Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

ÁREA DE INGENIERÍA TELEMÁTICA

TRABAJO FIN DE GRADO Nº 17010523

"Aplicación web para la gestión de un repositorio"

Autor:
Raúl García Fernández

Tutor:
Raquel Blanco Aguirre

Junio de 2017

Diseño Del Sistema

Índice

Ilustraciones

Tablas

1. Arquitectura del sistema

1.1 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 mínimo 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.

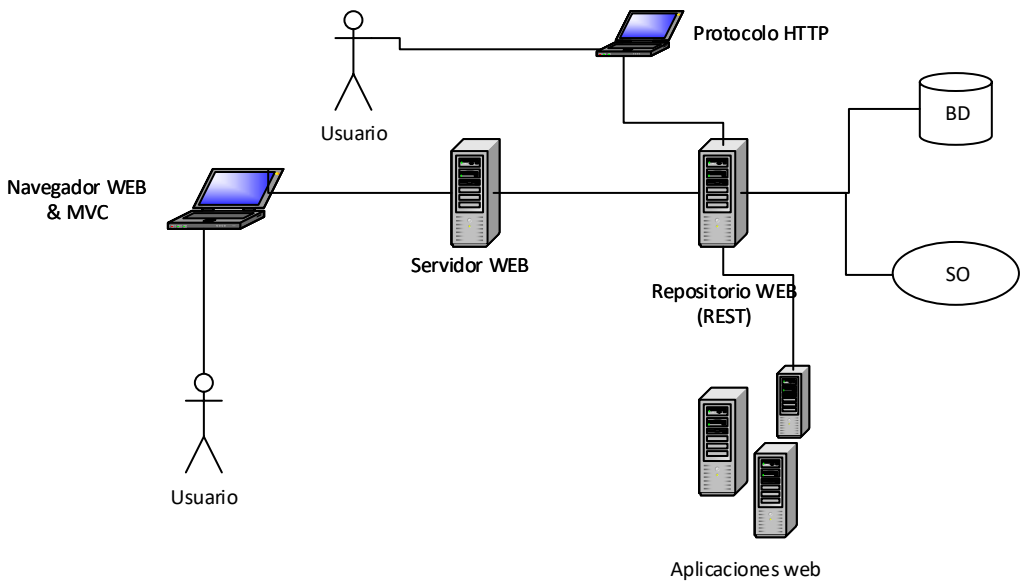


Ilustración 1-1 (Esquema general aplicación)

1.2 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.

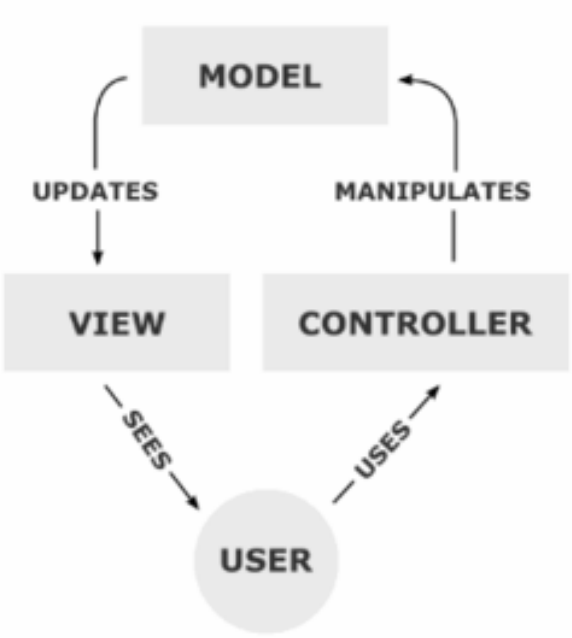


Ilustración 1-2 (Arquitectura MVC)

1.3 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…)

