DDD en TypeScript: Modelado y arquitectura

Codely Tv Pro

1. [Gratis] 🚀 Creación del proyecto: Bounded Context y Submodules

1.1. 👋 Bienvenida al curso: Objetivos, ¿qué aprenderás?

Vamos a partir de un Monolito que tendrá toda la lógica acoplada, y poco a poco la iremos separando por responsabilidades.

- Priorizando la escalabilidad, NO solo a nivel de rendimiento computacional, sino que también como escalabilidad tolerante al paso del tiempo y separación de equipos (promoción).

1.2. 📂 Estructura de carpetas: DDD en TypeScript

- El Frontal de la App de Cursos es el ***MOOC*** o Massive Online Open Courses.
  - Donde los Users se pueden registrar en los cursos.
    - Tiene su Front y su Back
- Backoffice: Donde los admins de Codely pueden ver estadísticas y demás.
  - Este *Backoffice* tiene tanto Front como Back
- La HomeOffice
  - Que también tiene sus cositas

Tenemos:

- Tenemos los ***BC*** (Bounded Context)
  - Que tendrán sus Módulos específicos, en donde estos pueden repetirse a nivel de nombre entre diferentes BD, pero no serán lo mismo a nivel de funcionalidad.
    - El Módulo de *Courses* de MOOC no será el mismo que tenga Backoffice.

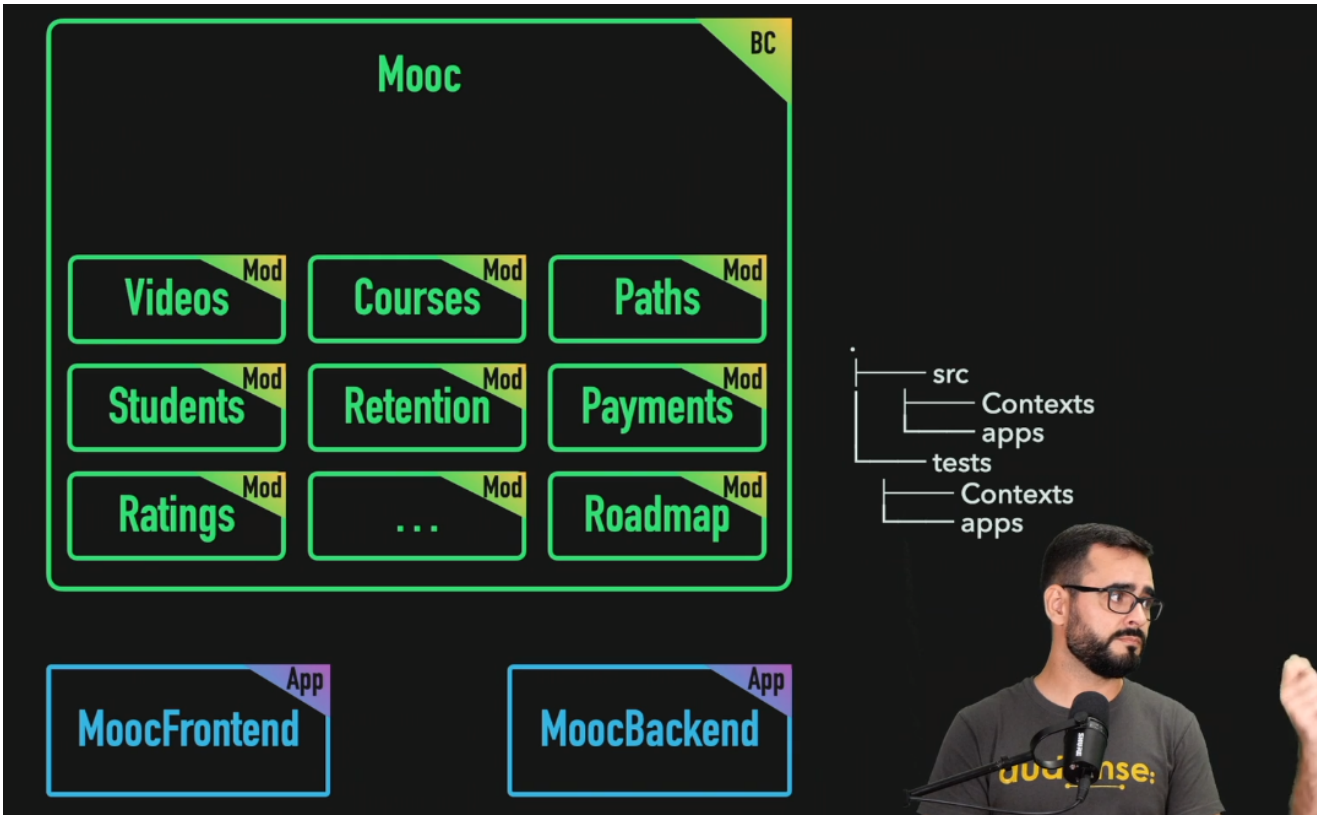
- Las Aplicaciones que consumen los ***BC***
  - MoocFrontend: App en React, Vue, etc.
    - Consume todos esos *Casos de Uso* del BC.
  - MoocBackend: API que está ejecutando código dentro de los BC.

