |  |  |
| --- | --- | --- |
| Entidad | UserLogin | |
| Variable | Tipo | Descripción |
| User | String (Email) | Define el email de la identidad del usuario. |
| Pass | String (SHA-1) | Define la contraseña del usuario cifrada con el algoritmo de firma SHA-1. |
| Rol | Enum (UserType:  {USER,ADMIN}) | Define el rol que tiene el usuario. |
| CreationTime | Date | Fecha de creación de la entidad. |
| Hashcode | String (SHA-1) | Usando los datos no modificables de la entidad (User y creationTime) se realiza una firma única que identifica a la entidad frente a otras. |

### MD-Entidades: Person

Su objetivo es proporcionar datos personales del usuario al sistema para su posible representación.

|  |  |  |
| --- | --- | --- |
| Entidad | Person | |
| Variable | Tipo | Descripción |
| Name | String | Nombre de la persona. |
| Subname | String | Apellidos de la persona. |
| Birthday | Date | Fecha de nacimiento de la persona. |
| Birthplace | String | Lugar donde vive la persona. |
| Province | String | Provincia donde vive la persona. |
| Country | String | País donde vive la persona. |
| Biografy | String | Pequeño texto para que la persona se pueda expresar. |
| profileImageUrl | String(URL) | Espacio para URL para qué la persona pueda poner una imagen representativa de cualquier repositorio de imágenes. |
| dateCreation | Date | Fecha de creación de la cuenta. |
| hashcode | String (SHA-1) | Usando los datos no modificables de la entidad (dateCreation) se realiza una firma única qué identifica a la entidad frente a otras. |

Tabla ‑ (Person)

### MD-Nodos: Project

Su objetivo es almacenar los datos necesarios para la ejecución del proyecto en el sistema.

|  |  |  |
| --- | --- | --- |
| Entidad | Project | |
| Variable | Tipo | Descripción |
| Name | String | Nombre del proyecto |
| Type | Enum (ProjectType:{  JAVA,  PYTHON,OCTAVE}) | Tipo del proyecto, definirá a que servicio del sistema debe acceder. |
| Description | String | Descripción explicativa de lo que hace el proyecto. |
| GitRepositoryURL | String (url) | Dirección URL del repositorio GIT de donde sacar el código del programa. |
| User | String (email) | Usuario del repositorio GIT, por general suelen ser email’s. |
| Password | String | Contraseña del repositorio GIT |
| MainName | String | Nombre del archivo main del código a ejecutar. |
| ResponseName | String | Nombre del archivo que se espera como respuesta tras la ejecución. |
| InputDescription | String | Descripción de los argumentos de entrada que puede aceptar el proyecto. |
| OutputDescription | String | Descripción de los datos de salida que aparecerán tras la ejecución del proyecto. |
| DefaultInputs | String[] | Conjunto de argumentos por defecto en el caso de que el usuario no pueda introducir entradas. |
| ModifyDate | Date | Fecha que determina la modificación de la entidad. |
| CreationDate | Date | Fecha que determina la creación de la entidad. |
| HashCode | String (SHA-1) | Usando los datos no modificables de la entidad (Name y creationDate) conseguimos utilizando el algoritmo SHA-1 una firma única para la entidad. |

Tabla ‑ (Project)

### MD-Nodos: Group

Su objetivo es almacenar los datos necesarios para crear un grupo qué almacene proyectos y grupos. Además, contendrá los permisos de las acciones.

|  |  |  |
| --- | --- | --- |
| Entidad | Group | |
| Variable | Tipo | Descripción |
| Name | String | Nombre del grupo |
| Type | Enum (GroupType:{  PUBLIC,  PRIVATE,MAIN}) | Tipo del grupo, definirá que propiedades de acceso tendrá el grupo. |
| Description | String | Descripción explicativa del grupo |
| CreationDate | Date | Fecha de creación del grupo. |
| SharingGroupPermissions | String[] | Conjunto de permisos “YES” o “NO” Que modifican los permisos |
| GroupCreationPermissions | String[] | Conjunto de permisos “YES” o “NO” Que modifican los permisos |
| MemberGestionPermissions | String[] | Conjunto de permisos “YES” o “NO” Que modifican los permisos |
| ProjectPropertiesPermissions | String[] | Conjunto de permisos “YES” o “NO” Que modifican los permisos |
| Hashcode | String(SHA-1) | Usando los datos no modificables de la entidad (Name , creationDate y type) conseguimos utilizando el algoritmo SHA-1 una firma única para la entidad. |

Tabla ‑ (Group)

### MD-Nodos: Ejecuciones

Su objetivo es almacenar los datos de una ejecución. Para saber su estado y su resultado.

