2021

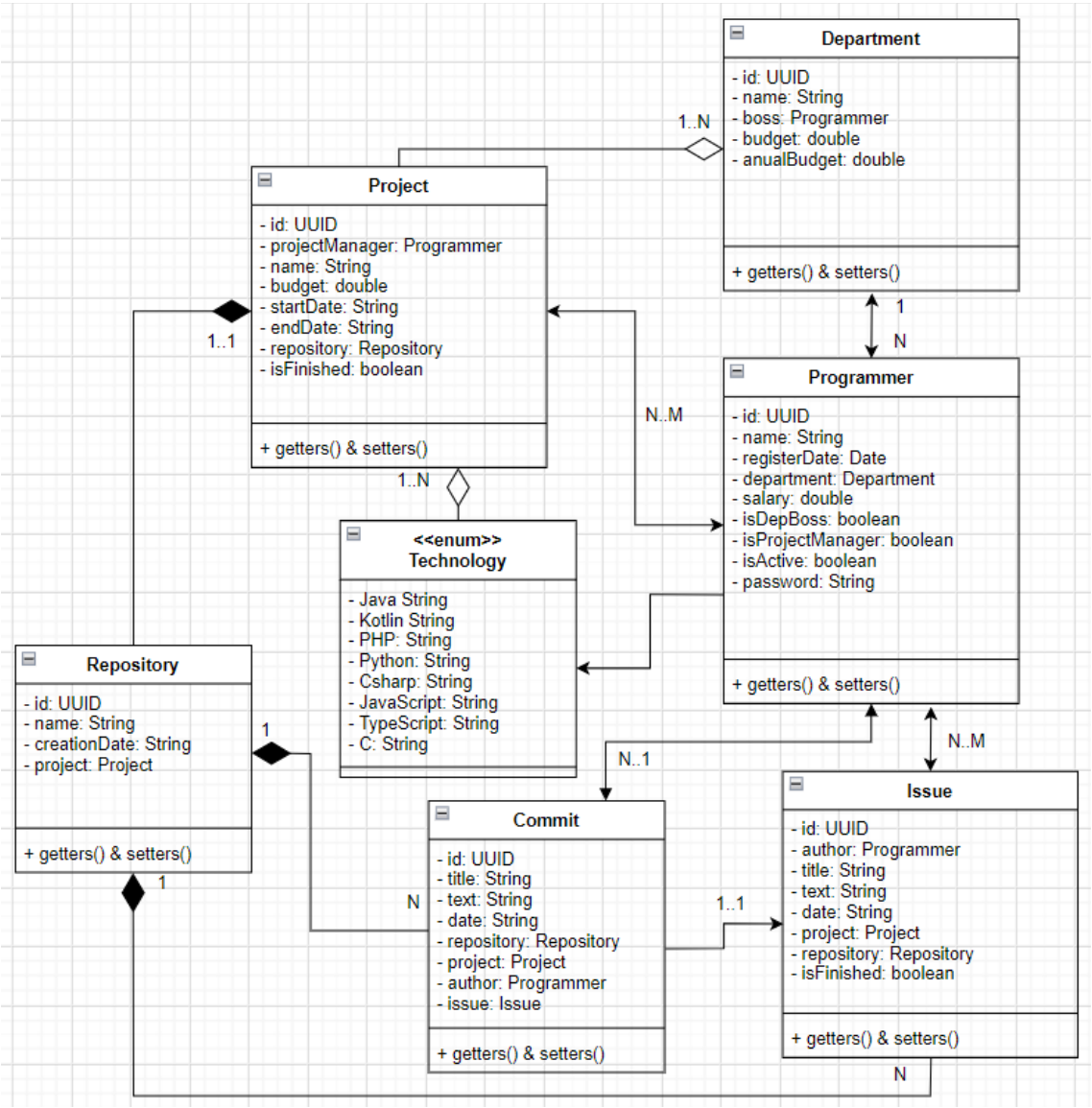
Recuperación JPA

ACCESO A DATOS

DANIEL RODRIGUEZ MUÑOZ – JAIME SALCEDO VALLEJO

Diagrama de Clases:

Imagen:



Explicación:

Department:

Consiste en una clase con un id, nombre, un programador jefe, un presupuesto, un presupuesto anual y, como indican las relaciones, una lista de proyectos (realmente son dos listas, pues luego se subdividen en terminados y no terminados, y sólo el presupuesto de los no terminados cuenta para el presupuesto anual) y un histórico de jefes (programadores).

Su relación con Programmer es bidireccional porque los programadores pertenecen a un departamento (tienen un departamento), y el departamento, si bien no tiene una lista de todos sus programadores, sí la tiene de sus jefes, que son programadores.

Su relación con Project es una agregación, ya que a un departamento se le asignan proyectos, pero si el departamento fuese eliminado (por recortes en la empresa, por ejemplo), no tendría sentido eliminar los proyectos, pues estos se pueden asignar a otro departamento. Como queremos que, además de tener departamento una lista de proyectos terminados, proyecto tenga asignado un departamento (porque viva complicarse la vida innecesariamente, así practicamos), implementaremos bidireccional en esta relación.

Project:

Consiste en una clase con un id, un jefe de proyecto (programador), nombre, presupuesto, fecha de inicio, fecha de fin (si es que ha finalizado), un booleano que dicta si está terminado o no y una lista de programadores. Además, a este proyecto se le agregan tecnologías (agregadas porque las tecnologías siguen existiendo si el proyecto desaparece) y está compuesto por un y solo un repositorio (compuesto porque si desaparece el proyecto, no tiene sentido que el repositorio siga en pie).

Su relación con programador es bidireccional: Por una parte, el proyecto cuenta no solo con un jefe de proyecto, sino también con una serie de programadores (o no, pues por ejemplo si el proyecto está terminado y descontinuado, sus programadores pueden ser desasignados del proyecto si este no requisiese mantenimiento), y, por otro lado, los programadores a su vez tienen asignada una lista de proyectos.

Su relación con repository es, como ya dijimos anteriormente, una composición, pues si el proyecto es destruido, no nos interesa conservar el repositorio. Esta relación es bidireccional, por lo que tendrá proyecto un objeto repositorio y repositorio un objeto proyecto.

Repository:

Se trata de una clase con id, nombre, fecha de creación y un proyecto asociado, así como está compuesto de una lista de issues y commits.

Su relación con commit es una composición, puesto, que si borramos el repositorio, no tiene ningún sentido que se conserven los commits, y es bidireccional, por lo que commit tendrá un objeto repository y repository una lista de commits. Exactamente lo mismo aplica para issue.

Commit:

Es una clase con id, titulo, texto, un autor (el cual es un programador), una fecha, un repositorio y un proyecto al que pertenece (innecesario, pues ya tiene el repositorio), así como una issue asignada.

Un commit puede tener un solo autor, pero un mismo programador puede tener múltiples commits, por lo que la relación entre estas dos clases es 1 a muchos.

Un commit pertenece a una issue, y en el momento en que la issue recibe un commit, se cierra, haciendo que sea una relación 1 a 1.

Un commit solo pertenece a un repositorio, pero ese repositorio puede tener múltiples commits, y además si el repositorio desaparece, los commits que tenga también lo harán, por lo que es una composición (un repositorio está compuesto por commits).

Issue:

Clase con id, autor(jefe de proyecto), titulo, fecha, proyecto asociado (innecesario), repositorio asociado y un booleano de si esta terminada o no, así como una lista de programadores implicados.

Una issue solo pertenece a un repositorio, pero ese repositorio puede tener múltiples issues, y además si el repositorio desaparece, las issues que tenga también lo harán, por lo que es una composición (un repositorio está compuesto por issues).

Su relación con programmer es muchos a muchos puesto que una issue tiene una lista de programadores y estos pueden tener asignadas múltiples issues a la vez.

Programmer:

Clase con id, nombre, fecha de registro, departamento, salario, lista de proyectos activos, commits, issues y tecnologías que usa, así como una contraseña y booleanos para saber si está activo en la empresa, es jefe de departamento o es jefe de proyecto.