1.2.1. Bounded Contex de MOOC

- Comenzamos con el BC de Mooc, y aquí iniciamos con la estructura del árbol de directorios.
  - Comenzamos con **2** carpetas, ***/src*** y ***/tests***
    - De esta manera todo el *código* que va a ir a ***producción*** va a estar en ***/src***, por lo tanto será el código que compilamos.
      - De esta manera podemos tener al mismo nivel a */Context* y */apps*.
    - En ***/test*** Replicamos la estructura del */src*
      - Los TEST de ***/apps*** van a ser tests de *Aceptación*, de Caja Negra o tests E2E (end to end). Tests desde donde vamos a testear una funcionalidad *desde el punto más Externo SIN conocer la implementación que hay por dentro*.
        - s
      - Los TESTS en ***/Contexts*** en su mayoría van a ser *Tests Unitarios*, pero en el caso de testear algún elemento de *Infraestructura* unos tests de *Integración* como podría ser un test de un Repository contra una DB.
        - Para estos *Unit Tests*: ***Jest***
        - Tests de Caja Negra Emulando lo que puede hacer un Usuario Final con las *Apps* que estamos exponiendo

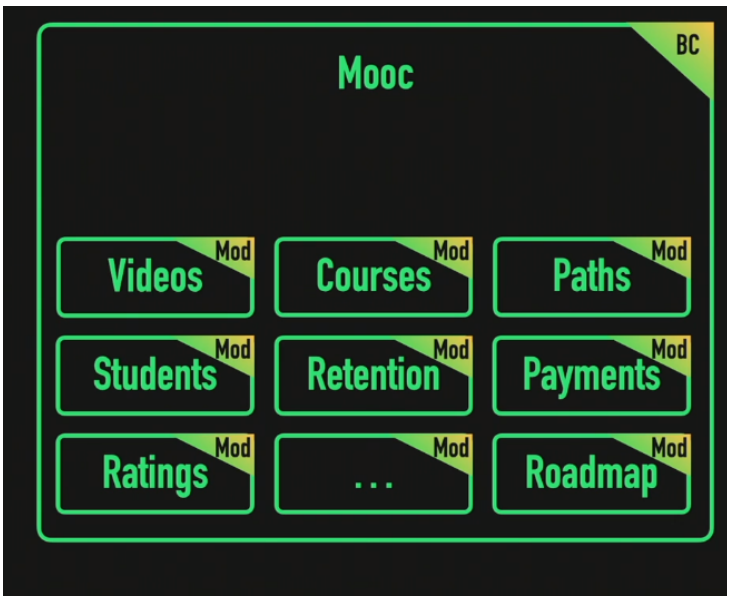
con el código de los Controllers o las vistas renderizadas dentro de /src/apps

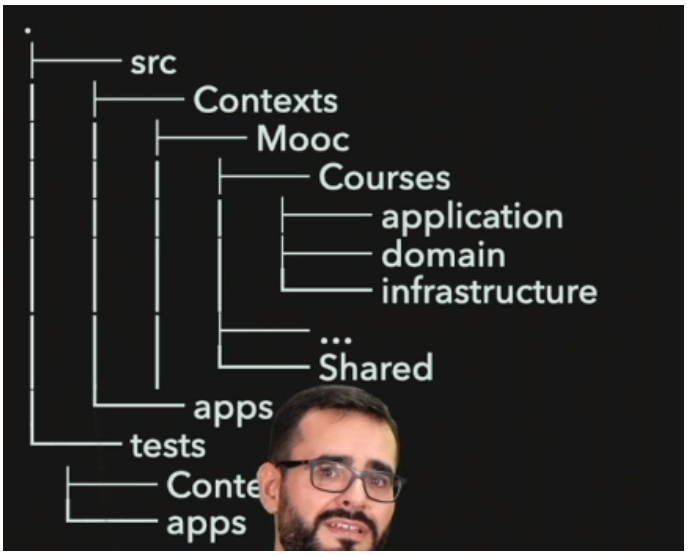
- Para estos tests de *Caja Negra* utilizaremos ***Cucumber***