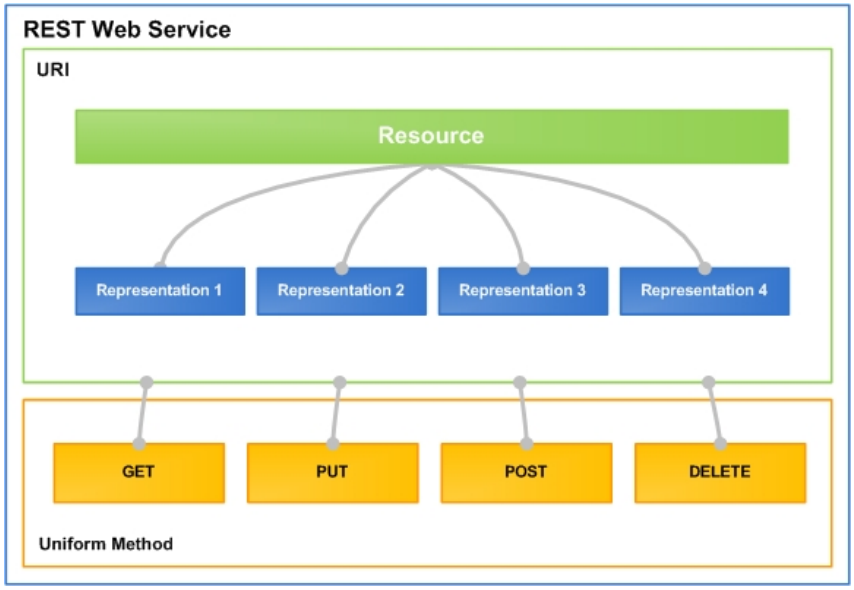


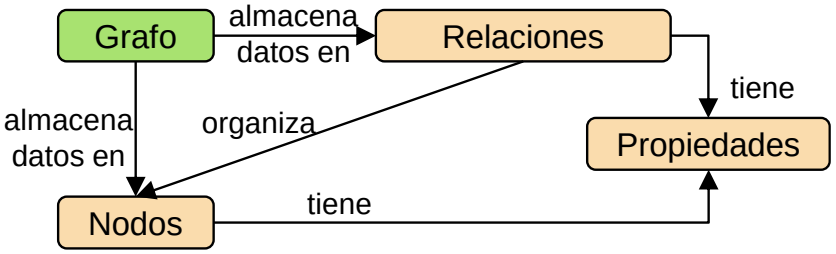
Ilustración 1-3 (Arquitectura-REST)

2. Diseño de datos

2.1 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.



2.2 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.

2.2.1 MD-Nodos:UserLogin:

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

Tabla 2-1 (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.

2.2.2 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 2-2 (Person)

2.2.3 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 2-3 (Project)

2.2.4 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 2-4 (Group)

2.2.5 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.

2.3 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 2-5 (Neo4j: conjunto de entidades)

2.4 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.

