



Clase 40. Programación Backend

Arquitectura del servidor: Persistencia



OBJETIVOS DE LA CLASE

- Aplicar los patrones de la capa de persistencia DAO, DTO y Repository, en conjunto con el patrón Factory.
- Comprender cada uno de ellos con ejemplos y diferencias.
- Presentar ORM y ODM como técnicas para la conversión de datos.

CRONOGRAMA DEL CURSO

Clase 39



**Arquitectura del servidor:
Diseño**

Clase 40



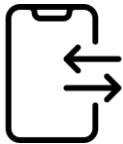
**Arquitectura del
servidor: Persistencia**

Clase 41



**Arquitectura cliente
servidor + Api REST**

PATRÓN DAO (DATA ACCESS OBJECT)



¿De qué se trata?



- Prácticamente todas las aplicaciones de hoy en día, requieren acceso al menos a una fuente de datos. Como suelen ser base de datos relacionales, muchas veces no tenemos problema en acceder a los datos.
- Sin embargo, podemos necesitar más de una fuente de datos, o la que tenemos puede variar, lo que nos obligaría a refactorizar gran parte del código.
- Para ésto, tenemos el patrón **Arquitectónico Data Access Object (DAO)**, que permite separar la lógica de acceso a datos de los **Business Objects u Objetos de negocios**, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.



Problemas de acceso a datos



La implementación y formato de la información pueden variar según la fuente de los datos.



Implementar la lógica de acceso a datos en la capa de lógica de negocio implicaría lidiar con dicha lógica en sí, más la implementación para acceder a los datos.



Además, si tenemos múltiples fuentes de datos o estas pueden variar, tendríamos que implementar las diferentes lógicas para acceder a las diferentes fuentes de datos.



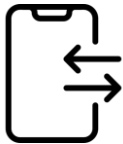
¿Cómo lo solucionamos?



- Para solucionar este problema, el patrón DAO propone separar por completo la lógica de negocio de la lógica para acceder a los datos.
- De esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información.



La capa de negocio solo se preocupa por la lógica de negocio y utiliza el DAO para interactuar con la fuente de datos. Éste es simplemente un nexo entre la lógica de negocio y la capa de persistencia (en general, base de datos).



Pasos del patrón DAO

1

Nuestra aplicación encapsula la información en un DTO.

2

El DAO toma ese DTO, extrae la información y construye la lógica necesaria para comunicarse con la fuente de datos (sentencias SQL, manejo de archivos, etc).

3

La fuente de datos recibe la información en el formato adecuado para tratarla.



Pasos del patrón DAO en sentido contrario



1

Nuestra aplicación hace un pedido de datos al DAO.

2

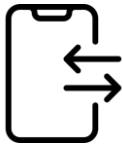
El DAO realiza una petición de datos a la fuente de datos

3

La fuente de datos envía al DAO la información.

4

El DAO recopila esa información, la encapsula en el DTO (o en otro elemento que la aplicación entienda) y se la devuelve a nuestra lógica de negocio.



Vocabulario relacionado



- **BusinessObject**: representa un objeto con la lógica de negocio.
- **DataAccessObject (DAO)**: representa una elemento de la capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos.
- **DataTransferObject (DTO)**: es un objeto plano que implementa el patrón de diseño homónimo, el cual sirve para transmitir la información entre la capa de negocio y la capa de persistencia.
- **DataSource**: representa de forma abstracta la fuente de datos, la cual puede ser una base de datos, un web service, un archivo de texto, etc.

DAO y el patrón Abstract Factory



¿De qué se trata?



- Como vimos, el DAO nos es útil cuando tenemos una sola fuente de datos, sin importar de qué tipo sea.
- Sin embargo, es común que tengamos que requerir datos de diferentes fuentes. Y es ahí dónde usamos el patrón **Abstract Factory**.
- Mediante este patrón podemos definir una serie de familias de clases que permitan conectarnos a las diferentes fuentes de datos.
- Entonces, vamos a tener un DAO por cada fuente de datos diferente que tengamos, de modo de poder usarlo de “traductor” en cada una de ellas y no tener que modificar la lógica de negocio si alguna cambia.



Usando DAO

```
import CustomError from '../errores/CustomError.js'

class ProductosDao {

  async getAll() {
    throw new CustomError(500, 'falta implementar getAll!')
  }

  async getById(id) {
    throw new CustomError(500, 'falta implementar getById!')
  }

  async add(prodNuevo) {
    throw new CustomError(500, 'falta implementar add!')
  }

  async deleteById(id) {
    throw new CustomError(500, 'falta implementar deleteById!')
  }

  async deleteAll() {
    throw new CustomError(500, 'falta implementar deleteAll!')
  }

  async updateById(id, nuevoProd) {
    throw new CustomError(500, 'falta implementar updateById!')
  }
}

export default ProductosDao
```