|  |  |  |
| --- | --- | --- |
| Entidad | Execution | |
| Variable | Tipo | Descripción |
| creationDate | Date | Fecha de creación de la entidad ejecución. También servirá como fecha de inicio de la ejecución. |
| StateOfExecution | Enum (ExecutionType:{  STARTED,  FINISH\_WITH\_ERROR,  FINISH\_SUCCESS,  RUNNING}) | Estado de ejecución. Representará el estado actual en el que se encuentra la ejecución. |
| GroupOfExecution | String | Nombre del grupo en el que se está ejecutando el proyecto. Es mera información para el usuario. |
| InputJson | String[] | Conjunto de entradas utilizadas para la ejecución del proyecto. |
| Response | String(JSON) | Respuesta de la ejecución deberá estar en formato JSON Stringify. |
| Console | String | Respuesta interna de los servicios. |
| FinishDate | Date | Fecha que indica la finalización de la ejecución. |
| HashCode | String (SHA-1) | Conjunto de permisos “YES” o “NO” Que modifican los permisos |
| NameExecution | String | Nombre del Project que va se va a ejecutar. Este será el nombre de la ejecución. |

### MD – Relaciones de la base de datos

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nombre | Entidad A | Dirección | Entidad B | Descripción |
| MAKE\_REFERENCE  (HACE\_REFERENCIA) | UserLogin | -> | Person | La persona a la que referencia la identidad UserLogin es suya. |
| IS\_OWNER  (ES\_PROPIETARIO) | UserLogin | -> | Group | El usuario hace referencia al grupo que le pertenece. |
| KNOWS  (CONOCE) | UserLogin | -> | Group | El usuario que hace referencia conoce el grupo, es decir es visible. |
| IS\_CREATOR | UserLogin | -> | Project | El usuario que hace referencia es el creador del proyecto referenciado. |
| IS\_SUBGROUP  (ES\_SUBGRUPO) | Group | -> | Group | El grupo que referencia es subgrupo del grupo que es referenciado. |
| CONTAINS(CONTIENE) | Group | -> | Project | El grupo que referencia contiene (que no es propietario) del proyecto referenciado. |
| USE (USA) | Execution | -> | Project | La ejecución referencia al proyecto que está utilizando. |
| GENERATE (GENERA) | UserLogin | -> | Execution | El usuario que referencia genera a la ejecución que está generando. |

Tabla ‑ (Neo4j: conjunto de entidades)

### MD – Estructura general de la base de datos

En la siguiente figura se describe la estructura general de datos del sistema, donde sepuede ver un ejemplo de los nodos y las relaciones entre ellos.



Ilustración ‑ (Estructura general de datos)

# Diseño de sistema: Subsistema repositorio

El Subsistema repositorio es un sistema con dos objetivos principales:

* **Ejecutar proyectos**: Ejecutar proyectos de diferentes índoles, sincronizarlos y devolver la ejecución al usuario.
* **Gestionar el negocio del repositorio**: Gestionar todo el negocio referente a la ejecución de proyectos, los miembros, los grupos y los subgrupos que lo contienen.

El sistema repositorio estará dividido en varias capas. Las capas harán funciones independientes y todas estarán implementadas en Java. Todas las capas se comunicarán con la otras mediante interfaces que proporcionarán servicios compatibles para cada capa.

Algunas capas del sistema repositorio utilizaran tecnologías y frameworks que facilitaran el realizar los objetivos de las capas. Las diferentes capas y sus objetivos son las siguientes:

* Capa REST: Proporciona servicios de Comunicación entre el sistema y los diversos clientes.
* Capa Negocio: Proporciona servicios de Gestión de las diversas opciones del sistema.
* Capa de Datos: Proporciona acceso a la información de la base de datos.
* Capa de ejecución: Proporciona servicios de ejecución de los códigos de los proyectos.

### Capa REST

La capa REST proporcionara servicios de comunicación entre el sistema y los diversos clientes del repositorio utilizando la arquitectura REST para ello.

**El protocolo HTTP** es un protocolo utilizado por las tecnologías web. Este protocolo utiliza el texto como método de comunicación. Aunque también permite la comunicación de bits y los cifrados…

#### Arquitectura REST:



Ilustración ‑ (Logo Spring)

Para la realización del servidor REST se usará el framework Spring REST. Éste se ocupa del mapeado de las peticiones HTTP, enlazando las peticiones con clases contenedoras denominadas controladores que gestionan los métodos HTTP de un dominio específico. También se ocupa de la obtención de los datos recibidos y del procesado de datos para ser enviados.

Como la arquitectura REST está basada en recursos, se crearán controladores para cada tipo de recurso. Cada controlador tendrá los métodos HTTP básicos implementados (GET, PUT, DELETE…), pudiendo, por ejemplo, crear una entidad grupo, listar una ejecución, borrar un grupo, etc. Existirán tantos controladores como entidades existan dentro del modelo de datos.