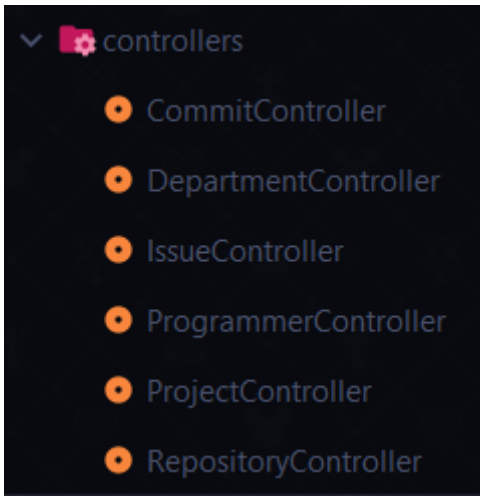
Sus relaciones han sido descritas con anterioridad en los otros elementos del diagrama.

Paquetes:

Hablaremos de los paquetes en general porque la mayoría de las clases son lo mismo cambiando un par de detalles.

Controllers:

Explicación:



Los controladores son aquellos que se encargan de que si, por ejemplo, quieres insertar un departamento, llamen a las clases necesarias mediante la delegación a la correspondiente clase service para llevar a cabo esa acción y, una vez llevada a cabo, te muestren el resultado de dicha operación en el formato que le hayas pedido (xml o json)

Un [x]Controller es capaz de devolver en json o xml las operaciones CRUD (create, find all, find by id, update, delete) de [x]

Código de uno de ellos:

```
Esta clase se encarga de llamar al DepartmentService y mostrar lo que reciba como xml o json
Author: Daniel Rodriguez
object DepartmentController {
    private val service = DepartmentService()

    Este metodo muestra todos los department como xml o json dependiendo del parámetro que le pases
    Params: returnMode - String
    Returns: String
    Author:  Daniel Rodriguez
    fun findAllDepartments(returnMode: String): String {
        return when (returnMode.toUpperCase()) {
            "XML" -> findAllDepartmentsXML()
            "JSON" -> findAllDepartmentsJSON()
            else -> throw Exception("Invalid parameter. Admitted parameters: XML , JSON")
        }

    }
```

```
    Este metodo coge todos los department y las muestra como xml
    Returns: String xml
    Author:  Daniel Rodriguez
    private fun findAllDepartmentsXML(): String {
        val list = service.getAllDepartments()
        list.forEach { x -> Jaxb.departmentToXML(x) }
        return ""
    }

    Este metodo coge todos los department y las muestra como json
    Returns: String json
    Author:  Daniel Rodriguez
    private fun findAllDepartmentsJSON(): String {
        return GsonBuilder().setPrettyPrinting().create()
                .toJson(service.getAllDepartments())
                ?: throw SQLException("Error at DepartmentController.findAllDepartmentsJSON")
    }
```

```
Este metodo coge todos los department dependiendo de un id que le pasemos y las muestra en xml o json en
función del parámetro pasado
Params: id - String
        returnMode - String
Returns: String
Author: Daniel Rodriguez
fun getDepartmentById(id: String, returnMode: String): String {
    return when (returnMode.toUpperCase()) {
        "XML" → getDepartmentByIdXML(id)
        "JSON" → getDepartmentByIdJSON(id)
        else → throw Exception("Invalid parameter. Admitted parameters: XML , JSON")
    }
}

Este metodo coge todos los department dependiendo de un id que le pasemos y las muestra como xml
Params: id - String
Returns: String xml
Author: Daniel Rodriguez
private fun getDepartmentByIdXML(id: String): String {
    val res = service.getDepartmentById(id)
    Jaxb.departmentToXML(res)
    return ""
}
```

```
Este metodo coge todos los department dependiendo de un id que le pasemos y las muestra como json
Params: id - String
Returns: String json
Author: Daniel Rodriguez
private fun getDepartmentByIdJSON(id: String): String {
    return GsonBuilder().setPrettyPrinting().create()
            .toJson(service.getDepartmentById(id))
            ?: throw SQLException("Error at DepartmentController.getDepartmentByIdJSON:" +
                    " Department with id $id not found.")
}

Este metodo inserta un department y lo muestra como xml o json, dependiendo del parámetro que le pasemos
Params: dep - DepartmentDTO
        returnMode - String
Returns: String
Author: Daniel Rodriguez
fun insertDepartment(dep: DepartmentDTO, returnMode: String): String {
    return when (returnMode.toUpperCase()) {
        "XML" -> insertDepartmentXML(dep)
        "JSON" -> insertDepartmentJSON(dep)
        else -> throw Exception("Invalid parameter. Admitted parameters: XML , JSON")
    }
}
```

```
Este metodo inserta un department y lo muestra como xml
Params: dep - DepartmentDTO
Returns: String xml
Author: Daniel Rodriguez
private fun insertDepartmentXML(dep: DepartmentDTO): String {
    val res = service.createDepartment(dep)
    Jaxb.departmentToXML(res)
    return ""
}

Este metodo inserta un department y lo muestra como json
Params: dep - DepartmentDTO
Returns: String json
Author: Daniel Rodriguez
private fun insertDepartmentJSON(dep: DepartmentDTO): String {
    return GsonBuilder().setPrettyPrinting().create()
            .toJson(service.createDepartment(dep))
            ?: throw SQLException("Error at DepartmentController.insertDepartmentJSON:" +
                    " Could not insert Department with id ${dep.id}")

}
```

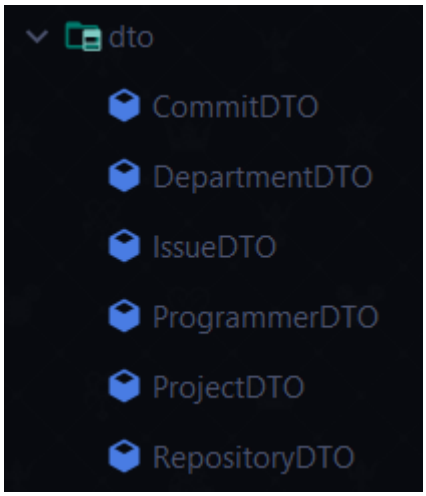
```
Este metodo actualiza un department y lo muestra como xml o json, dependiendo del parámetro que le pasemos
Params: dep - DepartmentDTO
        returnMode - String
Returns: String
Author:  Daniel Rodriguez
fun updateDepartment(dep: DepartmentDTO, returnMode: String): String {
    return when (returnMode.toUpperCase()) {
        "XML" -> updateDepartmentXML(dep)
        "JSON" -> updateDepartmentJSON(dep)
        else -> throw Exception("Invalid parameter. Admitted parameters: XML , JSON")
    }
}

Este metodo actualiza un department y lo muestra como xml
Params: dep - DepartmentDTO
Returns: String xml
Author:  Daniel Rodriguez
private fun updateDepartmentXML(dep: DepartmentDTO): String {
    val res = service.updateDepartment(dep)
    Jaxb.departmentToXML(res)
    return ""
}
```

```
Este metodo inserta un department y lo muestra como json
Params: dep - DepartmentDTO
Returns: String json
Author: Daniel Rodriguez
private fun updateDepartmentJSON(dep: DepartmentDTO): String {
    return GsonBuilder().setPrettyPrinting().create()
            .toJson(service.updateDepartment(dep))
            ?: throw SQLException("Error at DepartmentController.updateDepartmentJSON:" +
                    " Could not update Department with id ${dep.id}")

}

Este metodo elimina un department y lo muestra como xml o json, dependiendo del parámetro que le pasemos
Params: dep - DepartmentDTO
        returnMode - String
Returns: String
Author: Daniel Rodriguez
fun deleteDepartment(dep: DepartmentDTO, returnMode: String): String {
    return when (returnMode.toUpperCase()) {
        "XML" -> deleteDepartmentXML(dep)
        "JSON" -> deleteDepartmentJSON(dep)
        else -> throw Exception("Invalid parameter. Admitted parameters: XML , JSON")
    }
}
```

```
Este metodo elimina un department y lo muestra como xml
Params: dep - DepartmentDTO
Returns: String xml
Author: Daniel Rodriguez
private fun deleteDepartmentXML(dep: DepartmentDTO): String {
    val res = service.deleteDepartment(dep)
    Jaxb.departmentToXML(res)
    return ""
}

Este metodo elimina un department y lo muestra como json
Params: dep - DepartmentDTO
Returns: String json
Author: Daniel Rodriguez
private fun deleteDepartmentJSON(dep: DepartmentDTO): String {
    return GsonBuilder().setPrettyPrinting().create()
            .toJson(service.deleteDepartment(dep))
            ?: throw SQLException("Error at DepartmentController.deleteDepartmentJSON:" +
                    " Could not delete Department with id ${dep.id}")
}
```