- Para comenzar, creamos la clase ***ProductosDao***, que va a ser la clase base.
- DAO es un objeto que tiene los métodos necesarios para acceder al sistema de persistencia.
- La clase de CustomError es simplemente:

```
class CustomError {
  constructor (estado, descripcion, detalles) {
    this.estado = estado
    this.descripcion = descripcion
    this.detalles = detalles
  }
}

export default CustomError
```



Usando DAO

- *ProductosDao* utiliza estas dos clases (*DbClient* y *DbClientMongo*) para conectarse a la base de datos, que en este caso es de MongoDB.

```
class DbClient {  
  
  async connect() {  
    throw new Error("falta implementar 'connect' en subclase!")  
  }  
  
  async disconnect() {  
    throw new Error("falta implementar 'disconnect' en subclase!")  
  }  
}  
  
export default DbClient
```

```
const Config = {  
  db: {  
    name: 'my_database',  
    collection: 'productos',  
    cnxStr: 'mongodb://localhost/',  
    //projection: {_id:0, __v:0}  
    projection: {__v:0}  
  }  
}  
  
export default Config
```

- Archivo Config:

```
import Config from '../config.js'  
import CustomError from '../errores/CustomError.js'  
import mongoose from 'mongoose'  
import DbClient from './DbClient.js'  
  
class MyMongoClient extends DbClient {  
  constructor() {  
    super()  
    this.connected = false  
    this.client = mongoose  
  }  
  
  async connect() {  
    try {  
      await this.client.connect(Config.db.cnxStr+Config.db.name, {  
        useNewUrlParser: true,  
        useUnifiedTopology: true,  
        useFindAndModify: false,  
        useCreateIndex: true  
      })  
      console.log('base de datos conectada')  
    } catch (error) {  
      throw new CustomError(500, 'error al conectarse a mongodb 1', error)  
    }  
  }  
  
  async disconnect() {  
    try {  
      await this.client.connection.close()  
      console.log('base de datos desconectada')  
      this.connected = false  
    } catch (error) {  
      throw new CustomError(500, 'error al conectarse a mongodb 2', error)  
    }  
  }  
}  
  
export default MyMongoClient
```



Usando DAO

```
import ProductosDaoDB from './dao/productosDaoDB.js'

class ProductosApi {

  constructor() {
    this.productosDao = new ProductosDaoDB()
  }

  async agregar(prodParaAgregar) {
    const prodAgregado = await this.productosDao.add(prodParaAgregar)
    return prodAgregado
  }

  async buscar(id) {
    let productos
    if (id) {
      productos = await this.productosDao.getById(id)
    } else {
      productos = await this.productosDao.getAll()
    }
    return productos
  }

  async borrar(id) {
    if(id) {
      await this.productosDao.deleteById(id)
    }
    else {
      await this.productosDao.deleteAll()
    }
  }

  async reemplazar(id, prodParaReemplazar) {
    const prodReemplazado = await this.productosDao.updateById(id, prodParaReemplazar)
    return prodReemplazado
  }

  exit() {
    this.productosDao.exit()
  }
}
```

- En un archivo llamado **index.js** tenemos entonces la lógica de nuestra aplicación, lo que sería la lógica de negocio.
- La clase **ProductosApi** utiliza la clase **ProductosDaoDB** para realizar los métodos de un CRUD sobre el modelo de productos que tenemos en este ejemplo.
- Vemos entonces, que si se modifica el tipo de persistencia, solo va a cambiar lo que hacen los métodos de la clase ProductosDaoDB, sin necesidad de modificar nuestra lógica de negocio.



Usando DAO

- Finalmente, para poder ejecutar los métodos de *ProductosApi*, vamos a usar un ***minimist*** (visto en clases anteriores) con el que podemos ejecutar los métodos con simples comandos por consola. De esta forma no tenemos la necesidad de crear un servidor y hacer algunas vistas. Lo vamos a ver en detalle más adelante.
- Para ejecutar cada método, en consola escribimos por ejemplo estos comandos para agregar nuevo producto y listar todos los que haya:

```
import minimist from 'minimist'

console.log('Instanciando la API')
const productosApi = new ProductosApi()

async function ejecutarCmds() {
  const argv = minimist(process.argv.slice(2))
  const {cmd,id,nombre,precio,stock} = argv
  try {
    switch(cmd.toLowerCase()) {
      case 'buscar':
        console.log(cmd)
        console.log(await productosApi.buscar(id))
        break

      case 'agregar':
        console.log(cmd)
        console.log(await productosApi.agregar({nombre,precio,stock}))
        break

      case 'reemplazar':
        console.log(cmd)
        console.log(await productosApi.reemplazar(id,{nombre,precio,stock}))
        break

      case 'borrar':
        console.log(cmd)
        await productosApi.borrar(id)
        break

      default:
        console.log('comando no válido:',cmd)
    }
  } catch(error) {
    console.log(error)
  }

  productosApi.exit()
}

ejecutarCmds()
```

```
$ node index.js --cmd Agregar --nombre TV --precio 12.34 --stock 99
```

```
$ node index.js --cmd Buscar
```




PATRÓN DAO

Tiempo: 10 minutos

PATRÓN DAO

Desafío
generico



Tiempo: 10 minutos

Crear una clase DAO (Data Access Object) llamada PersonasDaoMem que permita almacenar, obtener en forma total y leer/modificar/borrar por id los datos de personas recibidas (nombre, apellido, dni) utilizando la memoria RAM del servidor.