#### Autenticación y sesión

Aunque la gestión de la autenticación o el acceso podría haber sido gestionado por un framework o protocolo de autenticación y sesión como OpenMP, OAuth o Spring sesión, se va a optar por la implementación de un sistema propio, buscando la simplicidad y la rapidez.

A cada usuario que se conecte al sistema se le asociará una sesión, la cuál será individual para cada acceso y será gestionada por la capa REST. La capa de sesión, la cual está ubicada en el servidor REST, mantendrá un conjunto de sesiones que analizará y borrará pasado un tiempo de inactividad. Los usuarios que deseen autenticarse deberán introducir su identidad indicada en el modelo de datos (email y contraseña) para poder obtener el token asociado a su sesión. A continuación, se describe el diagrama de iteración del proceso de autenticación y sesión.



Ilustración ‑ Diagrama secuencia loggin

Cuando el proceso de autenticación se termina exitosamente, se genera un token de sesión que identifica a la sesión del sistema. Toda iteración con el sistema para la obtención de recursos deberá realizarse con el token de sesión, el cual, será analizado por la capa de sesión.

#### Diagrama de iteración general

Para la obtención de cualquier recurso del sistema se tendrá que seguir una iteración general, esta iteración siempre será la misma y no variará para los diferentes recursos. Todas las solicitudes de recursos serán devueltas utilizando un recurso general genérico, denominado mensaje, que será un contenedor de los recursos y los estados solicitados.



* + 1. **Protocolo de comunicación:**

El lenguaje que entiende el repositorio es el protocolo HTTP. Son ordenes realizadas mediante un protocolo que debe ser conocido para poder comunicarse a este nivel. En este protocolo se entiende que TODAS las entidades (proyectos, grupos, usuarios) están representadas por un ID personal que será lo que lo identifique. Cuando se quiera realizar una acción se deberá saber los ID’s de los afectados.

http://DIRECCIONIP:8080/ACCION/QUE/SE/ORDENA

Para poder conectarnos al repositorio deberemos conocer la dirección IP o el DNS donde se alberga el servidor, tras eso se tendrá que conectar al puerto :8080 donde se albergara el repositorio. Luego para ordenar una acción se utiliza los dominios URL. Posteriormente se ha realizado una serie de tablas para facilitar la comprensión del repositorio.

http://127.0.0.1:8080/raulgf92@gmail.com/contrasena1

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salidas |
| /{email}/{contraseña}  (HTTP method:GET) | Acción que permite la autenticación en el sistema. | Vacío | Mensaje con el ID del usuario. |
| /{idUser}/whoami  (HTTP method:GET) | Acción que devuelve la entidad del usuario que usa el id | Vacío | Devolverá un mensaje con los datos de autenticación y personales. |

* + - 1. Autenticación:
      2. Grupos:

Todas estas acciones llevan por delante el comando **“/group/”** se deberá de concatenar por delante para que acción funcione **(dir+/group/+accion)**.

* + - * 1. Grupos acciones básicas:

Tabla 4‑2.Protocolo repositorio: Grupo acciones básicas

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salida |
| /{idUser}/{groupID}  (HTTP method:GET) | Devuelve la entidad correspondiente al id del grupo entregado | Vacío | Devuelve un mensaje con la entidad de grupo solicitada. |
| /{idUser}/{groupID}  (HTTP method:DELETE) | Elimina la entidad correspondiente al id del grupo entregado | Vacío | Devuelve el resultado de la acción realizada. |
| /{idUser}/{groupID}  (HTTP method:PATH) | Modifica la entidad correspondiente al id del grupo entregado | Entidad grupo con los datos modificados que se va a modificar | Devuelve el resultado de la acción realizada |
| /{idUser}/all  (HTTP method:GET) | Devuelve todos los grupos creados por el usuario | Vacío | Devuelve un listado de ID’s de grupos que fueron creados por el usuario. |
| /{idUser}/create  (HTTP method:GET) | Facilita la creación de los grupos mediante un ejemplo | Vacío | Devuelve una entidad grupo con datos rellenados con ejemplos para poder crear el grupo. |
| /{idUser}/create  (HTTP method:POST) | Crea el grupo entregado por el usuario | Entidad grupo con los datos que se quieren crear | Devuelve el resultado de la acción realizada |

* + - * 1. Grupos miembros:

Tabla 4‑3.Protocolo repositorio: Grupo miembros

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salidas |
| /{idUser}/{groupID}/member  (HTTP method:GET) | Acción que permite visualizar todos los miembros de un grupo | Vacío | Devuelve un conjunto de ID’s de usuarios. |
| /{idUser}/{groupID}/member/create  (HTTP method:GET) | Acción que permite conseguir un ejemplo de método de entrada para crear miembros de grupos | Vacío | Devuelve una entidad mensaje que se podrá usar para comunicar con el sistema. |
| /{idUser}/{groupID}/member/create  (HTTP method:POST) | Acción que permite crear miembros para un grupo. | Mensaje de comunicación para generar un miembro | Devolverá el resultado sobre la ejecución de la acción. |
| /{idUser}/{groupID}/  member/{memberID}  (HTTP method:DELETE) | Acción que permite borrar un miembro de un grupo | Vacío | Devolverá el resultado sobre la ejecución de la acción |