Dto:

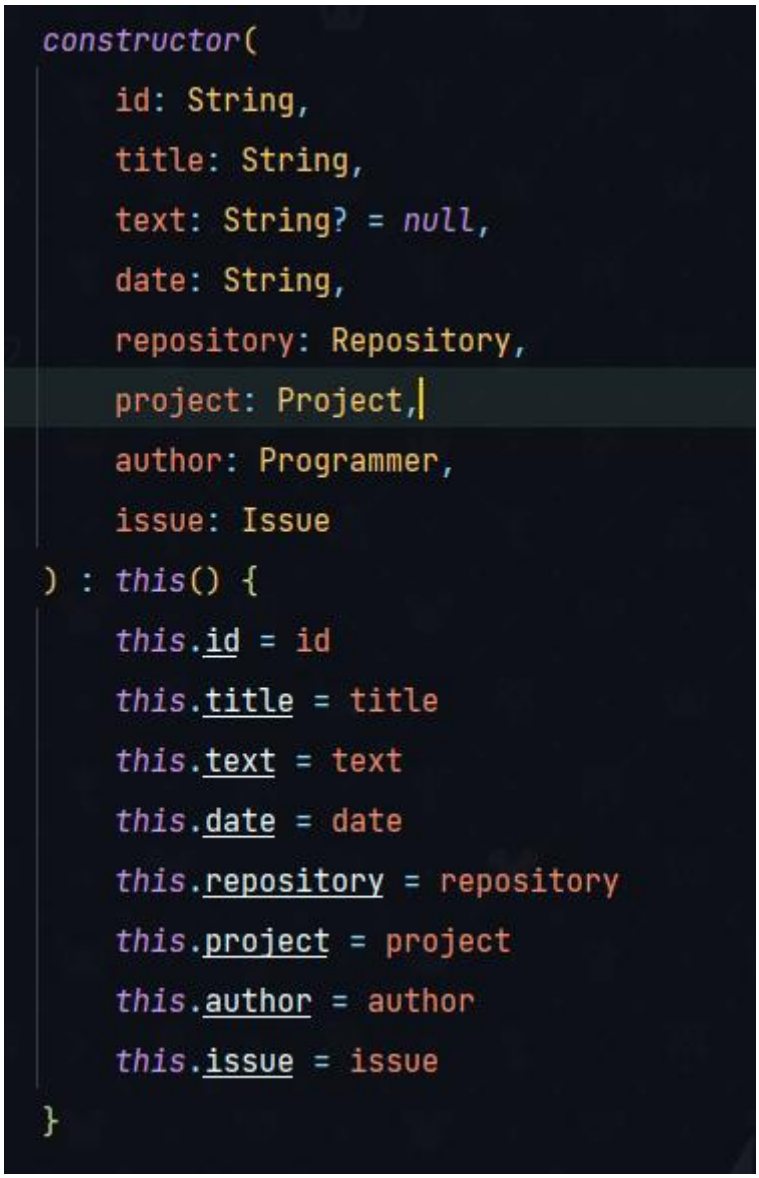
Explicación:



Los DTO(data transfer object) son los objetos que usaremos para todo lo que no sea interactuar con la base de datos (para eso tenemos model). Por ejemplo, para mostrar los json o los xml.

Codigo de uno de ellos:

```
Data transfer Object de Commit. Preparado para ser sacado en formato XML y JSON.
Author: Daniel Rodríguez
@XmlRootElement(name = "commit")
@XmlAccessorType(XmlAccessType.FIELD)
class CommitDTO() {
    @XmlAttribute
    lateinit var id: String

    @XmlAttribute
    lateinit var title: String
    var text: String? = null

    @XmlAttribute
    lateinit var date: String
    lateinit var repository: Repository
    lateinit var project: Project
    lateinit var author: Programmer
    lateinit var issue: Issue
```

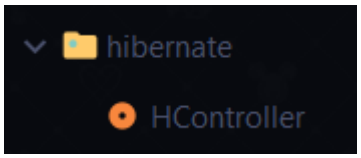


```
constructor(
    id: String,
    title: String,
    text: String? = null,
    date: String,
    repository: Repository,
    project: Project,
    author: Programmer,
    issue: Issue
) : this() {
    this.id = id
    this.title = title
    this.text = text
    this.date = date
    this.repository = repository
    this.project = project
    this.author = author
    this.issue = issue
}
```

```
    De un string JSON lo convierte en un CommitDTO
    Params: json - String
    Returns: CommitDTO
    Author: Daniel Rodríguez
fun fromJSON(json: String): CommitDTO? {
    return Gson().fromJson(json, CommitDTO::class.java)
}

    De un CommitDTO lo convierte en un string JSON
    Returns: String
    Author: Daniel Rodríguez
fun toJSON(): String {
    return GsonBuilder().setPrettyPrinting().create().toJson( src: this)
}
}
```

Hibernate:

Explicación:



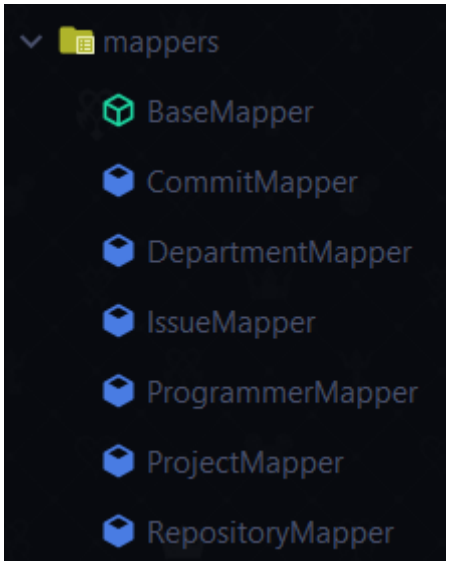
El controlador de hibernate. Es un singleton con el cual nos aseguramos de siempre estar llamando al mismo controlador.