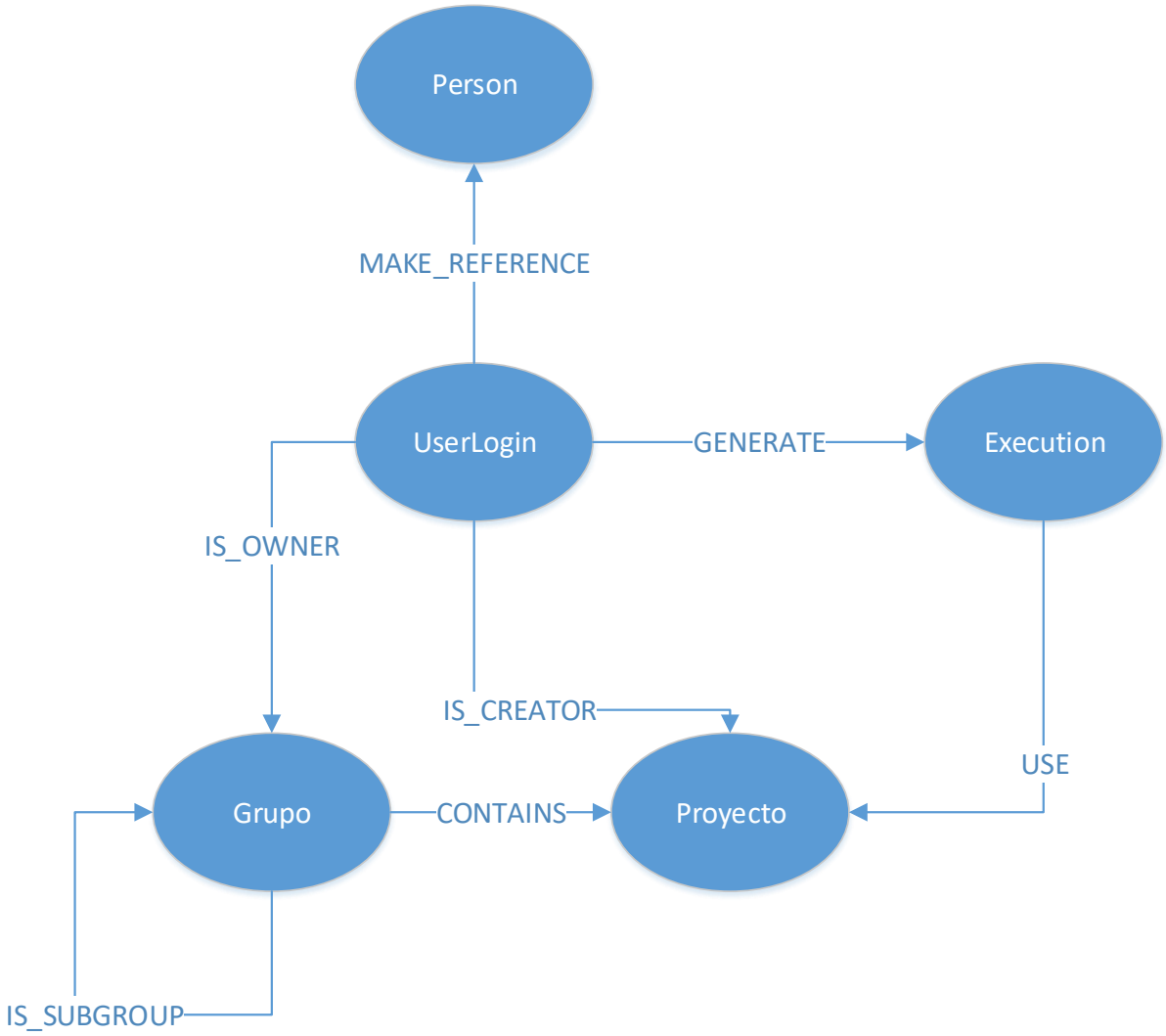


Ilustración 2-1 (Estructura general de datos)

3. 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.

3.1 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…

3.1.1 Arquitectura REST

Ilustración 3-1 (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.

3.1.2 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.

