Pinia ORM

Pinia ORM simula una "base de dades" usant les store com a base.

Això ens permet guardar les dades com una store, de forma global, i accedir a elles de qualsevol component. Al guardar el resultat d'un llistat, podem recuperar les dades, d'un element en concret, al accedir a la seva pàgina específica.

A més ens permet realitzar filtres, ordenacions... com si fos una base de dades. Si ho juntam amb la reactivitat els filtres de les nostres pantalles poden resultar trivials.

Implementació

La part d'implementació es la més costosa, ja que s'ha de definir el model de dades i les relacións entre tots els elements.

Abans de continuar deixar clar que aquest model no es connectarà directament a la base de dades real, es a dir, ens podem prendre les llicencies que considerem. Per exemple: si un camp esta definit a la base de dades però no s'usarà mai al frontal es pot obviar

Pinia ORM disposa de dues formes d'implementació per camps o per decoradors. Encara que les dues funciones igual de bé, i ja que usam typescript ens decantarem per els decoradors que queda més clara.

El primer que hem de fer es definir els models de dades. Com es va veure al capítol de l'estructura i ha una carpeta on s'han de posar. Anam a veure un exemple amb un expedient

```
import { Model } from 'pinia-orm'
import { Num, Str } from 'pinia-orm/decorators'

// Hem d'extendre la nostre class amb el Model de pinia-orm
export class Expedient extends Model {
  // També s'ha de definir la variable entity que serà el nom de l'store,
per tant no es pot repetir
  static entity = 'Expedient'

  // Declaració dels camps
  @Num(null) declare id: number
  @Str('') declare num_expedient: string
  ...
}
```

Com s'ha pogut veure, la declaració dels camps del model es realitza indicant el decorador (entre parantesis el valor per defecte), següit de `declare`, el nom de la propietat (convendria coincidis amb el de la api) i finalment indicant el tipus.

Els decoradors disponibles son `@Num`, `@Str`, `@Bool` i `@Attr`, aquest darrer per si no es cap dels tipus anteriors. En general bastant trivial completar la majoria de camps que ens queden.

Un dels casos que encara no hem contemplat son els enumerats. Encara que es pot resoldre amb un `Str`, també ho podem fer amb un enumerat i `Attr`

```
export enum TipusExpedient {
  normal = 'N',
  expres = 'E'
}

export class Expedient extends Model {
  static entity = 'Expedient'

  @Attr(null) declare tipus: TipusExpedient
}
```

Casts

Per a continuar primer hem d'introduir el casts, aquests s'apliquen a un atribut i ajusten el tipus. Per exemple un número o una data poden venir com un string.

Per defecte, pinia ORM inclou els casts més habituals i a la web oficial podeu trobar el detall. També podem definir els nostres propis en cas de necessitat. Així i tot els que més usarem son el: `BooleanCast` i `DateCast`.

Una propietat pot tenir distints decoradors, per tant afegir un cast es tant senzill com afegir el decorador corresponent a la propietat.

```
import { Attr, Cast, Bool } from 'pinia-orm/decorators'
import { DateCast, BooleanCast } from 'pinia-orm/casts'

export class Expedient extends Model {
  static entity = 'Expedient'

  @Cast(() => DateCast)
  @Attr(null)
  declare data: Date
  // Si les dades venen en format string ISO aquest cast deixarà el tipus
Date llest per usar.

  @Cast(() => BooleanCast)
  @Bool(null)
  declare urgent: boolean
  // Es habitual que els booleans puguin venir com a 1/0 en lloc de
true/false
}
```

Relacions

Finalment ens queden les relacions entre objectes. Encara que hi ha de tot tipus ens centrarem en les 1-N i N-M que son les habituals.

Es important indicar que les relacions han de ser entre objectes Model de pinia ORM, no es pot relacionar amb classes normals.

Belongs to (1-N)

Suposem la relació 1-N entre expedient-persona, on una persona pot tenir diversos expedients.

```
import { Attr, BelongsTo } from 'pinia-orm/decorators'

export class Expedient extends Model {
  static entity = 'Expedient'

  @Num(null) declare persona_id: number
  @BelongsTo(() => Persona, 'persona_id') declare persona: Persona
}
```

Com heu pogut notar hem de declarar tant l'id com l'objecte, indistintament de que el back envii únicament una de les dades.

En el cas que el back ens envii l'expedient amb l'objecte persona el que farà pinia ORM es guardar l'objecte de la persona dins l'store la persona i dins l'store de l'expedient l'id de la persona.

Desde l'objecte persona tendriem la relació inversa.

```
import { Attr, HasMany } from 'pinia-orm/decorators'

export class Persona extends Model {
  static entity = 'Persona'

  @HasMany(() => Expedient, 'persona_id') declare expedients: Expedient[]
}
```

De igual manera que el cas anterior, en cas de rebre del back una persona amb un llistat d'expedients, aquests es guardarien dins l'store de l'expedient al guardar la persona.

No es necessari declarar les dues relacions `BelongsTo` i `HasMany` si amb una ens es suficient.

Belongs to many (N-M)

Finalment ens queda la relació N-M `BelongsToMany`. Per aquesta relació si que ens serà necessari haver de declarar la classe pivot.

Suposem la relació usuari-rol

```
```

```
export class UsuariRol extends Model {
  // Per a que funcioni correctament hem de declarar una PK composada,
encara que la realitat pugui tenir un id
  static primaryKey = ['usuari_id', 'rol_id']

  @Num(null) declare usuari_id: number
  @Num(null) declare rol_id: number
}
```

Posteriorment ja podem declarar la relació als objectes Usuari i Rol. Ens limitarem al primer que te més sentit

```
import { BelongsToMany } from 'pinia-orm/decorators'

export class UsuariRol extends Model {

  @BelongsToMany(() => Rol, () => UsduariRol, 'usuari_id', 'rol_id')
declare rols: Rol[]
}
```

Ús

L'ús es semblant al de una base de dades, podem guardar, eliminar i consulta les dades i pinia ORM s'encarregarà de mantenir l'ordre i la duplicitat.

Per obtenir un repositori el primer que hem de fer es declarar-lo de la seguent manera:

```
const ExpedientRepo = useRepo(Expedient)
```

Indistintament de les vegades que cridem això únicament es crearà una instancia del repositori d'expedient, per tant no podem tenir multiples repositoris, sempre serà el mateix compartit globalment.

Per convenció, dins el **fitxer index de la carpeta models**, es creen les instancies per a cada model i es recupera la variable al haves d'usar-la.

Consultes

El sistema de consultes es molt semblant al d'un ORM

```
// Obtenir un registre
ExpedientRepo.find(1)
ExpedientRepo.where('persona_id',25).first()

// Tots
ExpedientRepo.all()
```

```
// Uns registres per id
ExpedientRepo.find([1,2,3])
```

Podem realitzar consultes

```
ExpedientRepo.where('tipus',TipusExpedinet.normal)
  .where('actiu', true)
  .get()

// El where pot tenir com a segon paràmetre una funció
ExpedientRepo.where('data', (data) =>
dataEntre(data,'01/01/2024','01/01/2025'))
// O únic paràmetre una funció
ExpedientRepo.where(expedient => expedient.id > 10 && expedient.actiu ==
false)
```

Per a no allargar no entrarem en molt més detall, podeu trobar més informació a la web oficial

Guardar

Per a guardar simplement hem de cridar al mètode save del respositori i passar com a paràmetre un element o un array d'elements del mateix tipus

```
const expedients = [
  { id: 1, num_expedient: "1/2024"},
  { id: 2, num_expedient: "2/2024", persona: { id: 25, nom: "Miquel" }}
]
// Guardam un array
ExpedientRepo.save(expedients)

// Guardam un element
ExpedientRepo.save({ id: 3, num_expedient: "3/2024"})
```

Es important que els registres tenguin emplenats els camps que formen la PK.

L'execució anterior ens deixarà dues estores amb el contingut seguent:

```
expedients: [
  { id: 1, num_expedient: "1/2024"}
  { id: 2, num_expedient: "2/2024", persona_id: 25}
  { id: 3, num_expedient: "3/2024"}
]

persones: [
  { id: 25, nom: "Miquel" }
]
```

En el cas dels rols tendriem un funcionament semblant:

```
UsuariRepo.save({id: 1, nom: "Joan", rols: [{id:1, nom: "Zona nord"}, {id:
2, nom: "Zona est"}]})
```

Obtindriem:

```
usuaris: [
  {id: 1, nom: "Joan"}
]

rols: [
  {id:1, nom: "Zona nord"},
  {id: 2, nom: "Zona est"}
]

usuariRols: [
  {usuari_id: 1, rol_id: 1}
  {usuari_id: 1, rol_id: 2}
]
```

Finalment indicar semgre que guardem una dada amb un id aquest s'actualitzarà, no es duplicarà. Si la propietat es buida aquesta es buidarà però si no es defineix no es tindrà en compte, vegem un exemple.

```
PersonaRepo.save({id:1, nom: "Marc", llinatge1: "Barceló", llinatge:
"Gelabert"})

PersonaRepo.save({id:1, llinatge1: "Pujol", llinatge2: undefined})

//El resultat serà
persones: [
  {id:1, nom: "Marc", llinatge1: "Pujol", llinatge2: undefined}
]
```

Eliminar

Podem eliminar dades de dues maneres principalment, per l'id o a través d'una consulta.

```
ExpedientRepo.destroy(1)

ExpedientRepo.destroy([1,2,3])

ExpedientRepo.where('actiu', false).delete()
```

Conclusió

Pinia ORM ens permet gestionar les dades al frontal com si d'una Base de dades es tractes. Això no ens ha de confondre ja que no te cap relació ni sincronització amb la base de dades real i tampoc ha de ser una copia d'aquesta, únicament hem de gestionar les dades que ens son necessaries a la sessió.

Si volem gestionar les dades correctament hem de tenir en compte que hem d'afegir els elements, actualitzar-los i eliminar-los després de les respostes positives amb l'API. Això ho veurem més endavant.