Codigo:

```
object HController {
    private val eManagerFact = Persistence.createEntityManagerFactory( persistenceUnitName: "default")
    val manager: EntityManager = eManagerFact.createEntityManager()
    val transaction : EntityTransaction = manager.transaction
}
```

Mappers:

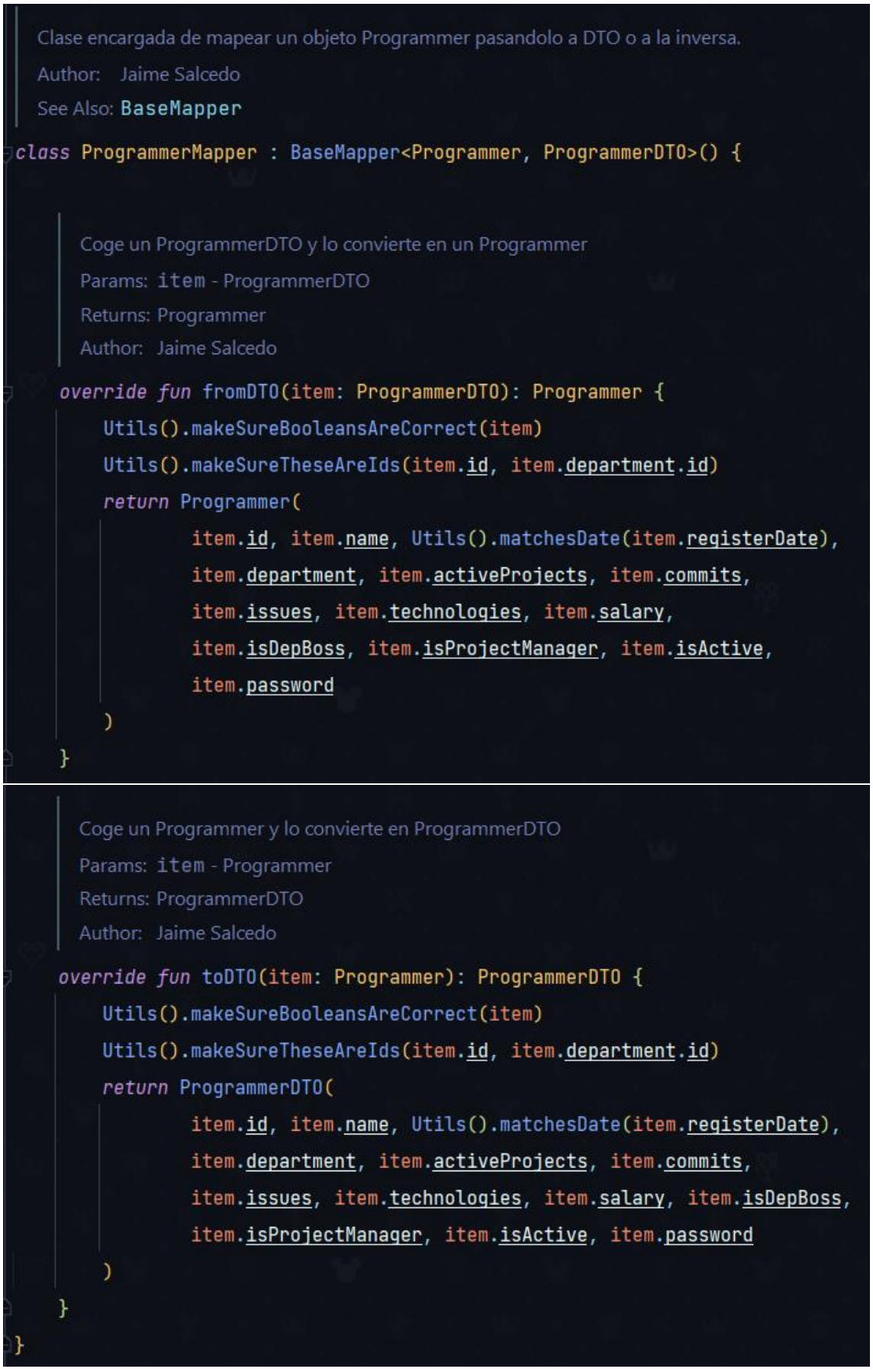
Explicación:



Se encargan de pasar de DTO a modelo y de modelo a DTO.

Código de uno de ellos (y de la clase abstracta):

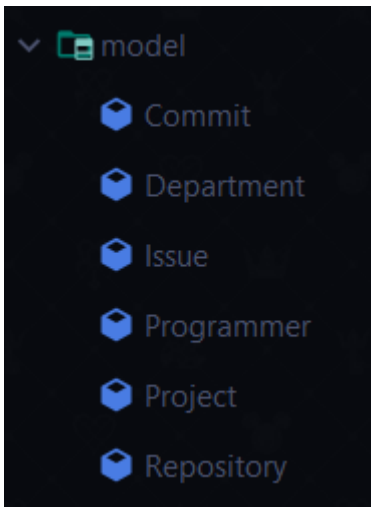

Clase abstracta que hace de esqueleto para los mapeadores.
Author: Daniel Rodriguez Muñoz
abstract class BaseMapper<T, DTO> {
fun fromDTO(items: List<DTO>): List<T> {
return items.map { fromDTO(it) }
}
fun toDTO(items: List<T>): List<DTO> {
return items.map { toDTO(it) }
}
abstract fun fromDTO(item: DTO): T
abstract fun toDTO(item: T): DTO
}



```
Clase encargada de mapear un objeto Programmer pasandolo a DTO o a la inversa.
Author:  Jaime Salcedo
See Also: BaseMapper
class ProgrammerMapper : BaseMapper<Programmer, ProgrammerDTO>() {

    Coge un ProgrammerDTO y lo convierte en un Programmer
    Params: item - ProgrammerDTO
    Returns: Programmer
    Author:  Jaime Salcedo
    override fun fromDTO(item: ProgrammerDTO): Programmer {
        Utils().makeSureBooleansAreCorrect(item)
        Utils().makeSureTheseAreIds(item.id, item.department.id)
        return Programmer(
                item.id, item.name, Utils().matchesDate(item.registerDate),
                item.department, item.activeProjects, item.commits,
                item.issues, item.technologies, item.salary,
                item.isDepBoss, item.isProjectManager, item.isActive,
                item.password
        )
    }

    Coge un Programmer y lo convierte en ProgrammerDTO
    Params: item - Programmer
    Returns: ProgrammerDTO
    Author:  Jaime Salcedo
    override fun toDTO(item: Programmer): ProgrammerDTO {
        Utils().makeSureBooleansAreCorrect(item)
        Utils().makeSureTheseAreIds(item.id, item.department.id)
        return ProgrammerDTO(
                item.id, item.name, Utils().matchesDate(item.registerDate),
                item.department, item.activeProjects, item.commits,
                item.issues, item.technologies, item.salary, item.isDepBoss,
                item.isProjectManager, item.isActive, item.password
        )
    }
}
```

Model:

Explicación:



Las clases que interactuarán con la base de datos. Gracias a JPA, estas pueden tener objetos y listas, pues JPA se encarga por debajo de pasar todo ello a ids y lo que considere necesario con tal de que case con la base de datos.

Código de uno de ellos:

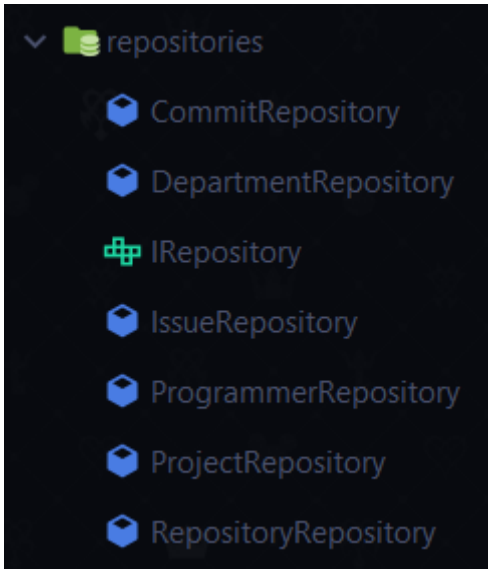
```
@Entity
@Table(name = "Department")
@NamedQuery(name = "Department.findAll", query = "SELECT r FROM model.Department r")
class Department() {
    @Id
    @Column(name = "id", nullable = false)
    lateinit var id: String
    @Basic
    @Column(name = "name", nullable = false, length = 255)
    lateinit var name: String
    @OneToOne
    lateinit var boss: Programmer
    @Basic
    @Column(name = "budget", nullable = false)
    var budget: Double = 0.0
    @OneToMany(fetch = FetchType.EAGER, mappedBy = "Department", cascade = arrayOf(CascadeType.REMOVE))
    var finishedProjects: List<Project>? = null
    @OneToMany(fetch = FetchType.EAGER, mappedBy = "Department", cascade = arrayOf(CascadeType.REMOVE))
    var developingProjects: List<Project>? = null
    @Basic
    @Column(name = "anualBudget", nullable = false)
    var anualBudget: Double = 0.0
    @OneToMany(fetch = FetchType.EAGER, mappedBy = "Department", cascade = arrayOf(CascadeType.REMOVE))
    lateinit var bossHistory: List<Programmer>
```