- Debe proveer los métodos necesarios para interactuar con una estructura tipo array de objetos que contenga toda la información.
- Realizar las pruebas necesarias por código para comprobar el correcto funcionamiento.
- La nueva clase creada debe estar en un archivo separado cuyo nombre será similar al de su clase contenida.

PATRÓN DTO (DATA TRANSFER OBJECT)



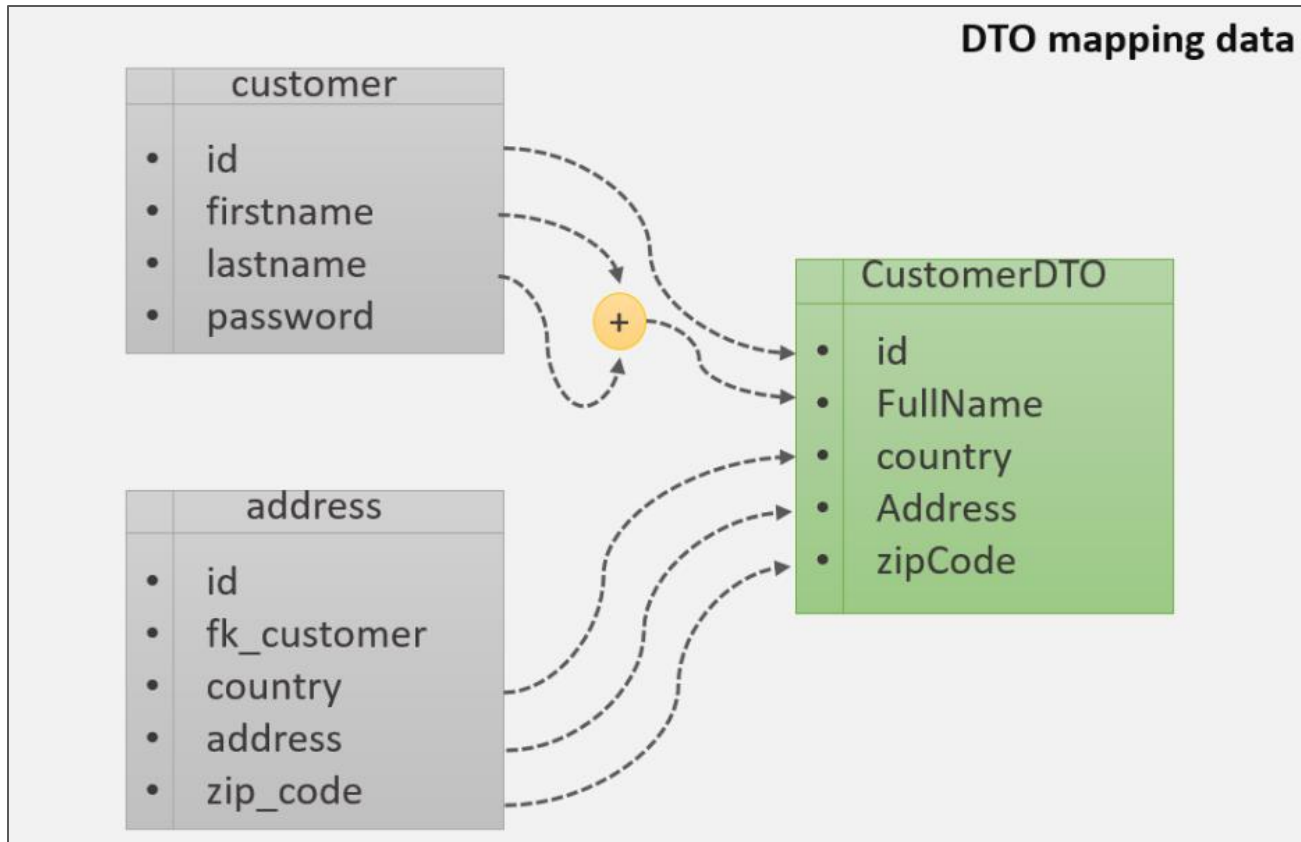
¿De qué se trata?



- Una de las problemáticas más comunes cuando desarrollamos aplicaciones, es diseñar la forma en que la información debe viajar desde la capa de servicios a las aplicaciones o capa de presentación.
- Muchas veces utilizamos las clases de entidades para retornar los datos, lo que ocasiona que retornemos más datos de los necesarios o incluso, tengamos que ir en más de una ocasión a la capa de servicios para recuperar los datos requeridos.
- El **patrón DTO** tiene como finalidad crear un objeto plano (POJO: Plain Old Javascript Object) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple.