* + - * 1. Grupos subgrupos:

Tabla 4‑4.Protocolo repositorio: Grupo subgrupos

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salidas |
| /{token}/{groupID}/subgroups  (HTTP method:GET) | Accion que devuelve todos los subgrupos de un grupo | Vacío | Devuelve un conjunto de ID’s de grupos |
| /{token}/{groupID}/subgroups/{subgroupID} (HTTP method:GET) | Acción que permite conseguir la información de que un grupo es subgrupo de otro | Vacío | Devuelve una entidad mensaje que contiene la información de la orden |
| /{token}/{groupID}/subgroups/{subgroupID}  (HTTP method:POST) | Acción que permite hacer que un grupo sea subgrupo de otro grupo | Vacío | Devolverá el resultado sobre la ejecución de la acción. |
| /{token}/{groupID}/subgroups/{subgroupID} (HTTP method:DELETE) | Acción que permite revertir que un grupo sea subgrupo de otro | Vacío | Devolverá el resultado sobre la ejecución de la acción |

* + - * 1. Grupos proyectos:

Tabla 4‑5.Protocolo repositorio: Grupo proyectos

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salidas |
| /{idUser}/{groupID}/contain/project  (HTTP method:GET) | Acción que permite ver todos los proyectos contenidos en un grupo. | Vacío | Devuelve un mensaje con un conjunto de ID’s de proyectos contenidos en un grupo |
| /{idUser}/{groupID}/contain/project/{projectID}  (HTTP method:GET) | Acción que permite visualizar la información de cuando un proyecto es contenido en un grupo | Vacío | Devuelve un mensaje con la información de la acción |
| /{idUser}/{groupID}/contain/project/{projectID}  (HTTP method:POST) | Acción que permite introducir proyectos en el grupo | Vacío | Devolverá el resultado sobre la ejecución de la acción. |
| /{idUser}/{groupID}/contain/project/{projectID}  (HTTP method:DELETE) | Acción que permite borrar un proyecto de un grupo | Vacío | Devolverá el resultado sobre la ejecución de la acción |

* + - 1. Proyectos:

Todas estas acciones llevan por delante el comando **“/project/”** se deberá de concatenar por delante para que acción funcione **(dir+/project/+accion).**

Tabla 4‑6 .Protocolo repositorio: Proyectos

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salida |
| /{idUser}/{projectID}  (HTTP method:GET) | Devuelve la entidad correspondiente al id del proyecto entregado | Vacío | Devuelve un mensaje con la entidad de proyecto solicitada. |
| /{idUser}/{projectID}  (HTTP method:DELETE) | Elimina la entidad correspondiente al id del proyecto entregado | Vacío | Devuelve el resultado de la acción realizada. |
| /{idUser}/{projectID}  (HTTP method:PATH) | Modifica la entidad correspondiente al id del proyecto entregado | Entidad proyecto con los datos modificados que se va a modificar | Devuelve el resultado de la acción realizada |
| /{idUser}/all  (HTTP method:GET) | Devuelve todos los proyectos creados por el usuario | Vacío | Devuelve un listado de ID’s de proyecto que fueron creados por el usuario. |
| /{idUser}/create  (HTTP method:GET) | Facilita la creación de los proyectos mediante un ejemplo | Vacío | Devuelve una entidad proyectos con datos rellenados con ejemplos para poder crear el proyecto. |
| /{idUser}/create  (HTTP method:POST) | Crea el proyecto entregado por el usuario | Entidad proyecto con los datos que se quieren crear | Devuelve el resultado de la acción realizada |

* + - 1. Ejecuciones:

Todas estas acciones llevan por delante el comando **“/execution/”** se deberá de concatenar por delante para que acción funcione **(dir+/execution/+accion)**.