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "Department", cascade = arrayOf(CascadeType.REMOVE))
lateinit var bossHistory: List<Programmer>

constructor(
    id: String,
    name: String,
    boss: Programmer,
    budget: Double,
    finishedProjects: List<Project>?,
    developingProjects: List<Project>?,
    anualBudget: Double,
    bossHistory: List<Programmer>
) : this() {
    this.id = id
    this.name = name
    this.boss = boss
    this.budget = budget
    this.finishedProjects = finishedProjects
    this.developingProjects = developingProjects
    this.anualBudget = anualBudget
    this.bossHistory = bossHistory
}
```

```
    override fun toString(): String {
        return ("Department:{ id:$id, name:$name, boss:${boss.id}, budget:$budget, " +
                "finishedProjects:${finishedProjects?.forEach { "${it.id};" } ?: ""}, " +
                "developingProjects:${developingProjects?.forEach { "${it.id};" } ?: ""}, " +
                "anualBudget:$anualBudget, bossHistory:${bossHistory.forEach { "${it.id};" }} }")
    }
}
```

Repositories:

Explicación:



Los repositorios se encargarán de las operaciones CRUD (y cualquier otra operación adicional, si es que hemos creado una NamedQuery para dicha operación en la clase del modelo correspondiente). Gracias a JPA, el proceso de programar esto, pese a que sigue siendo repetitivo, al menos no hace que me den ganas de pegarme un tiro.

Código de uno de ellos (y de la interfaz):

```
Interfaz que obliga a las clases que la implementen a implementar las operaciones CRUD.
interface IRepository<T, ID> {
    fun findAll(): List<T>
    fun getById(id: ID): T
    fun insert(t: T): T
    fun update(t: T): T
    fun delete(t: T): T
}
Clase encargada de hacer las operaciones CRUD de Repository.
Author: Jaime Salcedo
See Also: IRepository
class ProjectRepository : IRepository<Project, String> {
    Encuentra todos los repositories presentes en la BD y los devuelve como una lista de objetos Project
    Returns: List
    Author: Jaime Salcedo
    override fun findAll(): List<Project> {
        try {
            val query = HController.manager
                .createNamedQuery(
                    "Project.findAll",
                    Project::class.java
                )
            return query.resultList
        } catch (e: Exception) {
            throw SQLException("Error at ProjectRepository.findAll")
        }
    }
```

```
Encuentra el project cuyo ID casa con el parámetro introducido y lo devuelve como un objeto Project, si lo
encuentra.
Params: id - String
Returns: Project
Author: Jaime Salcedo
override fun getById(id: String): Project {
    return HController.manager.find(Project::class.java, id) ?:
    throw SQLException("Error at ProjectRepository.getById: Project with id:$id does not exist.")
}
```

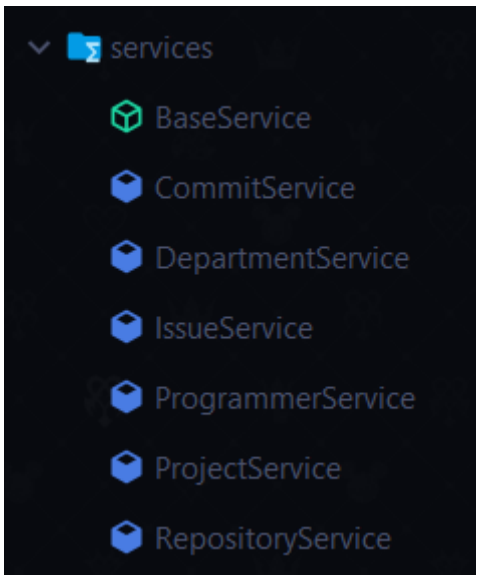
```
Inserta un project en la base de datos, donde cada atributo del project va a un campo de la tabla project,
devolviendo dicho project si lo consigue.
Params: project - Project
Returns: Project
Author: Jaime Salcedo
override fun insert(project: Project): Project {
    try {
        HController.transaction.begin()
        HController.manager.persist(project)
        HController.transaction.commit()
        return project
    } catch (e: Exception) {
        HController.transaction.rollback()
        throw SQLException("Error at ProjectRepository.insert: Could not insert Project into BD: ${e.message}")
    }
}
```

```
Modifica un project, si existe, devolviendo dicho project si lo consigue.
Params: project - Project
Returns: Project
Author: Jaime Salcedo
override fun update(project: Project): Project {
    try {
        HController.transaction.begin()
        HController.manager.merge(project)
        HController.transaction.commit()
        return project
    } catch (e: Exception) {
        HController.transaction.rollback()
        throw SQLException("Error at ProjectRepository.update: Could not update Project: ${e.message}")
    }
}
```

```
    Borra un project, si existe, devolviendo dicho project si lo consigue.
    Params: project - Project
    Returns: Project
    Author: Jaime Salcedo
override fun delete(project: Project): Project {
    try {
        val itemDelete = HController.manager.find(Project::class.java, project.id)
        HController.transaction.begin()
        HController.manager.remove(itemDelete)
        HController.transaction.commit()
        return itemDelete
    } catch (e: Exception) {
        HController.transaction.rollback()
        throw SQLException("Error at ProjectRepository.delete: Could not delete Project with id: ${project.id}: ${e.message}")
    }
}
}
```

Services:

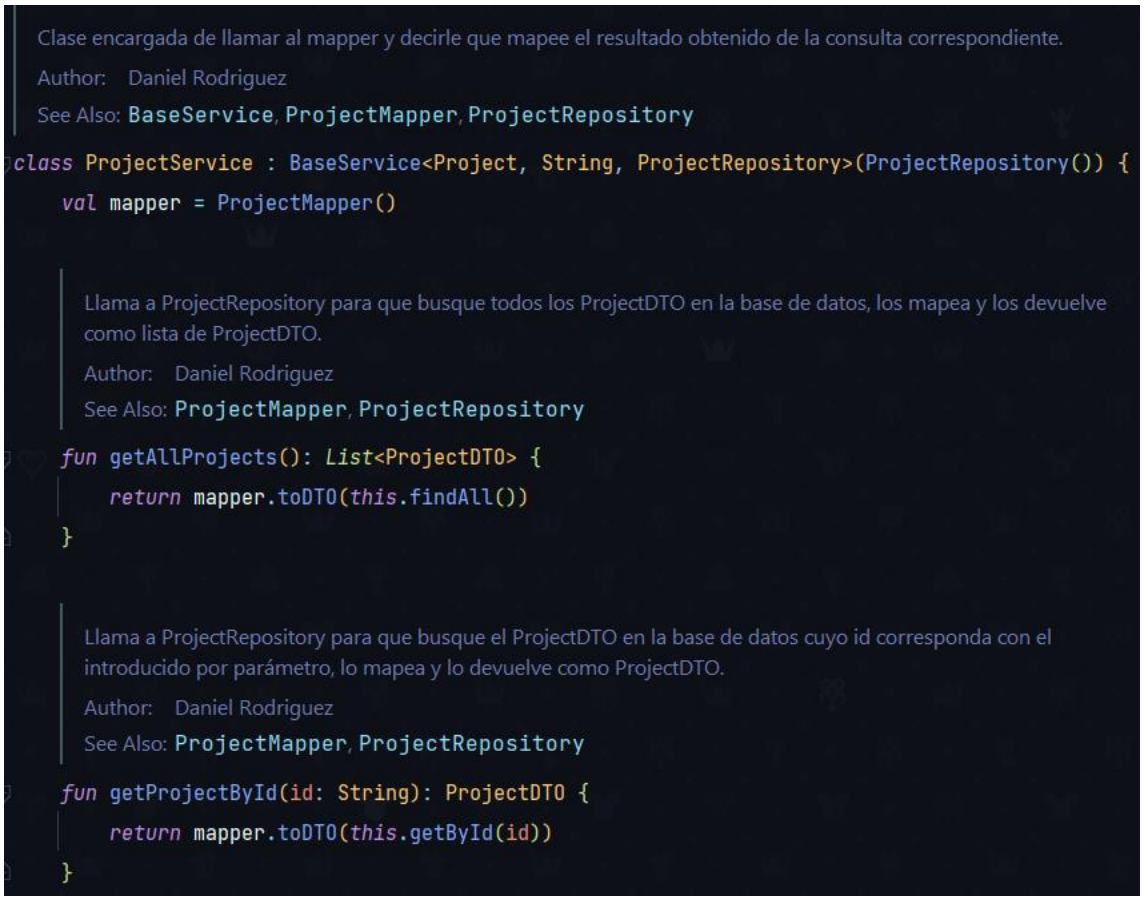
Explicación:



Se encarga de llamar al mapper(o no, si la operación no requiere que le pases un DTO previamente, como para el caso del findAll) para que pase el DTO en cuestión a un objeto del modelo con el que operar, luego de llamar al repositorio correspondiente y, tras llevar a cabo la operación, llamar de nuevo al mapper para que el resultado de dicha operación sea transformado a un DTO, y devolverlo.

Código de uno de ellos (y de la clase abstracta):