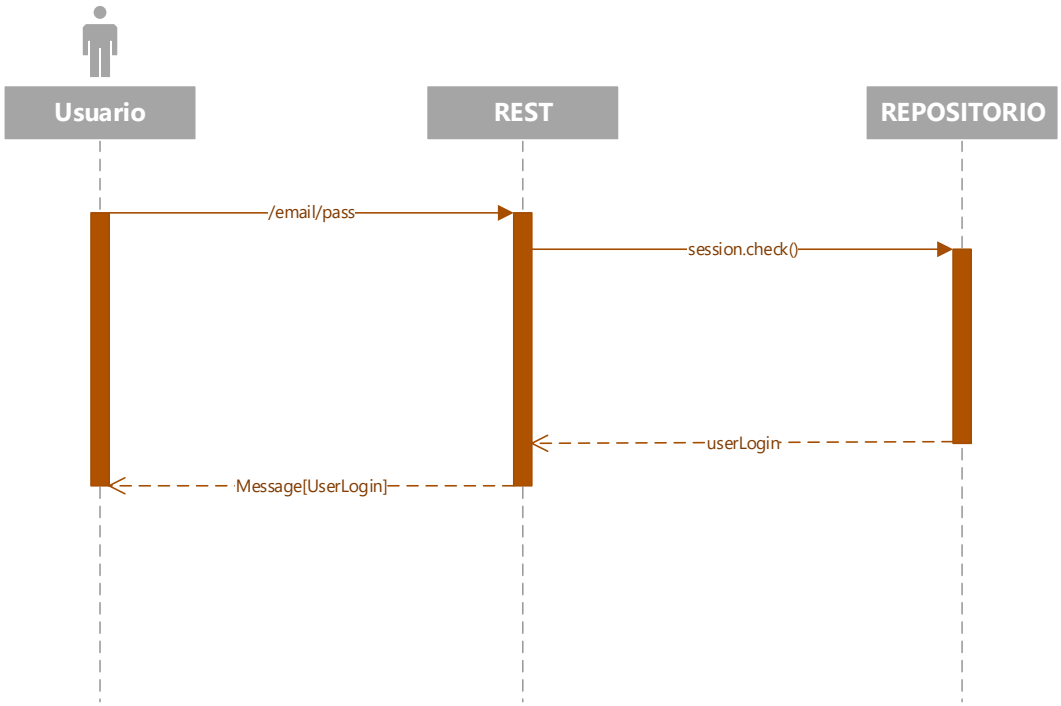
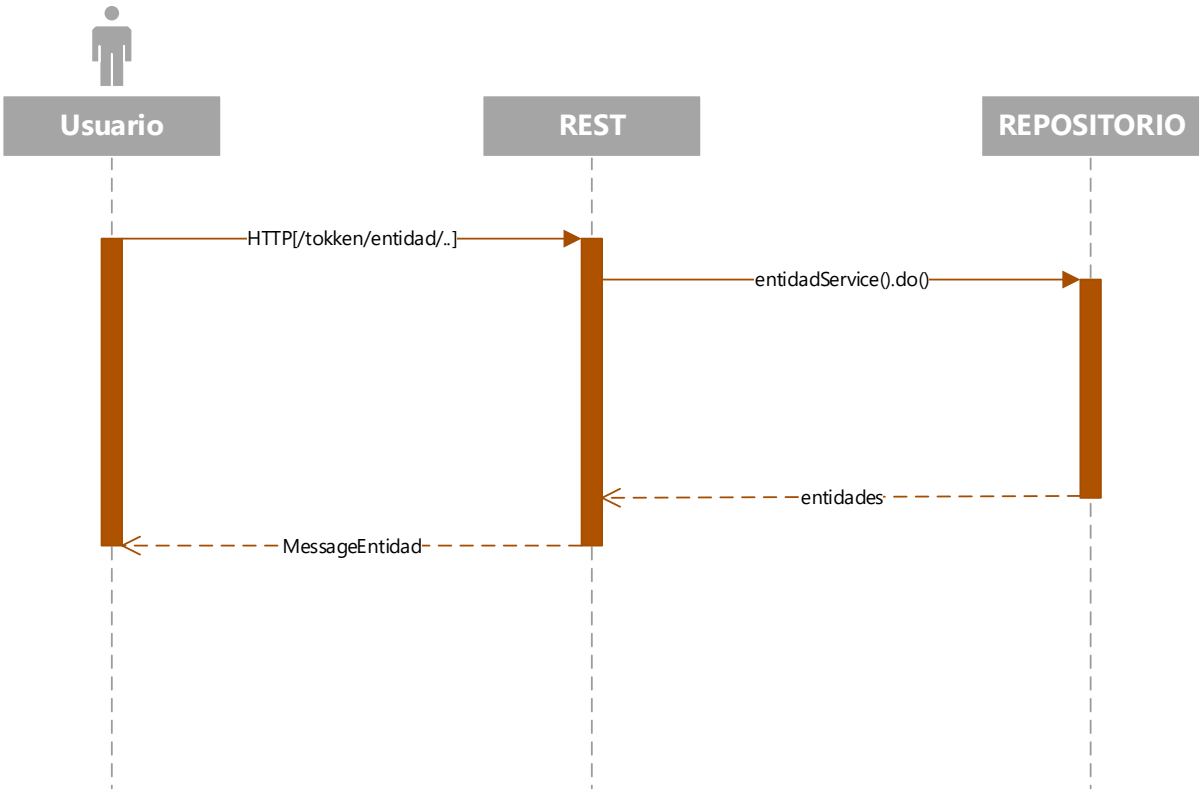


Ilustración 3-2 Diagrama secuencia loggin

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.

3.1.3 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.



3.1.4 Protocolo de comunicación

El lenguaje que entiende el repositorio es el protocolo HTTP. Son órdenes 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 las 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 conectarse al repositorio se deberá conocer la dirección IP o el DNS donde se alberga el servidor, tras eso se tendrá que conectar al puerto :8080 donde se albergará el repositorio. Posteriormente, para ordenar una acción se utiliza los dominios URL.

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

A continuación, se muestran una serie de tablas para facilitar la comprensión del repositorio.

3.1.4.1 Autenticación:

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.

3.1.4.2 Grupos:

Estas acciones tendrán siguiente formato: **dir+/group/+accion**.

3.1.4.2.1 Grupos acciones básicas:

Tabla 3-1.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

3.1.4.2.2 Grupos miembros:

Tabla 3-2.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}**	Acción que permite borrar	Vacío	Devolverá el resultado sobre la

(HTTP method:DELETE)	un miembro de un grupo		ejecución de la acción

3.1.4.2.3 Grupos subgrupos:

Tabla 3-3.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

3.1.4.2.4 Grupos proyectos:

Tabla 3-4.Protocolo repositorio: Grupo proyectos

Método	Descripción	Entradas (Body)	Salidas
/{idUser}/{groupID}/contain/project	Acción que permite ver	Vacío	Devuelve un mensaje con

(HTTP method:GET)	todos los proyectos contenidos en un grupo.		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

3.1.4.3 Proyectos:

Estas acciones tendrán siguiente formato: **dir+/project/+accion.**

Tabla 3-5 .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	Vacío	Devuelve el resultado de la acción realizada.

	proyecto entregado		
/{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

3.1.4.4 Ejecuciones:

Estas acciones tendrán siguiente formato: **dir+/execution/+accion**.

Tabla 3-6.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

3.1.4.5 Administración:

Estas acciones tendrán siguiente formato: **dir+/admin/+accion**.

Tabla 3-7.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

3.1.4.6 Navegación:

Estas acciones tendrán siguiente formato: **dir+/path/+accion**.

Tabla 3-8.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

3.2 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.

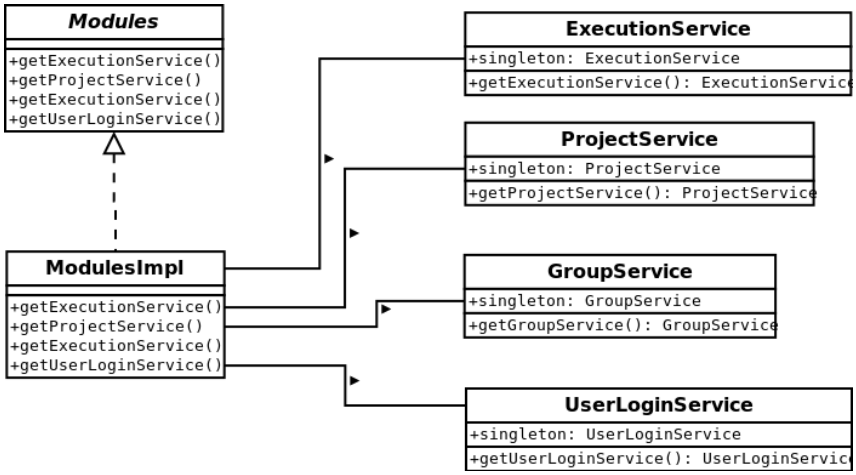


Ilustración 3-3 Esquema Servicios Negocio

3.3 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.

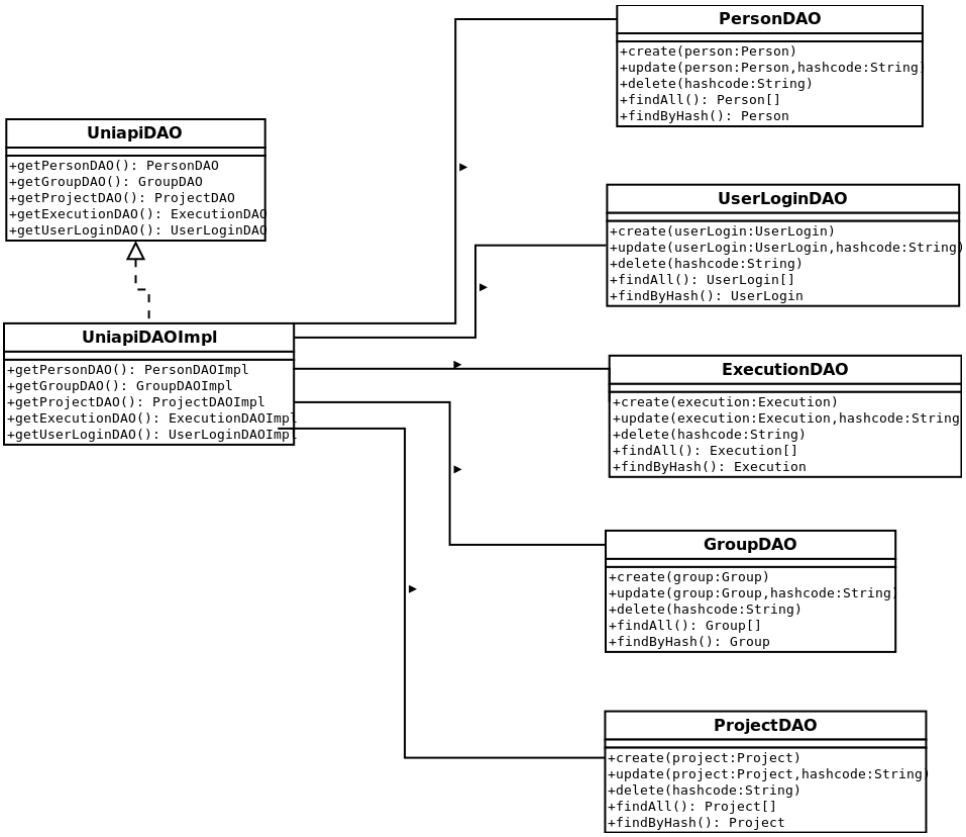


Ilustración 3-4 Esquema DAO Entidades

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.

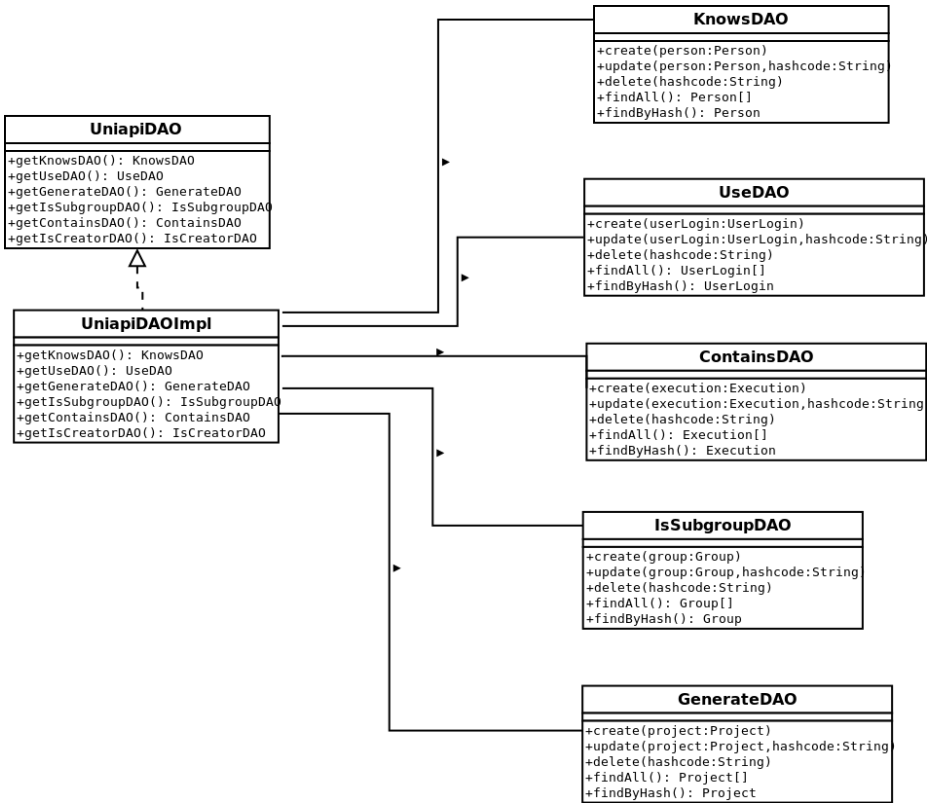


Ilustración 3-5 Esquema DAO Relaciones

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

4. 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.

4.1 Lado servidor

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

4.1.1 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.

4.1.2 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.

4.2 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.

4.2.1 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.

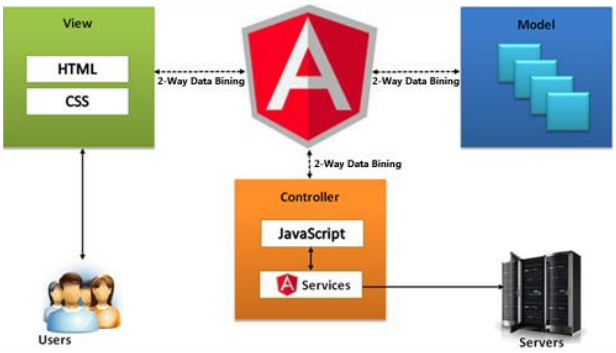


Ilustración 4-1 Arquitectura Angular.js MVC

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.

4.2.1.1 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).

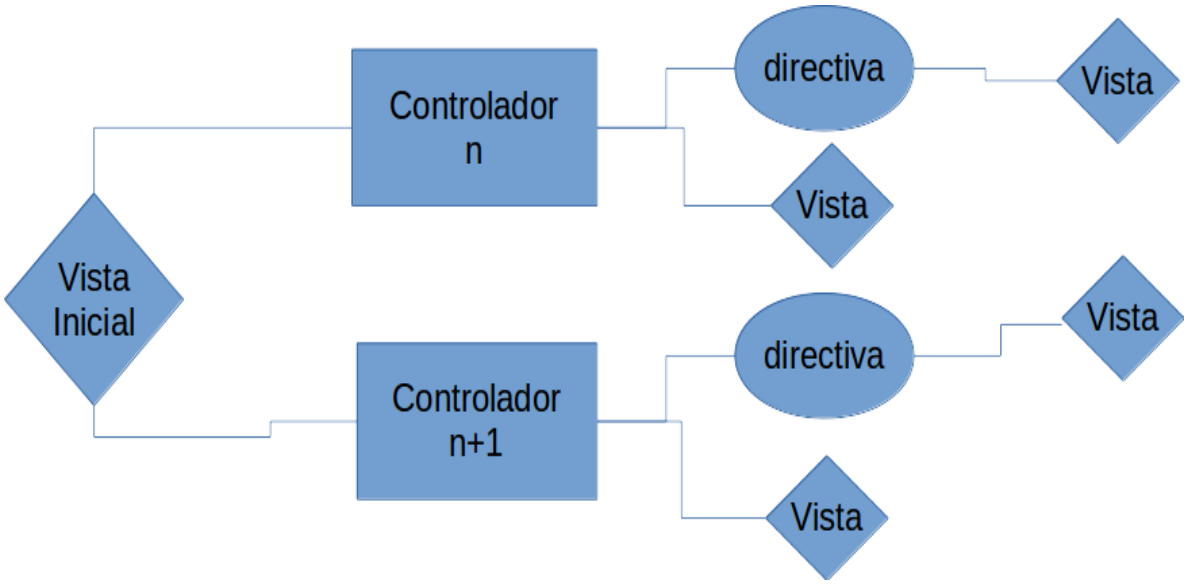


Ilustración 4-2 Arquitectura básica de app cliente Angular.js

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.

4.2.1.2 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**()	