- Dentro de la carpeta */src/**Contexts***
  - Tenemos el primero de nuestros contextos o ***Bounded Context*** que en este caso es *Mooc*.
    - Se mapea 1 a 1 el Nivel de Jerarquía que vemos en el diagrama para cada *Módulo*. Viendo así el *Módulo* de ***Courses***.
      - En donde cada ***Módulo*** va a tener ya las carpetas referentes a *Application*, *Domain* e *Infrastructure*.
        - Application, domain & infrastructure vendrían a ser cada una de *las capas de la Arquitectura Hexagonal*.
        - De esta forma, el Módulo de Courses es muy *fácil de promocionar* si llegase a crecer demasiado ya que tiene toda la lógica bien estructurada y encapsulada con Arquitectura hexagonal.

- Al ser fácil de *promocionar*, al realizar algún cambio NO tenemos que modificar cada uno de los clientes que hacen uso de lo que expone ese módulo.
  - Promocionar *a Bounded Context*, al hacerlo, todo queda como está gracias a la arquitectura utilizada.



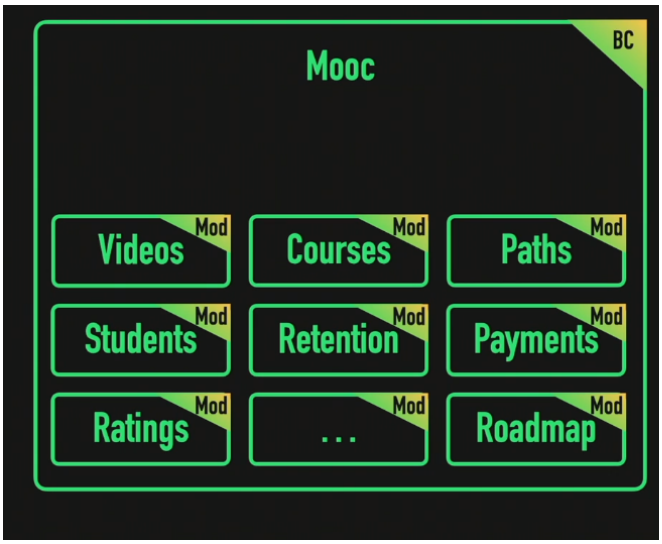


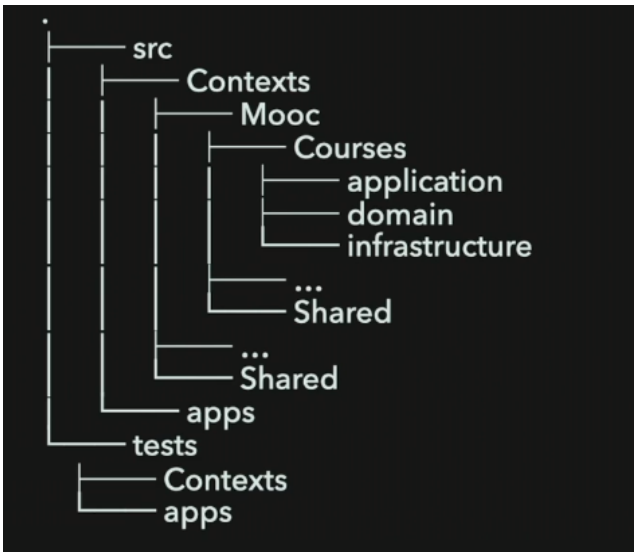
- Las carpetas compartidas o ***Shared*** en este caso lo tenemos dentro de un *Contexto/BC*, por lo tanto tendremos únicamente la funcionalidad que podremos compartir DENTRO del BOUNDED CONTEXT, entre *Módulos*.
  - Es decir, todo el comportamiento que podremos compartir entre módulos de un mismo BC, ya sea, para este caso, Courses, Videos, Paths, Studies, etc.
    - Lo más habitual suelen ser cosas como el *Value Object de CourseId* o algún elemento del *Domain* que se comparte.
      - Ej. ***Value Object*** que es la Clase que modela el identificador de CourseId.
        - Estos *Value Objects:* Para el caso concreto de las *Entity* que modelamos se separan y no dependen de un JOIN o un Lazy para traer la data con el SELECT.
        - Ej. Un *Video* pertenece a 1 *Curso* (N:1): En ***DDD***, en lugar de hacer el tradicional

OneToMany y demás que nos traería todo el Course y el Video, *lo Relacionamos por Identificador (CourseID)*.