Diagrama





Características



- Si bien un DTO es simplemente un objeto plano, tiene que cumplir algunas reglas para poder considerar que hemos creado un DTO correctamente implementado:
 - **Sin lógica:** Dado que el objetivo de un DTO es utilizarlo como un objeto de transferencia entre el cliente y el servidor, es importante evitar tener operaciones de negocio o métodos que realicen cálculos sobre los datos, es por ello que solo deberemos de tener los métodos GET y SET de los respectivos atributos del DTO.
 - **Serializable:** Es claro que, si los objetos tendrán que viajar por la red, deberán de poder ser serializables, pero no hablamos solamente de la clase en sí, sino que también todos los atributos que contenga el DTO deberán ser fácilmente serializables.



Usando DTO

Ejemplo
en vivo



```
class ProductoDto {
  constructor(datos, cotizaciones) {
    this.nombre = datos.nombre
    this.precio = datos.precio
    this.stock = datos.stock
    for (const [ denominacion, valor ] of Object.entries(cotizaciones)) {
      this[ denominacion ] = valor
    }
  }
}
```

```
class Cotizador {
  static VALOR_DOLAR = 100

  getPrecioSegunMoneda(precio, moneda) {
    switch (moneda) {
      case 'USD':
        return precio * Cotizador.VALOR_DOLAR
      default:
        return precio
    }
  }
}
```

- Continuando con el ejemplo que vimos en DAO, vamos a sumarle un método más para usar DTO.
- DTO es un objeto que se crea para reunir la información de varios objetos o modificar uno existente para crear 'vistas' de información y devolverle al cliente datos adaptados.
- Creamos un archivo con el código del DTO, y otro con un componente capaz de calcular precios en otras monedas.



Usando DTO

Ejemplo
en vivo



```
class ProductoDto {  
  constructor(datos, cotizaciones) {  
    this.nombre = datos.nombre  
    this.precio = datos.precio  
    this.stock = datos.stock  
    for (const [ denominacion, valor ] of Object.entries(cotizaciones)) {  
      this[ denominacion ] = valor  
    }  
  }  
}
```

- Vemos que, en este caso, el producto no presenta su `_id`, y en cambio, incluye las cotizaciones que hayamos elegido al momento de instanciarlo.



De esta forma, cuando busquemos un producto por su id, nos va a devolver un objeto con estas propiedades del producto, a diferencia del método anterior de Buscar que nos devolvía `_id`, nombre, precio y stock, tal cual figura en la base de datos.



Usando DTO

Ejemplo
en vivo



```
async buscarConCotizacionEnDolares(id) {  
  if (id) {  
    const producto = await this.productosDao.getById(id);  
    const cotizaciones = {  
      precioDolar: this.cotizador  
        .getPrecioSegunMoneda(producto.precio, 'USD')  
    }  
  
    const productoDto = new ProductoDto(producto, cotizaciones)  
    return productoDto  
  } else {  
    const productos = await this.productosDao.getAll();  
    const productosDtos = productos.map(producto => {  
      const cotizaciones = {  
        precioDolar: this.cotizador  
          .getPrecioSegunMoneda(producto.precio, 'USD')  
      }  
  
      const productoDto = new ProductoDto(producto, cotizaciones)  
      return productoDto  
    })  
    return productosDtos;  
  }  
}
```

base de datos conectada

```
[  
  ProductoDto {  
    nombre: 'TV',  
    precio: 12.34,  
    stock: 99,  
    precioDolar: 1234  
  },  
  ProductoDto {  
    nombre: 'PC',  
    precio: 56.78,  
    stock: 50,  
    precioDolar: 5678  
  }  
]
```

base de datos desconectada

Luego, en el archivo *index.js* importamos y agregamos un nuevo método en la clase *ProductosApi*.

- Este método busca todos los productos o un producto por su id, agregando su precio en dólares.
- Agregamos un comando en nuestros scripts para probarlo.

Y obtenemos lo siguiente como respuesta.

CODER HOUSE



Resumiendo...

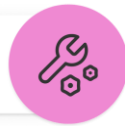


- Como vimos, los DTO son un patrón muy efectivo para transmitir información entre un cliente y un servidor, pues nos permite crear estructuras de datos independientes de nuestro modelo de datos, lo que nos permite crear cuantas “vistas” sean necesarias de un conjunto de tablas u orígenes de datos.
- Además, nos permite controlar el formato, nombre y tipos de datos con los que transmitimos los datos para ajustarnos a un determinado requerimiento.
- Finalmente, si por alguna razón, el modelo de datos cambió el cliente no se afectará, pues seguirá recibiendo el mismo DTO.