```
Clase abstracta que le da a las clases que la extiendan los metodos que llaman a los metodos del repository
correspondiente para hacer las operaciones crud
Author: Daniel Rodriguez
abstract class BaseService<T, ID, R : IRepository<T, ID>>(rep: R) {
    protected val repository = rep

    fun findAll(): List<T> {
        return repository.findAll()
    }

    fun getById(id: ID): T {
        return repository.getById(id)
    }

    fun insert(t: T): T {
        return repository.insert(t)
    }

    fun update(t: T): T {
        return repository.update(t)
    }

    fun delete(t: T): T {
        return repository.delete(t)
    }
}
```



```
Clase encargada de llamar al mapper y decirle que mapee el resultado obtenido de la consulta correspondiente.
Author: Daniel Rodriguez
See Also: BaseService, ProjectMapper, ProjectRepository
class ProjectService : BaseService<Project, String, ProjectRepository>(ProjectRepository()) {
    val mapper = ProjectMapper()

    Llama a ProjectRepository para que busque todos los ProjectDTO en la base de datos, los mapea y los devuelve
    como lista de ProjectDTO.
    Author: Daniel Rodriguez
    See Also: ProjectMapper, ProjectRepository
    fun getAllProjects(): List<ProjectDTO> {
        return mapper.toDTO(this.findAll())
    }

    Llama a ProjectRepository para que busque el ProjectDTO en la base de datos cuyo id corresponda con el
    introducido por parámetro, lo mapea y lo devuelve como ProjectDTO.
    Author: Daniel Rodriguez
    See Also: ProjectMapper, ProjectRepository
    fun getProjectById(id: String): ProjectDTO {
        return mapper.toDTO(this.getById(id))
    }
```

```
    Llama a ProjectRepository para que inserte el ProjectDTO en la base de datos, lo mapea y lo devuelve como
    ProjectDTO.
    Author:  Daniel Rodriguez
    See Also: ProjectMapper, ProjectRepository
fun createProject(proj: ProjectDTO): ProjectDTO {
    return mapper.toDTO(this.insert(mapper.fromDTO(proj)))
}

    Llama a ProjectRepository para que actualice el ProjectDTO en la base de datos, lo mapea y lo devuelve como
    ProjectDTO.
    Author:  Daniel Rodriguez
    See Also: ProjectMapper, ProjectRepository
fun updateProject(proj: ProjectDTO): ProjectDTO {
    return mapper.toDTO(this.update(mapper.fromDTO(proj)))
}

    Llama a ProjectRepository para que borre el ProjectDTO en la base de datos, lo mapea y lo devuelve como
    ProjectDTO.
    Author:  Daniel Rodriguez
    See Also: ProjectMapper, ProjectRepository
fun deleteProject(proj: ProjectDTO): ProjectDTO {
    return mapper.toDTO(this.delete(mapper.fromDTO(proj)))
}
}
```

Utils:

Explicación:

Una clase de utilidades varias para, por ejemplo, asegurarse de que los booleanos de un programador están correctos (0,0,1 – 1,0,0 – 0,1,0) [0==false; 1==true], o para asegurarse de que una fecha pasada como String cumple con una expresión regular específica, entre otras cosas.

Codigo:

```
Clase de utilidades.
class Utils {
    Se asegura de que la string introducida corresponde a una fecha entre el 01/01/1900 y el 31/12/9999
    dd/MM/yyyy, separados por barras y obligatoriamente dia es un rango de 1 a 31, mes es de 1 a 12 y año de
    1900 a 9999. Si no cumple este criterio, suelta excepcion.
    fun matchesDate(string: String): String {
        val splittedString = string.split( ...delimiters: "/")
        val year = splittedString[2].toInt()
        val month = splittedString[1].toInt()
        val day = splittedString[0].toInt()
        return if (day in 1..31 && month in 1..12 && year in 1900..9999) string
        else throw Exception("Invalid date.")
    }

    Si le pasas un nulo o cadena vacía, returnea nulo. De lo contrario llama al metodo matchesDate y returnea el
    resultado de dicho metodo.
    See Also: matchesDate
    fun matchesDateAcceptingNull(string: String?): String? {
        return if (string.isNullOrBlank()) null
        else matchesDate(string)
    }
```

```
Se asegura de que los booleanos de un objeto Programmer son correctos. De lo contrario, excepcion. Si son
correctos no hace nada.
fun makeSureBooleansAreCorrect(item: Programmer) {
    val depBoss = item.isDepBoss
    val pManager = item.isProjectManager
    val active = item.isActive
    when {
        depBoss && pManager → throw Exception(
                "A programmer can't be Department Boss and Project Manager at the same time."
        )
        depBoss && active → throw Exception(
                "A Department Boss can't be active at the same time."
        )
        pManager && active → throw Exception(
                "A Project Manager can't be active at the same time."
        )
        depBoss && pManager && active → throw Exception(
                "A programmer can't be Department Boss, Project Manager and be active at the same time."
        )
    }
}
```

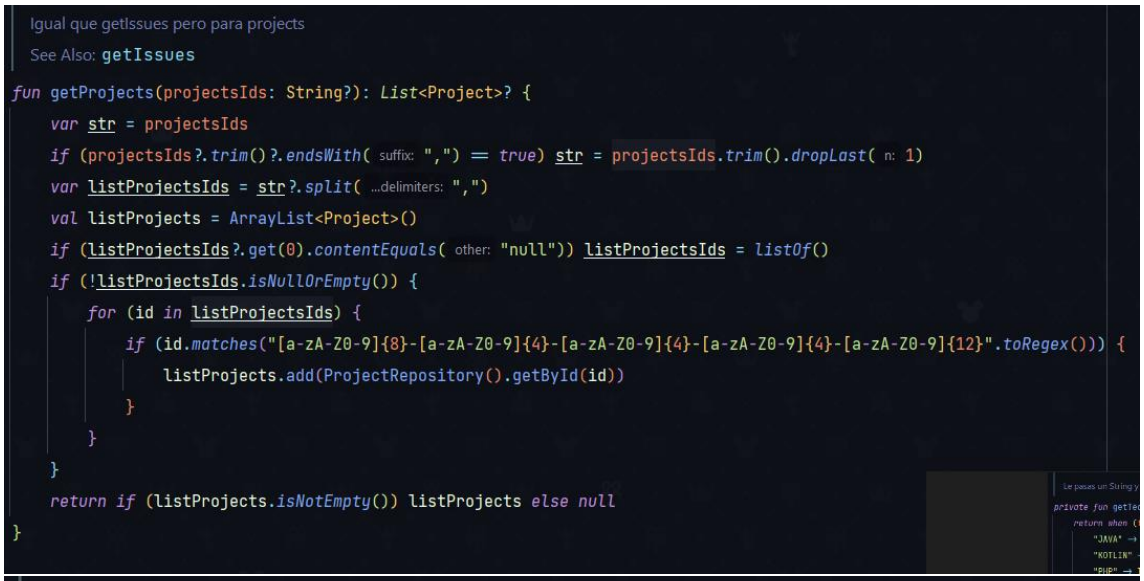
```
Lo mismo que el metodo de su mismo nombre pero para ProgrammerDTO
See Also: makeSureBooleansAreCorrect
fun makeSureBooleansAreCorrect(item: ProgrammerDTO) {
    when {
        item.isDepBoss && item.isProjectManager → throw Exception(
                "A programmer can't be Department Boss and Project Manager at the same time."
        )
        item.isDepBoss && item.isActive → throw Exception(
                "A Department Boss can't be active at the same time."
        )
        item.isProjectManager && item.isActive → throw Exception(
                "A Project Manager can't be active at the same time."
        )
        item.isProjectManager && item.isProjectManager && item.isActive → throw Exception(
                "A programmer can't be Department Boss, Project Manager and be active at the same time."
        )
    }
}
```

```
Se asegura de que la string introducida en cada argumento cumpla con la expresión regular del UUID. En caso
de no hacerlo, excepcion.
fun makeSureTheseAreIds(vararg args: String) {
    for (arg in args) {
        if (!(arg.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())))
            throw IllegalArgumentException("Error: The introduced value is not a valid id")
    }
}

Coge la string pasada por parametro y la convierte en una lista de issues, buscandolas por ID. La string ha de
contener las ids de las issues y estas deben estar separadas por comas.
fun getIssues(issuesIds: String?): List<Issue>? {
    val listIssues = issuesIds?.split( ...delimiters: ",")
    val listIssuesResult = ArrayList<Issue>()
    if (!listIssues.isNullOrEmpty()) {
        for (id in listIssues) {
            if (id.trim().matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex()))
                listIssuesResult.add(IssueRepository().getById(id))
        }
    }
    return if (listIssuesResult.isNotEmpty()) listIssuesResult else null
}
```

```
Igual que getIssues pero para commits
See Also: getIssues
fun getCommits(commitsIds: String?): List<Commit>? {
    val listCommits = commitsIds?.split( ...delimiters: ",")
    val listCommitsResult = ArrayList<Commit>()
    if (!listCommits.isNullOrEmpty()) {
        for (id in listCommits) {
            if (id.trim().matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                listCommitsResult.add(CommitRepository().getById(id))
            }
        }
    }
    return if (listCommitsResult.isNotEmpty()) listCommitsResult else null
}
```

```
Lo mismos que las anteriores, pero para technology
See Also: getCommits
fun getTechnologies(technologies: String?): List<Technology>? {
    val listTechnologies = technologies?.split( ...delimiters: ",")
    val listTechnologiesResult = ArrayList<Technology>()
    if (!listTechnologies.isNullOrEmpty()) {
        for (t in listTechnologies) {
            val tech = getTech(t.trim())
            if (tech != null) listTechnologiesResult.add(tech)
        }
    }
    return if (listTechnologiesResult.isNotEmpty())
        listTechnologiesResult else null
}
```

```
Le pasas un String y te returnea una tecnología, o nulo.
private fun getTech(t: String): Technology? {
    return when (t.toUpperCase()) {
        "JAVA" → Technology.JAVA
        "KOTLIN" → Technology.KOTLIN
        "PHP" → Technology.PHP
        "PYTHON" → Technology.PYTHON
        "CSHARP" → Technology.CSHARP
        "JAVASCRIPT" → Technology.JAVASCRIPT
        "TYPESCRIPT" → Technology.TYPESCRIPT
        "C" → Technology.C
        else → null
    }
}
```



```
Igual que getIssues pero para projects
See Also: getIssues
fun getProjects(projectsIds: String?): List<Project>? {
    var str = projectsIds
    if (projectsIds?.trim()?.endsWith( suffix: ",") == true) str = projectsIds.trim().dropLast( n: 1)
    var listProjectsIds = str?.split( ...delimiters: ",")
    val listProjects = ArrayList<Project>()
    if (listProjectsIds?.get(0).contentEquals( other: "null")) listProjectsIds = listOf()
    if (!listProjectsIds.isNullOrEmpty()) {
        for (id in listProjectsIds) {
            if (id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                listProjects.add(ProjectRepository().getById(id))
            }
        }
    }
    return if (listProjects.isNotEmpty()) listProjects else null
}
```

```
Le pasas una lista de technologies y te las devuelve en formato String, o devuelve una cadena vacia.
fun getTechnologiesAsString(technologies: List<Technology>?): String {
    var result = ""
    return if (!technologies.isNullOrEmpty()) {
        for (technology in technologies) {
            result += "${technology.name},"
        }
        result
    } else result
}
```

```
Le pasas una lista de projects y las devuelve como string, o returnea cadena vacia.
fun getProjectsIDS(projects: List<Project>?): String {
    var result = ""
    return if (!projects.isNullOrEmpty()) {
        for (project in projects) {
            if (project.id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                result += "${project.id},"
            } else throw Exception(
                    "Error at Utils.getProjectsIDS: " +
                            "Non-valid id"
            )
        }
        result
    } else result
}
```

```
Igual que getProjectsIDS pero para commits.
See Also: getProjectsIDS
fun getCommitsIDS(commits: List<Commit>?): String {
    var result = ""
    return if (!commits.isNullOrEmpty()) {
        for (commit in commits) {
            if (commit.id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                result += "${commit.id},"
            } else throw Exception(
                    "Error at Utils.getCommitsIDS: " +
                            "Non-valid id"
            )
        }
        result
    } else result
}
```

```
Igual que getProjectsIDS pero para issues.
See Also: getProjectsIDS
fun getIssuesIDS(issues: List<Issue>?): String {
    var result = ""
    return if (!issues.isNullOrEmpty()) {
        for (issue in issues) {
            if (issue.id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                result += "${issue.id},"
            } else throw Exception(
                    "Error at Utils.getIssuesIDS: " +
                            "Non-valid id"
            )
        }
        result
    } else result
}
```

```
Igual que getProjectsIDS pero para programmers.
See Also: getProjectsIDS
fun getProgrammersIDS(programmers: List<Programmer>?): String {
    var result = ""
    return if (!programmers.isNullOrEmpty()) {
        for (programmer in programmers) {
            if (programmer.id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                result += "${programmer.id},"
            } else throw Exception(
                    "Error at Utils.getProgrammersIDS: " +
                            "Non-valid id"
            )
        }
        result
    } else result
}
```

```
Igual que getIssues pero para programmers.
See Also: getIssues
fun getProgrammers(programmersIds: String?): List<Programmer>? {
    val listProgrammersIds = programmersIds?.split( ...delimiters: ",")
    val listProgrammers = ArrayList<Programmer>()
    if (!listProgrammersIds.isNullOrEmpty()) {
        for (id in listProgrammersIds) {
            if (id.matches("[a-zA-Z0-9]{8}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{4}-[a-zA-Z0-9]{12}".toRegex())) {
                listProgrammers.add(ProgrammerRepository().getById(id))
            }
        }
    }
    return if (listProgrammers.isNotEmpty()) listProgrammers else null
}
```

```
Se asegura de que el programmer introducido es jefe de departamento. En caso negativo, excepcion.
fun makeSureThisGuyIsDepBoss(boss: Programmer) {
    when {
        !boss.isDepBoss →
            throw Exception("Error: programmer with id ${boss.id} is not a Department boss.")
    }
    makeSureBooleansAreCorrect(boss)
}