Tabla 4‑7.Protocolo repositorio: Ejecuciones

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salida |
| /{idUser}/{groupID}/{projectID}  (HTTP method:POST) | Crear una entidad ejecución y la ejecuta. Usando un proyecto con projectID albergado en el grupo de groupID | Archivo JSON, que será utilizado como la entrada de la ejecución del proyecto. | Devuelve el resultado de la acción realizada |
| /{idUser}/{executionID}  (HTTP method:GET) | Devuelve la entidad correspondiente al id de la ejecución entregada | Vacío | Devuelve un mensaje con la entidad de ejecución solicitada. |
| /{idUser}/{executionID}  (HTTP method:DELETE) | Elimina la entidad ejecución representada por executionID. En el caso de que este en ejecución se detendrá. | Vacío | Devuelve un mensaje con la entidad de ejecución solicitada. |
| /{idUser}/{executionID}/project  (HTTP method:GET) | Devuelve la información del proyecto que está ejecutando la entidad ejecución representada por el executionID | Vacío | Devuelve un mensaje con la entidad del proyecto de la ejecución solicitada. |
| /{idUser}/running  (HTTP method:GET) | Devuelve todas las ejecuciones del usuario que se están ejecutando. | Vacío | Devuelve un mensaje con el conjunto de las identidades de las ejecuciones en ejecución |
| /{idUser}/finish  (HTTP method:GET) | Devuelve todas las ejecuciones del usuario que hayan finalizado | Vacío | Devuelve un mensaje con el conjunto de las identidades de las ejecuciones finalizadas |

* + - 1. Administración:

Todas estas acciones llevan por delante el comando **“/admin/”** se deberá de concatenar por delante para que acción funcione **(dir+/admin/+accion)**.

Tabla 4‑8.Protocolo repositorio: Administración

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salida |
| /{idUser}/createAccount  (HTTP method:POST) | Crea una cuenta con el rol que elija el administrador | Entidad userLogin con los datos rellenos para su creación | Devuelve el resultado de la acción realizada |
| /{idUser }/allUsers  (HTTP method:GET) | Devuelve todos los usuarios del sistema | Vacío | Devuelve un mensaje con el conjunto de identidades de usuarios |
| /{idUser}/allGroups  (HTTP method:GET) | Devuelve todos los grupos del sistema | Vacío | Devuelve un mensaje con el conjunto de identidades de grupos |
| /{idUser}/allProjects  (HTTP method:GET) | Devuelve todos los proyectos del sistema | Vacío | Devuelve un mensaje con el conjunto de identidades de proyectos |
| /{idUser}/allExecution  (HTTP method:GET) | Devuelve todas las ejecuciones del sistema | Vacío | Devuelve un mensaje con el conjunto de identidades de ejecuciones |

* + - 1. Navegación:

Navegación: Todas estas acciones llevan por delante el comando **“/path/”** se deberá de concatenar por delante para que acción funcione **(dir+/path/+accion)**.

Tabla 4‑9.Protocolo repositorio: Navegación

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Descripción | Entradas  (Body) | Salida |
| /{idUser}/main  (HTTP method:GET) | Devuelve los proyectos públicos y los grupos superiores del usuario | Vacío | Devuelve un mensaje con todas las identidades de proyectos y grupos que contiene la navegación actual |
| /{idUser}/enter/{groupID}  (HTTP method:GET) | Devuelve todos los proyectos y grupos que puede visualizar el usuario | Vacío | Devuelve un mensaje con todas las identidades de proyectos y grupos que contiene la navegación actual |
| /{idUser}/public  (HTTP method:GET) | Devuelve la información del grupo publico | Vacío | Devuelve un mensaje con la entidad del grupo publico |

## Capa Negocio

La función principal de la capa negocio es el gestionar y operar las diferentes opciones de los servicios que proporciona el repositorio. Esta capa estará enteramente implementada en java sin la utilización de ningún framework.

Esta capa estará implementada utilizando dos patrones de diseño. Estos patrones facilitan la utilización de códigos en su desarrollo y ampliaciones. La capa utiliza una factoría básica que crea los diversos servicios (capas), los cuales serán implementados utilizando el tipo *singleton* para que su propia existencia sea única.

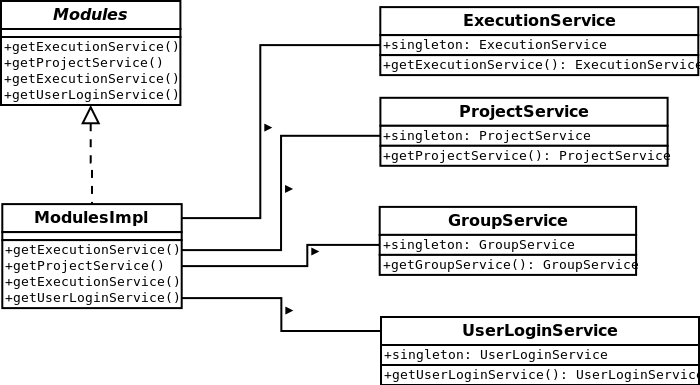


Ilustración ‑ Esquema Servicios Negocio

## Capa de datos

El objetivo de la capa de datos es el de ser el intermediario entre las demás capas y la base de datos. Esta capa traduce los datos existentes en la base de datos en entidades o clases del modelo e introduce datos nuevos en la base de datos. Para desarrollar esta capa se crearán clases para la manipulación de datos de cada entidad en la base de datos. Todas ellas serán creadas por una clase factoría básica que implementara una interfaz básica para comunicarse con los demás módulos del subsistema.