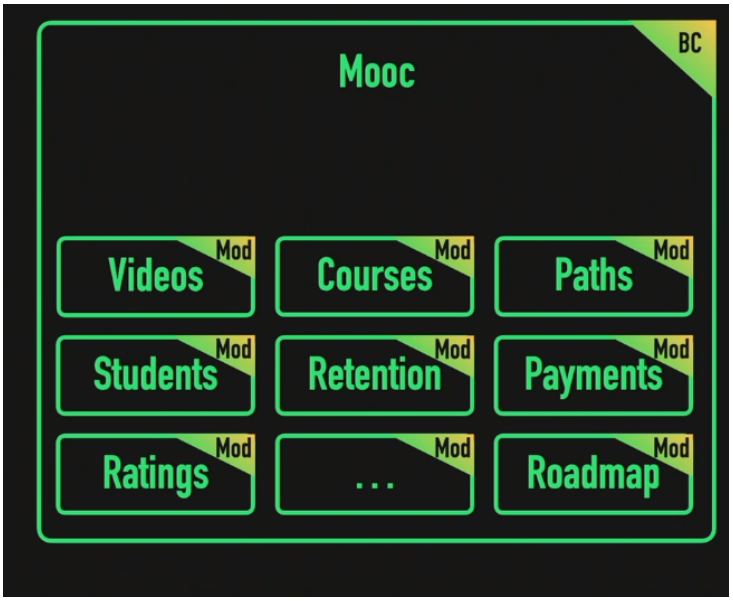
- De esta manera ganamos escalabilidad a nivel de costes computacionales, sino que también a escalabilidad del paso del tiempo a medida que ingresan nuevas personas al equipo, se implementan más casos de uso, etc.
- Con este CourseID evitamos que el *Video* conozca demasiadas cosas del *Course*.

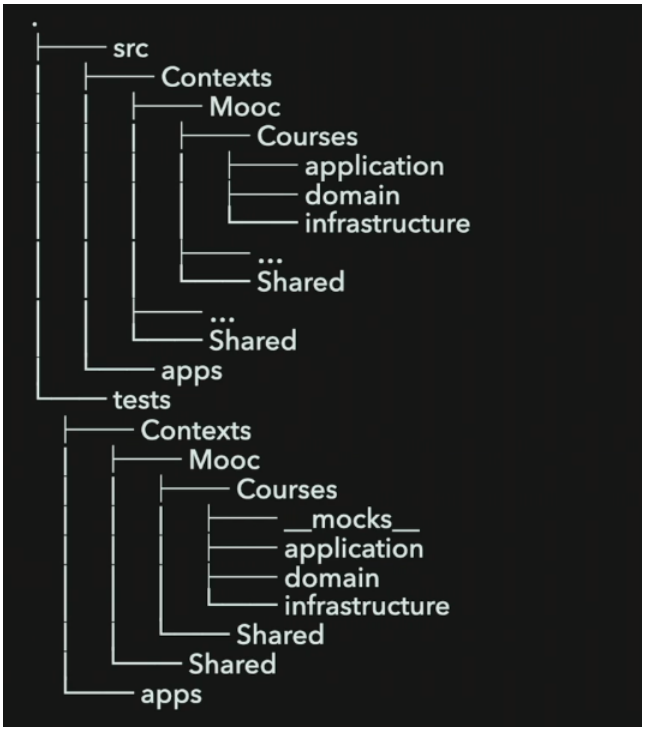
- ***Shared*** a nivel de Contextos con ***BC***:
  - Vamos a tener funcionalidad compartida entre distintos Contextos, ya no solo a nivel de Módulos de cada BC, sino que ya es más global a nivel de BC.
    - Es común encontrar elementos de *Infrastructure* como Conecciones a DB, temas de EventBus o si nos vamos a nivel de *domain* podemos tener elementos básicos como la Clase Value Object Genérica que NO referencia a ningún VO concreto, o la Clase Query Genérica, pero son las Clases de las que van a Extends todas las Query/VO, etc, o también el identificador único UUID q no lo provee por defecto TS.
    - Todo es genérico, nada particular de ningún Context: EmailAddress va en este Shared, pero StudentEmailAddress ya pertenece al Módulo de Students y Hereda del genérico.