PATRÓN DTO

Tiempo: 10 minutos



Tiempo: 10 minutos

Crear en base al desafío anterior, otro DAO llamado `PersonasDaoFile` que contenga los mismos métodos para interactuar con los datos de una persona, pero esta vez persistida en el file system (fs). Para asegurar el correcto funcionamiento del contenedor, agregar un método *init* que verifique que el archivo existe y sino que lo cree. Para mantener la consistencia entre las interfaces de ambos DAOs, agregar también este nuevo método al DAO de memoria.

En ambas clases (mem y file) en lugar de devolver objetos anónimos de javascript, instanciar y devolver objetos de la clase `PersonaDto`. Crear esta clase con los mismos campos que los almacenados para cada persona (ni uno más, ni uno menos). Las nuevas clases creadas deben estar en un archivo separado cuyo nombre será similar al de su clase contenida.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

PATRÓN FACTORY METHOD



¿De qué se trata?



- Este patrón actúa como una herramienta que podemos implementar para limpiar un poco nuestro código.
- En esencia, el patrón **Factory Method** nos permite centralizar la lógica de crear objetos (es decir, qué objeto crear y por qué) en un solo lugar. Esto nos permite olvidarnos de esa parte y concentrarnos en simplemente solicitar el objeto que necesitamos y luego usarlo.
- Es un patrón de creación que no requiere que usemos un constructor, pero proporciona una interfaz genérica para crear objetos.
- Este patrón puede resultar realmente útil cuando el proceso de creación es complejo.



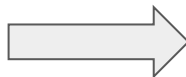
¿Cómo lo implementamos?



- En este ejemplo, primero creamos una clase Employee.

```
class Employee {  
  
    speak() {  
        return "Hi, I'm a " + this.type + " employee"  
    }  
  
}
```

- Luego, creamos 3 clases hijas de la clase Employee.



```
class FullTimeEmployee extends Employee{  
    constructor(data) {  
        super()  
        this.type = "full time"  
        //....  
    }  
}  
  
class PartTimeEmployee extends Employee{  
    constructor(data) {  
        super()  
        this.type = "part time"  
        //....  
    }  
}  
  
class ContractorEmployee extends Employee{  
    constructor(data) {  
        super()  
        this.type = "contractor"  
        //....  
    }  
}
```




¿Cómo lo implementamos?



- En el patrón Factory Method, debemos crear otra clase, que instancie a la clase creada anteriormente.
- En este caso, tenemos 3 clases a instanciar, que según el valor pasado como parámetro, decidimos cuál instanciar, como vemos en el código:

```
class MyEmployeeFactory {  
  
    createEmployee(data) {  
        if(data.type == 'fulltime') return new FullTimeEmployee(data)  
        if(data.type == 'parttime') return new PartTimeEmployee(data)  
        if(data.type == 'contractor') return new ContractorEmployee(data)  
    }  
}
```



¿Cómo lo usamos?



- Vemos ahora un ejemplo de cómo usar las clases que desarrollamos.
- La conclusión clave del este código es el hecho de que está agregando objetos al mismo array, todos los cuales comparten la misma interfaz (en el sentido de que tienen el mismo conjunto de métodos) pero realmente no necesitamos preocuparnos por qué objeto crear y cuándo hacerlo.

```
( _ => {  
  
    let factory = new MyEmployeeFactory()  
  
    let types = ["fulltime", "parttime", "contractor"]  
    let employees = [];  
    for(let i = 0; i < 100; i++) {  
        employees.push(factory.createEmployee({type: types[Math.floor( (Math.random(2) * 2) )]}))  
    }  
  
    //....  
    employees.forEach( e => {  
        console.log(e.speak())  
    })  
  
})()
```



Casos de uso



- Un caso de uso particular para este patrón es el manejo de la creación de objetos de error.
- Imaginemos tener una aplicación Express con aproximadamente 10 puntos finales, en la que cada punto final necesita devolver entre dos o tres errores según la entrada del usuario. Estamos hablando de 30 frases como las siguientes:

```
if(err) {  
  res.json({error: true, message: "Error message here"})  
}
```

- Ahora, eso no sería un problema, al menos hasta la próxima vez que tuviéramos que agregar repentinamente un nuevo atributo al objeto de error.



Casos de uso



- Ahora tenemos que repasar todo nuestro proyecto, modificando los 30 lugares. Ésto se resolvería moviendo la definición del objeto de error a una clase.
- Esta acción funcionará, a menos que tuviéramos más de un objeto de error y, de nuevo, tenemos que decidir qué objeto instanciar en función de una lógica que solo nosotros conocemos.
- Si tuviésemos que centralizar la lógica para crear el objeto de error, entonces todo lo que tendríamos que hacer a lo largo de nuestro código:

```
if(err) {  
  res.json(ErrorFactory.getError(err))  
}
```

✓ Con esto solucionamos el problema, y tenemos un solo objeto para manejar los errores.