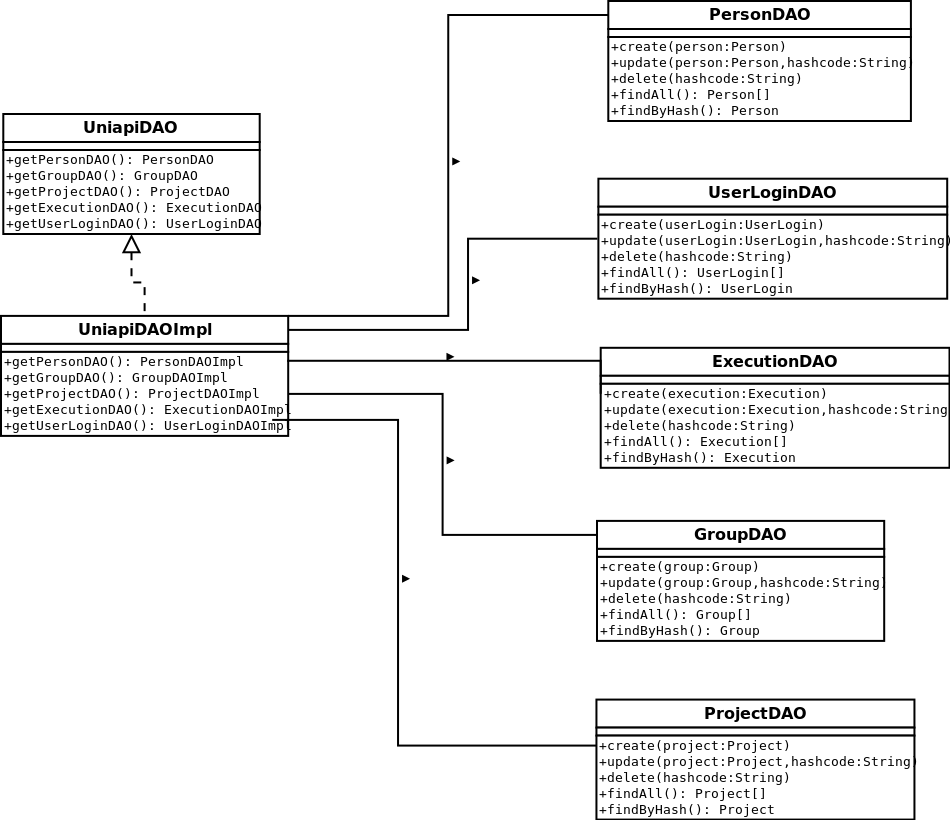


Ilustración ‑ Esquema DAO Entidades

Puesto que Neo4j también almacena las relaciones entre los nodos que representan a las entidades, es necesario crear clases para la gestión de dichas relaciones.

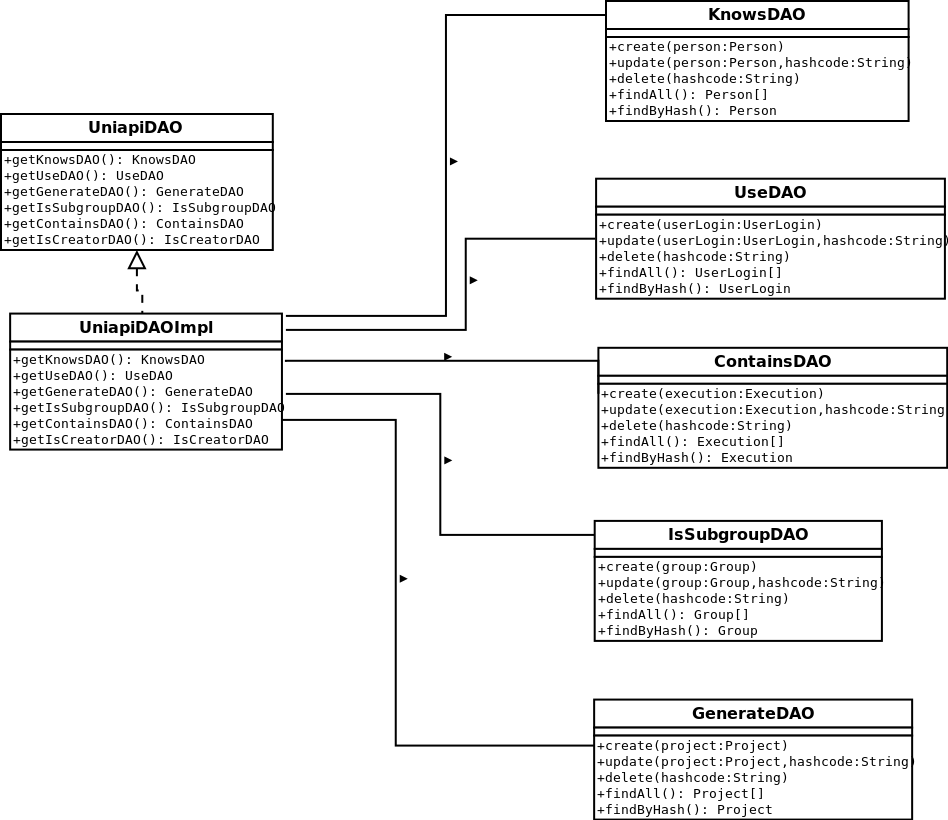


Ilustración ‑ Esquema DAO Relaciones

## Capa ejecución

La capa ejecución tiene como objetivo ejecutar proyectos, monitorizar su ejecución y recoger su respuesta para comunicarlo luego con las demás capas. Esta capa será implementada en java. La capa ejecución se comunica directamente con el Sistema Operativo y con los repositorios GIT, por lo que se utilizarán tanto librerías externas como librerías internas de java.

### Sincronización de proyectos:

La capa de ejecución creará una jerarquía de proyectos en el sistema de archivos del sistema operativo anfitrión, donde, usando una librería externa denominada **JGit**, actualizará y descargara los datos de los proyectos. Para cada proyecto que se vaya a crear se generará un espacio personal donde se descargará la información. También se creará un sistema de monitorización de los proyectos, actualizándose cada vez que se genera se un cambio.

**JGit** es una librería desarrollada por Eclipse para facilitar la comunicación entre los repositorios GIT por parte de los clientes. Cualquier operación GIT puede ser realizada usando sus códigos. La capa de ejecución contendrá un servicio denominado servicio GIT que contendrá todas las operaciones.

### Ejecución de proyectos

El principal objetivo de la capa de ejecución es la de ejecutar proyectos de diferentes índoles de manera genérica. Para ello se utilizará un patrón de diseño denominado factoría abstracta, donde dependiendo del tipo de proyecto que se ejecute se enlazará con el servicio que puede ejecutar el proyecto.

Mientras los proyectos sean ejecutados se mantendrá un conjunto de referencias a los mismos para comprobar sus estados, pudiendo parar las ejecuciones cuando desee el usuario.

### Respuestas de ejecuciones

La capa de ejecución tratará los datos de las ejecuciones de los proyectos y se comunicará con las demás capas con el objetivo de informar a los usuarios que lo soliciten.

# Diseño de sistema: Subsistema aplicación web

El subsistema aplicación web tiene varios objetivos como sistema general:

* **Comunicación con el usuario**: El subsistema tendrá que proporcionar los medios necesarios para facilitar una gestión y acceso a las diferentes tareas que podrá hacer en la aplicación por parte del usuario.
* **API de comunicación del repositorio**: El subsistema contendrá una librería de apoyo para la comunicación entre el repositorio y la librería. Esta librería será independiente de la IU (Interfaz de usuario) y podrá utilizarse en cualquier programa que no sea el propio subsistema de la aplicación web.

El subsistema sigue un **modelo - vista - controlador (MVC)** donde se crearán distintos elementos que se comunicarán con el usuario generando y recibiendo eventos.

## Lado servidor

El lado servidor será un servidor web en JavaScript que se creará utilizando Node.js.



### Gestión de recursos web:

El objetivo principal del servidor será el gestionar los recursos. Para ello, el servidor web creara una jerarquía y lo enlazara con un dominio URL para la petición de los recursos por parte del cliente. Cuando el cliente pida un recurso X en cierto dominio HTTP, El servidor interpretara este dominio obteniendo el recurso y enviándolo al cliente.

### Creación de cuentas:

Para una mejor seguridad en el sistema, soló las cuentas enviadas al repositorio mediante el servidor web serán gestionadas. El servidor web deberá proveer un controlador y un modelo para la gestión de la creación de cuentas. En caso de un evento de creación de cuentas, el servidor web deberá hacer una petición de cuentas y enviarla al repositorio.

## Lado cliente

El objetivo del lado del cliente es obtener todos los eventos que interactúa el usuario, comprenderlos, analizarlos y dar una vista nueva personalizada al evento procesado. El lado del cliente contendrá el total del cómputo de la aplicación web, pero manteniendo el modelo MVC.

### Angular.js: El MVC en el cliente

Angular.js es un framework de JavaScript que utiliza un modelo MVC en el cliente con apoyo AJAX y unos servidores backend (servidor web y repositorio), que le despliega un flujo de datos procesados para que sean mostrados al usuario.

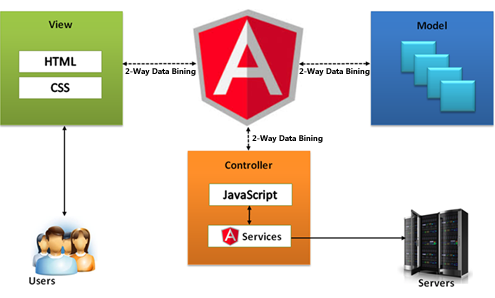


Ilustración ‑ Arquitectura Angular.js MVC

Como se puede ver en la imagen anterior, el modelo MVC de Angular mantiene los 3 componentes principales del modelo (Modelo, vista y controlador):

* **Vista**: La vista a modificar se realiza sobre el HTML como en las plantillas de modelos MVC de servidor. La diferencia es que las plantillas se transforman en variables que se iniciaran por los controladores cuando su ciclo de vida se inicie.
* **Controlador**: Los controladores gestionan los estados, crean eventos para el usuario e inyectan la información a los modelos. La mejora que Angular.js ofrece frente a los demás controladores convencionales es el uso de directivas, las cuales, son elementos HTML que llevan enlazada una estructura MVC fija.
* **Modelo**: Los modelos representan los datos que se van a mostrar en la interfaz de usuario, en función de lo entregado por el controlador. Angular aporta como mejora que el controlador puede manejar el CSS de la página web como si fuera un dato mismo del usuario.

#### Estructura General de la aplicación

La estructura general de la aplicación será una jerarquía de controladores y directivas (las directivas contienen a su vez controladores y otras directivas).

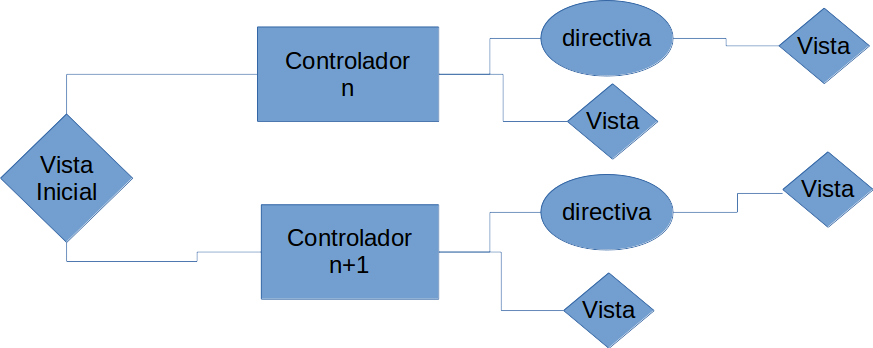


Ilustración ‑ Arquitectura básica de app cliente Angular.js

La vista general llama a un conjunto de controladores que cargan los modelos que contienen las vistas, generando una vista personalizada para cada usuario. Cada vista puede contener varias directivas, las cuales, a su vez, podrán contener un conjunto de controladores, modelos y vistas, siguiendo un árbol en profundidad de directivas con otros controladores...

Siguiendo estas directrices se creará un controlador con una vista asignada para cada elemento de los menús del interfaz de usuario (prototipos de interfaz de usuario del análisis del sistema). Para cada funcionalidad individual de los menús, se crearán directivas que se gestionarán con controladores internos.

#### API de comunicación

Los modelos MVC albergados en los clientes se nutren de información REST de servicios en otros servidores web, los cuales tienen unos protocolos de comunicación y autenticación. Para facilitar la comunicación con el servidor se creará una librería externa (denominada API).

Angular facilita mucho al modelo MVC la generación de API’s para sus sistemas con las clases Service. Una clase Service es una clase basada en el patrón *singleton* que se mantiene ejecutada todo el tiempo que la sesión del usuario esté en funcionamiento. Todos los componentes utilizaránn estas clases para obtener información para las vistas.

La información se obtendrá a través de una comunicación HTTP con el servidor REST. Esto será posible gracias al protocolo AJAX.

|  |  |  |
| --- | --- | --- |
| Nombre de las funciones | | |
| Whoami() | **getMyProjects**() | **deleteSubgroup**() |
| changePass() | **getMyGroups**() | **getMainPathGroups**() |
| changeBio() | **getProject**() | **getPathGroups**() |
| createAccount() | **updateProject**() | **projectsInsideGroup**() |
| getAllUsers() | **deleteProject**() | **getProjectsInsideGroup**() |
| getAllProjects() | **createProject**() | **getExecutionsRunning**() |
| getAllGroups() | **createGroup**() | **getAllExecutionsRunning**() |
| getGroup() | **updateGroup**() | **getExecution**() |
| deleteGroup() | **inviteToGroup**() | **getProjectOfExecution**() |
| getAllGroupMembers() | **removeMemberOfTheGroup**() | **getExecutionFinish**() |
| getSubgroups() | **getPathProjects**() | **getAllExecutionsFinish**() |
| createSubgroup() | **putGroupProject**() | **executedProject**() |
| getMainPathProjects() | **deleteProjectInsideGroup**() |  |