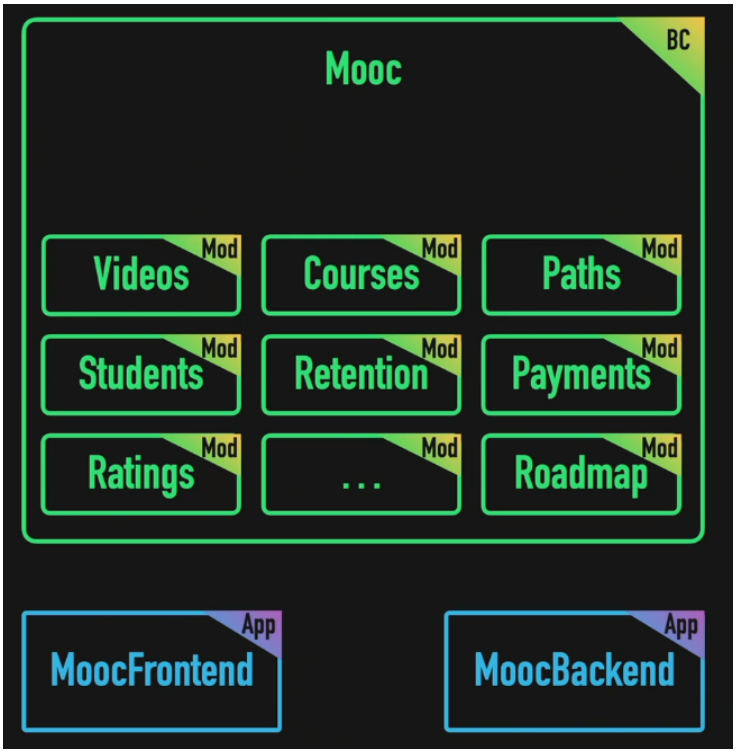
- Dentro de */tests/**Contexts***
  - Los tests dentro de Contexts NO testean puntos de entrada (entrypoints) por lo cual *NO* estamos testeando *Apps* (NO front/back), SINO que Testeamos los *Casos de Uso* por lo tanto vemos cómo se Replica la misma Estructura del context de /src.

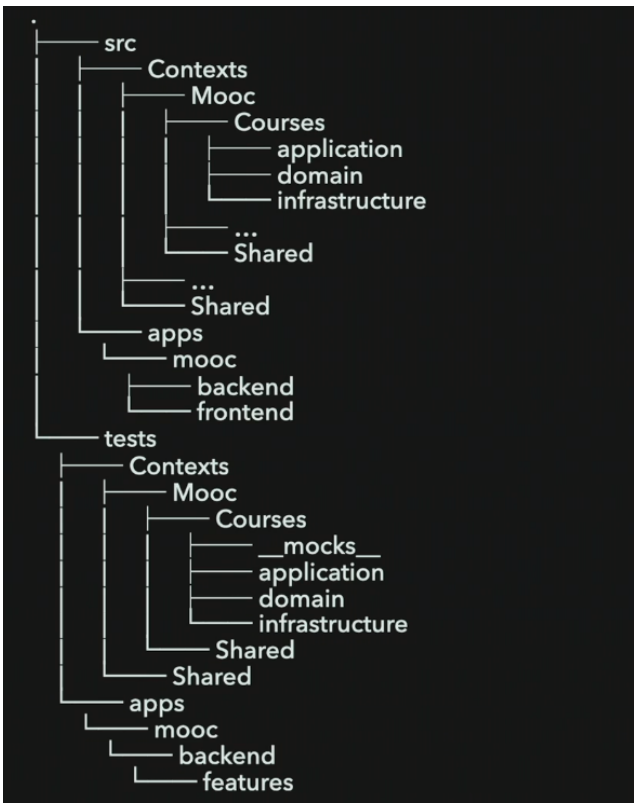




  - Replicamos la estructura del Context/BC en /src a nuestros tests (siguiendo convenciones del testing del front) para así poder identificar fácilmente el test específico que necesitamos.
    - Basado en la Pirámide de Test.

- Dentro de */tests/**apps***
  - Nos encontraremos los tests para las Apps tanto para front cómo back.
    - Ahora si testeamos las Apps
      - Los tests los hacemos con las Features del Lenguaje Gherkin implementado con Cucumber.
        - También podríamos usar Cypress, SuperTest.
      - Al utilizar L. Gherkin, los test que vemos para LP, JAVA, TS van a ser similares, solo cambia la implementación.





1.3. ⚡ Crea tu aplicación TypeScript siguiendo DDD en 5 minutos

Queremos un MONOREPO que soporte distintos Bounded Context que tengan Módulos específicos que sean fáciles de promocionar (a BC).

- Estos Bounded Context (BC) van a ser consumidos por *Apps* tanto de front como de back
  - El Front en Next.js por ejemplo, que hagan uso de todos esos *Casos de Uso*.
  - Todo en pro de conseguir una arquitectura altamente escalable, tolerante a cambios, altamente estable, fácil de promocionar y fácil de integrar nueva gente al proyecto.
- Esto además si queremos podríamos meterlo en diferentes repositorios, uno para cada BC.

- Clonamos el skeleton del repo

2. 👩‍⚕️ Health check de la aplicación: Nuestro primer endpoint

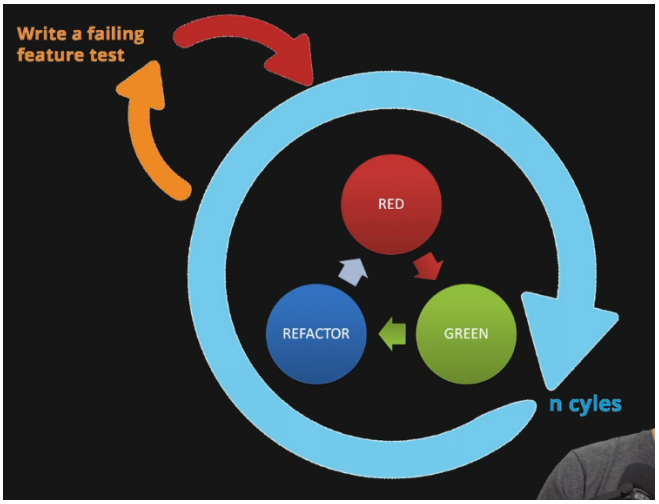
2.1. 🔚📍Crear Endpoint de health check: Controllers asíncronos con Express y declaración de rutas dinámica

- En la Sección creamos el Endpoint que únicamente valida si el Server se levanto o no, NO valida conexiones a DB ni nada
  - Inyección de dependencias, Yo use ***Awilix***

3. ♻️ Desarrollo Outside-in: Implementación del caso de uso para crear curso

3.1. ☝️ Implementación del endpoint y test de Aceptación

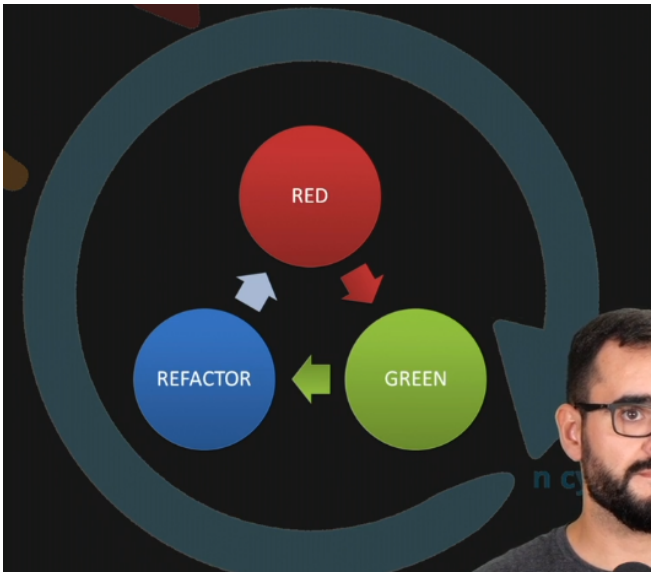
- Usaremos ATDD
  - Que es el ciclo que engloba a TDD



  - Escribiremos el test de Aceptación con Gherkin/Cucumber
    - Luego saltaremos a hacer ***n*** ciclos de Unit Tests

- Entonces, ***ATDD*** es empezar haciendo el *Test de Aceptación*
  - En donde tenemos nuestros Test con Cucumber
    - Testeamos el Controller
    - Luego entramos al Ciclo de Unit Tests

  - Al entrar al ciclo de N Unit Tests es donde Inicia el ***TDD***
    - Y aquí es donde entra *Jest*
      - Aquí debemos colocar en Verde los Unit Test y una vez consigamos eso, Salimos hacia afuera y Colocamos en Verde el *ATDD*.