PATRÓN FACTORY

Tiempo: 10 minutos

PATRÓN FACTORY

Desafío
generico



Tiempo: 10 minutos

- Agregar al ejemplo del último desafío:
 - Una clase PersonasDaoDb que realice la persistencia usando MongoDB (devolviendo instancias de PersonaDto). Debemos agregar a nuestra familia de DAOs un método para desconectar al DAO (aunque sólo realice algo útil en la versión para MongoDB).
 - Una factory en donde se defina qué sistema de almacenamiento de datos se utilizará entre las tres opciones disponibles:
 - Memoria
 - Archivos
 - MongoDB

PATRÓN FACTORY

Desafío
generico



Tiempo: 10 minutos

- El test no se debe enterar del cambio del sistema de persistencia. Ésta se establece a través de un parámetro que se pasa por línea de comandos con estas opciones:
 - **'Mem'**: selecciona Memory como sistema de persistencia.
 - **'File'**: selecciona File System como sistema de persistencia.
 - **'Mongo'**: selecciona MongoDB como sistema de persistencia.

PATRON REPOSITORY



¿De qué se trata?



- Repository es un patrón que se utiliza para mantener una conexión débilmente acoplada entre el cliente y los procedimientos de almacenamiento de datos del servidor que ocultan toda implementación compleja.
- Esto significa que el cliente no tendrá que preocuparse por cómo acceder a la base de datos, agregar o eliminar elementos de una colección, etc.
- Con este patrón realizamos una correspondencia entre los datos provenientes de la base de datos y los modelos del dominio del negocio.
- Un repositorio se comporta como una colección de datos, con los métodos que esperamos de ella, abstrayéndonos de su implementación.



¿De qué se trata?



- Es posible definir un repositorio genérico con las operaciones básicas, y luego mediante el mecanismo de herencia generar comportamientos personalizados para cada entidad según corresponda.
- La lógica empresarial está encapsulada en funciones dentro del Repositorio. Si la implementación cambia alguna vez, lo tenemos todo en un solo lugar para cambiarlo como deseemos.



Diferencias



Patron Repository

- Se ubica al mismo nivel de la capa de modelo de dominio, un poco más arriba que DAO.
- Un Repository usa y genera entidades del modelo de dominio completamente instanciadas.



DAO

- DAO se ubica en un nivel más bajo, mucho más cerca a la fuente de datos.
- Un DAO usa y genera objetos portadores de información.

Un Repository puede ser implementado sobre una capa de DAO, sin embargo, la operación contraria supondría una destrucción de las definiciones formales de ambos conceptos.



Diferencias



Patron Repository

- Sólo existe un repositorio para cada Agregado, solo algunas de las tablas en la base de datos pueden ser correspondidas con la existencia de un Repositorio en la capa de modelo de dominio.
- Su intención es proveer entidades de la capa de dominio.



DAO

- Entre tablas en una base de datos y DAO en la capa de acceso a datos suele existir una relación que tiende a ser directa, es decir, tiende a existir una relación uno a uno entre ambos.
- DAO ha sido diseñado para obtener y guardar información de una base de datos



Usando Repository

Ejemplo
en vivo



En este ejemplo desarrollaremos un repositorio de Personas. Para ello, crearemos primero la clase Persona, con sus correspondientes validaciones:

```
export default class Producto {  
  #id;  
  #nombre;  
  #precio;  
  #stock;  
  
  constructor({ id, nombre, precio, stock }) {  
    this.id = id  
    this.nombre = nombre;  
    this.precio = precio;  
    this.stock = stock;  
  }  
  
  get id() { return this.#id }  
  
  set id(id) {  
    if (!id) throw new Error('"id" es un campo requerido');  
    this.#id = id;  
  }  
  
  get nombre() { return this.#nombre }
```

```
  set nombre(nombre) {  
    if (!nombre) throw new Error('"nombre" es un campo requerido');  
    this.#nombre = nombre;  
  }  
  
  get precio() { return this.#precio }  
  
  set precio(precio) {  
    if (!precio) throw new Error('"precio" es un campo requerido');  
    if (isNaN(precio)) throw new Error('"precio" debe ser numérico');  
    if (precio < 0) throw new Error('"precio" debe ser positivo');  
    this.#precio = precio;  
  }  
  
  get stock() { return this.#stock }  
  
  set stock(stock) {  
    if (!stock) throw new Error('"stock" es un campo requerido');  
    if (isNaN(stock)) throw new Error('"stock" debe ser numérico');  
    if (stock < 0) throw new Error('"stock" debe ser positivo');  
    this.#stock = stock;  
  }  
}
```

CODER HOUSE



Usando Repository

Ejemplo
en vivo



Luego creamos el repositorio, el cual generalmente tendrá las mismas operaciones que una colección de objetos: agregar, quitar, listar, buscar, contar (no se incluyen todas en este ejemplo):

```
export default class ProductosRepo {  
  
  constructor() {  
    this.dao = getDao()  
  }  
  
  async getAll() {  
    const dtos = await this.dao.getAll({})  
    return dtos.map(dto => new Producto(dto))  
  }  
  
  add(prod) {  
    const dto = new ProductoDto(prod)  
    return this.dao.save(dto)  
  }  
  
  addMany(prods) {  
    const dtos = prods.map(p => new ProductoDto(p))  
    return this.dao.saveMany(dtos)  
  }  
}
```

Observen que el repositorio recibe Personas pero envía DTOs al DAO.