Se asegura de que el programador introducido este en la issue introducida.
fun makeSureThisProgrammerIsInThisIssue(idProgrammer: String, idIssue: String) {
    val issue = IssueRepository().getById(idIssue)
    var containsProgrammer = false
    issue.programmers?.forEach { x →
        run {
            if (x.id.contentEquals(idProgrammer)) {
                containsProgrammer = true
            }
        }
    }
    if (!containsProgrammer) throw Exception("Error: programmer with id $idProgrammer is not in Issue[$idIssue].programmers")
}
```

```
Se asegura de que el programmer introducido es jefe de proyecto. En caso negativo, excepcion.
fun makeSureThisGuyIsProjectManager(author: Programmer, id: String) {
    var containsID = false
    author.activeProjects?.forEach { x →
        run {
            if (x.id.contentEquals(id)) {
                containsID = true
            }
        }
    }
    when {
        !author.isProjectManager →
            throw Exception("Error: programmer with id ${author.id} is not a Project Manager.")
        !containsID →
            throw Exception("Error: programmer with id ${author.id} is not in this project.")
    }
    makeSureBooleansAreCorrect(author)
}
```

Resources > META-INF:

Explicación:

El fichero de persistencia de JPA, ubicado en resources/META-INF.

En el podemos definir, entre otras cosas, la base de datos a usar, el lenguaje sql, el driver jdbc, usuario, contraseña, tipo de persistencia de los datos (por ejemplo, crear la base de datos de cero al iniciar la app y borrarla al terminar la ejecución [create-drop]), y en el tenemos también que poner la ubicación de las clases que serán nuestras entidades, en nuestro caso, las clases de la carpeta model.

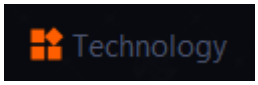
Codigo:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
             version="2.2">
    <persistence-unit name="default">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>model.Commit</class>
        <class>model.Department</class>
        <class>model.Issue</class>
        <class>model.Programmer</class>
        <class>model.Project</class>
        <class>model.Repository</class>

        <properties>
            <property name="hibernate.connection.url" value="jdbc:h2:~/test2"/>
            <property name="hibernate.connection.driver_class" value="org.h2.Driver"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.connection.user" value="sa"/>
            <property name="hibernate.connection.password" value=""/>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
            <property name="show_sql" value="true"/>
            <property name="hibernate.temp.use_jdbc_metadata_defaults" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Otras clases:

Technology:

Explicación:

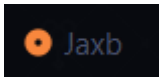


Clase enum para las tecnologías.

```java
enum class Technology {
    JAVA, KOTLIN, PHP, PYTHON, CSHARP, JAVASCRIPT, TYPESCRIPT, C
}
```

Jaxb:

Explicación:



Clase encargada de, gracias a las anotaciones hechas en los dto, volcar la información de un dto a un fichero xml y luego leer ese fichero para imprimirlo por pantalla.

Codigo:

```
Clase encargada de pasar el DTO introducido a xml.
Author: Jaime Salcedo
object Jaxb {
    private val fileDirectory = File( pathname: "${System.getProperty("user.dir")}${File.separator}temporalFiles")
    val file = File( pathname: "${System.getProperty("user.dir")}${File.separator}temporalFiles${File.separator}temp.txt")

    Si el directorio donde se va a almacenar el xml a mostrar no existe, lo crea para que vaya correcto todo
    init {
        if (!fileDirectory.exists()) {
            fileDirectory.mkdirs()
        }
    }

    Este metodo se encarga de pasar el DepartmentDTO que le pasemos a xml
    Params: x - DepartmentDTO
    Author: Jaime Salcedo
    fun departmentToXML(x: DepartmentDTO) {
        val jaxbContext = JAXBContext.newInstance(DepartmentDTO::class.java)
        printXML(jaxbContext, x)
    }
```

```
Este metodo se encarga de pasar el ProgrammerDTO que le pasemos a xml
Params: x - ProgrammerDTO
Author: Jaime Salcedo
fun programmerToXML(x: ProgrammerDTO) {
    val jaxbContext = JAXBContext.newInstance(ProgrammerDTO::class.java
    printXML(jaxbContext, x)
}

Este metodo se encarga de pasar el CommitDTO que le pasemos a xml
Params: x - CommitDTO
Author: Jaime Salcedo
fun commitToXML(x: CommitDTO) {
    val jaxbContext = JAXBContext.newInstance(CommitDTO::class.java)
    printXML(jaxbContext, x)
}

Este metodo se encarga de pasar el ProjectDTO que le pasemos a xml
Params: x - ProjectDTO
Author: Jaime Salcedo
fun projectToXML(x: ProjectDTO) {
    val jaxbContext = JAXBContext.newInstance(ProjectDTO::class.java)
    printXML(jaxbContext, x)
}
```

```
Este metodo se encarga de pasar el IssueDTO que le pasemos a xml
Params: x - IssueDTO
Author: Jaime Salcedo
fun issueToXML(x: IssueDTO) {
    val jaxbContext = JAXBContext.newInstance(IssueDTO::class.java)
    printXML(jaxbContext, x)
}

Este metodo se encarga de pasar el RepositoryDTO que le pasemos a xml
Params: x - RepositoryDTO
Author: Jaime Salcedo
fun repositoryToXML(x: RepositoryDTO) {
    val jaxbContext = JAXBContext.newInstance(RepositoryDTO::class.java)
    printXML(jaxbContext, x)
}
```

```
Este metodo coge el context que le pasas y crea un archivo formateado en un path, lee el archivo línea a línea y lo printea
Params: jaxbContext - JAXBContext
        x - : Any
Author: Jaime Salcedo
private fun printXML(jaxbContext: JAXBContext, x: Any) {
    val marshaller = jaxbContext.createMarshaller()
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)
    marshaller.marshal(x, file)
    val result = Files.readAllLines(file.toPath())
    result.forEach { y → println(y) }
}
```

Empresa:

Explicación:

Empresa

Esta clase es la que se dedicaría a llamar a los diferentes controladores dándoles las ordenes de hacer las operaciones crud y printear los resultados por consola en el formato deseado (xml o json), pero son las 2 de la mañana, llevo 2 horas y media haciendo documentación y todo el día haciendo la práctica y no me quedan fuerzas para terminar esto (básicamente porque para hacerlo tengo que crear un montón de objetos de prueba y eso se me hace tremendamente tedioso a estas horas de la noche). El código sería algo tal que así:

```
Hace un insert - find all - get by id - update - delete a Department
Author: Daniel Rodriguez
fun departments(x: String) {
    /*
    println("INSERT Department:")
    println(DepartmentController.insertDepartment(depart, x))
    println("\n\nFIND ALL Departments:")
    println(DepartmentController.findAllDepartments(x))
    println("\n\nGET Department with ID = ${depart.id}:")
    println(DepartmentController.getDepartmentById(depart.id, x))
    println("\n\nUPDATE Department with ID = ${depart.id}:")
    println(DepartmentController.updateDepartment(depart2, x))
    println("\n\nDELETE Department with ID = ${depart.id}:")
    println(DepartmentController.deleteDepartment(depart2, x))
    println("\n\n\n")

     */
}
```

(igual para el resto de objetos, cambiando obviamente el tipo de controller y objeto)

App:

Explicación:

App.kt

Función que inicializa el programa. Si no le has pasado exactamente un argumento, finaliza el programa con el código de error 1707 y te salta un mensaje diciendo que introduzcas exactamente un argumento.

Tras esto, llama a todos los métodos de Empresa.kt

```kotlin
fun main(args: Array<String>) {
    if (args.size != 1) {
        println("Invalid number of arguments. Only 1 argument required [xml/json]")
        println("Current number of arguments: ${args.size}")
        exitProcess( status: 1707)
    }

    Empresa.departments(args[0])
    //Empresa.departments("json")

    Empresa.projects(args[0])
    //Empresa.projects("json")

    Empresa.programmers(args[0])
    //Empresa.programmers("json")

    Empresa.commits(args[0])
    //Empresa.commits("json")

    Empresa.issues(args[0])
    //Empresa.issues("json")

    Empresa.repositories(args[0])
    //Empresa.repositories("json")
}
```