- g
  - s
    - n
  - s
    - n
      - s
  - s
    - n
      - s

- g
  - s
    - n
  - s

4. 💪 Refactorizando aprovechando el potencial de TypeScript

4.1. 🔀 Mocks más semánticos y mantenibles

- Creamos la Clase que extiende de repository, y esa es el mock que tiene un jest.fn() para testear que cualquier method del Repository sea llamado con los params que requerimos
  - Esto es más semántico y nos da mejores errores
    - Como es el Caso de Uso quien Instancia el Curso, este se compara con el Mock y si falla en algo, nos dice con que fue llamado y no solo 1 true o false
    - Esto mejora tremendamente la experiencia de desarrollo ya que nos ayuda a encontrar rápidamente el error.

- g
  - s
    - n

5. 👤 Modelando el dominio: Agregado Course

5.1. ⛏️ Utilizando objetos Request y Response para comunicarnos con el Application Service

- g
  - s
    - n

- g
  - s
    - n

5.1.1. s

- g
  - s
    - n

- g
  - s
    - n

- g
  - s
    - n

5.2. s

- g
  - s
    - n

- g

        - s
            - n

- g
    - s
        - n

5.3. s

- g
    - s
- g
    - s
        - n
- g
    - s
        - n
- g

5.4. j

- g
    - s
- g
    - s
        - n
- g

5.5. j

- g
    - s
- g
    - s
        - n
- g

5.6. j

- g
  - s
- g
  - s
    - n
- g

5.7. j

- g
  - s
- g
  - s
    - n
- g

6. s

7. g

7.1. j

- g
  - s
- g
  - s
    - n
- g

7.2. j

- g
  - s
- g
  - s
    - n
- g

7.3. j

- g
  - s
- g
  - s
    - n
- g

7.4. j

- g
  - s
- g
  - s
    - n
- g

7.5. j

- g
  - s
- g
  - s
    - n
- g

7.6. j

- g
  - s
- g
  - s
    - n
- g

7.7. j

- g
  - s
- g
  - s
    - n
- g

7.8. j

- g
  - s
- g
  - s
    - n
- g

7.9. j

- g
  - s

- g
  - s
    - n
- g

8. g

8.1. j

- g
  - s
- g
  - s
    - n
- g

8.2. j

- g
  - s
- g
  - s
    - n
- g

8.3. j

- g
  - s
- g
  - s
    - n
- g

8.4. j

- g
  - s
- g
  - s
    - n
- g

8.5. j

- g
  - s
- g
  - s
    - n
- g

8.6. j

- g
  - s
- g
  - s
    - n
- g

8.7. j

- g
  - s

- g
  - s
    - n
- g

8.8. j

- g
  - s
- g
  - s
    - n
- g

8.9. j

- g
  - s
- g
  - s
    - n
- g

9. g

9.1. j

- g
  - s

- g
  - s
    - n
- g

9.2. j

- g
  - s
- g
  - s
    - n
- g

9.3. j

- g
  - s
- g
  - s
    - n
- g

9.4. j

- g
  - s
- g
  - s
    - n
- g

9.5. j

- g
  - s
- g
  - s
    - n
- g

9.6. j

- g
  - s
- g
  - s
    - n
- g

9.7. j

- g
  - s
- g
  - s
    - n
- g

9.8. j

- g
  - s
- g
  - s

    - n
- g

9.9. j

- g
  - s
- g
  - s
    - n
- g

9.10.j

- g
  - s
- g
  - s
    - n
- g

9.11. j

- g
  - s
- g
  - s
    - n
- g

9.12.j

- g
  - s
- g
  - s
    - n
- g

9.13.j

- g
  - s
- g
  - s
    - n
- g

9.14.j

- g
  - s
- g
  - s
    - n
- g

9.15.j

- g
  - s
- g
  - s

    - n
- g