De igual manera, el DAO devuelve DTOs, pero el repositorio los transforma en Personas antes de devolverlos.

El repositorio cumple entonces la función de nexo entre la capa de negocio y la capa de persistencia (aunque más del lado del negocio, en cuanto a nivel de abstracción).



PATRÓN REPOSITORY

Tiempo: 10 minutos

PATRÓN REPOSITORY

Desafío
generico



Tiempo: 10 minutos

Aplicar el patrón repositorio sobre el desafío anterior.

- El repositorio estará implementado en una clase `PersonaRepository`, y contará con métodos para realizar las operaciones de CRUD sobre instancias de la clase `Persona` (nuestro modelo de dominio). Dispondrá también de un método que indique la cantidad de personas almacenadas.
- El constructor de `PersonaRepository` obtendrá el DAO de personas que utilizará para la persistencia desde un factory.

APLICANDO REPOSITORY

Desafío
generico



Tiempo: 10 minutos

Probar el repositorio utilizando un código de test llamando a cada acción del repositorio con los datos apropiados ó mediante un menú de test que permita por línea de comandos ejecutar las distintas tareas implementadas en el repositorio, representando en todos los casos los resultados por consola.

- Utilizar diferentes DAOs y verificar que el funcionamiento no se ve afectado de ninguna manera al cambiar entre uno y otro.
- Verificar que los datos se persisten de forma correcta en cada persistencia.

ORM ***(OBJECT RELATIONAL MAPPING)***

¿De qué se trata?



- Es una técnica para convertir datos entre el sistema de tipos del lenguaje de programación y la base de datos.
- Va dirigido solamente a las bases de datos relacionales (SQL). Esto crea un efecto “objeto base de datos virtual” sobre la base de datos relacional, el cual es lo que nos permite manipular la base de datos a través del código.
- **Object:** Hace referencia a los objetos que podemos usar en nuestro lenguaje.
- **Relational:** Hace referencia a nuestro Sistema Gestor de Base de Datos (MySQL, MSSQL, PostgreSQL).
- **Mapping:** Hace referencia a la conexión entre los objetos y las tablas.

¿De qué se trata?



- ORM es una técnica que nos permite hacer queries y manipular datos de la base de datos desde un lenguaje de programación.
- Tiene las siguientes ventajas:
 - **Abstracto:** Diseño de una estructura o modelo aislado de la base de datos.
 - **Portable:** Nos permite transportar la estructura de nuestro ORM a cualquier DBMS.
 - **Anidación de datos:** En caso de que una tabla tenga una o varias relaciones con otras.

¿De qué se trata?



- Tiene las siguientes desventajas:
 - **Lento:** Si se compara el tiempo de respuesta entre una raw query y un query hecho por objetos, raw query es mucho mas rápido debido a que no existe una capa intermedia (mapping).
 - **Complejidad:** Algunas veces necesitaremos hacer queries complejas. Para eso, tenemos Sequelize que nos permite ejecutar raw queries.

Sequelize



- Sequelize es un ORM basado en promesas para Node.
- Soporta PostgreSQL, MySQL, SQLite y MSSQL, y entrega características sólidas de transacciones, relaciones entre tablas, mecanismos de migraciones y carga de datos, etc.
- Sequelize maneja sus objetos como promesas, algo que va de la mano con el event loop de Node.
- Tiene los mismos alcances que Mongoose y se usa para lo mismo, pero cuando trabajamos con bases de datos relacionales.

ODM ***(OBJECT DOCUMENT MAPPER)***



¿De qué se trata?



- Es como un ORM para bases de datos no relacionales o bases de datos distribuidas como MongoDB.
- Por ejemplo, mapea un modelo de objeto y una base de datos NoSQL (bases de datos de documentos, base de datos de gráficos, etc.).
- MongoDB expresa los datos que se guardarán en un formato similar a JSON y los guarda como un documento. ODM es la función de asociar tal documento con un objeto en un lenguaje de programación.
- **Mongoose** es un ODM. Esto significa que nos permite definir objetos con un esquema fuertemente tipado que se asigna a un documento MongoDB.



Mongoose - Schemas



- Un esquema en Mongoose es una estructura JSON que contiene información acerca de las propiedades de un documento. Puede también contener información acerca de la validación y de los valores por default y si una propiedad en particular es requerida. Los esquemas pueden contener lógica y otro tipo de información importante.
- Sirven como guías de la estructura de los documentos. Estos son necesarios para la creación del modelo. Así que antes de utilizar los modelos de manera apropiada, es necesario definir sus esquemas.
- Mongoose ignora todas las propiedades que no sean definidas dentro del modelo de un esquema.
- Se pueden conectar entre sí. Lo que significa que cierta funcionalidad puede ser extendida a través de todos los esquemas de la aplicación.



