

11 Construire un framework WEB

☰ Tags	
🕒 Created	@June 30, 2024 8:05 PM
🕒 Updated	@October 31, 2024 10:19 PM

DISCLAIMER: Ce projet va nous faire écrire du code, en effacer, revenir en arrière, poser des postulats pour les balayer juste après. Le but étant de passer par toutes les étapes que les développeurs passent pour mettre en place un framework tout en découvrant les subtilités de Typescript pour mener à bien ce projet

Pour ce projet, on va utiliser parcel, qui va nous permettre de lancer un fichier TS depuis une page HTML

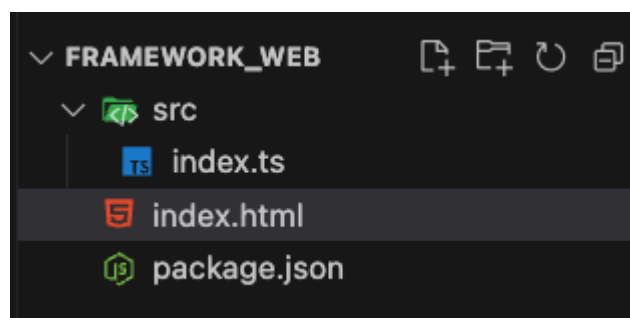
```
npm i -g parcel-bundler  
npm install --save-dev parcel
```

On le lancera avec la commande

```
npx parcel index.html
```

On devra mettre dans notre fichier html

```
<script type="module" src="./src/index.ts"></script>
```



Dans `index.ts`

```
console.log('Hello, world!')
```

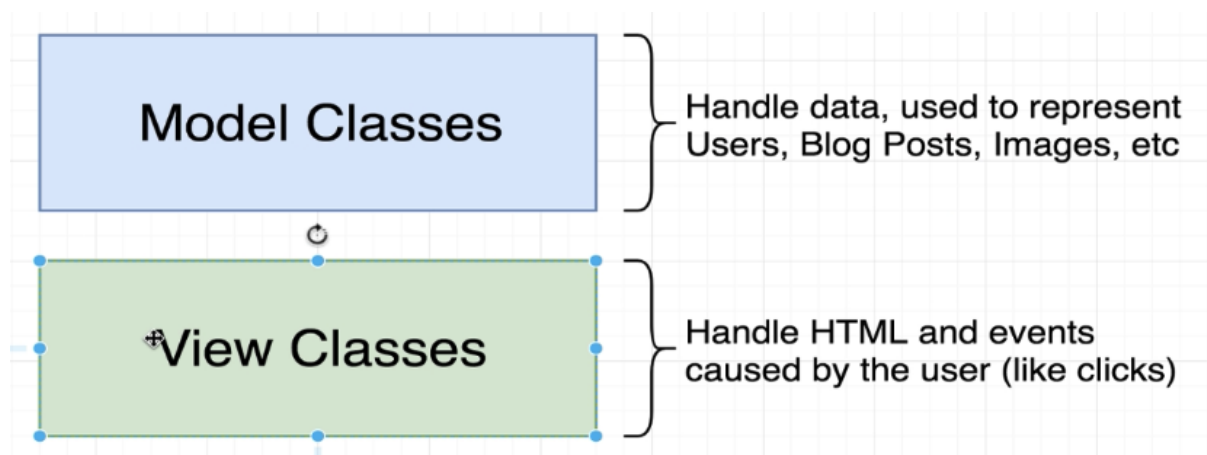
Dans `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="./src/index.ts" type="module" defer></script>
  <title>Document</title>
</head>
<body>

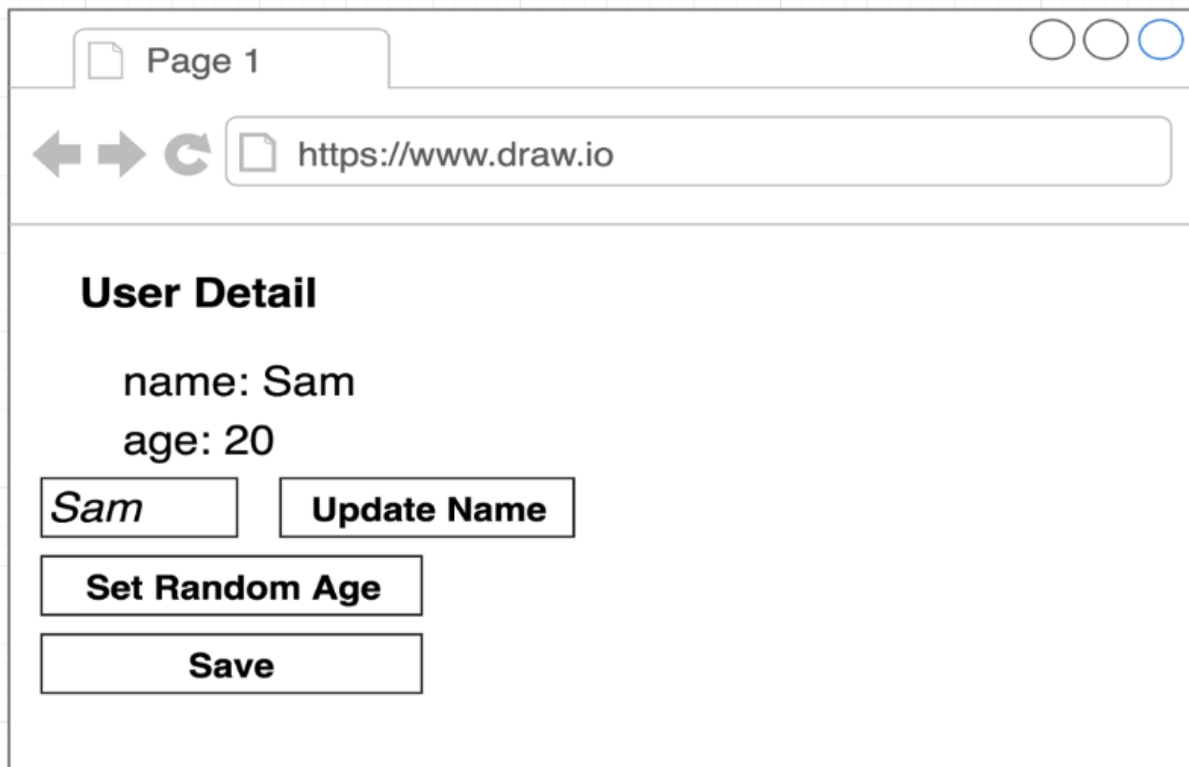
</body>
</html>
```

Si on lance `parcel index.html` on a bien notre console log dans le navigateur

L'idée est la suivante, dans notre framework web on aura 2 types de classes:



Attention, même si il y a un projet derrière, le but ce n'est pas de coder une application en TS, le but ici est de construire un framework en TS qui va nous permettre de créer l'exemple suivant



Donc notre framework doit pouvoir me permettre de créer une classe `User` et les propriétés qui vont avec comme son nom et son âge

Mon framework doit également me donner un moyen de récupérer les attributs de mon `User`, voire de les mettre à jour et donc être capable de les stocker quelque part

J'aurais des boutons qui vont me permettre de mettre à jour le nom et générer un âge aléatoirement, ce qui va mettre à jour ma vue et les données de mon `User`.

Donc ma classe doit avoir la possibilité d'informer mon application quand ces données ont changé

Et enfin ma classe `User` doit être donc capable de persister les données et de les récupérer quand elle le voudra

On devra donc être capable de construire une classe `User` du type:

class User	
private data: UserProps	Objet où stocker les information
get(propName: string): (string number)	Getter de n'importe quel attribut
set(update: UserProps): void	Setter de n'importe quel attribut
on(eventName: string, callback: () => {})	Enregistrer un gestionnaire d'event avec cet objet et en informer l'application
trigger(eventName: string): void	Déclencher un event pour signifier aux autres que quelque chose a changé
fetch(): Promise	Récupérer des données
save(): Promise	Sauvegarder des données

On crée un dossier `models` à l'intérieur de `src` et dans un fichier `User.ts` (à l'intérieur de `models`)

```
interface UserProps {
  name: string;
  age: number;
}

export class User {
  constructor(private data: UserProps) {}
}
```

Pour le getter:

```
get(propName: string): (number | string) {
  return this.data[propName];
}
```

Comme ça dans le `index.ts` je peux faire

```
import { User } from "../models/User";

const user = new User({ name: "myname", age: 20 });
```

```
user.get("name");
user.get("age");
```

Pour le setter:

```
set(update: UserProps): void {
    Object.assign(this.data, update);
}
```

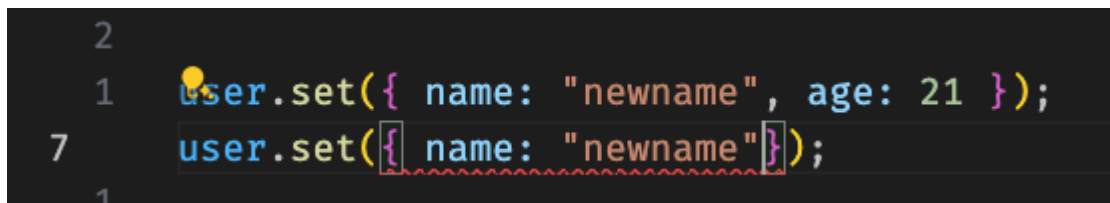
`Object.assign()` va prendre ce qu'il trouvera dans l'objet `update` et mettre à jour l'attribut `data`

Dans `index.ts`

```
const user = new User({ name: "myname", age: 20 });
user.set({ name: "newname", age: 21 });

console.log(user.get("name"));
console.log(user.get("age"));
```

Le problème c'est si je souhaite mettre à jour une seule propriété



```
2
1  user.set({ name: "newname", age: 21 });
7  user.set({ name: "newname" });
1
```

Normal ! notre setter attend un objet de type `UserProps`

Pour ça, on va mettre des propriétés optionnelles

```
interface UserProps {
    name?: string;
    age?: number;
}
```

Listener support

```
on(eventName: string, callback: () => void) {  
  
}
```

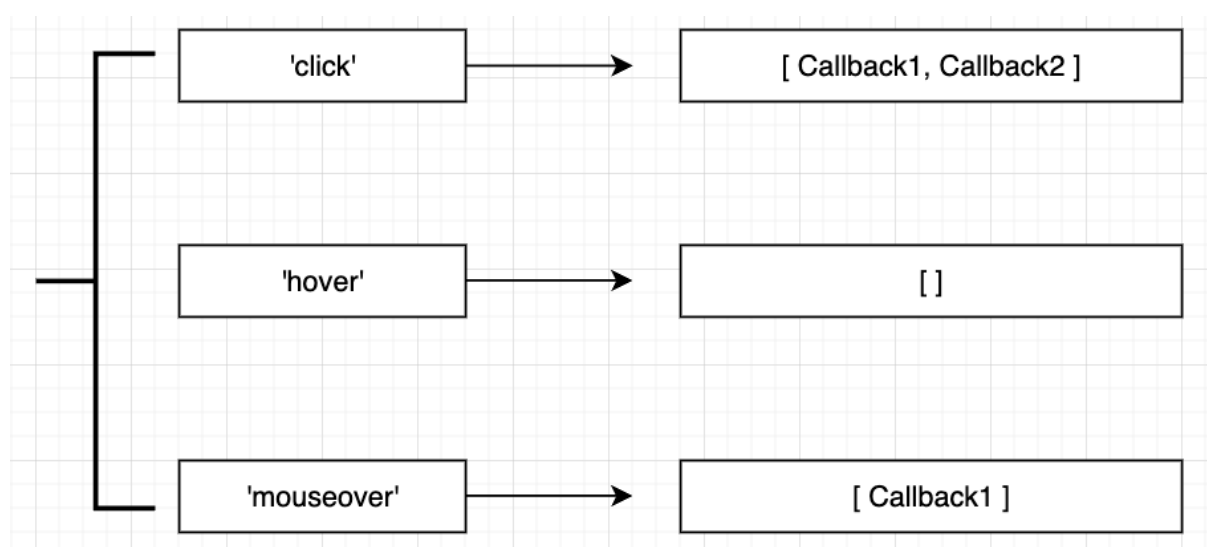
On peut creer notre type `Callback`

```
src > models > User.ts > Callback  
5 interface UserProps {  
4   name: string;  
3   age: number;  
2 }  
1  
6 type Callback = () => void;
```

Notre methode `on()` doit pouvoir gerer tout type d'event, comme le `click` le `hover` le `mouseup` etc... comme ceux qu'on voit dans le `addEventListener()` et lancer la callback

Pour ce faire, on va utiliser un objet pour stocker tous les differents noms d'evenements avec leurs fonctions de rappels (callback)

Les cles de cet objet seront les events et les valeurs seront les listes de callback gerer par cet event



On va donc initialiser un objet `events` dans notre classe User

```
export class User {  
  events: { [key: string]: Callback[] } = {};
```

Dynamic Array creation

```
on(eventName: string, callback: Callback) {  
  const handlers = this.events[eventName] || [];  
  handlers.push(callback);  
  this.events[eventName] = handlers;  
}
```

Dans `index.ts`

```
const user = new User({ name: "myname", age: 20 });  
user.on("change", () => {  
  console.log("Change #1");  
})  
console.log(user);
```

Declencher les events callback

```
trigger(eventName: string): void {  
  const handlers = this.events[eventName]; // on recupere nos callbacks pour l'event precisement  
  if (!handlers || !handlers.length) {  
    return;  
  }  
  handlers.forEach(callback => {  
    callback(); // on lance chaque callback liee a notre event  
  });  
}
```

Dans `index.ts`

```
const user = new User({ name: "myname", age: 20 });

user.on("change", () => {
  console.log("Change #1");
})

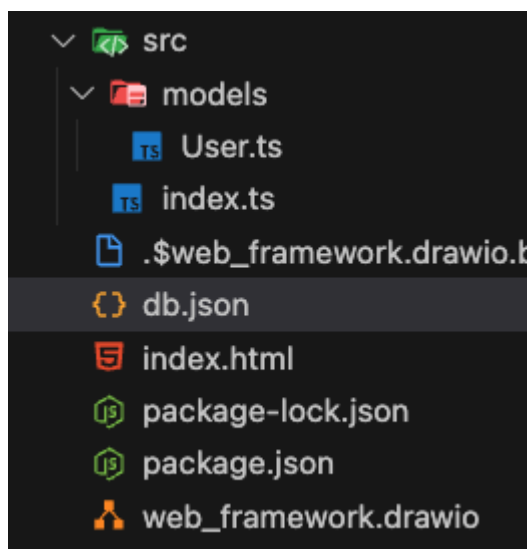
user.trigger("change");
console.log(user);
```

Pour la suite:

```
sudo npm i -g json-server
npm i axios concurrently
```

Ca va nous permettre d'enregistrer nos données User dans un serveur backend, et pour cela on va utiliser `json-server`

Chaque fois que nous exécutons le serveur JSON, nous devons le pointer vers un fichier JSON réel, ou on déposera un fichier contenant des données JSON, ça jouera le rôle de base de données



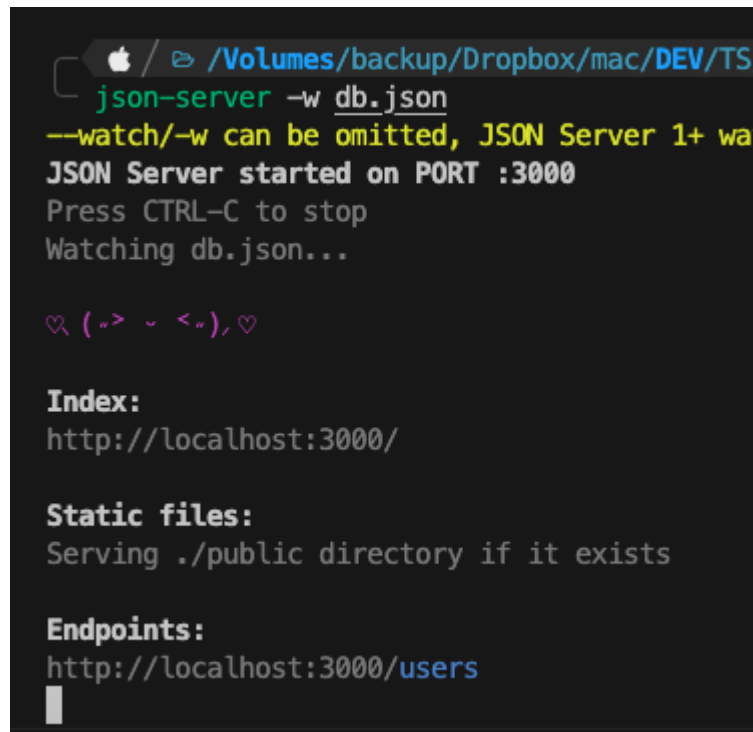
```
{
  "users" : []
}
```



```
}
```

Dans un 2eme terminal on va lancer la commande

```
json-server -w db.json
```

A terminal window on a Mac showing the command 'json-server -w db.json' being executed. The output indicates that the JSON Server is started on port 3000 and is watching db.json. It also displays the index URL (http://localhost:3000/) and the static files directory (./public).

```
Apple /Volumes/backup/Dropbox/mac/DEV/TS
json-server -w db.json
--watch/-w can be omitted, JSON Server 1+ wa
JSON Server started on PORT :3000
Press CTRL-C to stop
Watching db.json...

♡ ( " > ~ < " ), ♡

Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

Endpoints:
http://localhost:3000/users
```

On peut voir ainsi une URL ressemblant a un endpoint d'API

Et dans `package.json`

```
"scripts": {
  "start:parcel": "npx parcel index.html",
  "start:db": "json-server --watch db.json --port 3001",
  "start": "concurrently npm:start:*"
},
```

Dans `index.ts`

```
import axios from 'axios';

const postData = async () => {
```

```

    try {
      const response = await axios.post('http://localhost:3001/users', {
        name: 'John',
        age: 30
      });
    } catch (error) {
      console.error('Erreur:', error);
    }
  };

  postData();

```



```

db.json > [ ] users
2   {
1   "users": [
3   {
1   "name": "John Doe",
2   "age": 25,
3   "id": 1
4   }
5   ]
6   }

```

Du coup tous nos modèles qui seront synchroniser avec le serveur auront besoin d'un ID, mais si on récupère un Utilisateur du serveur, pas besoin de fournir un `id`

```

interface UserProps {
  id?: number;
  name: string;
  age: number;
}

```

Si un utilisateur a un ID c'est qu'il a une representation sauvegardee, et s'il n'a pas d'ID c'est que le modele est nouveau, donc pas sauvegarde

On peut donc implementer `fetch` et `save` dans la classe User

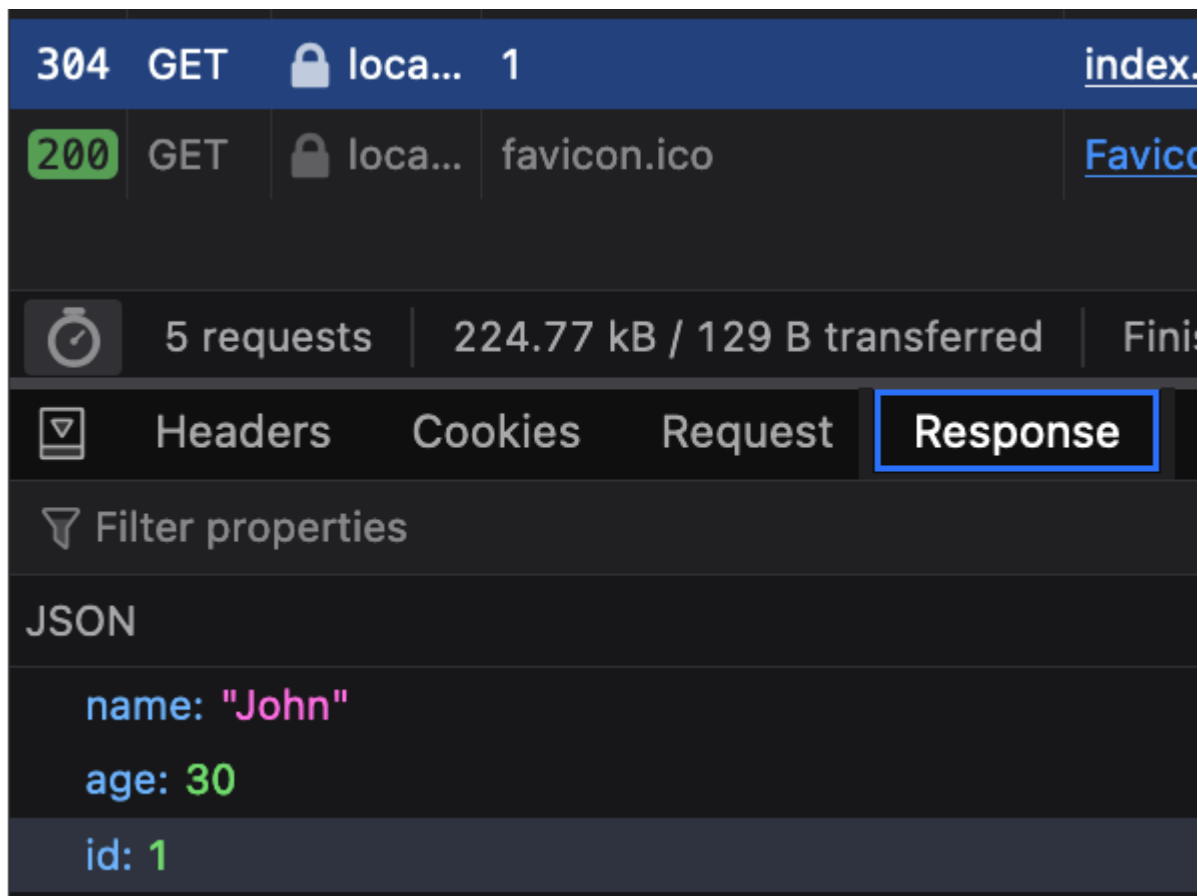
```
    fetch(): void {  
        axios.get(`http://localhost:3001/users/${this.get  
( 'id' )}`)  
            .then((response: AxiosResponse) : void => {  
                this.set(response.data);  
            })  
    }
```

On va tester dans `index.ts`

