Watch: **Refactorizando Student Sign Up aplicando repositorio**

Refactorizando Student Sign Up aplicando repositorio ✈️

En videos anteriores comenzamos a desarrollar el caso de uso para el registro de un nuevo usuario en la plataforma, sin embargo en ese momento el almacenamiento en memoria se estableció en un array dentro de la propia clase

Clase `StudentSignUpper`:

```
final class StudentSignUpper
{
    private static $students = [];

    public function __invoke(string $studentId, StudentName
$name, StudentPassword $password)
    {
        $this->ensureStudentDoesntExist($studentId);

        $student = new Student ($studentId, $name, $password);

        self::$students[$studentId] = $student;
    }

    private function ensureStudentDoesntExist($studentId):
void
    {
        $existentStudent = get($studentId, self::$students);

        if(null !== $existentStudent){
            throw new StudentAlreadyExist($studentId);
        }
    }
}
```

Puesto que no es conveniente tener toda la lógica de persistir el nuevo usuario dentro del caso de uso y mucho menos cargándose en

un array en memoria, vamos a refactorizarlo usando el **Patrón Repository**, consistente en extraer la lógica de almacenamiento a la infraestructura, así el caso de uso StudentSignUpper no tendrá conocimiento de donde estamos persistiendo

Clase `StudentRepositoryInMemory`:

```
final class StudentRepositoryInMemory
{
    private static $students = [];

    public function save(Student $student){
        self::$students[$student->studentId()] = $student;
    }

    public function search(StudentId $id): ?Student
    {
        return get($id->value(), self::$students)
    }
}
```

De este modo, estamos extrayendo la lógica de persistir un usuario a una clase a parte, en este caso una implementación para guardarlos en memoria. Es importante ver que lo que recibe la función save es `Student`, puesto que **Los repositorios siempre van a interacturar con los Aggregate Root**. Si no hicieramos esto, permitiríamos que desde los clientes se interactuase con los elementos que están dentro del Aggregate Root y estaríamos rompiendo con las reglas de consistencia que permiten gestionarlo

Clase `StudentSignUpper`:

```
final class StudentSignUpper
{
    private $repository;

    public function __construct(StudentRepositoryInmemory
$repository)
    {
        $this->repository = $repository;
    }

    public function __invoke(StudentId $id, StudentName $name,
StudentPassword $password)
    {
```

```
            $this->ensureStudentDoesntExist($id);

            $student = new Student ($id, $name, $password);

            $this->repository->save($student);
        }

        private function ensureStudentDoesntExist($studentId):
    void
        {
            $existentStudent = $this->repository-
    >search($studentid);

            if(null !== $existentStudent){
                throw new StudentAlreadyExist($studentId);
            }
        }
    }
```

En el caso de uso estamos inyectando el repositorio vía constructor, puesto que aún no hemos realizado ninguna inversión de dependencias, `StudentSignUpper` estará por el momento acoplada a la capa de infraestructura (concretamente a la clase `StudentRepositoryInMemory`), violando así la regla de dependencia de Arquitectura Hexagonal

Aunque aún tenemos este acoplamiento en el caso de uso, estamos ganando de cara a que hemos encapsulado la responsabilidad de almacenar los datos en el repositorio, mientras que en el caso de uso de lo que se encargará es de orquestar toda la lógica necesaria

¿Alguna Duda?

Hasta aquí ya hemos visto cómo aplicar el patrón Repository en nuestro caso de uso, en el siguiente video daremos un pasito más invirtiendo las dependencias y veremos la diferencia entre Role interfaces y Header interfaces

Si tienes dudas o sugerencias sobre este video recuerda que puedes abrir una nueva discusión ¡Justo aquí abajo! 👇👇👇