Mongoose - Schemas



```
/**
 * Module dependencies.
 */

'use strict';

const mongoose = require('mongoose');
const Schema = mongoose.Schema;

/**
 * Schema definition
 */

// recursive embedded-document schema

const Comment = new Schema();

Comment.add({
  title: {
    type: String,
    index: true
  },
  date: Date,
  body: String,
  comments: [Comment]
});
```

```
const BlogPost = new Schema({
  title: {
    type: String,
    index: true
  },
  slug: {
    type: String,
    lowercase: true,
    trim: true
  },
  date: Date,
  buf: Buffer,
  comments: [Comment],
  creator: Schema.ObjectId
});
```

```
const Person = new Schema({
  name: {
    first: String,
    last: String
  },
  email: {
    type: String,
    required: true,
    index: {
      unique: true,
      sparse: true
    }
  },
  alive: Boolean
});
```

- Para crear un esquema, lo requerimos de mongoose, e instanciamos la clase.
- En este caso creamos el esquema Comment, BlogPost y Person.



Mongoose - Schemas



```
/**  
 * Accessing a specific schema type by key  
 */
```

```
BlogPost.path('date')  
  .default(function() {  
    return new Date();  
  })  
  .set(function(v) {  
    return v === 'now' ? new Date() : v;  
  });
```

```
/**  
 * Pre hook.  
 */
```

```
BlogPost.pre('save', function(next, done) {  
  /* global emailAuthor */  
  emailAuthor(done); // some async function  
  next();  
});
```

- Podemos acceder a un tipo de esquema específico por el nombre de su key.
- En este caso, accede a 'date' diciendo que el valor por default de este campo será la fecha actual.



Mongoose - Schemas



```
/**
 * Methods
 */

BlogPost.methods.findCreator = function(callback) {
  return this.db.model('Person').findById(this.creator, callback);
};

BlogPost.statics.findByTitle = function(title, callback) {
  return this.find({ title: title }, callback);
};

BlogPost.methods.expressiveQuery = function(creator, date, callback) {
  return this.find('creator', creator).where('date').gte(date).run(callback);
};

/**
 * Plugins
 */

function slugGenerator(options) {
  options = options || {};
  const key = options.key || 'title';

  return function slugGenerator(schema) {
    schema.path(key).set(function(v) {
      this.slug = v.toLowerCase().replace(/^[^a-z0-9]/g, '').replace(/-+/g, '');
      return v;
    });
  };
}

BlogPost.plugin(slugGenerator());
```

- Luego, tenemos los métodos, en donde por ejemplo buscamos un creador por su id o buscamos por título.
- Tenemos también los plugins, que son una herramienta para reutilizar la lógica en múltiples esquemas.



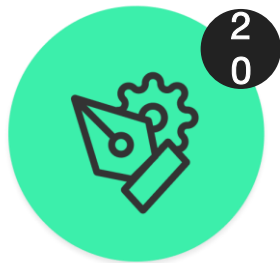
Mongoose - Schemas



```
/**
 * Define model.
 */

mongoose.model('BlogPost', BlogPost);
mongoose.model('Person', Person);
```

- Finalmente, definimos el modelo como se muestra en el código.



MEJORAR LA ARQUITECTURA DE NUESTRA API

Retomemos nuestro trabajo para implementar los patrones de diseño Factory, DAO y DTO.

MEJORAR LA ARQUITECTURA DE NUESTRA API

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Desafío
entregable



>> Consignas:

- Modificar la capa de persistencia incorporando los conceptos de Factory, DAO, y DTO.
- Los DAOs deben presentar la misma interfaz hacia la lógica de negocio de nuestro servidor.
- El DAO seleccionado (por un parámetro en línea de comandos como lo hicimos anteriormente) será devuelto por una Factory para que la capa de negocio opere con el.
- Cada uno de estos casos de persistencia, deberán ser implementados usando el patrón singleton que impida crear nuevas instancias de estos mecanismos de acceso a los datos.
- Comprobar que si llamo a la factory dos veces, con una misma opción elegida, devuelva la misma instancia.
- Implementar el patrón Repository para la persistencia de productos y mensajes.

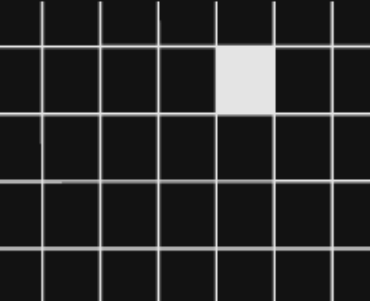
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Patrones de diseño:
 - Singleton
 - Factory Method y Abstract Factory
 - DAO, DTO y Repository
 - ORM y ODM
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN