Contenido

[Ilustraciones 2](#_Toc481185480)

[Tablas 2](#_Toc481185481)

[1. Introducción: 3](#_Toc481185482)

[2. Análisis general del sistema: 4](#_Toc481185483)

[2.1 Introducción al sistema: 4](#_Toc481185484)

[2.2 Arquitectura Modelo-vista-controlador: 5](#_Toc481185485)

[2.3 Arquitectura REST: 6](#_Toc481185486)

[3. Diseño de datos: 7](#_Toc481185487)

[3.1 Introducción: 7](#_Toc481185488)

[3.2 Modelo de los datos-Entidades: 7](#_Toc481185489)

[3.2.1 MD-Entidades: UserLogin 8](#_Toc481185490)

[3.2.2 MD-Entidades: Person 9](#_Toc481185491)

[3.2.3 MD-Entidades: Project 10](#_Toc481185492)

[3.2.4 MD-Entidades: Group 11](#_Toc481185493)

[3.2.5 MD-Entidades: Ejecuciones 12](#_Toc481185494)

[3.3 Modelo de datos - Neo4j: 14](#_Toc481185495)

[3.3.1 MD-Neo4j Introducción: 14](#_Toc481185496)

[3.3.2 MD - Neo4J: Relaciones 15](#_Toc481185497)

[3.3.3 MD – Neo4j: Estructura general de datos 16](#_Toc481185498)

[3.4 Diseño de sistema: Subsistema repositorio 17](#_Toc481185499)

[3.4.1 DS-Subsistema repositorio Web: Capa REST 18](#_Toc481185500)

[3.5 DS-Subsistema repositorio Web: Capa Negocio 20](#_Toc481185501)

[3.6 DS- Subsistema repositorio Web: Capa de datos 21](#_Toc481185502)

[3.7 DS-Subsistema repositorio: Capa ejecución 23](#_Toc481185503)

[3.7.1 Sincronización de proyectos: 23](#_Toc481185504)

[3.7.2 Ejecución de proyectos: 24](#_Toc481185505)

[3.7.3 Respuestas de ejecuciones: 24](#_Toc481185506)

[4. Diseño de sistema: Subsistema aplicación web 25](#_Toc481185507)

[4.1 DS- Subsistema aplicación web: JavaScript y el peso del cómputo 26](#_Toc481185508)

[4.2 DS – Subsistema aplicación web: Lado servidor 27](#_Toc481185509)

[4.2.1 Gestión de recursos: 27](#_Toc481185510)

[4.2.2 Creación de cuentas: 28](#_Toc481185511)

[4.3 DS-Subsistema aplicación web: Lado cliente 28](#_Toc481185512)

[4.3.1 Angular.js: El MVC en el cliente 28](#_Toc481185513)

Ilustraciones

[Ilustración 1 (Esquema general aplicación) 4](file:///C:\Users\rulo-\Desktop\UniApi\documentación\individual\DiesñoDelSistema.docx#_Toc481185448)

[Ilustración 2 (Arquitectura MVC) 5](file:///C:\Users\rulo-\Desktop\UniApi\documentación\individual\DiesñoDelSistema.docx#_Toc481185449)

[Ilustración 3 (Arquitectura-REST) 6](#_Toc481185450)

[Ilustración 4 (neo4j: Ejemplo básico estructura) 14](#_Toc481185451)

[Ilustración 5 (Neo4j: Ejemplo entidad-relación) 15](#_Toc481185452)

[Ilustración 6 (Estructura general de datos) 17](#_Toc481185453)

[Ilustración 7 (Logo Spring) 19](#_Toc481185454)

[Ilustración 8 Esquema Servicios Negocio 21](#_Toc481185455)

[Ilustración 9 Esquema DAO Entidades 22](#_Toc481185456)

[Ilustración 10 Esquema DAO Relaciones 23](#_Toc481185457)

[Ilustración 11 Arquitectura Angular.js MVC 29](#_Toc481185458)

[Ilustración 12 Arquitectura básica de app cliente Angular.js 30](#_Toc481185459)

Tablas

[Tabla 1 (UserLogin) 8](#_Toc481185460)

[Tabla 2 (Person) 9](#_Toc481185461)

[Tabla 3 (Project) 11](#_Toc481185462)

[Tabla 4 (Group) 12](#_Toc481185463)

[Tabla 5 (Neo4j: conjunto de entidades) 16](#_Toc481185464)

# Introducción:

Este documento denominado “Diseño del sistema”, es un documento de un conjunto de documentos cuya finalidad es el definir y diseñar una “Aplicación web que gestione un repositorio”.

Para poder describir mejor el problema deberemos desarrollar el significado de lo que se va a definir, es decir, “Aplicación web que gestione un repositorio”. Un repositorio es según la Wikipedia: *Un* ***repositorio de software*** *es un lugar de almacenamiento del cual pueden ser recuperados e instalados los paquetes de software en un ordenador.*

Nuestro repositorio mantendrá un lugar de almacenamiento dentro del ordenador que lo instale, con el fin de contener diversos proyectos que albergarán programas de diversos lenguajes que podrán ser ejecutados. Nuestro repositorio devolverá no es programa almacenado, sino, la información de interactuar con él.

Nuestra aplicación web, gestionara la comunicación con nuestro repositorio. Pidiéndole información y mostrándosela al usuario de una manera sencilla para su entendimiento. Se insta que si se quiere saber más sobre el objetivo de la aplicación se lea el documento general de la memoria.

El objetivo de este documento es el realizar un **diseño del sistema de información.** El cual su función es definir la arquitectura de hardware y software, componentes, módulos y datos de un sistema de cómputo, a efectos de satisfacer ciertos requerimientos. Requerimientos que han sido proporcionados por el análisis del sistema, documento descrito en el conjunto de documentos técnicos.

# Análisis general del sistema:

### Introducción al sistema:

El sistema estará dividido en dos grandes subsistemas que a su vez están divididos en capas independientes, buscando el minino 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 entendimiento del usuario más que el del repositorio. Se basa en una arquitectura modelo- vista-controlador.
* **Repositorio**: Subsistema cuyo principal objetivo es la realización de la actividad del sistema. Prima la realización de actividades, frente a sus mandatos. Es necesario un conocimiento de sus mandatos o protocolo para poder realizar actividades. Se basa en una arquitectura REST.

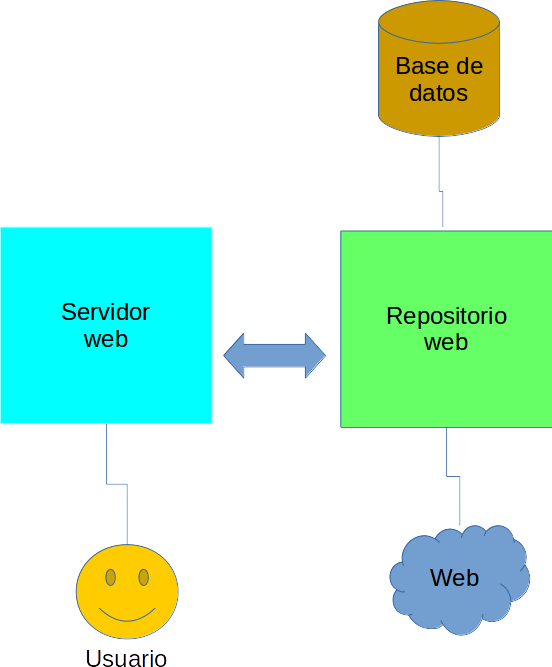


Ilustración (Esquema general aplicación)

## 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 del usuario. El modelo MVC genera para cada interacción del usuario con el sistema una vista personalizada para el mismo o el momento en el que se encuentre. Sus componentes principales son:

* **Modelo**: Parte de la arquitectura que gestiona el acceso a los datos y los maneja para envolverlos en condiciones correctas para el usuario y las capas superiores.
* **Controlador**: Es el intermediario 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 posteriormente realizar una vista personalizada para el usuario con los datos del modelo.
* **Vista**: Presenta los datos manejados por el modelo y proporciona las herramientas necesarias para generar los eventos que posteriormente avisaran al controlador.

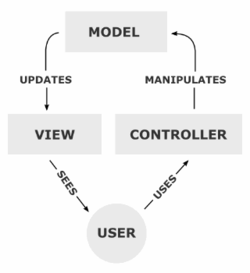


Ilustración (Arquitectura MVC)

## Arquitectura REST:

La Arquitectura REST es una arquitectura de sistemas web que utilizan la tecnología HTTP para el transporte de información o indicación de acciones. Esta arquitectura facilita la creación de comunicación de servidor-cliente. Ya que sigue el protocolo HTTP, protocolo universal para cualquier tipo de aplicación o sistema operativo.

Esta arquitectura se basa en un sistema de Petición-Respuesta asíncrona sin estado, es decir, pides algo y se te responde algo cuando sea posible. La arquitectura es sin estado, es decir, el sistema no guarda estados, cada petición es independiente y acaba cuando es finalizada. Los elementos principales de la arquitectura REST son:

* **Recursos**: Un recurso es una entidad o elemento definido por una identidad. En REST los recursos son enviados, recibidos, manipulados o accionados por servicios de negocios. Cuando solicitamos a un servicio REST recibiremos un Recurso, que puede contener varios recursos a su vez.
* **Acciones**: Las acciones son una derivación de recursos, es la abstracción en recursos de una acción. Por ejemplo: Si mandamos correr en vez enviaremos un recurso “Anda” con un módulo de dirección y una velocidad. El servidor REST realizará la acción y devolverá un recurso indicando el estado.
* **Métodos prefijados**: Los servidores REST proveen de un conjunto de acciones a realizar sobre sobre los recursos. Cuando llamamos a un recurso. Tendremos una lista de acciones (GET, PUT, POST, DELETE…)

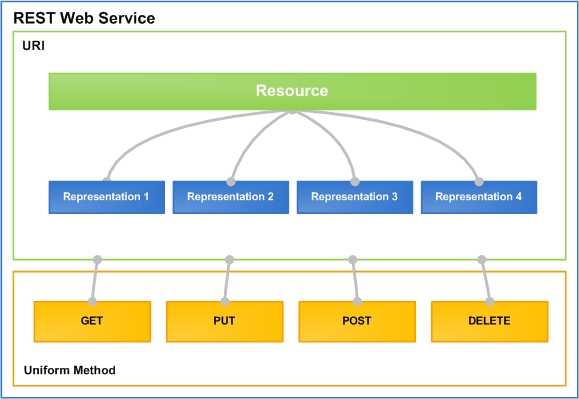


Ilustración (Arquitectura-REST)

# Diseño de datos:

## Introducción:

El diseño de datos proporcionara la información e indicaciones para realizar el modelo de datos del sistema. Esto modelo será permanente y será el utilizado por la aplicación. Para la realización del modelo de datos utilizaremos una tecnología **NO-SQL.** Esta tecnología se basa en no utilizar las bases comunes relacionales, sino que utilizan otros medios para almacenar los datos. En concreto a petición del organizador de este trabajo se utilizará la base de datos **Neo4j.**

## Modelo de los datos-Entidades:

En este punto se describirá las diferentes clases que contendrá el sistema, siguiendo el modelo de negocio del análisis del sistema, describiremos de forma final todas las clases del sistema, listando sus atributos y su tipo de información (Utilizaremos el inglés para denominar los atributos, simplemente por gusto del programador).

Dado cualquier tipo de Base de datos, esta, va tener unas entidades con unos atributos indistintos a la implementación. A continuación, describiremos las entidades básicas y después añadiremos a estas entidades propiedades en función de la implementación de la base de datos.

### MD-Entidades: UserLogin

|  |  |  |
| --- | --- | --- |
| Entidad | UserLogin | |
| Objetivo | Identificar al usuario del sistema y facilitar información para su autenticación. | |
| 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. |

Tabla (UserLogin)

### MD-Entidades: Person

|  |  |  |
| --- | --- | --- |
| Entidad | Person | |
| Objetivo | Proporcionar datos personales del usuario al sistema para su posible representación. | |
| 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 que 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 que identifica a la entidad frente a otras. |

Tabla (Person)

### MD-Entidades: Project

|  |  |  |
| --- | --- | --- |
| Entidad | Project | |
| Objetivo | Almacena los datos necesarios para la ejecución del proyecto en el sistema. | |
| 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-Entidades: Group

|  |  |  |
| --- | --- | --- |
| Entidad | Group | |
| Objetivo | Almacena los datos necesarios para crear un grupo que almacene proyectos y grupos. Además, contendrá los permisos de las acciones. | |
| 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-Entidades: Ejecuciones

|  |  |  |
| --- | --- | --- |
| Entidad | Execution | |
| Objetivo | Almacena los datos de una ejecución. Para saber su estado y su resultado. | |
| 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. |

## Modelo de datos - Neo4j:

### MD-Neo4j Introducción:

Neo4j es una base de datos NO-SQL orientada a grafos. NO-SQL significa que no es una base de datos basada en el motor relacional de SQL. Neo4j en vez de guardar la información en disco mediante tablas como podría hacer SQL. Este lo hace en grafos con referencias o “flechas” que unen las entidades. Un ejemplo de la estructura seria la siguiente:

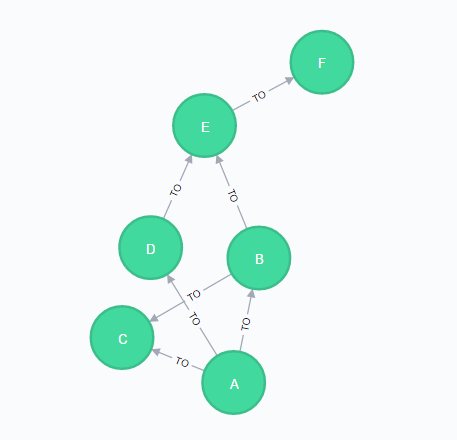


Ilustración (neo4j: Ejemplo básico estructura)

Si se quiere saber más sobre Neo4j se aconseja ir a la memoria donde se mostrará los porqués de esta tecnología.

### MD - Neo4J: Relaciones

Las relaciones son las propiedades especiales de las bases de dato Neo4j. Al contrario de las bases de datos relacionales donde se inyectas claves primarias o secundarias en las tablas, las claves forman parte del framework de la base de datos. Y las relaciones entre entidades son externas, es decir, no realiza modificaciones en las entidades.

Podemos considerar las entidades como verbos o acciones que realiza las entidades entre ellas. La siguiente imagen es un ejemplo de lo mostrado, dos entidades persona realizan una acción que es conocerse.

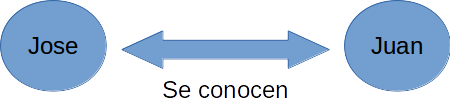


Ilustración (Neo4j: Ejemplo entidad-relación)

Las entidades tienen una dirección es decir una flecha, que indica que referencia a que. A continuación, listaremos las relaciones entre entidades del modelo 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 – Neo4j: Estructura general de datos

En la siguiente figura describiremos la estructura general de datos del sistema. Veremos un ejemplo de entidades y relaciones individuales, ya que si generamos un escenario complicado es casi indescifrable.

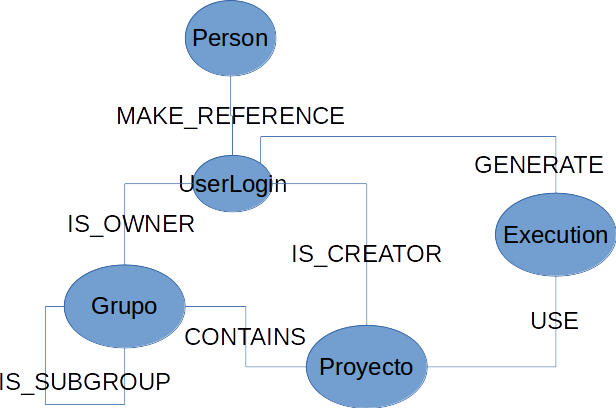


Ilustración (Estructura general de datos)

# Diseño de sistema: Subsistema repositorio

El Subsistema repositorio es un sistema con dos objetivos principales:

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

El sistema repositorio estará dividido en varias capas. Estas capas harán funciones independientes y todas estarán implementadas en Java. Java es un lenguaje orientado a objetos. Cada capa se comunicará con la otra mediante interfaces que proporcionaran servicios compatibles para cada capa.

Algunas capas del sistema repositorio utilizaran tecnologías y frameworks que facilitaran realizar los objetivos de estas capas. Estos frameworks o tecnologías son de carácter público, pudiéndose utilizar gratis. 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 un acceso a la base de datos.
* Capa de ejecución: Proporciona servicios de ejecución de códigos.

### DS-Subsistema repositorio Web: Capa REST

La capa REST proporciona servicios de comunicación entre el sistema y los diversos clientes del repositorio. Para realizar este objetivo la capa utiliza la arquitectura REST. Esta arquitectura fue creada como un método de comunicación cliente-servidor estándar para diversos clientes, utilizando para ello el protocolo HTTP.

**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:

Como se ha explicado en el punto 3.1 “Introducción”. La arquitectura REST es una arquitectura de transporte de recursos (objetos o entidades) mediante el protocolo HTTP. Esta arquitectura utiliza servidores web que utilizan el mapeado URL de las páginas de los dominios web, la diferencia es, en vez de devolver un mensaje HTTP con el cuerpo de una página web (HTML). Es enviado una entidad o recurso en un lenguaje unificado de comunicación con (JSON O XML) construido en String.

Podríamos realizar a mano esta capa, generaríamos un servidor que atendiera a peticiones HTTP en cierto puerto. Que analizara las peticiones HTTP y que las interpretara, enlazando cada tipo de petición con una clase que gestionara el servicio. Pero esto alejaría mucho el objetivo del TFG y supera con crecer el ámbito de este trabajo.

Para facilitar el trabajo de aplicaciones web existen framework o librerías, que proporcionan códigos para facilitar la tarea de servicios web sin tener que crear servidores web especializados. Hay muchas tecnologías que abarcan un sinfín de problemas.



Ilustración (Logo Spring)

Una de ellas es Spring. Spring es un framework web basado en Java. Este framework está basado en una solución para los problemas de inyección de dependencia. Más adelante evoluciono facilitando soluciones de servicios web como modelos MVC o REST además de un conjunto más amplio de soluciones.

Spring REST se ocupa del mapeado de las peticiones HTTP enlazando las peticiones con clases contenedoras de métodos que gestionan las peticiones llamadas controladores. También se ocupa de la obtención de los datos recibidos y del procesado de datos para ser enviados.

Como REST es una arquitectura basada en recursos, crearemos 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 o borrar un grupo.

#### 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 session. Se ha optado por la implementación de un sistema propio, buscando la simplicidad y la rapidez.

Cada usuario que se conecte al sistema se le asociará una sesión, La cual será gestionada por la capa REST. La capa 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). A continuación, se describe el diagrama de iteración del proceso de autenticación y sesión.

PONER AQUÍ DIAGRAMA DE SECUENCIA

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 sesion.

#### Protocolo o diagrama de iteración general:

Para la obtención de cada recurso se tendrá que seguir una iteración general. Esta iteración siempre será la misma y no variará por diferentes recursos. Si se quiere saber qué tipos de recursos existen y como obtenerlos se recomienda leer el manual del usuario donde se indicarán todos los recursos.

PONER AQUÍ DIAGRAMA DE SECUENCIA

## DS-Subsistema repositorio Web: 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, los cuales serán implementados utilizando el tipo singleton para que su propia existencia sea única.

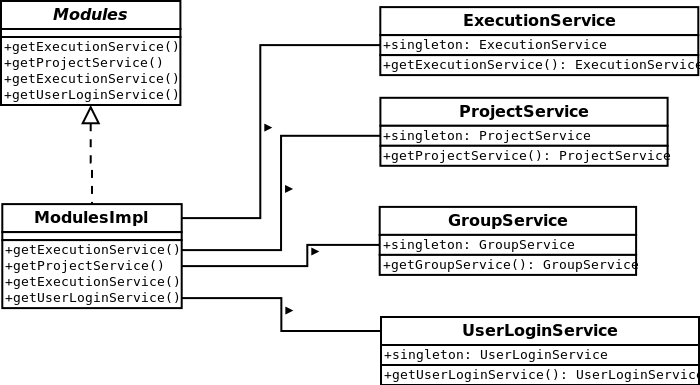


Ilustración Esquema Servicios Negocio

## DS- Subsistema repositorio Web: Capa de datos

El objetivo de la capa de datos es el de ser intermediario entre las demás capas y la base de datos. Esta capa traduce los datos de la base de datos en entidades del modelo e introduce datos nuevos en la base de datos. Para desarrollar esta capa se creará 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.

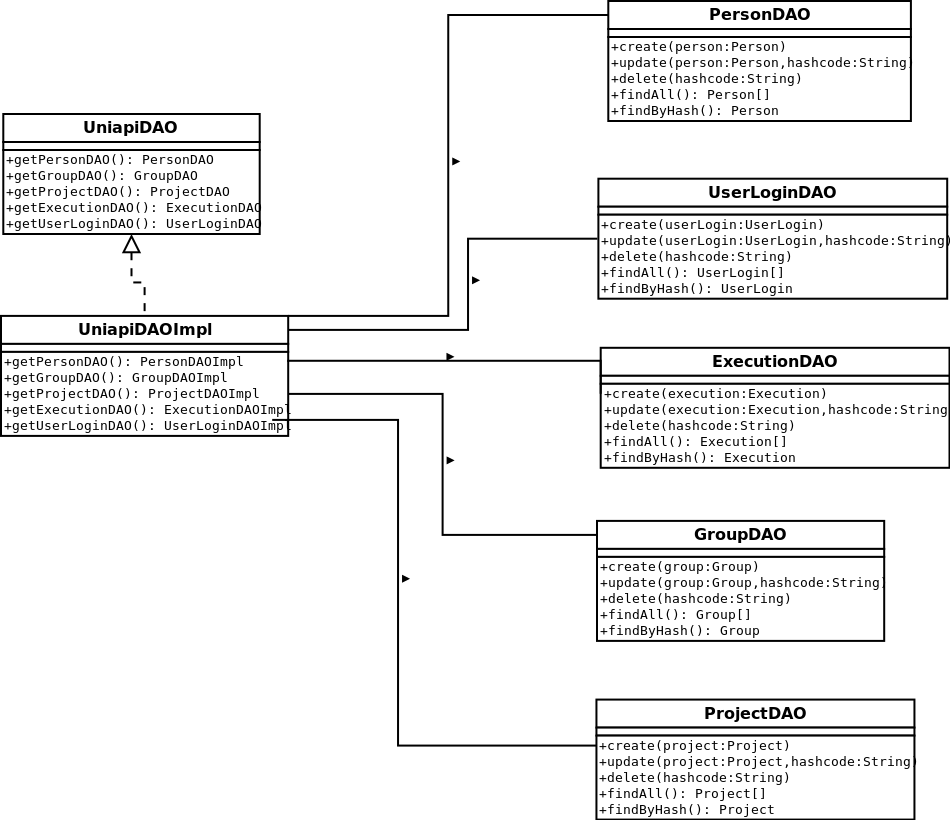


Ilustración Esquema DAO Entidades

Neo4j tiene a parte de las entidades relaciones entre ellas es por eso que necesitamos crear clases para la gestión de las relaciones.

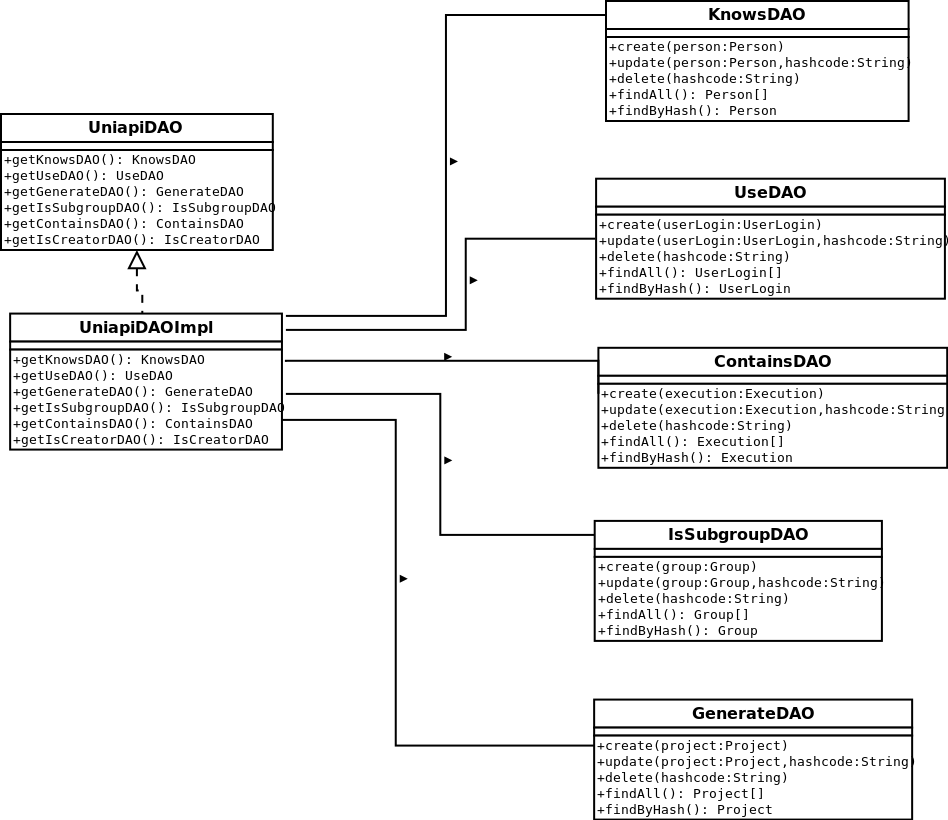


Ilustración Esquema DAO Relaciones

## DS-Subsistema repositorio: 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 está totalmente implementada en java. La capa ejecución se comunica directamente con los Sistemas operativos y con los repositorios GIT. Para poder comunicarse con estos sistemas la capa utilizara librerías internas de java y externas.

### Sincronización de proyectos:

La capa de ejecución creara una jerarquía de proyectos en el sistema de archivos del sistema operativo, donde usando una librería externa, **JGIT**, actualizara y descargara los datos de los proyectos. Cada proyecto que se vaya a crear se generara un espacio personal donde se descargara la información. También se creará un sistema de monitorización de los proyectos actualizándose cada vez que se genera un cambio.

**JGit** es una Liberia 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 proyectos para comprobar los estados de los mismos, pudiendo parar las ejecuciones cuando desee el usuario.

### Respuestas de ejecuciones:

La capa de ejecución tratara los datos de las ejecuciones de los proyectos. Tratando los datos y comunicándose con las demás capas con el objetivo de comunicárselo a los usuarios que los soliciten.

# Diseño de sistema: Subsistema aplicación web

El subsistema aplicación web tiene varios objetivos del sistema general:

* **Comunicación con el usuario**: El subsistema tendrá que proporcionar los medios necesarios para facilitar la comunicación entre el usuario y el repositorio.
* **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. Pudiendo utilizarse en cualquier programa que no sea el propio subsistema de 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. El modelo - vista - controlador es un modelo WEB para la realización de aplicaciones web. Las aplicaciones web contienen dos partes o lados. Cada lado está separado por su el lugar del programa (“localización” lógica), estos son los elementos:

* **Lado servidor (backend):** El lado servidor es todo código que se ejecuta en el computador que contiene el servidor web. Los servidores suelen contener toda la información y códigos que luego serán enviados a los clientes cuando estos los solicitan.
* **Lado cliente (frontend):** El lado cliente son los navegadores web de los clientes que solicitan los servicios web. Todo código dentro del navegador o recursos multimedia que interactúan con el cliente es el lado servidor.

El modelo MVC es un modelo de arquitectura web que tiene en cuenta los dos lados gestionando la comunicación entre ellos mediante el HTTP. Aunque podríamos realizar un servidor web y una funcionalidad que fabriquen páginas web personalizadas en función de los datos computados por el servidor. Existen muchos frameworks que facilitan la creación de estos sistemas.

## DS- Subsistema aplicación web: JavaScript y el peso del cómputo

Desde que nació JavaScript (JS), un lenguaje funcional que podía ser utilizado por los navegadores web. Los navegadores web empezaron a hacer cómputos de bajo calibre hasta llegar a cómputos complejos. Antes de la existencia de JS los modelos MVC, como fue explicado en la introducción, todo el peso de la computación del modelo era generado en el servidor. Era el servidor el que leía los eventos, los trataba y modificaba las páginas para enviársela al cliente. Todo esto conlleva un coste de red (a veces limitado como en los móviles) y de cómputo en el servidor (teniendo miles o millones de clientes).

Con la llegada de JS llego la creación de modelos MVC que se computaban en el cliente. Estos modelos atraen los eventos y son tratados dentro del cliente, sin tener que llamar al servidor, salvo en casos muy generales, que necesite un cambio de información importante y será necesario la llamada al servidor que será el que provea la información.

¿Entonces que utilizar? ¿Todo el computo en el servidor? ¿Todo el computo en el cliente? ¿Mixto?

En este TFG utilizaremos un sistema de todo el cómputo en el cliente. Para poder demostrar que en aplicaciones de clientes son igual de potentes que las tradicionales, donde todo el peso lo lleva el servidor. En este TFG todo el peso de cómputo en lo referente a IU de usuarios será llevado por el cliente. Siendo el servidor un mero repartidor de elementos y recursos web. Las razones de esta decisión serán explicadas en la memoria, donde se defenderá las razones del uso de esta decisión.

Cabe destacar que para la realización de este subsistema utilizaremos JavaScript. Ya que es este lenguaje el que entienden los clientes web y es el lenguaje en el que están hechos los diversos frameworks.

## DS – Subsistema aplicación web: Lado servidor

Como se explica en el punto 5.1 el objetivo del servidor será el proporcionar y enviar por el medio de las tecnologías WEB, HTTP. HTTP es un protocolo de transporte de información genérico, que puede ser usado por cualquier lenguaje. Podremos crear el servidor web en cualquier lenguaje. Siempre que el lenguaje utilizado pueda enviar peticiones HTTP e interpretar las peticiones HTTP recibidas por el cliente.

Utilizaremos framework básicos que proporcionan los lenguajes más utilizados por los usuarios para realizar el servidor web. Aunque podríamos usar el mismo lenguaje que el repositorio web (Java), los servicios que provee Java son complejos y llevan más tiempo gestionarlos que otros lenguajes como python o JavaScript, cuyos servidores web son muy sencillos y fáciles de usar.

Es por eso que haremos un servidor web en JavaScript. La parte de servidores se denomina Node.js . Node.js no es más que un compilador de navegador web pero externalizado para ejecución en un servidor. Node.js tiene una librería propia denominada $http() esta librería gestiona toda petición entrante del protocolo HTTP. Para la interpretación de las peticiones y la realización de servidores web se utiliza el framework express.js.



### Gestión de recursos:

El objetivo principal del servidor será el gestionar los recursos. Para ello el servidor web creara una jerarquía y un espacio dominio http, para la petición de los recursos por parte del cliente. Cuando el cliente pida un recurso X en el dominio HTTP. Este dominio será interpretado por el servidor, obtenido el recurso y enviado al cliente.

### Creación de cuentas:

Para una mejor seguridad en el sistema, solo las cuentas enviadas al sistema mediante el servidor web al repositorio serán gestionadas. El servidor deberá proveer un controlador y un modelo para la gestión de la creación de cuentas. El servidor deberá tratar los eventos de la creación de la cuenta. Tratar la cuenta y enviarla al repositorio. También deberá esperar a la respuesta, tratarla e enviarla.

## DS-Subsistema aplicación web: Lado cliente

El objetivo del lado del cliente es obtener todos los eventos que interactúa el cliente. Comprenderlos, analizarlos y dar una respuesta coherente al evento. El lado del cliente como se ha explicado en el punto 5.1 contendrá el total del cómputo de la aplicación. Aunque todo el cómputo de la aplicación lo lleva el cliente, se seguirá usando el modelo MVC. Ya que es el predominante en aplicaciones web, sea la ejecución en el cliente o en el servidor.

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

Con la evolución del JavaScript nacieron frameworks que facilitaban la ejecución de cómputos en el frontend (lado cliente), uno de ellos es Angular.js. Angular utiliza un modelo MVC en el cliente, con apoyo mediante AJAX a un servidor backend que le despliega un chorro de datos no elaborados para su muestra al usuario.

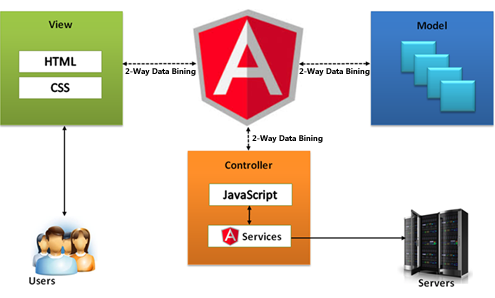


Ilustración Arquitectura Angular.js MVC

Como se puede ver en la imagen 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. Esto no varía mucho de los MVC tradicionales, la mejora de Angular.js frente a los demás controladores convencionales son las directivas, las cuales, son elementos HTML que llevan enlazado una estructura MVC fija.
* **Modelo**: los modelos rellenan muestran la interfaz de usuario en función de los datos entregados por el controlador. En angular no se difiere mucho de los modelos tradicionales MVC, salvo la mejora de que el controlador pueda 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).

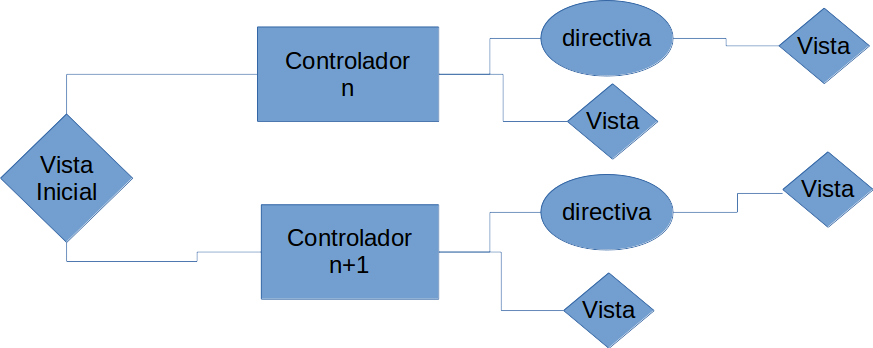


Ilustración Arquitectura básica de app cliente Angular.js

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 una directiva la cual a su vez está cargada por un conjunto de controladores, modelos y vistas, siguiendo un árbol en profundidad de directivas con controladores...

Siguiendo estas directrices crearemos para cada elemento de los menús un controlador con una vista asignada (prototipos de IU del análisis), y para cada funcionalidad individual de los menús crearemos directivas que se gestionen con controladores internos.

#### API de comunicación:

Los modelos MVC albergados en los clientes se nutren de información REST de servicios en servidores web. Todos ellos tienen unos protocolos de comunicación y autenticación. Si queremos realizar una comunicación con un servidor puede volverse muy repetitivo. Es por ello que vamos a generar una librería que facilite la comunicación con los servidores. Esto es lo que se denomina como API una librería externa que te facilita la comunicación con puntos de información.

Angular facilita mucho al modelo MVC la generación de API 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 se mantiene en funcionamiento. Todos los componentes utilizaran la clase servicio en cuestión para obtener la información.

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