```
import { User } from './models/User';  
  
const user = new User({id: 1})
```

Comme je fournis un `id` ca indique que cet enregistrement a une representation dans le backend

Je peux voir dans le navigateur que la requete a bien ete lancee



Alors pour l'instant nous n'avons pas de mecanisme qui nous dit que la recuperation des donnees c'est fait avec succes, car la methode `fetch()` ne renvoie rien

On va implementer la methode `save()` mais cette fois-ci on a 2 cas de figures, si le User a un `id` on va sauvegarder les modifications sinon c'est un nouvel User

```
save(): void {
  const id = this.get('id');
  if(id) {
    axios.put(`http://localhost:3001/users/${id}`,
this.data)
  } else {
    axios.post(`http://localhost:3001/users`, this.
data)
  }
}
```

On peut voir qu'on fait appel plusieurs fois a notre URL, mais demain ce sera pas un localhost notre API, on y reviendra

Dans `index.ts` on va tester:

```
import { User } from './models/User';

const user = new User({id: 1})
user.set({name: 'new name', age: 99})
user.save()
```

En sauvegardant, on peut constater la modification



```
"users": [
  {
    "id": 1,
    "name": "new name",
    "age": 99
  },
]
```

On va creer un nouvel user maintenant

```
import { User } from './models/User';

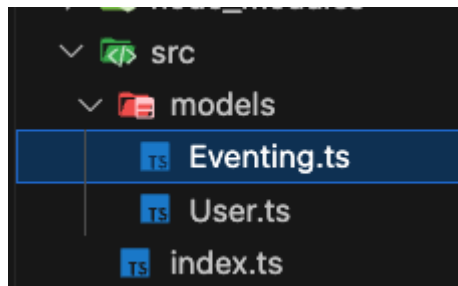
const user = new User({name: "Henry", age: 40})
user.save()
```



```

  {
    "name": "Henry",
    "age": 40,
    "id": 3
  }
]
```

Refactorisation de la classe User pour utiliser la composition



Dans cette classe, on va mettre les methodes `on()` et `trigger()`

```
type Callback = () => void;

export class Eventing {

  events: { [key: string]: Callback[] } = {};

  on(eventName: string, callback: Callback) {
    const handlers = this.events[eventName] || [];
    handlers.push(callback);
    this.events[eventName] = handlers;
  }

  trigger(eventName: string): void {
    const handlers = this.events[eventName];
    if (!handlers || !handlers.length) {
      return;
    }
    handlers.forEach(callback => {
      callback();
    });
  }
}
```

Et notre classe User se resume a:

```
import axios, { AxiosResponse } from "axios";

interface UserProps {
  id?: number;
  name: string;
  age: number;
}

export class User {
  constructor(private data: UserProps) {}

  get(propName: string): (number | string) {
    return this.data[propName];
  }

  set(update: Partial<UserProps>): void {
    Object.assign(this.data, update);
  }

  fetch(): void {
    axios.get(`http://localhost:3001/users/${this.get('id')}`)
      .then((response: AxiosResponse) : void => {
        this.set(response.data);
      })
  }

  save(): void {
    const id = this.get('id');
    if(id) {
```

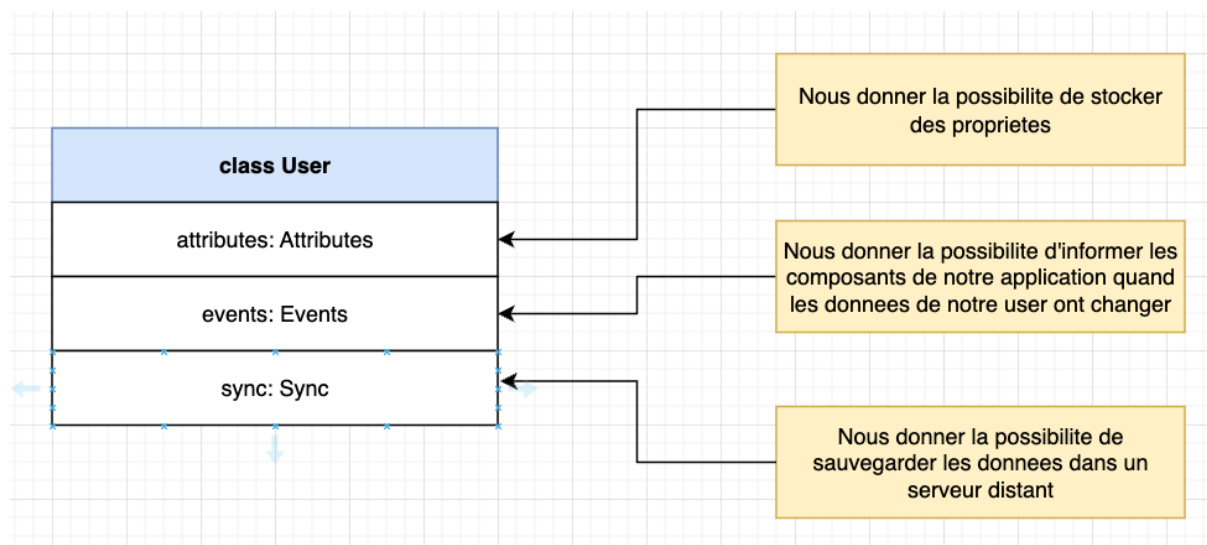
```

        axios.put(`http://localhost:3001/users/${id}`,
this.data)
    } else {
        axios.post(`http://localhost:3001/users`, this.
data)
    }
}
}
}

```

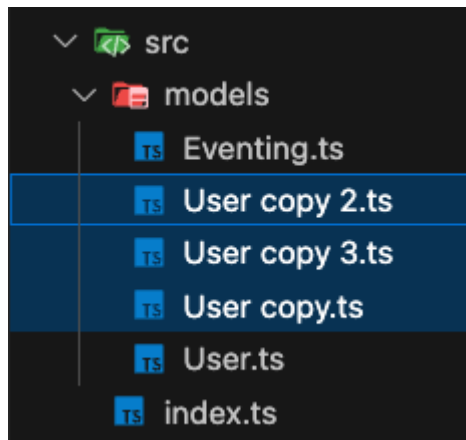
On a donc extrait note gestion d'evenement de la classe user, maintenant il faut connecter notre classe User a cette classe Eventing

Le but est de creer une "super" class qui contiendrait par composition des elements nous permettant:



C'est ce qu'on avait fait dans `stats project` avec la classe `MatchReader`

On va explorer 3 options, en gardant une copie a chaque fois, et garder celle qui serait la meilleure pour nous



Option 1: Accepter les dependances comme second argument de notre constructeur

Option 2: Accepter les dependances dans notre constructeur, definir une methode static pour preconfigurer le User et lui assigner les proprietes

Option 3: Accepter seulement les proprietes dans le constructeur et hard coder les dependances comme proprietes de classes

Dans la premiere copie:

```
export class User {  
  
    constructor(  
        private data: UserProps,  
        private events: Eventing  
    ) {}  
  
    ...  
}
```

Ce qui fait qu'on devra construire le user en lui passant une instance de `Eventing`

(Dans le meme fichier, tout en bas a l'exterieur de la classe)

```
new User({id: 1}, new Eventing());
```

Ce qui n'est pas top en vrai, car a chaque creation de User, il nous faut creer une classe de Eventing

Si on revine a notre dernier diagramme, ca fait que notre classe User aura egalement:

```
new User({id: 1}, new Eventing(), new Attributes(), new Sync());
```

Voyons l'option 2: Accepter les dependances dans notre constructeur, definir une methode static pour preconfigurer le User et lui assigner les proprietes

```
export class User {  
  
  private data: UserProps;  
  
  constructor(private events: Eventing) {}  
  
  static fromData(data: UserProps): User {  
    const user = new User(  
      new Eventing()  
    );  
    user.set(data);  
    return user;  
  }  
  
  ...  
}
```

Ce serait la 2eme approche, elle est plutot bonne, mais ce qui va pas, c'est l'initialisation avec `set(data)`, meme si dans ce cas c'est assez simple, on pourrait tomber sur des classes plus compliquees qui necessitent plus de ligne d'initialisation

Voyons l'option 3: Accepter seulement les proprietes dans le constructeur et `hard coder` les dependances comme proprietes de classes

```
export class User {
  events: Eventing = new Eventing(); // proprietes de classe 'events'

  constructor(private data: UserProps) {}
}
```

De cette maniere, User continue d'avoir acces a `events`

Le probleme avec cette approche, c'est que nous perdons les benefices de la composition, quand on va creer un User, on aura exactement meme la classe `Eventing` servant de module d'evenements a l'interieur

L'avantage de la composition est qu'on peut echanger la source que nous utilisons a la volee

Apres dans ce projet ci, je ne pense pas qu'on devra un jour remplacer notre classe Eventing par autre chose

Donc ce n'est pas risquer de se dire, que la classe `User` utilisera toujours la classe `Eventing`

Donc on va partir sur l'option 3

```
export class User {

  public events: Eventing = new Eventing();

  constructor(private data: UserProps) {}

  get(propName: string): (number | string) {
    return this.data[propName];
  }
}
```

(On garde pour l'instant les copies de User)

Maintenant, le petit souci, c'est que lorsque `on()` et `trigger()` faisait partie de `User`, on pouvait les invoquer directement

```
3
2  const user = new User({name: "Henry", age: 40})
1  user.on('change')
5  |
```

Maintenant on doit passer par `events`

```
import { User } from './models/User';

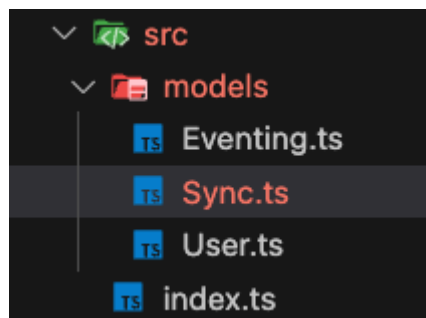
const user = new User({name: "Henry", age: 40})
user.events.on('change', () => {
  console.log('Change!');
})

user.events.trigger('change');
```

NB: Si le `Change!` ne s'affiche pas dans la console, supprimer les fichiers caches dans vscode, ils commencent par un point `.`

Classe Sync

On va maintenant extraire les methodes `fetch` et `save` dans une autre classe



```
import axios, { AxiosResponse } from "axios";
```

```

export class Sync {
  fetch(): void {
    axios.get(`http://localhost:3001/users/${this.get(
      'id')}`)
      .then((response: AxiosResponse) : void => {
        this.set(response.data);
      })
  }

  save(): void {
    const id = this.get('id');
    if(id) {
      axios.put(`http://localhost:3001/users/${id}`,
        this.data)
    } else {
      axios.post(`http://localhost:3001/users`, this.
        data)
    }
  }
}

```

Maintenant les erreurs se trouvent au niveau des `getters` , `setters` et `data`

Notre classe `Sync` essaie de se referer a des proprietes et methodes qui ne s'y trouvent pas

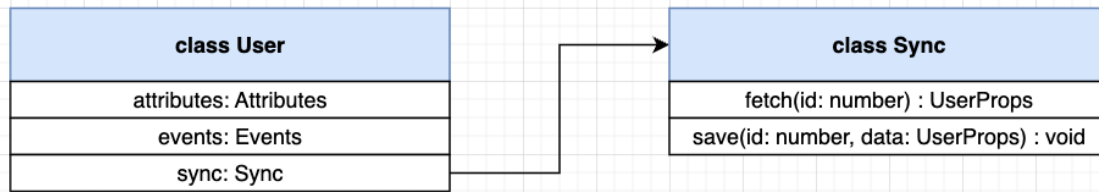
La relation entre Sync et User doit etre bien definie

La aussi nous avons 3 possibilites de resoudre ce probleme:

On va utiliser un type de composition qui s'appelle la `delegation` , la classe User va deleguer cette demande de sauvegarde a la classe `Sync`

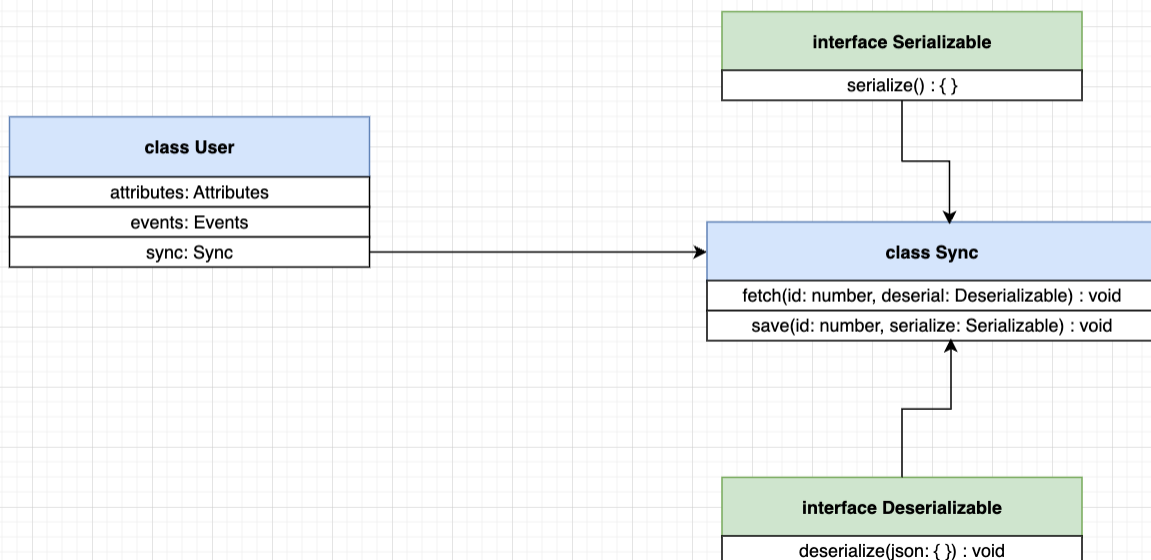
Dans la classe `Sync` nous allons prendre des donnees de l'utilisateur et les enregistrer

Option 1: Les methodes de Sync auront des arguments



Ca fonctionnera, mais le probleme c'est que la classe `Sync` sera configurer pour fonctionner uniquement avec l'utilisateur, trop de couplage.

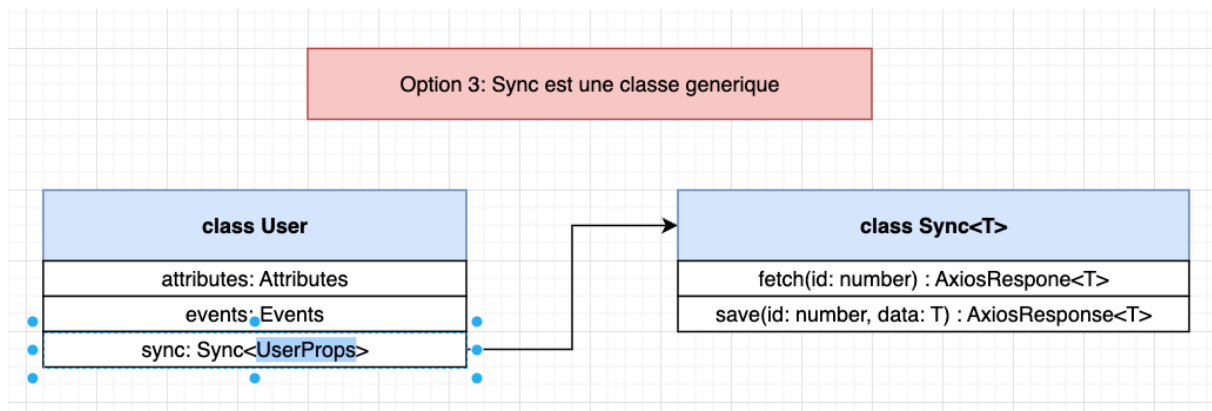
Option 2: Les methodes de Sync attendent des arguments qui satisfassent les interfaces Serialize et Deserialise



`Serializer` c'est convertir les donnees depuis un objet vers un format sauvegardable (json)

`Deserializer` C'est convertir des donnees depuis un format sauvegardable (json) vers un objet

Et enfin l'option 3, on l'a deja vu, c'est l'utilisation de classe generique



Attention, on va faire 2 factorisation, On veut transformer la classe `Sync` en classe generique et d'un autre cote, on veut egalement mettre a jour nos methodes `save` et `fetch` pour recevoir de nombreuses informations

Alors pour refactoriser, on va suivre etape par etape les options 1 a 3

On va exporter dans un premier temps l'interface `UserProps`

```

3  export interface UserProps {
1      id?: number;
2      name?: string;
3      age?: number;
4  }
  
```

Ensuite on va mettre dans le constructeur de `Sync` l'url de l'api

```

export class Sync {

    constructor(public baseUrl: string) {}

    ...

}
  
```

Et on le remplace dans nos methodes:

```

fetch(): void {
  axios.get(`${this.rootUrl}/${this.get('id')}`)
    .then((response: AxiosResponse) : void => {
      this.set(response.data);
    })
}

save(): void {
  const id = this.get('id');
  if(id) {
    axios.put(`${this.rootUrl}/${id}`, this.data)
  } else {
    axios.post(this.rootUrl, this.data)
  }
}

```

Dans la methode `save()`

```

save(data: UserProps): void {
  const { id } = data;
  if(id) {
    axios.put(`${this.rootUrl}/${id}`, data)
  } else {
    axios.post(this.rootUrl, data)
  }
}

```

Maintenant la methode `fetch()`

```

fetch(id: number): void { // ici
  axios.get(`${this.rootUrl}/${id}`) // ici
    .then((response: AxiosResponse) : void => {
      this.set(response.data);
    })
}

```


Mais on a un probleme a le `this.set()` , ca va etre la partie difficile, car fetch n'a pas acces au setter de User

On doit donc trouver un moyen de mettre cette `response` a la disposition de la classe User. Une des facons et de changer la signature de la methode `fetch()` et de renvoyer une promesse et la classe User recupere la response et mets a jour ces donnees

Donc la classe `Sync` ne sera responsable que de faire la demande

```
fetch(id: number): AxiosPromise {  
    return axios.get(`${this.rootUrl}/${id}`)  
}
```

On revient sur la methode `save()` , le souci c'est qu'on n'a pas d'indication que les donnees ont persistees avec succes

On va faire comme avec la methode `fetch()`

```
save(data: UserProps): AxiosPromise {  
    const id = data.id;  
    if(id) {  
        return axios.put(`${this.rootUrl}/${id}`, data)  
    }  
    return axios.post(this.rootUrl, data)  
}
```

On continue en transformant la classe `Sync` en classe generique

```
export class Sync<T> {
```

Et on retire tout ce qui fait reference a `UserProps` pour le remplacer par `T`

```

save(data: T): AxiosPromise {
  const { id } = data;
  if(id) {
    return axios.put(`${this.rootUrl}/${id}`, data)
  }
  return axios.post(this.rootUrl, data)
}

```

Le probleme avec les generiques c'est que par moment on aura du mal a faire reference aux proprietes sur une valeur qui est un type generique

Et ici le type n'etant pas encore fourni, il ne peut pas savoir s'il aura un `id` ou non

Pour regler ce probleme on doit ajouter une contrainte generique, pour dire a TS que quelque soit la valeur de `T` il aura un `id`

Pour cela on doit implementer une interface

```

src > models > Sync.ts > HasId
4   import axios, { AxiosPromise } from
3
2   interface HasId {
1     id: number;
5   }
1
2   export class Sync<T> {

```

```

1
7   export class Sync<T extends HasId> {
1
2     constructor(public rootUrl: string) {}
3

```

Donc quelque soit le type `T`, il aura une contrainte, a savoir contenir un `id`

Un autre probleme va apparaitre quand on va vouloir utiliser cette classe qui sera lier a cette contrainte de type,

On revient a notre classe User, toujours avec notre approche composition

Ce qu'on a fait avec `Eventing` ce n'est pas l'idéal, surtout quand on parle de composition, mais à la limite ça passe

Par contre ça ne va être aussi évident de hard coder `Sync` dans la classe `User`

Car dans `Sync` nous avons une implémentation d'une classe qui va toujours atteindre une API définie, donc on peut très bien se dire qu'on souhaite synchroniser les données d'un modèle vers une source externe en plus d'une API

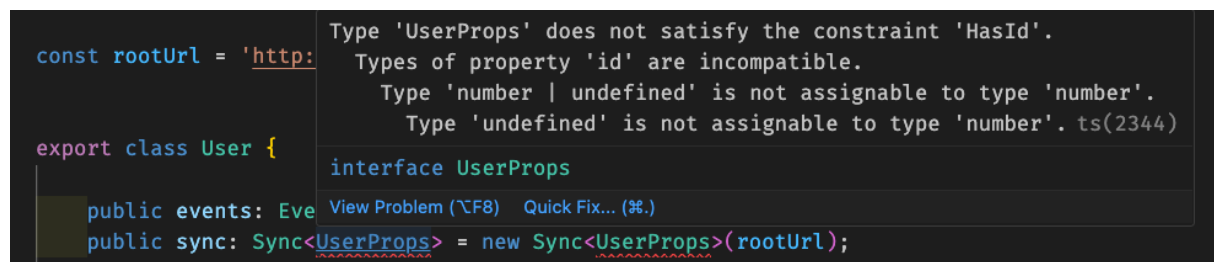
Donc dans un premier temps on va faire comme pour `Eventing`

```
const rootUrl = 'http://localhost:3000/users';

export class User {

    public events: Eventing = new Eventing();
    public sync: Sync<UserProps> = new Sync<UserProps>(rootUrl);
}
```

Donc là on arrive à notre première erreur



En effet `UserProps` ne satisfait pas l'interface `HasId`, car `id` est optionnel dans `UserProps`

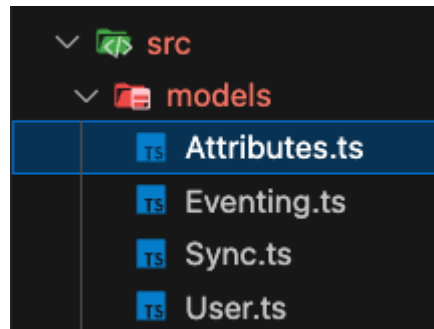
Donc on peut corriger cette erreur dans l'interface `HasId`

```
interface HasId {
    id?: number;
}
```

Maintenant, pourquoi on avait opté pour un `id` optionnel, souvenez-vous c'était dû à la méthode `save()`

Si on a un `id` on met a jour nos donnees, sinon on cree une nouvelle donnee.
On ameliorera notre class User au fur et a mesure que nous avancerons

Extraction des attributs



Et on y met

```
export class Attributes {
  constructor(private data: UserProps) {}

  get(propName: string): (number | string) {
    return this.data[propName];
  }

  set(update: UserProps): void {
    Object.assign(this.data, update);
  }
}
```

La aussi on a une reference a `UserProps` donc on peut transformer cette classe en classe generique

```
export class Attributes<T> {
  constructor(private data: T) {}

  get(propName: string): (number | string) {
```

```

        return this.data[propName];
    }

    set(update: T): void {
        Object.assign(this.data, update);
    }
}

```

On s'arrete 2 secondes sur le getter, il est baser sur `UserProps` donc c'est pour ca qu'on dit qu'il renvoie soit un number soit un string, mais `T` pourrais etre different et le getter pourrait renvoyer un boolean ou un objet

On pourrait faire:

```

get(propName: string): (number | string | boolean) {
    return this.data[propName];
}

```

En dessous de la classe on va mettre:

```

const attrs = new Attributes<UserProps>({id: 5, name: 'Joe', age: 30});
const id = attrs.get('id');

```

Si je survole `id` on peut voir son type

```

5      Object.assign(this.data, update);
4  }
3  }
2
1  const
const id = attrs.get('id');

```

'id' is declared but its value is never read. ts(6133)

const id: string | number | boolean

Quick Fix... (⌘.)

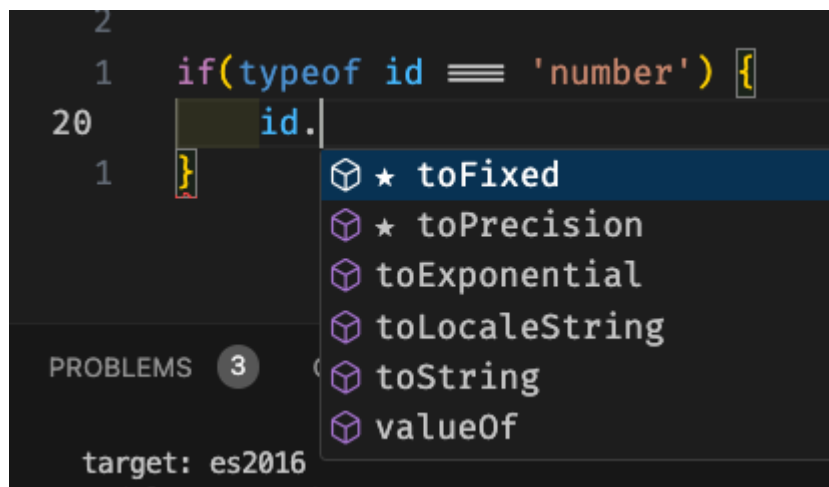
Il me dit que `id` peut etre un `string` `number` ou `boolean`

si je fais:



Je vois que vs code me propose que les methodes communes a string, number et boolean

Je suis limite obliger d'ecrire:



pour voir apparaitre les methodes relatives a `number`

L'ideal serait d'avoir le bon type des que je recupere la propriete de `attrs`

On peut pas typer `id` , on aurait une erreur

```
const id : number = attrs.get('id');
```

On peut utiliser l'assertion de type:

```
const id = attrs.get('id') as number;
```

Mais si on reviens modifier notre code, et qu'on souhaite recupere le `name` et qu'on oublie de modifier l'assertion de type, pour TS il n'y aura pas d'erreur, mais a l'execution il risque d'y en avoir

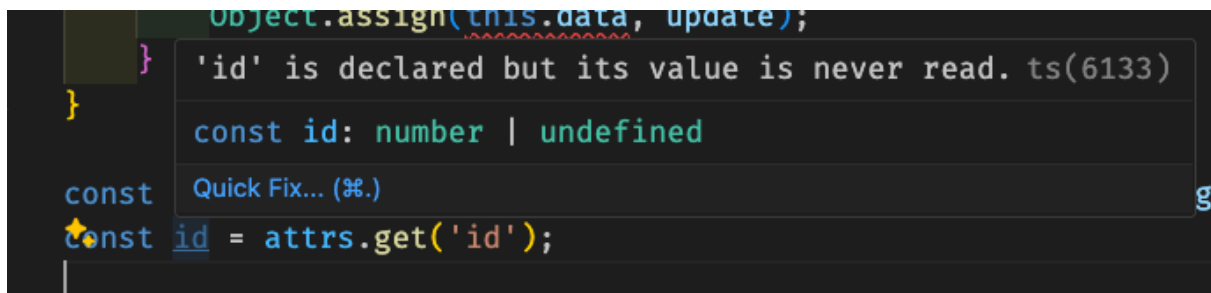
Donc pour regler ce probleme on va devoir utiliser une syntaxe un peu plus compliquee autour des generiques, on va utiliser les contraintes generiques

avancees

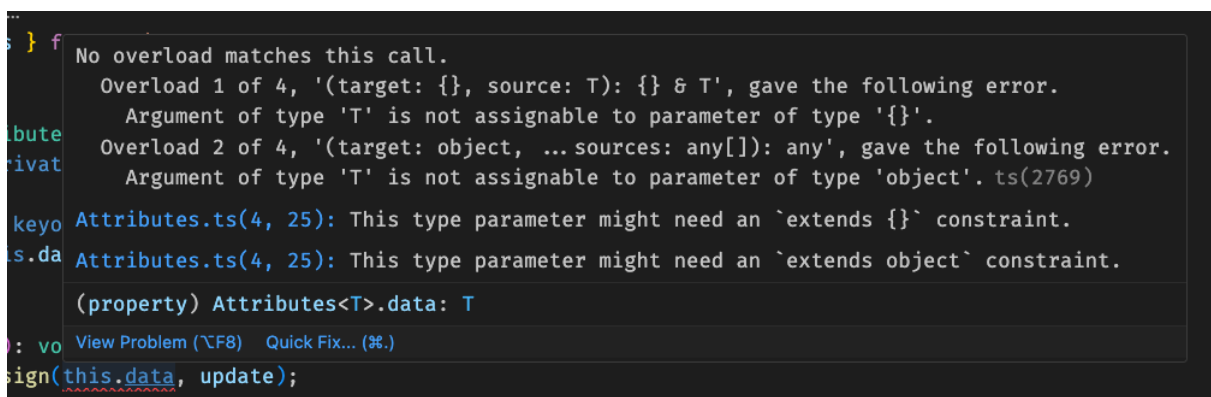
```
get<K extends keyof T>(key: K): T[K] {  
    return this.data[key];  
}
```

Juste apres le `get` on a notre contrainte generique, et en fait on dit ici, que `K` ne peut etre qu'une cle de l'objet `T`

Et la on peut voir



Enfin on a une erreur dans le `setter()`



On le corrige avec

```
export class Attributes<T extends object> { // ici  
    constructor(private data: T) {}  
  
    get<K extends keyof T>(key: K): T[K] {  
        return this.data[key];  
    }  
}
```

```

    set(update: T): void {
        Object.assign(this.data, update);
    }
}

```

Pourquoi ?

TypeScript ne peut pas garantir que `this.data` aura les propriétés nécessaires pour le setter, et cela peut entraîner des erreurs de compilation.

En ajoutant `extends object` à la déclaration générique `T`, on contraint `T` à être un type qui est un objet (c'est-à-dire qu'il ne peut pas être une valeur primitive comme un nombre ou une chaîne de caractères). Cela permet à TypeScript de mieux comprendre les propriétés de `this.data` et de ne plus signaler d'erreur pour l'accès aux propriétés de `this.data`.

Integration de Attributes dans la class User

```

export class User {

    public events: Eventing = new Eventing();
    public sync: Sync<UserProps> = new Sync<UserProps>(root
Url);
    public attributes: Attributes<UserProps>;

    constructor(attrs: UserProps) {
        this.attributes = new Attributes<UserProps>(attrs);
    }

}

```

Testons un peu tout ca dans notre `index.ts`


```
import { User } from "../models/User";

const user = new User({name: 'Henry', age: 30});
```

On veut enregistrer ce `user` dans notre serveur json

Je ferai

```
user.save()
```

J'ai oublier on doit passer par `sync`

```
user.sync.save()
```

Maintenant je dois lui passer les donnees a sauvegarder, bizarre ? car c'est `user` il devrait connaitre ses donnees a sauvegarder

Okay pas grave, on peut les recuperer via `attrs`

```
user.attributes.data
user.sync.save()
```

Ah j'ai oublier, ce n'est plus `data` car c'est une propriete private, on doit passer par les getters pour les avoirs

```
user.attributes.get('id')
user.attributes.get('name')
user.attributes.get('age')

user.sync.save()
```

On doit donc appeler le getter 3 fois

Vous commencez a comprendre que malgre l'approche elegante de notre refactorisation, vous voyez comment ca va etre une galere dans l'index

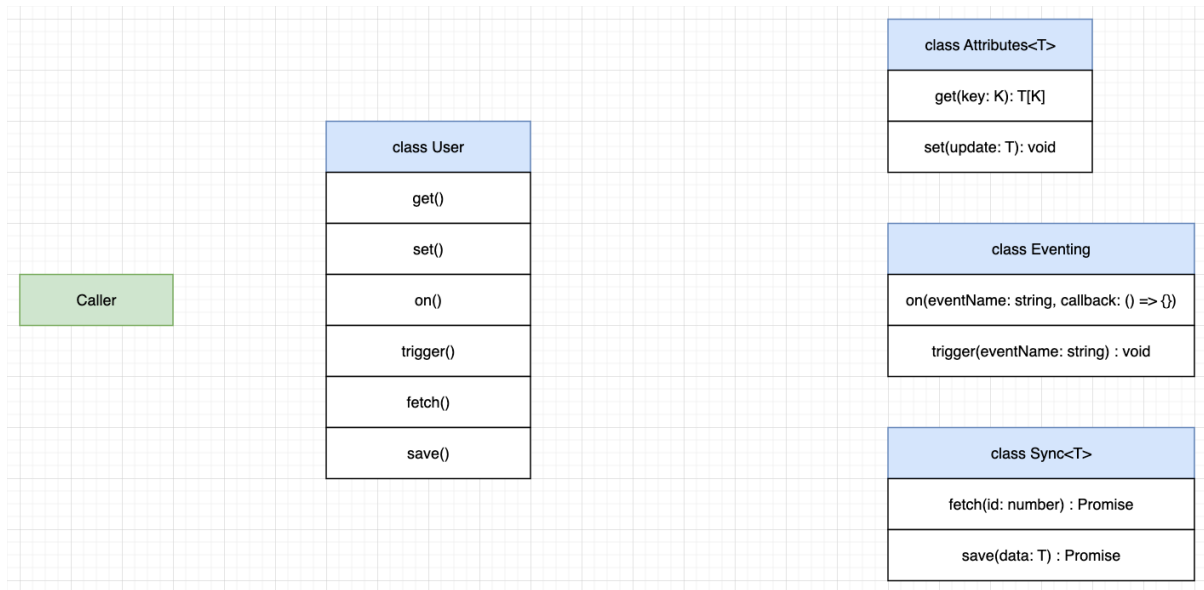
Alors qu'avant, on avait juste a faire

```
user.save()
```

clair et facile !

Ca ne veut pas dire que notre refactorisation est bonne a jetee, mais on doit l'ameliorer

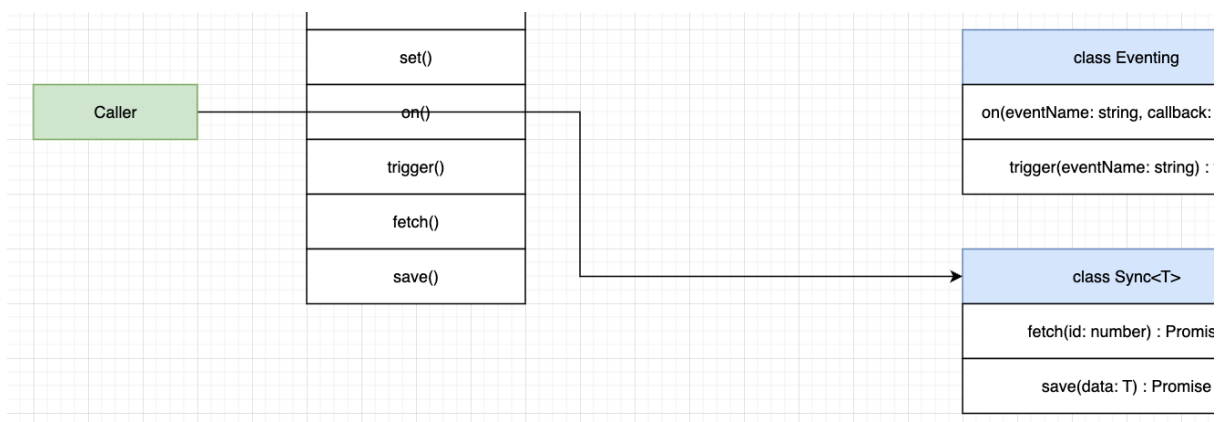
Revoyons notre diagramme:



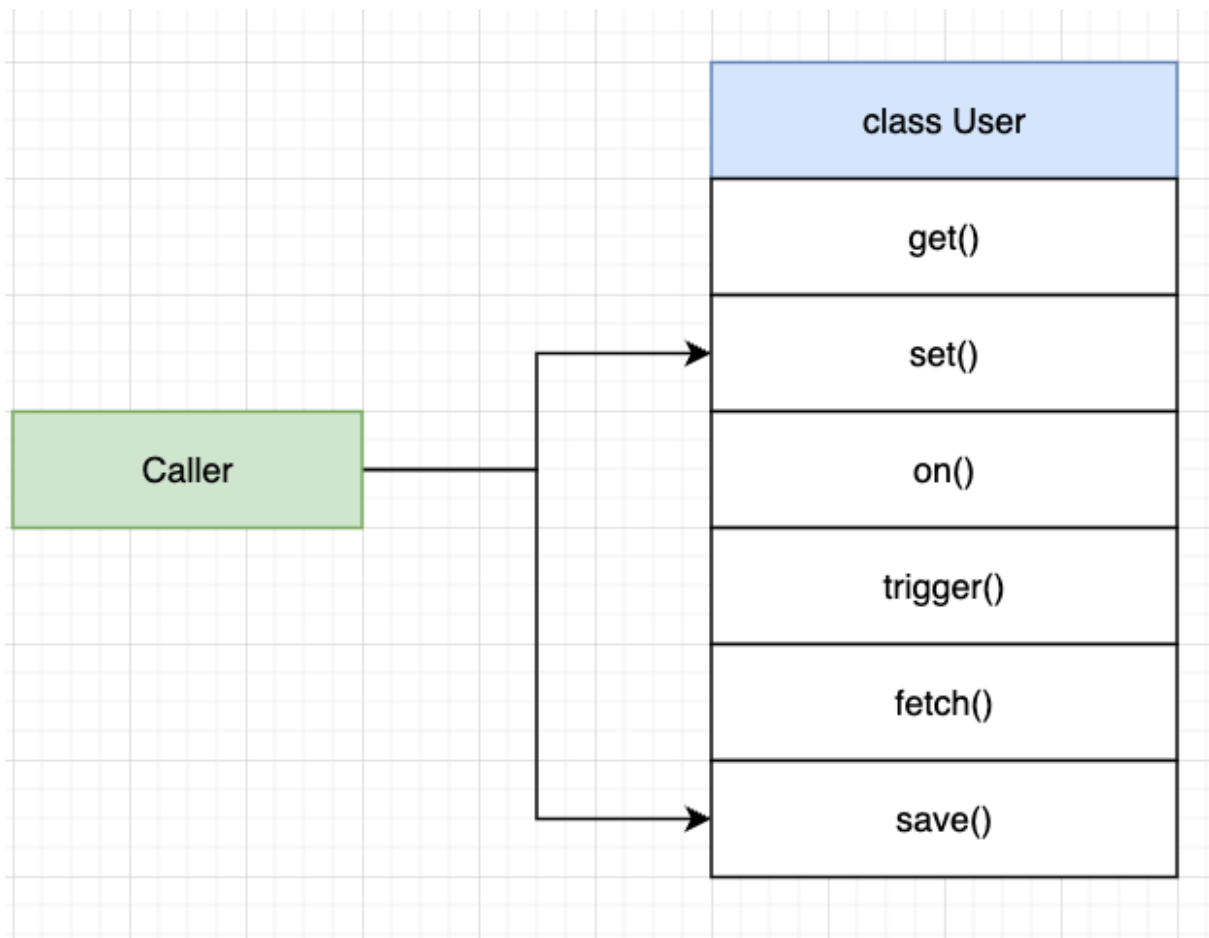
A gauche nous avons ce qu'on va appeler le **Caller** , celui qui va appeler une methode de la classe User

N'oubliez pas que l'idée de départ dans l'utilisation de la composition c'était pour mettre en oeuvre la delegation,

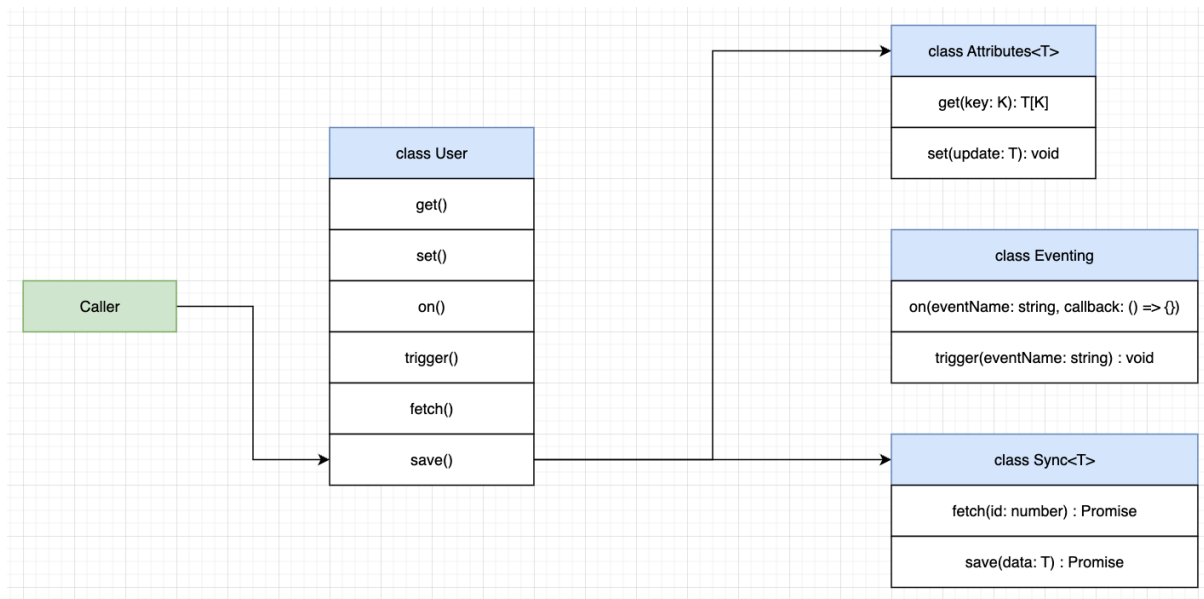
Avec la delegation, on ne veut pas vraiment que Caller atteigne la classe **Sync** directement, ou la classe **Attributes** ou **Eventing**



L'idée avec la delegation est que notre utilisateur de classe va toujours avoir toutes ces même propriétés ou ces même méthodes et le `Caller` va appeler ces méthodes



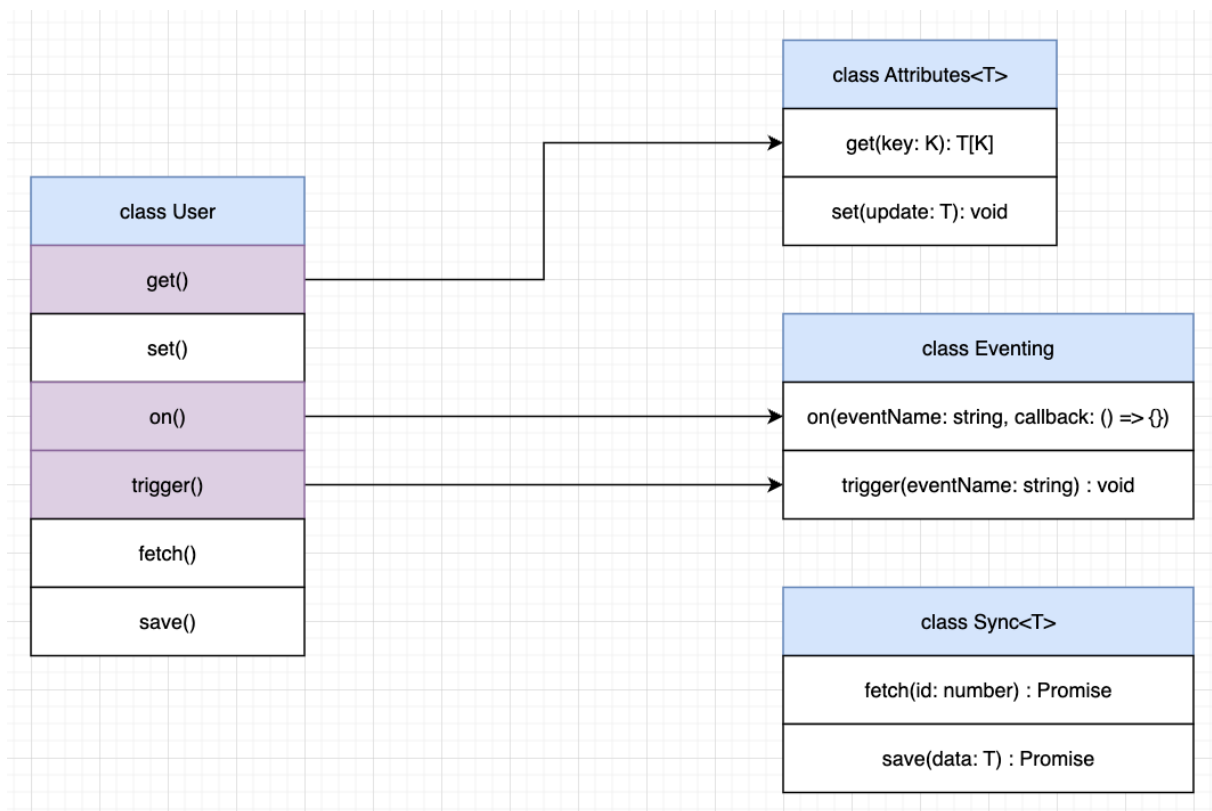
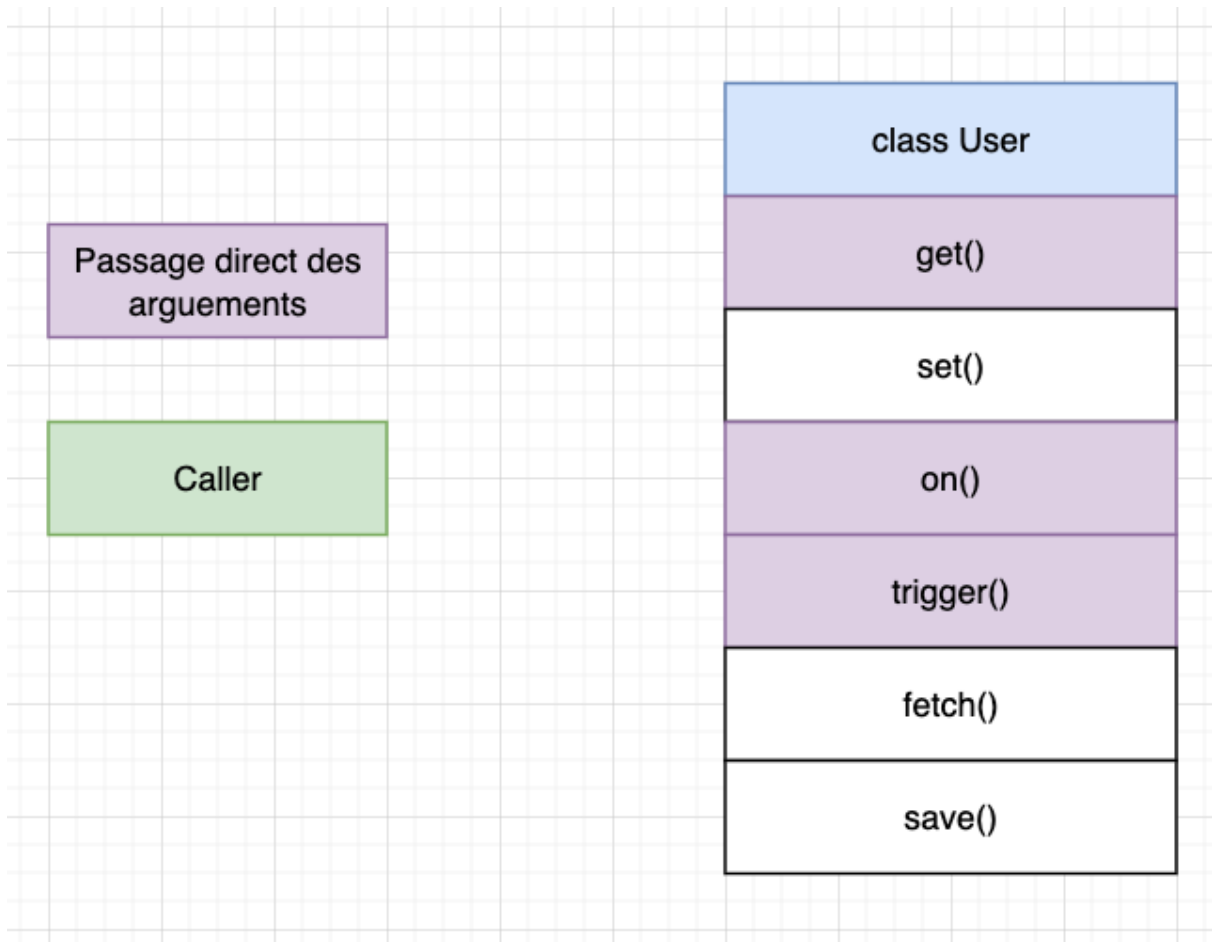
Et à l'intérieur de la méthode `save()` par exemple, la classe `User` va utiliser les différentes classes qu'elle compose pour implémenter un comportement, donc `save()` n'est pas vraiment activé par la classe `User`, mais par un de ses composants, ici `Sync` et `Attributes`



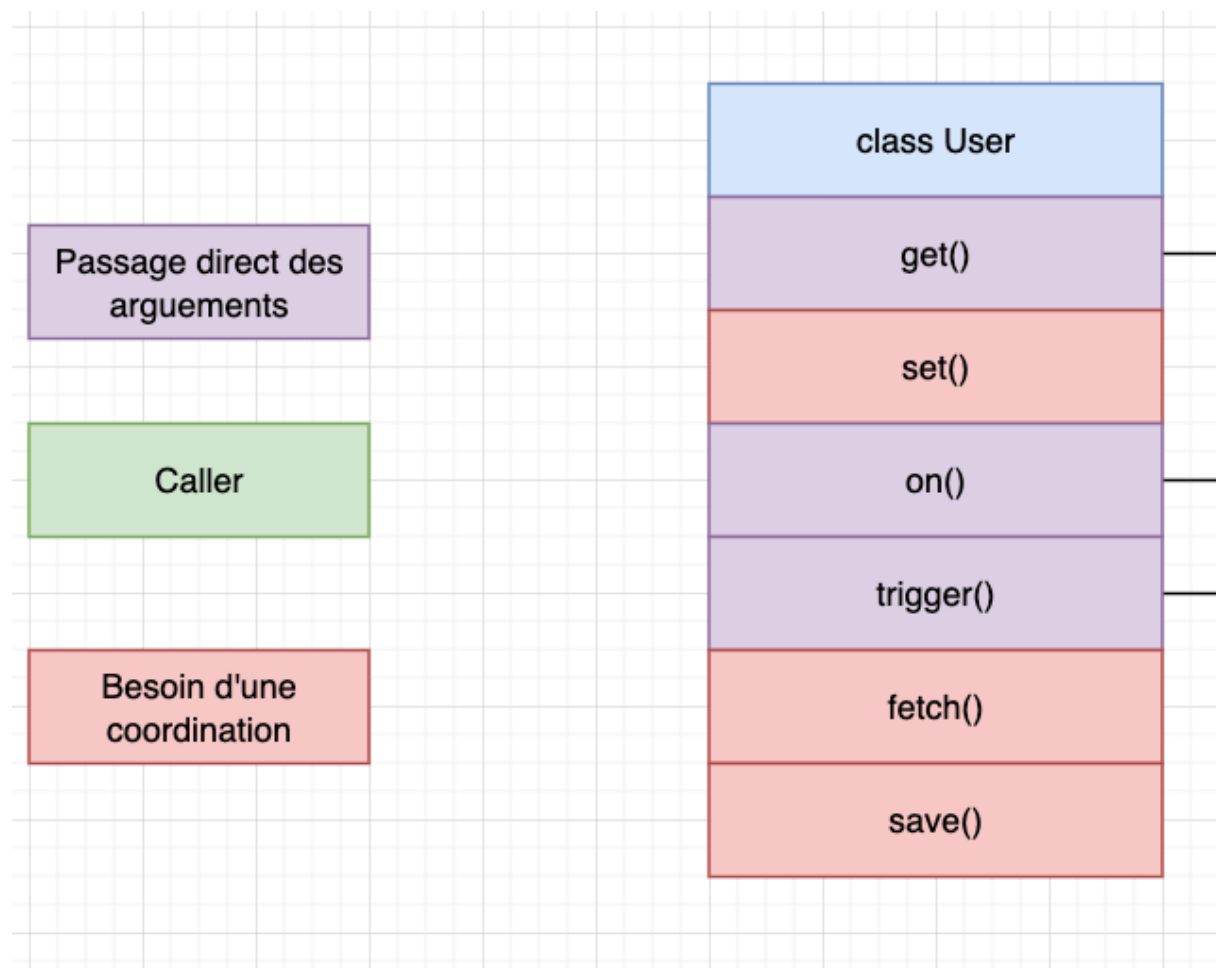
C'est ça la delegation et c'est cette partie qui nous manque, les elements sont la

Donc vous l'avez compris, on va devoir reecrire toutes les methodes initiales de la classe User, mais qui appelleront cette fois-ci le composant adequat, mais avec des ajustements a faire.

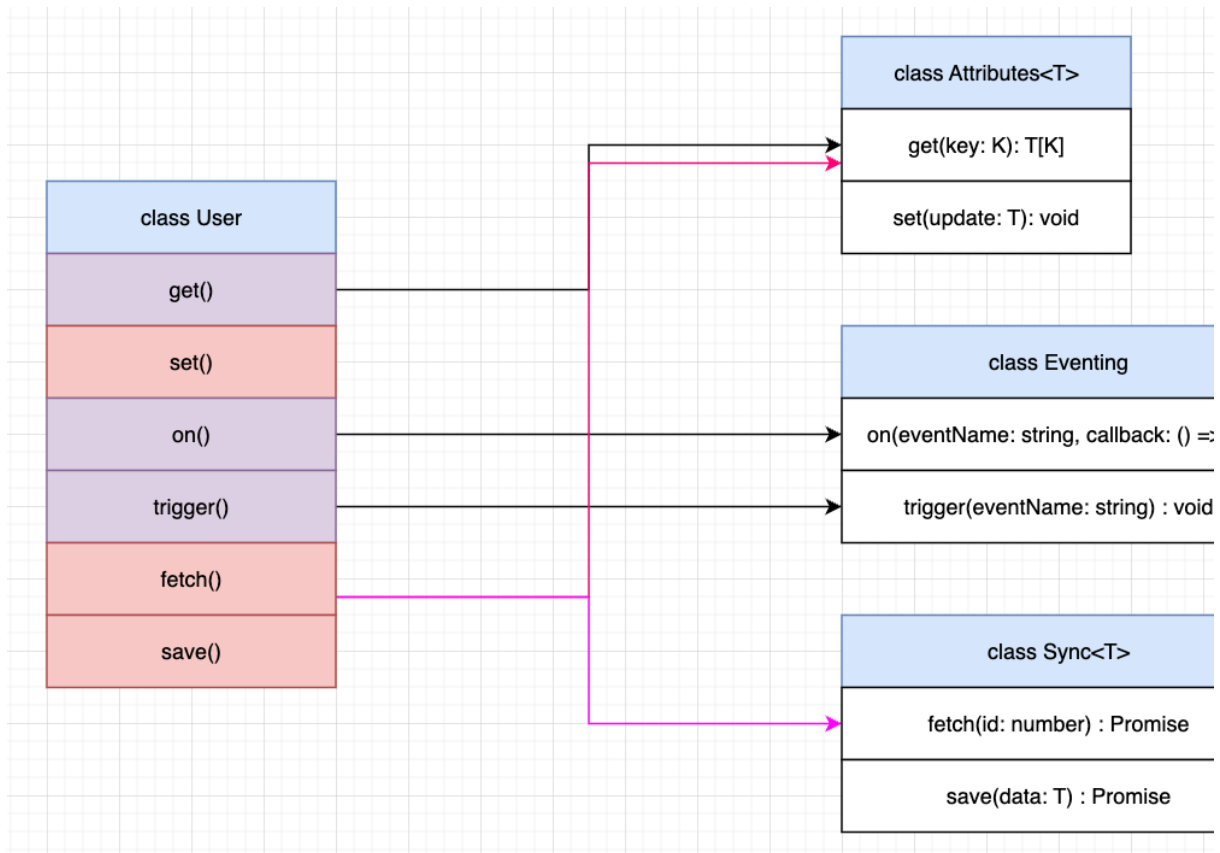
On a des methodes qui vont implementer le passage direct des arguments



Et on a des methodes un peu plus complexes qui ont besoin d'une coordination entre User et ces composants



Par exemple, en appelant `fetch()` on va devoir faire en premier un `get()`



Donc il va avoir une coordination entre `Attributes` et `Sync`

On commence par les methodes les plus simple

Dans la classe `User` on ajoute la methode suivante:

```

on(eventName: string, callback: Callback) {
  this.events.on(eventName, callback);
}
  
```

Alors attention, car si demain on modifie la signature de la methode `on()` et notamment les parametres qu'on y passe, dans la classe `Eventing`, on devra aller toutes les classes faisant appel a Eventing, pour le modifier, car souvenez vous, on fait du passage directe d'arguments, et ca c'est un vrai cauchemar pour un dev

L'ideal c'est d'utiliser un `getter`

A chaque fois qu'on fera reference a la methode `on` de User, nous allons retourner la methode `on` de events

```
get on() {  
    return this.events.on;  
}
```

En fait je renvoie une reference a `events` sur la methode `on()`

Dans `index.ts` on va faire un test

```
import { User } from "../models/User";  
  
const user = new User({name: 'Henry', age: 30});  
  
const on = user.on;  
on('change', () => {  
    console.log('Change #1');  
});
```

En fait, en omettant les parentheses, je n'invoque pas la methode, je retourne juste une reference de cette methode

On va donc faire la meme chose avec les 2 autres methodes

```
get on() {  
    return this.events.on;  
}  
  
get trigger() {  
    return this.events.trigger;  
}  
  
get get() {  
    return this.attributes.get;  
}
```


On a donc fini avec les méthodes de passage, et encore on va pouvoir refactoriser encore plus, on y reviendra

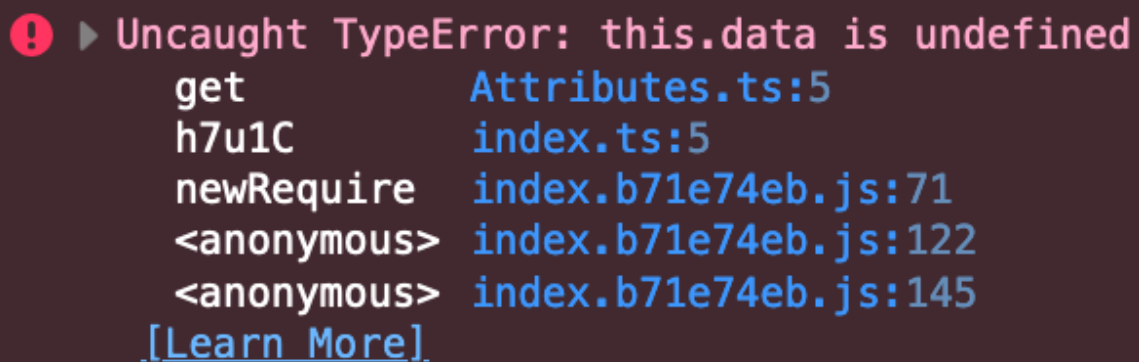
On va tester tout ça dans notre `index.ts`

```
import { User } from "../models/User";

const user = new User({name: 'Henry', age: 30});

console.log(user.get('name'));
```

On ne voit pas d'erreur a priori, mais si on regarde notre navigateur



```
! ▶ Uncaught TypeError: this.data is undefined
    get      Attributes.ts:5
    h7u1C    index.ts:5
    newRequire index.b71e74eb.js:71
    <anonymous> index.b71e74eb.js:122
    <anonymous> index.b71e74eb.js:145
    [Learn More]
```

Pour comprendre cette erreur, il faut se rappeler comment `this` fonctionne en JS

```
const couleurs = {
  couleur: 'rouge',
  printColor() {
    console.log(this.couleur);
  }
}

couleurs.printColor();
```

Et on voit apparaître `rouge` dans la console

Si par contre je fais:

```
const couleurs = {
  couleur: 'rouge',
  printColor() {
    console.log(couleurs.couleur);
  }
}

const printColor = couleurs.printColor;
printColor();
```

On obtient `undefined` dans le navigateur, on dit que `this` a perdu son contexte

Si on revient a notre code

```
console.log(user.get('name'));
```

On passe par le `getter` dans la classe `User`

```
get get() {
  return this.attributes.get;
}
```

Qui nous renvoie dans le `getter` de la classe `Attributes`

Donc c'est comme si on ecrivait:

```
2
1  get<K extends keyof T>(key: K): T[K] {
5  1  return user.data[key];
1  1  }
2
```

Et c'est pour ca que nous obtenons une erreur, donc a ce stade `data` est `undefined`

Maintenant comment resoudre ca ?

En fait notre `getter` dans `Attributes` doit être une fonction liée ou une fonction fléchée

```
get = <K extends keyof T>(key: K): T[K] => {  
    return this.data[key];  
}
```

Avec une fonction fléchée, `this` ne perd plus le contexte, quoiqu'il arrive, `this` fera toujours référence à l'instance `attributes` présent dans la classe `User`

On peut voir dans notre navigateur qu'on récupère bien la donnée

Ce problème sera présent également dans la classe `Eventing`, donc on transforme les méthodes `on()` et `trigger()` en fonctions fléchées

```
on = (eventName: string, callback: Callback) => {  
    const handlers = this.events[eventName] || [];  
    handlers.push(callback);  
    this.events[eventName] = handlers;  
}
```

```
trigger = (eventName: string): void => {  
    const handlers = this.events[eventName];  
    if (!handlers || !handlers.length) {  
        return;  
    }  
    handlers.forEach(callback => {  
        callback();  
    });  
}
```

Maintenant quand choisir des fonctions normales et des fonctions fléchées, en vrai on devrait toujours utiliser des fonctions fléchées

On continue nos tests dans `index.ts`

```
import { User } from "../models/User";
```

```
const user = new User({name: 'Henry', age: 30});

console.log(user.get('name'));

user.on('change', () => {
  console.log('Change #1');
});

user.trigger('change');
```

Pour les autres methodes, on va commencer par `set()`, on va la combiner avec la methode `on()`

En fait, a chaque fois qu'on va vouloir mettre a jour les donnees d'un `User` on va creer un evenement `change`, et on va declencher cet evenement avec la methode `trigger()`

Ca va permettre aux autres elements de notre application, d'ecouter les evenements declenches et de recevoir une notification chaque fois que les donnees de l'utilisateur ont ete modifiees

Donc dans la classe `User` on ajoute la methode `set()`

```
set(update: UserProps): void {
  this.attributes.set(update);
  this.events.trigger('change');
}
```

On va tester dans `index.ts`

```
import { User } from "../models/User";

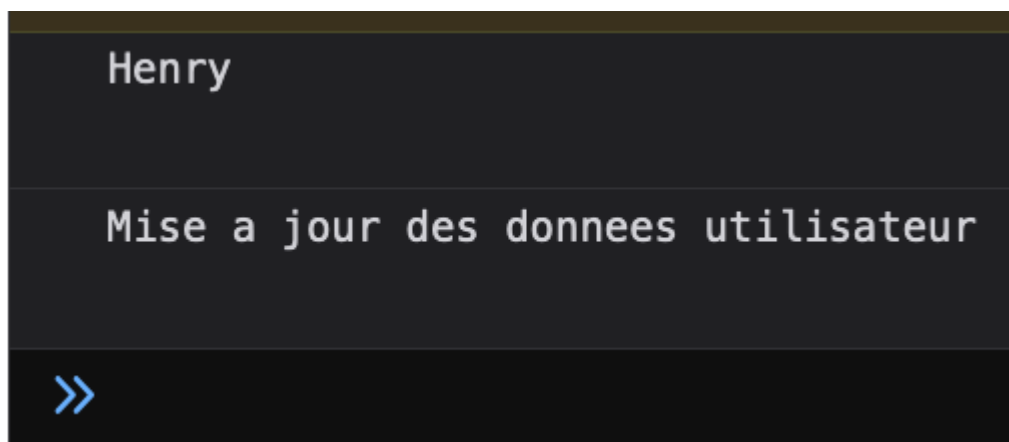
const user = new User({name: 'Henry', age: 30});

console.log(user.get('name'));
```

```
user.on('change', () => {
  console.log('Mise a jour des donnees utilisateur');
});

user.set({name: 'John'});
```

Donc si tout se passe bien on devrait voir dans le navigateur:



Maintenant on va s'occuper de la methode `fetch()`

Elle devra d'abord recuperer l'ID en passant par l'instance `attributes` et ensuite invoquer `fetch()` de l'instance `sync`

Dans la classe `User`

```
fetch(): void {
  const id = this.attributes.get('id');

}
```

N'oubliez pas que l'ID est une propriete facultative, elle peut ne pas etre definie
Si le type de l'ID n'est pas un nombre, en d'autres termes, si elle n'existe pas

```
fetch(): void {
  const id = this.attributes.get('id');
```

```

        if (typeof id !== 'number') {
            throw new Error('Cannot fetch without an id');
        }
    }
}

```

La vérification de type dans la méthode `fetch` est nécessaire pour garantir que l'ID est un nombre et non pas un autre type de valeur, comme `undefined`, `null`, ou même une chaîne vide.

Pourquoi Pas `if (id)` ?

Dans un langage comme JavaScript, une condition `if (id)` serait souvent utilisée pour vérifier si une valeur est "falsy". Cependant, cette approche a des limitations et peut introduire des bugs

En vérifiant explicitement que `id` est de type `number`, on élimine les ambiguïtés liées aux valeurs "falsy".

Donc apres cette verification:

```

    fetch(): void {
        const id = this.attributes.get('id');
        if (typeof id !== 'number') {
            throw new Error('Cannot fetch without an id');
        }
        this.sync.fetch(id).then((response: AxiosResponse)
: void => {
            this.attributes.set(response.data);
        });
    }
}

```

Maintenant, la question c'est est-ce que je `set` en passant par les attributs comme on vient de faire, ou en passant par la classe User

Comme je souhaite declencher un `trigger` il est preferable que je passe par le `set` de ma classe User

```

fetch(): void {
    const id = this.get('id');
    if (typeof id !== 'number') {
        throw new Error('Cannot fetch user with id: ' + id);
    }
    this.sync.fetch(id).then((response) => {
        this.set(response.data);
    });
}

```

Et d'ailleurs on peut le faire également quand on recupere l'ID

```

fetch(): void {
    const id = this.get('id');
    if (typeof id !== 'number') {
        throw new Error('Cannot fetch user with id: ' + id);
    }
    this.sync.fetch(id).then((response) => {
        this.set(response.data);
    });
}

```

On teste dans `index.ts`

```

import { User } from "../models/User";

const user = new User({id: 1});
user.on('change', () => {
    console.log(user);
})

```

```
user.fetch()
```

```
▼ Object { events: {...}, sync: {...}, attributes: {...} }
  ▼ attributes: Object { data: {...}, get: get(key) ↗ }
    ▶ data: Object { id: 1, name: "new name", age: 99 }
    ▶ get: function get(key) ↗
    ▶ <prototype>: Object { ... }
  ▼ events: Object { events: {...}, on: on(eventName, callback) ↗
    trigger: trigger(eventName) ↗ }
    ▶ events: Object { change: (1) [...] }
    ▶ on: function on(eventName, callback) ↗
    ▶ trigger: function trigger(eventName) ↗
    ▶ <prototype>: Object { ... }
  ▼ sync: Object { baseUrl: "http://localhost:3001/users" }
    baseUrl: "http://localhost:3001/users"
    ▶ <prototype>: Object { ... }
  ▶ <prototype>: Object { ... }
```

Il nous reste la methode `save()`

Nous avons dans `attributes` une methode pour recuperer les proprietes une par une,

Il nous faudrait une methode pour toutes les recuperer

Dans la classe `Attributes` on va ajouter la methode suivante:

```
getAll(): T {
  return this.data;
}
```

Et dans la classe `User` on va ajouter la methode `save()`

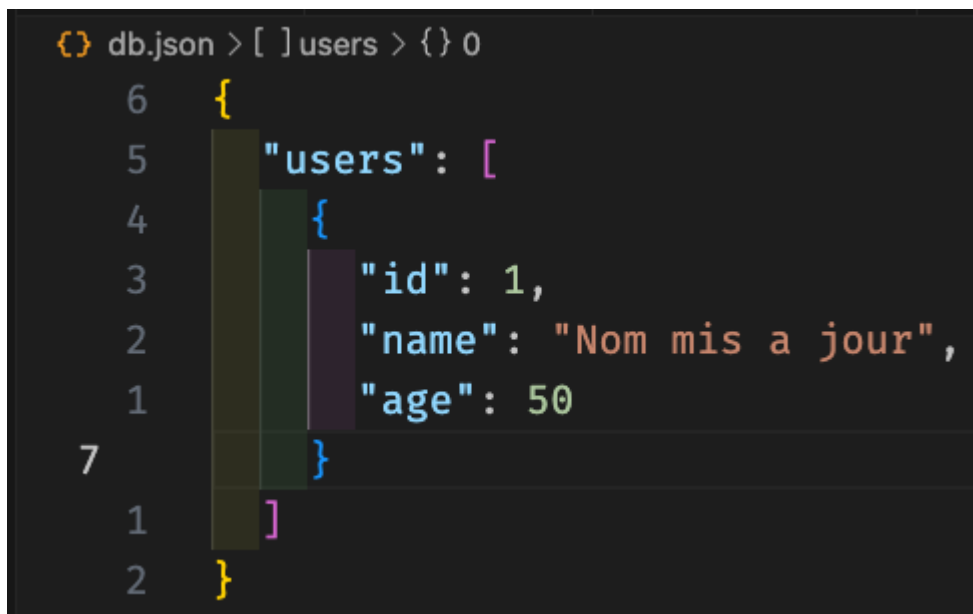
```
save(): void {
  this.sync.save(this.attributes.getAll())
    .then((response: AxiosResponse) : void => {
      this.trigger('save');
    })
    .catch(() => {
      this.trigger('error');
    });
}
```



```
});  
}
```

Dans `index.ts` on teste

```
import { User } from "../models/User";  
  
const user = new User({id: 1, name: 'Nom mis a jour', age: 50});  
user.on('save', () => {  
  console.log(user);  
})  
  
user.save()
```



```
{ } db.json > [ ] users > { } 0  
6 {  
5   "users": [  
4     {  
3       "id": 1,  
2       "name": "Nom mis a jour",  
1       "age": 50  
7     }  
1   ]  
2 }
```

Notre classe `User` est finie, et grace a la composition, on lui a donner pas mal de fonctionnalites

Maintenant on va voir comment creer d'autres types de modeles ayant les memes fonctionnalites que User

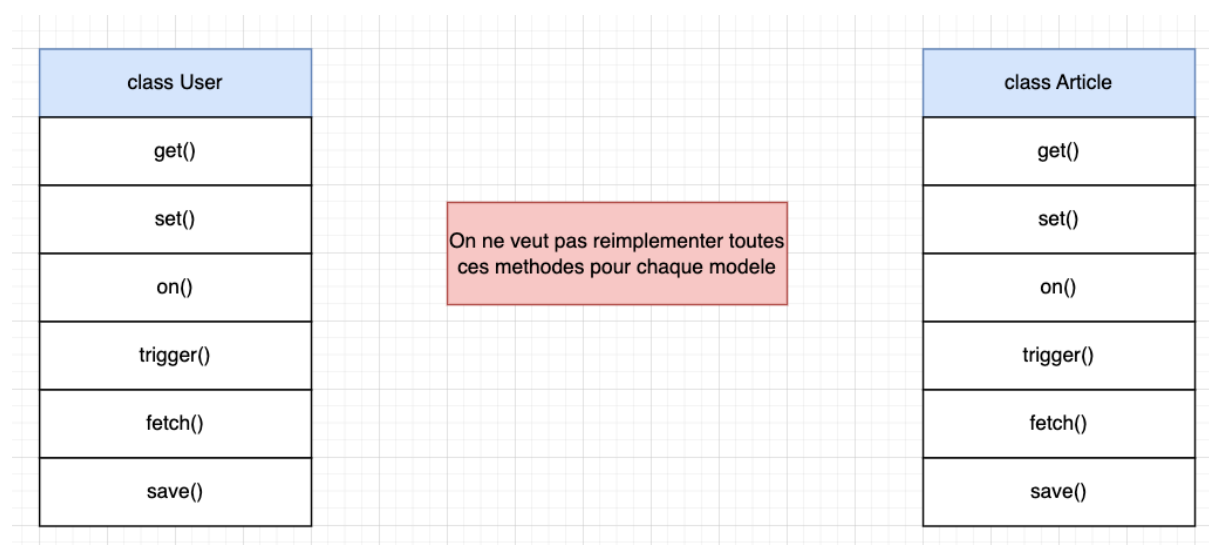
Mais avant ca, revenons sur notre classe User, elle contient des objets qui sont public et ca ne devraient pas etre le cas, on veut forcer les developpeurs a

passer par nos methodes

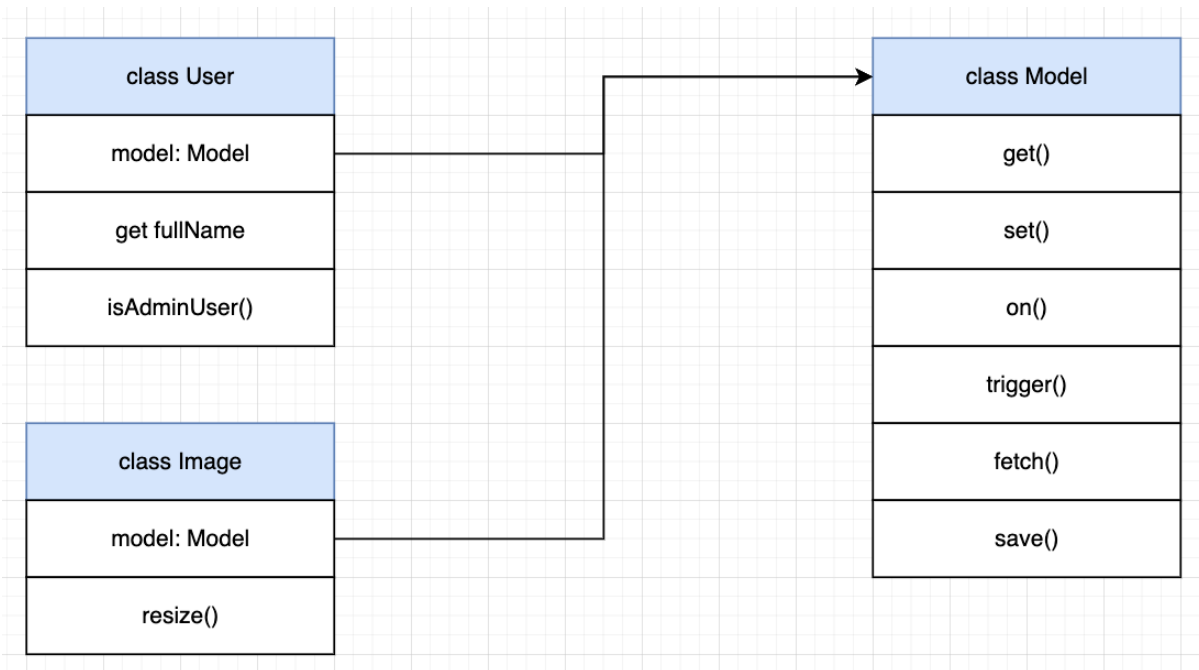
De plus si on part du principe que `events` et `attributes` ne changeront jamais, d'ou le fait de les avoir coder en dur, je ne peux pas etre sur que `sync` le sera egalement, on peut penser a un scenario, ou les donnees seraient synchroniser en local

De plus l'autre probleme c'est que notre classe depend des implementations, des classes reelles, et vous savez maintenant qu'on ne depend jamais des implementations mais de l'abstraction (interfaces)

Autre gros probleme, on veut creer un framework :



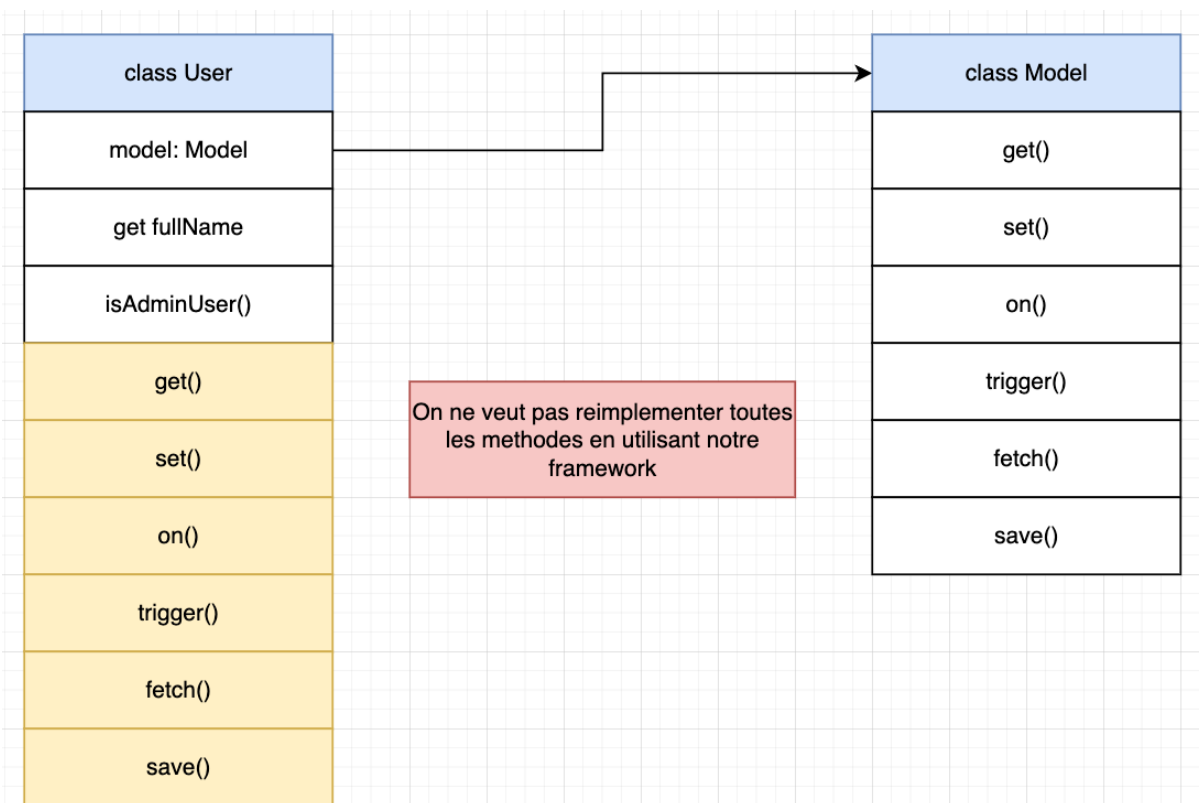
Voici une solution possible



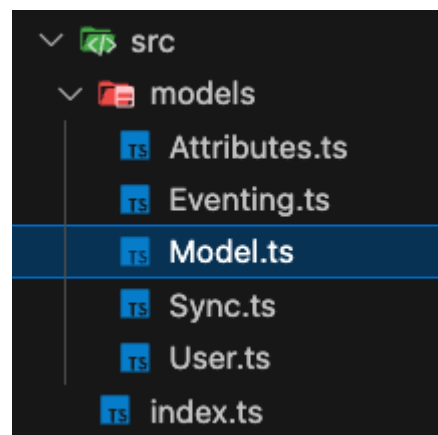
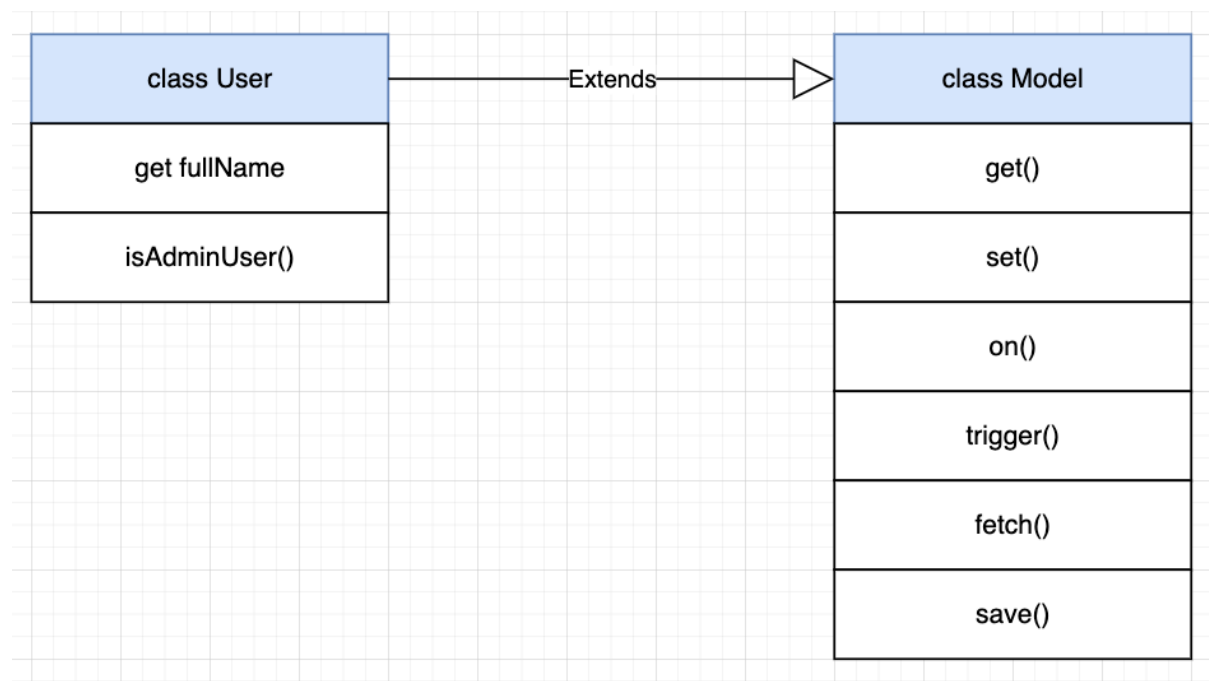
Une classe User qui serait composer d'une instance de model, et d'autres methodes liees a User

On pourrait ajouter ainsi facilement d'autres modeles construits de la meme maniere

Il y a un inconvenient a cette methode



Et on comprend ici que le meilleur scenario c'est l'utilisation de l'heritage



```
export class Model {  
  
}
```

Pour eviter la dependance de classes reelles, on va creer des maintenant nos interfaces

```
// Model.ts  
interface Events {  
    on(eventName: string, callback: () => void): void;  
    trigger(eventName: string): void;
```

```

}

interface ModelAttributes<T> {
  set(value: T): void;
  getAll(): T;
  get<K extends keyof T>(key: K): T[K];
}

interface Sync<T> {
  fetch(id: number): AxiosPromise;
  save(data: T): AxiosPromise;
}

```

Notre classe `Model` aura donc pour constructeur

```

export class Model<T> {
  constructor(
    private attributes: ModelAttributes<T>,
    private events: Events,
    private sync: Sync<T>
  ) {}

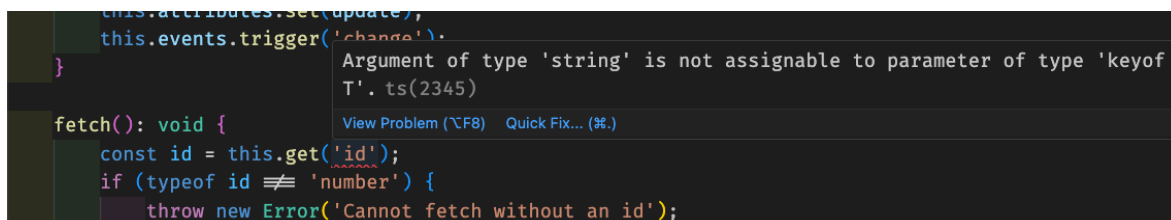
}

```

On prend toutes les methodes presentes dans `User` et on les coupe-colle dans la classe `Model`

On remplace `UserProps` par `T` et on corrige les imports

Il nous reste une erreur:



```

this.attributes.set(update);
this.events.trigger('change');
}

fetch(): void {
  const id = this.get('id');
  if (typeof id !== 'number') {
    throw new Error('Cannot fetch without an id');
  }
}

```

Il suffit de ramener l'interface `HasId` dans `Model.ts` et d'ajouter la contrainte

```
interface HasId {
    id?: number;
}

export class Model<T extends HasId> {
    ...
}
```

On revient a la classe User

le fichier `User.ts` se resume alors a:

```
import { Model } from "../Model";

export interface UserProps {
    id?: number;
    name?: string;
    age?: number;
}

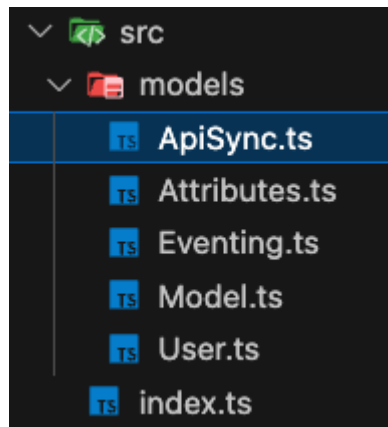
const rootUrl = 'http://localhost:3001/users';

export class User extends Model<UserProps> {

}
```

Il nous reste a finaliser la classe `User`

Juste avant, la classe `Sync` , on va la renommer en `ApiSync`



```
export class ApiSync<T extends HasId> {  
    ...  
}
```

Donc dans la classe User, on ecire une methode static dont son role sera de creer un user

```
import { ApiSync } from "../ApiSync";  
import { Attributes } from "../Attributes";  
import { Eventing } from "../Eventing";  
import { Model } from "../Model";  
  
export interface UserProps {  
    id?: number;  
    name?: string;  
    age?: number;  
}  
  
const rootUrl = 'http://localhost:3001/users';  
  
export class User extends Model<UserProps> {  
    static buildUser(attrs: UserProps): User {  
        return new User(  

```

```

        new Attributes<UserProps>(attrs),
        new Eventing(),
        new ApiSync<UserProps>(rootUrl)
    );
}
}

```

Comme ça dans `index.ts` mon code se resume a :

```

const user = User.buildUser({ name: 'Henry', age: 40 });

user.on('change', () => {
    console.log('User was changed');
});

user.set({ age: 99 });

```

Ainsi avec ce model d'architecture je peux permettre par exemple la persistance des donnees dans une bdd locale

```

    static buildLocalUser(attrs: UserProps): User {
        return new User(
            new Attributes<UserProps>(attrs),
            new Eventing(),
            new LocalSync<UserProps>(rootUrl) // inexistant
        );
    }
}

```

NB: On peut retirer cette methode

On va continuer maintenant et mettre en place la suite de notre framework

Juste avant, en revenant sur la classe `Model.ts` on peut raccourcir nos getters

```

on = this.events.on;
trigger = this.events.trigger;

```



```
get = this.attributes.get;
```

Attention, on ne peut utiliser cette notation que parce qu'on n'a pas initialiser les proprietes, car on passe par un constructeur

Okay on continue

Dans le fichier `ApiSync.ts`, on a la methode `fetch`

Cette methode peut etre appele que si nous avons un `id` et en fait on s'en ait pas rendu compte, mais c'est un gros probleme du style `l'oeuf ou la poule`

Le probleme c'est que quand on demarre notre application pour la premiere fois, nous ne savons pas reellement quels identifiants sont disponibles

C'est ce qu'on fait dans `index.ts` quand on ecrit:

```
import { User } from "../models/User";

const user = User.buildUser({id: 1});

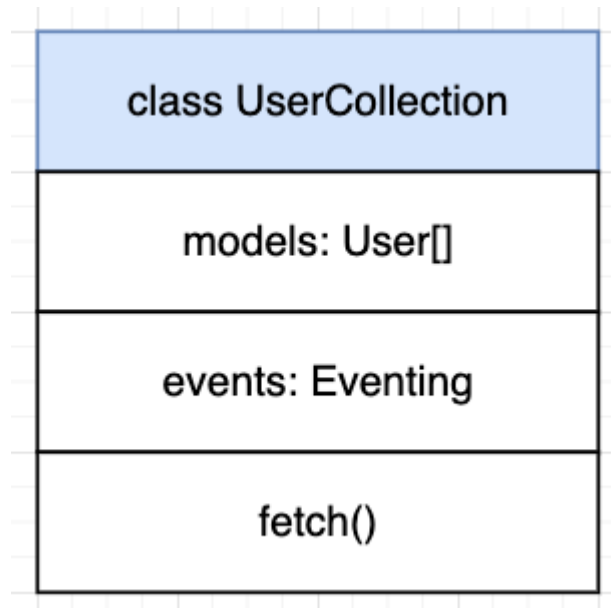
user.on('change', () => {
  console.log(user);
});

user.fetch();
```

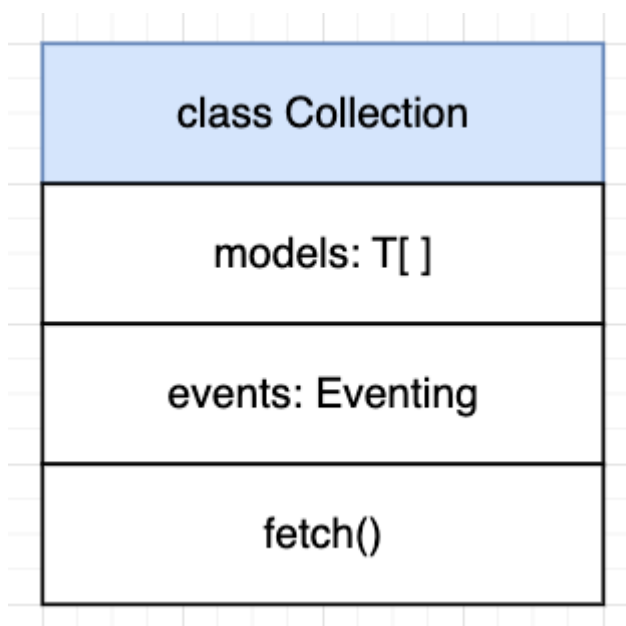
Si on a une grande collection d'utilisateurs stockes dans notre serveur json, on ne peut pas obtenir une liste de tous ces differents identifiants et savoir quels utilisateurs existent

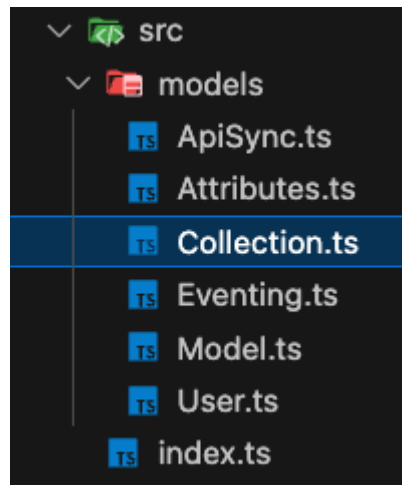
On a besoin d'un moyen pour determiner quels utilisateurs existent dans notre back-end

On va creer une classe `UserCollection`



Mais je sais des maintenant que je vais avoir besoin d'une collection des differents models que je vais creer, donc autant factoriser des maintenant





```
import { Eventing } from "../Eventing";
import { User } from "../User";

export class Collection {
  models: User[] = [];
  events: Eventing = new Eventing();

  get on() {
    return this.events.on;
  }

  get trigger() {
    return this.events.trigger;
  }
}
```

Ici on ne peut pas utiliser la version courte de nos getters, quand on initialise les propriétés (ce qui n'était pas le cas dans la classe `Model`, on avait un constructeur)

On continue notre classe `Collection` en lui ajoutant la méthode `fetch()`

```
fetch(): void {
  axios.get()
```

```
}
```

On va avoir besoin de l'URL, qu'on va mettre dans le constructeur au moment d'instancier la Collection

```
export class Collection {  
  models: User[] = [];  
  events: Eventing = new Eventing();  
  constructor(public rootUrl: string) {}  
}
```

On finit notre methode, mais dans cette methode, on va devoir boucler sur toutes nos donnees JSON pour `build` un user et le push dans notre array `models`

```
fetch(): void {  
  axios.get(this.rootUrl)  
    .then((response: AxiosResponse) => {  
      response.data.forEach((value: UserProps) => {  
        const user = User.buildUser(value);  
        this.models.push(user);  
      });  
      this.trigger('change'); // On informe le reste de l'application d'un changement  
    });  
}
```

Dans `index.ts`

```
import { Collection } from "../models/Collection";  
  
const collection = new Collection("http://localhost:3001/us
```

```

ers");

collection.on("change", () => {
    console.log(collection);
})

collection.fetch();

```

Okay, ça fonctionne, on peut transformer notre classe en classe generique

```

export class Collection<T> {
    models: T[] = [];
    events: Eventing = new Eventing();

    constructor(public rootUrl: string) {}

    ...
}

```

On a un souci au niveau de la methode `fetch()`

```

fetch(): void {
    axios.get(this.rootUrl)
        .then((response: AxiosResponse) => {
            response.data.forEach((value: UserProps) => {
                const user = User.buildUser(value);
                this.models.push(user);
            });
            this.trigger('change');
        });
}

```

On a une reference a `UserProps` , pour regler ce probleme, on va ajouter un autre type generique

```
export class Collection<T, P> { // P pour props
  ...
}
```

On le remplace dans la methode `fetch()`

```
fetch(): void {
  axios.get(this.rootUrl)
    .then((response: AxiosResponse) => {
      response.data.forEach((value: P) => {
        const user = User.buildUser(value);
        this.models.push(user);
      });
      this.trigger('change');
    });
}
```

Le 2eme probleme est l'invocation de la methode statique `buildUser` , meme si on met `T` rien ne dit que cette methode peut etre invoquee depuis `T`

Car ici ce n'est pas du typage, `User` fait reference a une classe reelle

Pour resoudre ce probleme, on va devoir passer au constructeur de `Collection` une fonction qu'on va appeler `deserialize` , dont son role sera de prendre des donnees JSON et de les transformer en une instance reelle d'un objet, cette fonction sera transmis a chaque appelle de l'instance `collection`

```
constructor(
  public rootUrl: string,
  public deserialize: (json: P) => T
) {}
```

et `fetch()` devient:

```

    fetch(): void {
        axios.get(this.rootUrl)
            .then((response: AxiosResponse) => {
                response.data.forEach((value: P) => {
                    const user = this.deserialize(value);
                    this.models.push(user);
                });
                this.trigger('change');
            });
    }
}

```

Dans notre fichier `index.ts`

```

import { Collection } from "../models/Collection";
import { User, UserProps } from "../models/User";

const collection = new Collection<User, UserProps>(
    "http://localhost:3001/users",
    (json: UserProps) => User.buildUser(json)
);

collection.on("change", () => {
    console.log(collection);
})

collection.fetch();

```

On va permettre maintenant à la classe User de récupérer sa collection d'utilisateurs

On va mettre en place une nouvelle méthode statique dans la classe User

```

    static buildUserCollection(): Collection<User, UserProps> {
        return new Collection<User, UserProps>(

```

```

        rootUrl,
        (json: UserProps) => User.buildUser(json)
    );
}

```

et notre `index.ts` devient:

```

import { User } from "../models/User";

const collection = User.buildUserCollection();

collection.on("change", () => {
    console.log(collection);
})

collection.fetch();

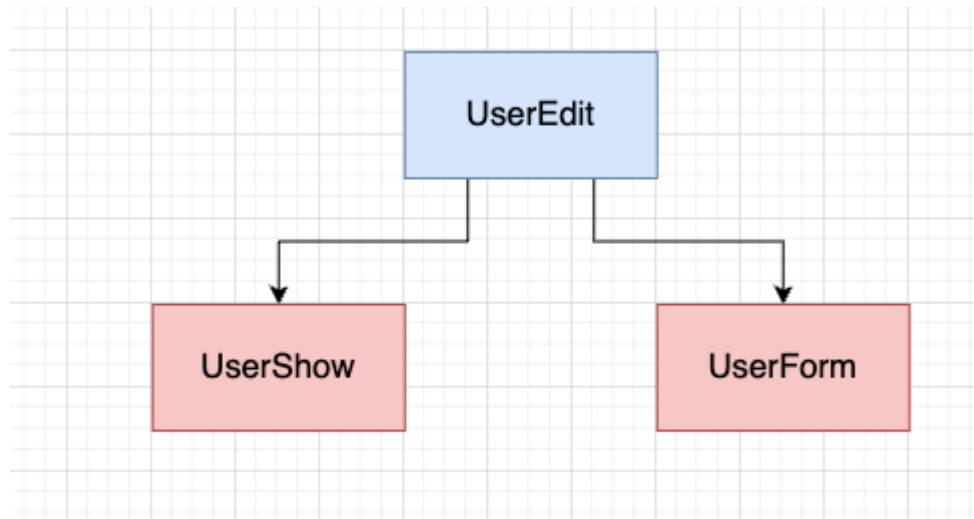
```

Maintenant que toute la partie logique de notre framework est terminée, on va pouvoir mettre en place la partie View ou le HTML

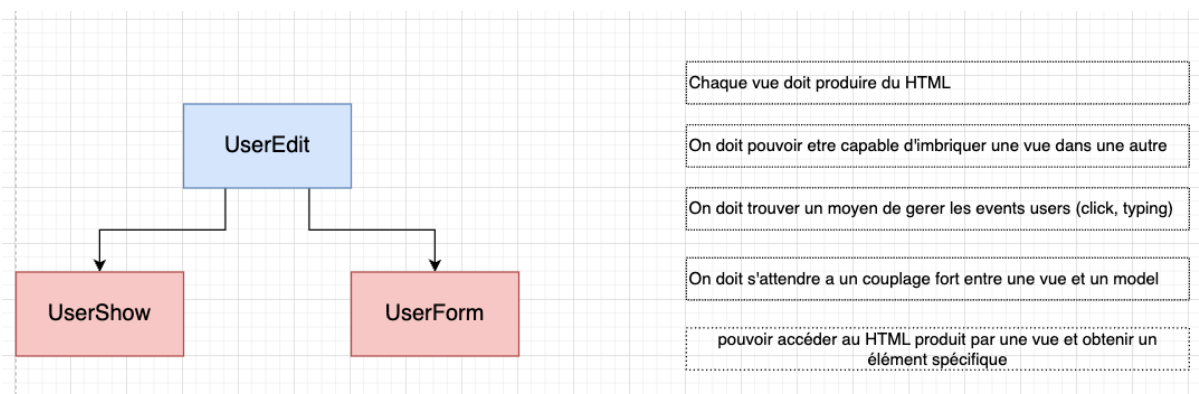
On va faire comme React ou Angular, on va créer des classes dédiées aux vues, charger d'afficher des sections de HTML pour notre application



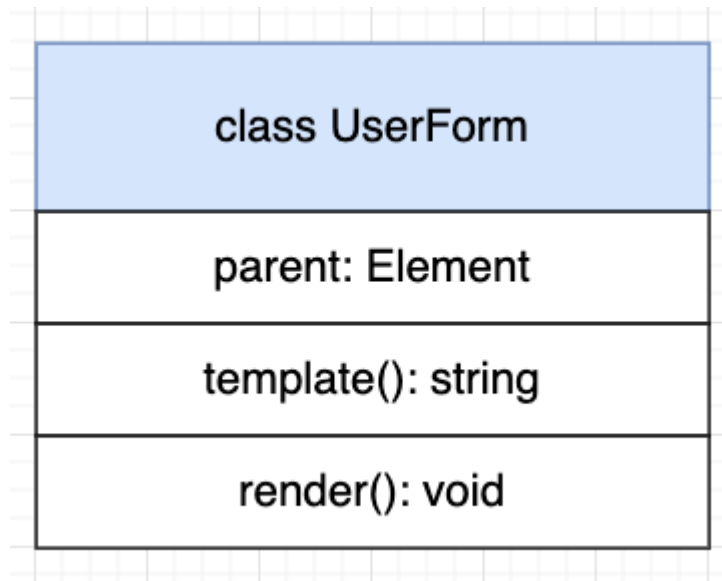
Nous allons mettre en place la hierarchie suivante:



Avec les specificites suivantes:



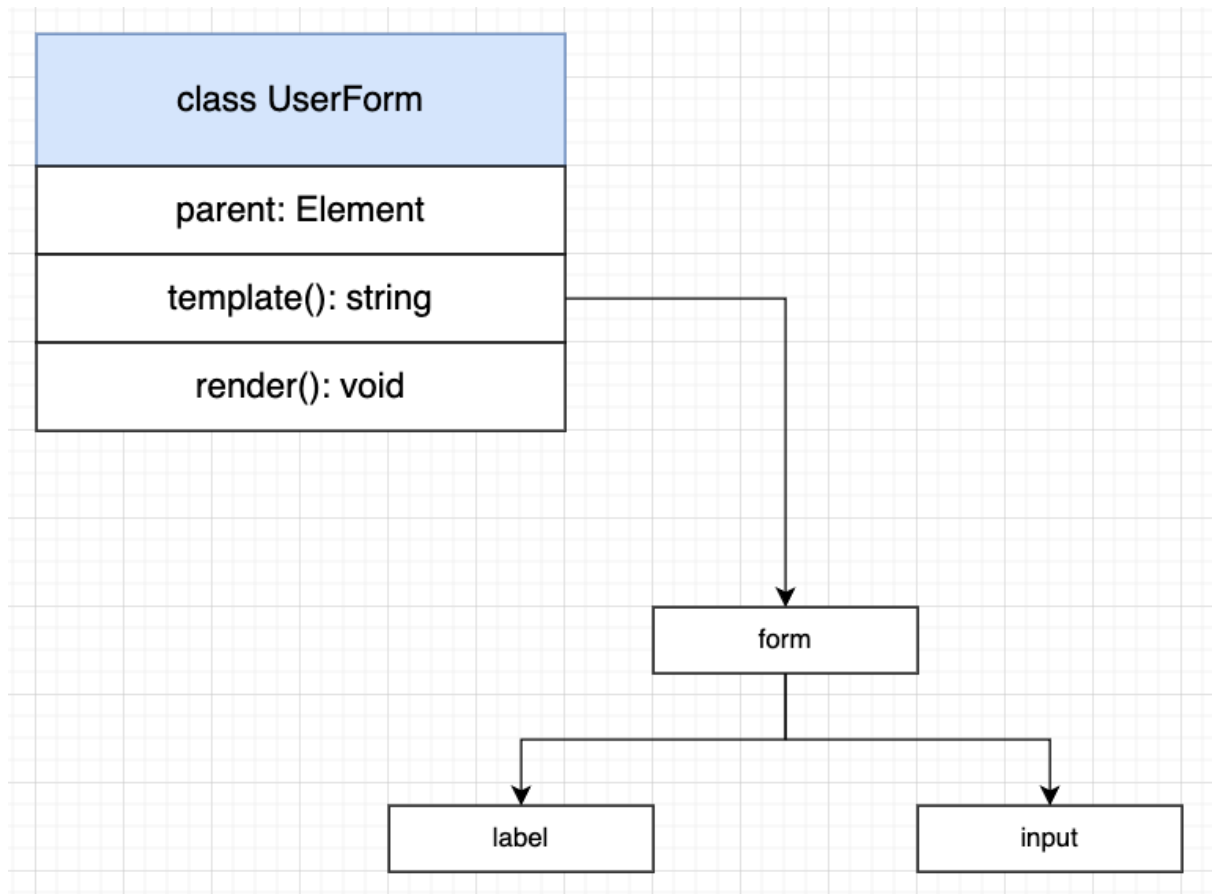
class UserForm



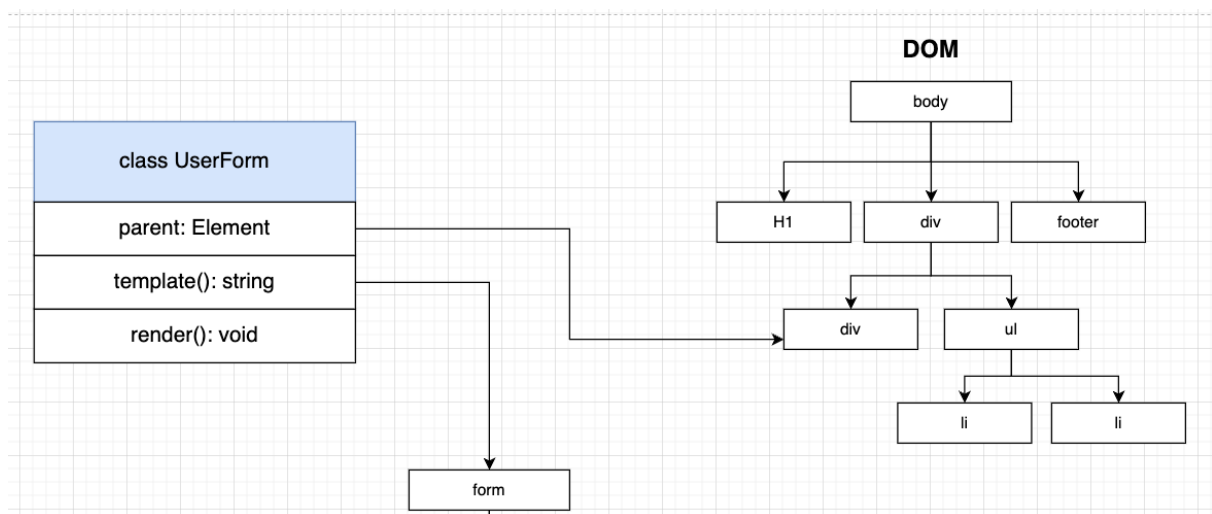
La methode `template()` va creer notre template, et la methode `render()` va l'insérer dans le DOM

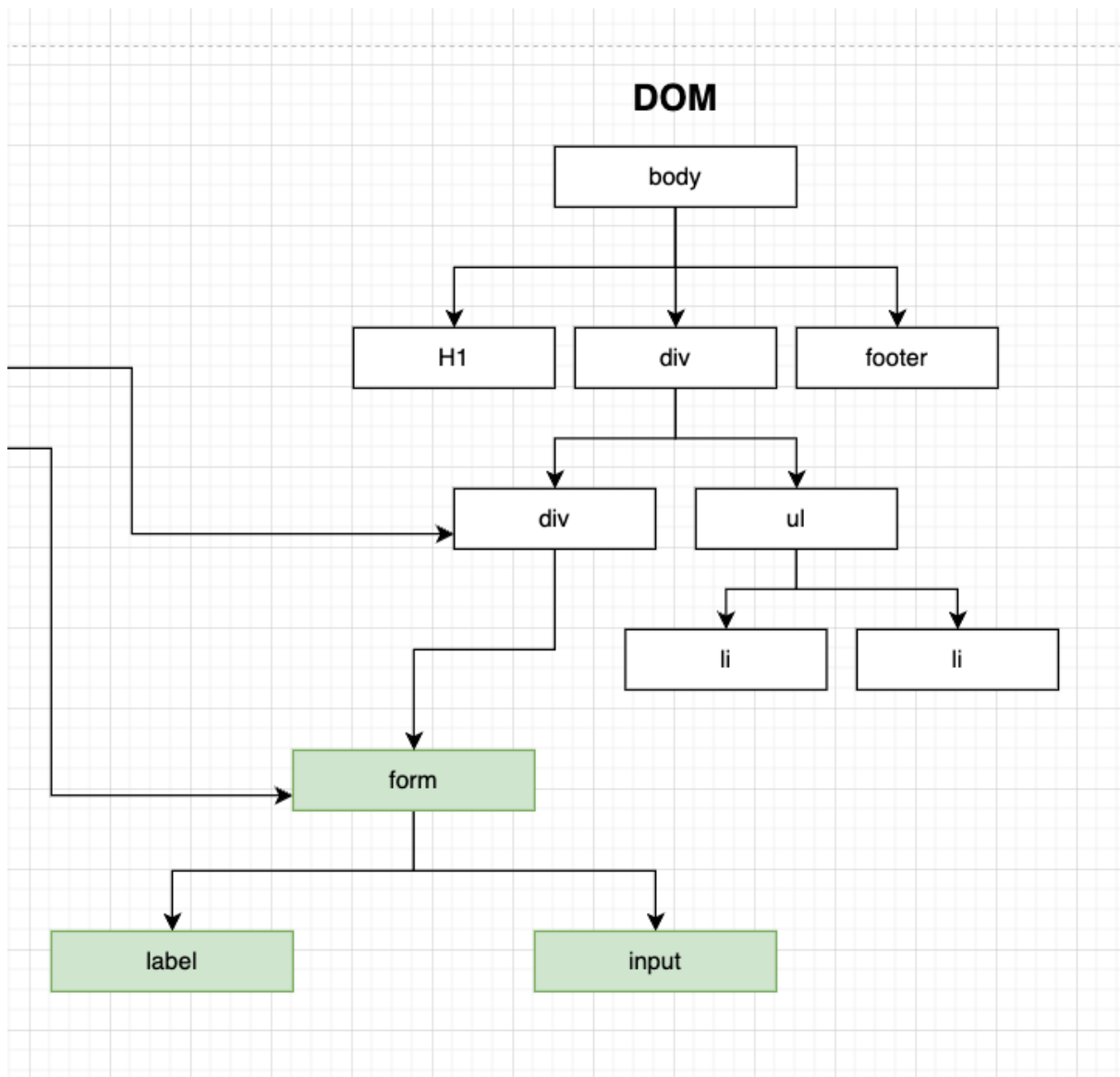
Pour nous assurer que nous pouvons reellement inserer notre `form` dans le DOM, on doit s'assurer qu'il a un parent, ce parent fera reference a un element HTML qui existe a l'interieur du DOM

`render` va appeler la methode `template` et `template` va retourner une chaine de caractere contenant de l'HTML

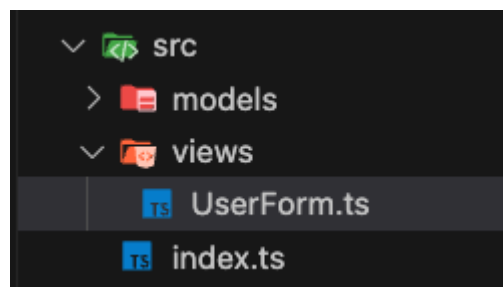


Et la methode `render` va alors prendre l'element et l'ajouter en tant que `enfant` a la propriete `parent` , qui sera un element du DOM





Alors notre HTML sera tres rudimentaire, pas de fioriture a la React, le principal est de se concentrer sur l'utilisation de TS pour creer notre framework



```
export class UserForm {
  parent: Element;
```

```

    template(): string {
        return `
            <div>
                <h1>User Form</h1>
                <input />
            </div>
        `;
    }
}

```

Pour la methode `render()` , alors attention on doit recuperer la chaine de caracteres de `template()` et la transformer en HTML, on va utiliser ce qu'on appelle un Template Element `<template></template>`

L'élément HTML `<template>` sert de mécanisme pour contenir des fragments HTML, qui peuvent être utilisés ultérieurement via JavaScript ou générés immédiatement dans le shadow DOM.

```

    render(): void {
        const templateElement = document.createElement('template');
        templateElement.innerHTML = this.template();
        this.parent.append(templateElement.content); // attention a ne pas oublier .content
    }

```

Il nous reste a initialiser le `parent` , a chaque fois qu'on creera une instance de `UserForm` on va devoir lui specifier le parent, ou s'insérer dans le HTML

```

export class UserForm {
    constructor(public parent: Element) {}

    template(): string {
        return `

```

On va tester tout ca dans le `index.ts`

Dans `index.html` on va mettre:

```
<body>
  <div id="root">

  </div>
</body>
```

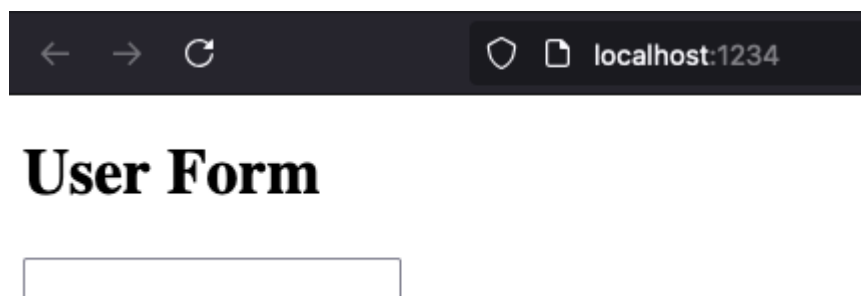
Dans `index.ts` on va mettre:

```
import { UserForm } from "../views/UserForm";

const rootElement = document.getElementById('root');

const userForm = new UserForm(rootElement!);
userForm.render();
```

Si on regarde notre navigateur:



Maintenant on doit egalement travailler sur notre gestionnaire d'evenements a ce HTML

Si je mets un bouton dans mon template

```

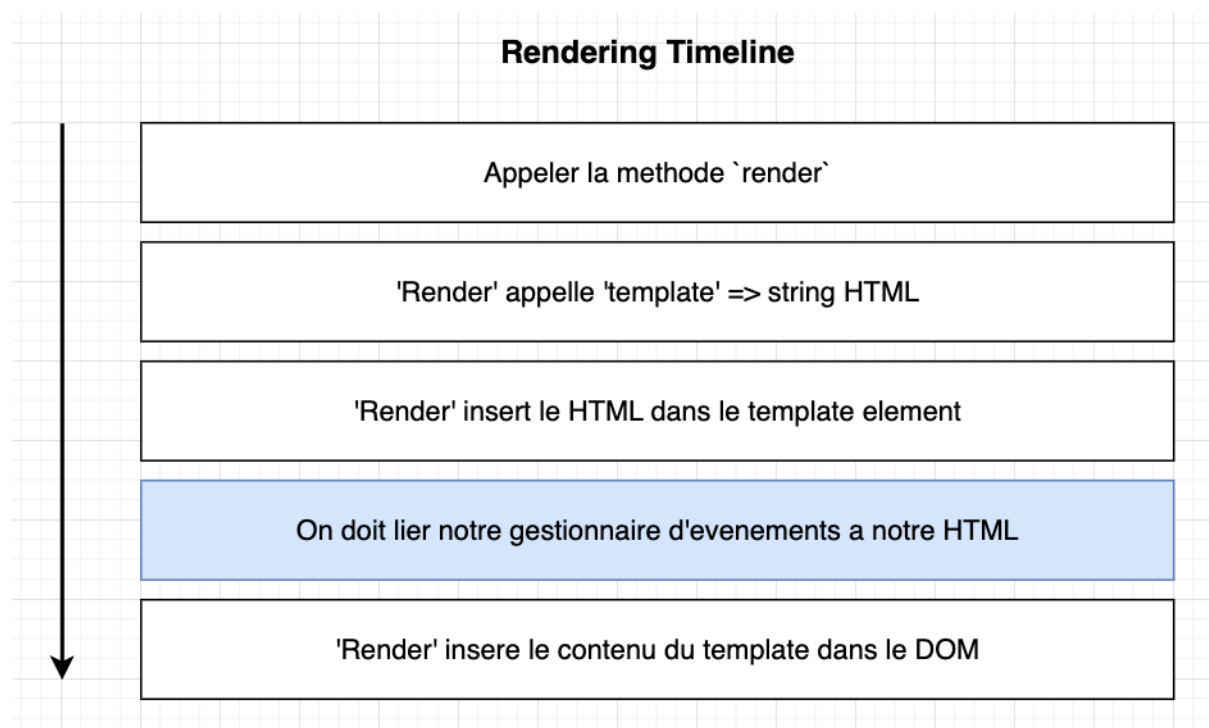
template(): string {
    return `
        <div>
            <h1>User Form</h1>
            <input />
            <button>Save</button>
        </div>
    `;
}

```

Il apparait dans mon navigateur et si je clique dessus, rien ne se passe

Et on n'a pas vraiment de moyen d'ajouter un gestionnaire d'evenement a ce bouton, je veux etre capable de detecter le click et d'executer un code a chaque clic

L'idee est la suivante:



On pourrait faire ca:

```

template(): string {
    return `
        <div onClick=${this.onClick}>
            <h1>User Form</h1>
            <input />
            <button>Save</button>
        </div>
    `;
}

```

Mais c'est beaucoup plus difficile a implementer qu'on ne le pense

Donc je veux ajouter un gestionnaire d'evenements que je veux executer chaque fois que je clique sur ce bouton

```

export class UserForm {

    constructor(public parent: Element) {}

    onClick(): void {
        console.log('Bouton cliquer');
    }

    ...

}

```

N'oubliez pas que tel quel, le string renvoyer par `template()` est sans valeur pour nous

Il n'est utile qu'une fois inserer dans le `templateElement` de la methode `render`

Je vais avoir besoin d'une methode `eventsMap`

```

export class UserForm {

    constructor(public parent: Element) {}

    ...

}

```



```

eventsMap() {
    // ici
}

onButtonClick(): void {
    console.log('Bouton cliquer');
}

```

Chaque fois que nous appellerons cette methode, je vais retourner un objet avec des cles et des valeurs speciales

```

eventsMap() {
    return {
        'click:button': this.onButtonClick
    };
}

```

Donc c'est un objet qui me dit en somme, si je clique sur le bouton, tu lances la methode `onButtonClick`

Et dans cet objet on pourra mettre d'autres cles et d'autres valeurs du genre:

```

eventsMap() {
    return {
        'click:button': this.onButtonClick,
        'hover:h1': this.onHoverHeader
    };
}

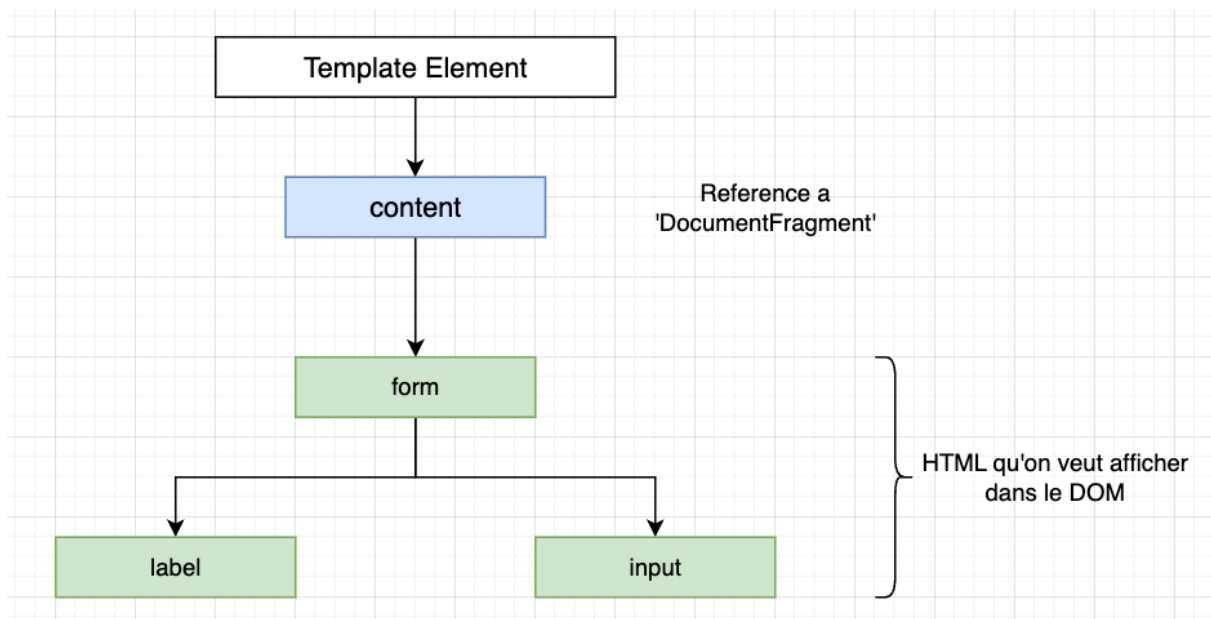
```

Une grande partie de la gestion des evenements etait prise en charge comme ca a l'epoque. Avec React ou Angular, la gestion est bien plus sophistiquée aujourd'hui

On va typer le retour de `eventsMap()`

```
eventsMap() : { [key: string]: () => void } {
  return {
    'click:button': this.onButtonClick
  };
}
```

Maintenant on va lier notre event a notre bouton, avant de faire ca, il y a une chose a savoir sur le template element



Comme on l'a vu, template element a une propriete `content`, c'est celle qui fait reference au code HTML,

le type de `content` est un `DocumentFragment`, un fragment de document est essentiellement un objet qui peut contenir une reference a du HTML

Son but est de conserver du HTML en memoire avant qu'il ne soit reellement inserer dans le DOM

On va donc creer une methode qui va prendre en arguments une reference a un fragment de document, puis parcourir le `eventsMap` et lier tous les evenements au HTML

```
bindEvents(fragment: DocumentFragment): void {
  const eventsMap = this.eventsMap();
```

```

        for (let eventKey in eventsMap) {
            const [eventName, selector] = eventKey.split
('::');
            fragment.querySelectorAll(selector).forEach(ele
ment => {
                element.addEventListener(eventName, eventsM
ap[eventKey]);
            });
        }
    }
}

```

Et dans la methode `render()` on fait appelle a cette methode, juste avant de l'attacher au `parent`

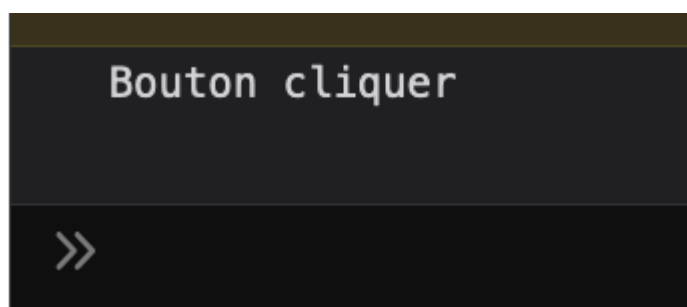
```

render(): void {
    const templateElement = document.createElement('tem
plate');
    templateElement.innerHTML = this.template();
    this.bindEvents(templateElement.content); // ici

    this.parent.append(templateElement.content);
}

```

maintenant si je clique sur le bouton, je peux voir que cela fonctionne



Maintenant, on veut s'assurer qu'a chaque fois qu'on creer une instance d'une vue, que ca transmet un modele

Et afficher les proprietes de ce modele a l'interieur de notre template

```

import { User } from '../models/User';

```

```
export class UserForm {

    constructor(
        public parent: Element,
        public model: User // ici
    ) {}

    ...
}
```

Dans `index.ts`

```
import { User } from "../models/User";
import { UserForm } from "../views/UserForm";

const rootElement = document.getElementById('root');
const user = User.buildUser({ name: 'HENRY', age: 20 });

const userForm = new UserForm(rootElement!, user);
userForm.render();
```

On revient dans `UserForm` dans la methode `template()`

```
template(): string {
    return `
        <div>
            <h1>User Form</h1>
            <div>User Name: ${this.model.get('name')}</di
div> <==== ici
            <div>User Age: ${this.model.get('age')}</di
```

```

v> <==== ici
        <input />
        <button>Save</button>
    </div>
    `;
}

```

Ca fonctionne:

User Form

User Name: HENRY

User Age: 20

Maintenant je veux mettre a jour les informations de l'utilisateur a partir du formulaire, en saisissant du texte, ou en cliquant sur le bouton save

Je vais ajouter un bouton qui va generer automatiquement l'age de l'utilisateur

```

template(): string {
    return `
        <div>
            <h1>User Form</h1>
            <div>User Name: ${this.model.get(
            <div>User Age: ${this.model.get(
            <input />
            <button>Save</button>
            <button>Random Age</button>
        </div>
    `;
}

```

Le souci c'est que si je clique sur les 2 boutons, j'ai le meme console log dut a notre `eventsMap`

On a donc besoin d'un meilleur moyen de specifier exactement a quel bouton nous devons lier tel event

Ce qu'on peut faire c'est ajouter un nom de classe ou un ID a nos differents boutons

```
<input />
<button>Save</button>
<button class= "set-age">Random Age</button>
</div>
;
```

et dans `eventsMap`

```
eventsMap() : { [key: string]: () => void } {
  return {
    'click:button': this.onButtonClick,
    'click:.set-age': this.onSetAgeClick
  };
}
```

Pour l'instant le premier va quand meme etre activer pour les deux, on va le retirer pour l'instant

```
eventsMap() : { [key: string]: () => void } {
  return {
    'click:.set-age': this.onSetAgeClick
  };
}
```

Et on enleve la methode `onButtonClick` , on y reviendra

On ajoute la methode `onSetAgeClick`

```
onSetAgeClick = (): void => {
    this.model.setRandomAge();
}
```

En fait on va demander au model `User` de definir randomly son age

Dans la classe `User` , on ajoute la methode

```
setRandomAge() {
    this.set({age: Math.floor(Math.random() * 99 + 1)})
}
```

Si on teste on obtient l'erreur suivante:

```
! ▶ Uncaught TypeError: this.model is undefined
onSetAgeClick  UserForm.ts:18
bindEvents    UserForm.ts:27
bindEvents    UserForm.ts:26
render        UserForm.ts:48
h7u1C         index.ts:9
```

C'est une erreur qu'on a deja rencontree, car le `this` a perdu son contexte

Donc on corrige en mettant une fonction flechee

```
onSetAgeClick = (): void => {
    this.model.setRandomAge();
}
```

Bon maintenant je ne vois pas le changement s'operer sur mon navigateur,
l'age ne change pas, car rien dans notre classe `UserForm` declenche un nouveau rendu

Re rendering

C'est la que la propriete `events` de la classe `Model` va entrer en jeu

Dans la methode `set` quand on met a jour les attributs de notre user, on declenche l'event `change`

Souvenez vous que ces events nous permette de communiquer avec les autres elements de notre application pour dire que quelque chose a changer

Donc des qu'on creer une instance d'une classe de vue, on va ecouter le modele qui est transmis et ecouter specifiquement l'evenement `change`

Et ca va etre tres simple, ca se passe dans le constructeur de la classe `UserForm`

```
export class UserForm {  
  
  constructor(  
    public parent: Element,  
    public model: User  
  ) {  
    this.model.on('change', () => {  
      this.render();  
    });  
  }  
}
```

On peut refactoriser rapidement avec un methode `bindModel()`, histoire de garder le constructeur aussi simple que possible

```
export class UserForm {  
  
  constructor(  
    public parent: Element,  
    public model: User  
  ) {  
    this.bindModel();  
  }  
  
  bindModel(): void {  
    this.model.on('change', () => {  
      this.render();  
    });  
  }  
}
```

Si on teste:

User Form

User Name: HENRY

User Age: 20

Save

Random Age

User Form

User Name: HENRY

User Age: 71

Save

Random Age

On obtient un doublon, mais ca fonctionne

Il nous reste donc une chose a faire:

```
render(): void {  
  this.parent.innerHTML = '';  
  const templateElement = document.createElement('template');  
  templateElement.innerHTML = this.template();  
  this.bindEvents(templateElement.content);  
  this.parent.append(templateElement.content);  
}
```

Mise a jour du nom

Maintenant nous voulons pouvoir mettre a jour le nom, en le saisissant dans le input et en cliquant sur le bouton save

Il nous faudra donc un gestionnaire d'evenements sur ce bouton

```

template(): string {
  return `
    <div>
      <h1>User Form</h1>
      <div>User Name: ${this.model.get('name')}</div>
      <div>User Age: ${this.model.get('age')}</div>
      <input />
      <button class="set-name">Save</button>
      <button class="set-age">Random Age</button>
    </div>
  `;
}

```

Dans `eventsMap`

```

eventsMap() : { [key: string]: () => void } {
  return {
    'click:.set-age': this.onSetAgeClick,
    'click:.set-name': this.onSetNameClick
  };
}

```

Et on écrit la méthode

```

onSetNameClick = (): void => {
  const input = this.parent.querySelector('input');
  const name = input.value;
  this.model.set({ name });
}

```

Si on modifie le nom dans le navigateur, tout se déroule bien

User Form

User Name: sadasdsad

User Age: 30

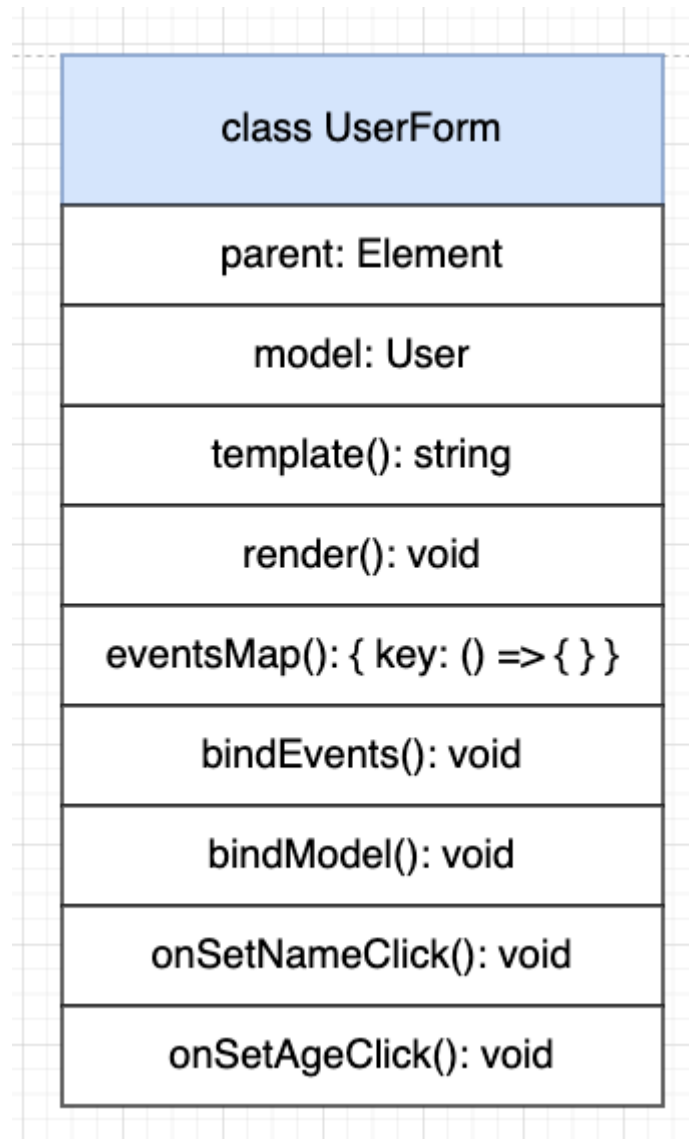
```
onSetNameClick = (): void => {  
  const input = this.parent.querySelector('input');  
  const name = input.value;  
  this.model.set({ name });  
}
```

On peut retirer cette erreur en mettant une condition

```
onSetNameClick = (): void => {  
  const input = this.parent.querySelector('input');  
  
  if(input) {  
    const name = input.value;  
    this.model.set({ name });  
  }  
}
```

Reutilisation de la Vue

Voici ce qu'on a:



`render()` est une methode reutilisable a 100%, rien a l'interieur n'est personnalise

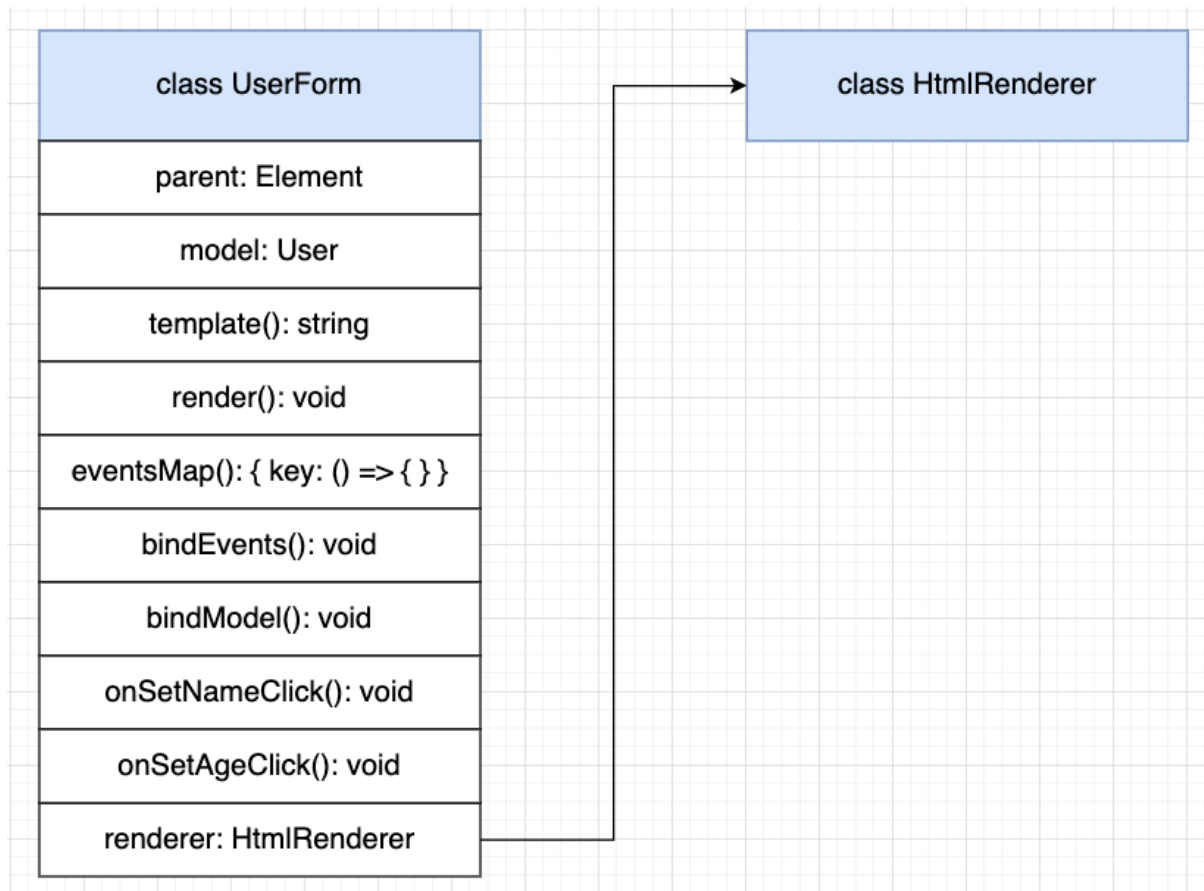
D'autres methodes ne sont pas reutilisables et dependent de User par exemple, comme `onSetNameClick` ou `onSetAgeClick`

On va encore utiliser la composition

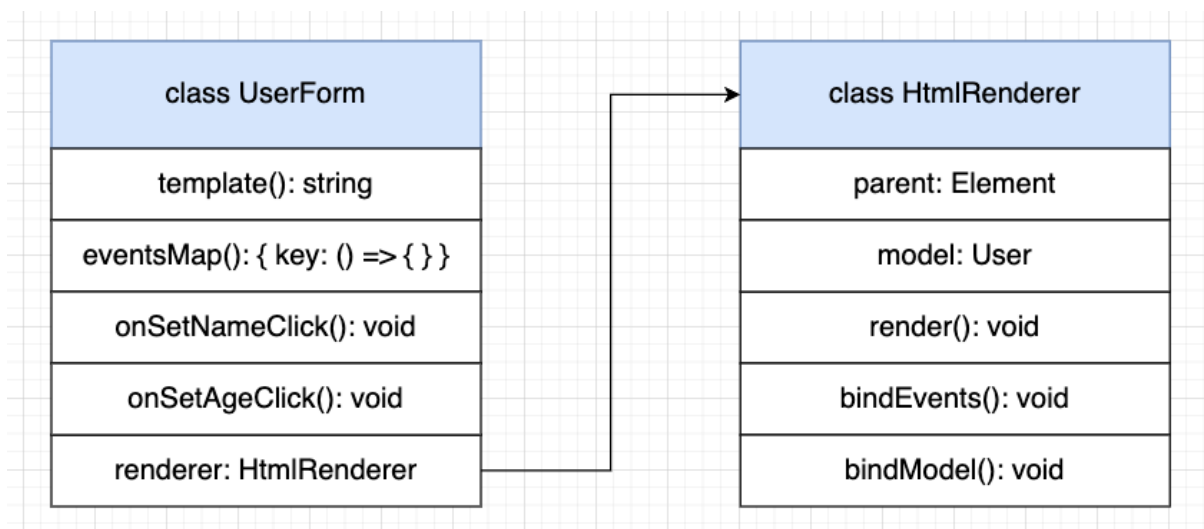
L'idée est de creer une classe `HtmlRenderer`, cette classe va prendre du code HTML et le mettre a l'ecran

Ca va encapsuler toute la logique de vue reutilisable actuelle que nous avons a l'interieur de `UserForm`

Une instance de `HtmlRenderer` dans `UserForm` sera responsable du rendu HTML



Maintenant la question est: quels sont les methodes a mettre dans `HtmlRenderer` et celles a laisser dans `UserForm`



Tout ce qui doit etre affiche a l'interieur du DOM doit avoir cette reference `parent` , donc il doit etre dans la classe `HtmlRenderer`

Pour le `model` de type `User`, c'est un peu confus, on veut vraiment que chaque html rendu, un model auquel il peut se lier afin qu'il puisse s'afficher automatiquement chaque fois que certaines donnees liees a ce model changent, d'un autre cote, le model ici est de type `User`, personnaliser donc au `UserForm`, on peut trouver un moyen d'utiliser les generiques pour rendre ce type un peu plus flexible

On va le mettre dans `HtmlRenderer`, comme ca a chaque fois que le model change, cela mette a jour le rendu

Qu'en est-il de la methode `template()`, celui-ci aussi il est un peu deroutant, car le rendu final fais appel au template, mais d'un autre cote le template est personnalisee pour chaque vue que nous assemblerons, on va donc le laisser dans `UserForm`

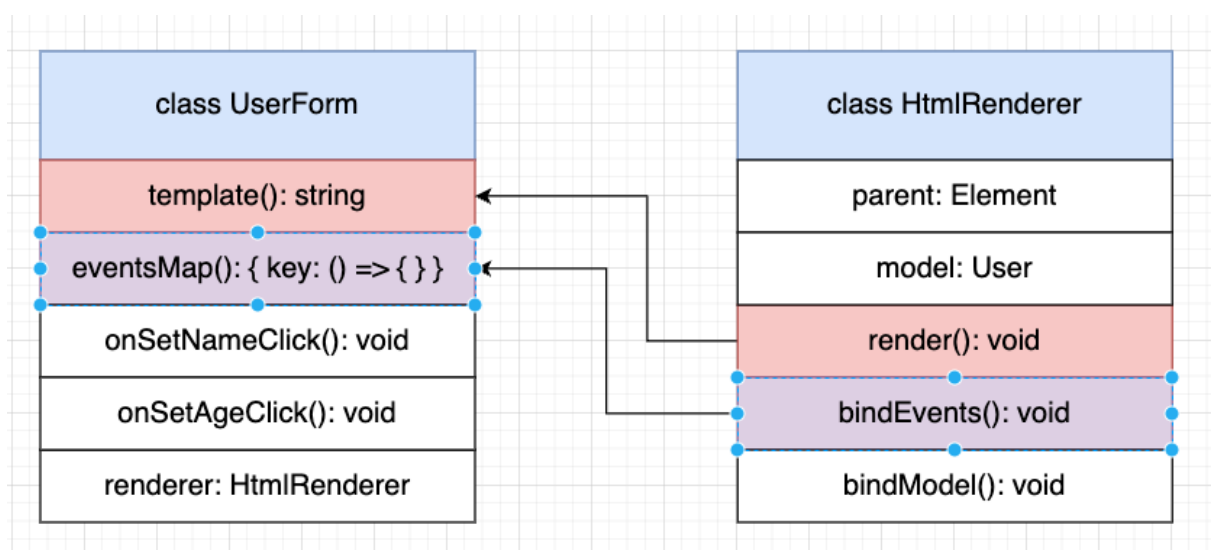
Pour la methode `render` pas de doute, comme son nom l'indique il doit etre dans `HtmlRenderer`

`eventsMap` c'est la methode qui va renvoyer un objet dont les cles sont les events et les valeurs les methodes a declencher, donc c'est dans `UserForm`, car il sera different en fonction de la vue

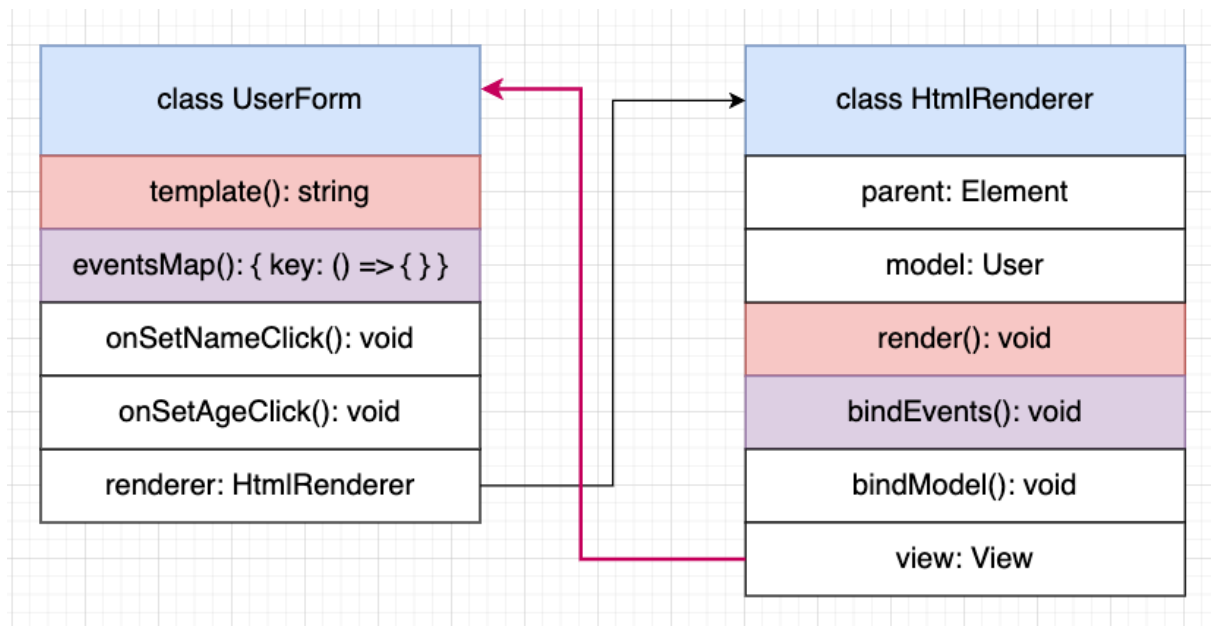
`bindEvents` et `bindModel` seront dans `HtmlRenderer`

`onSetNameClick` et `onSetAgeClick` sont deux methodes propre a `UserForm`

Le seul probleme avec cette approche c'est que la methode `render()` va faire appel a `template()` et la methode `bindEvents()` va faire appel a `eventsMap`

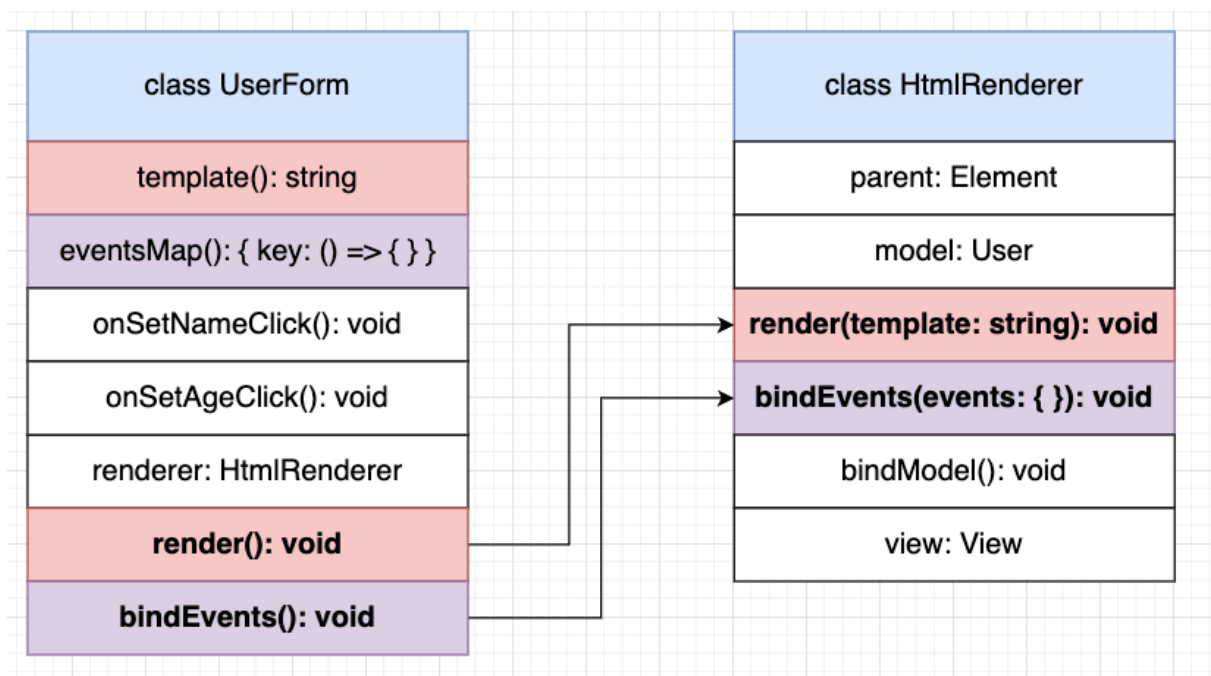


Donc on peut imaginer dans `HtmlRenderer` une reference a `UserForm` et dont le type serait `View` qui serait une interface qui aurait `template()` comme methode



Mais sachez que quand vous avez une relation bidirectionnelle, c'est généralement le signe que la composition n'est pas la meilleure idée, ou du moins la division des méthodes que nous avons mis en place

On pourrait éviter la bidirection via la délégation comme on l'a déjà fait:

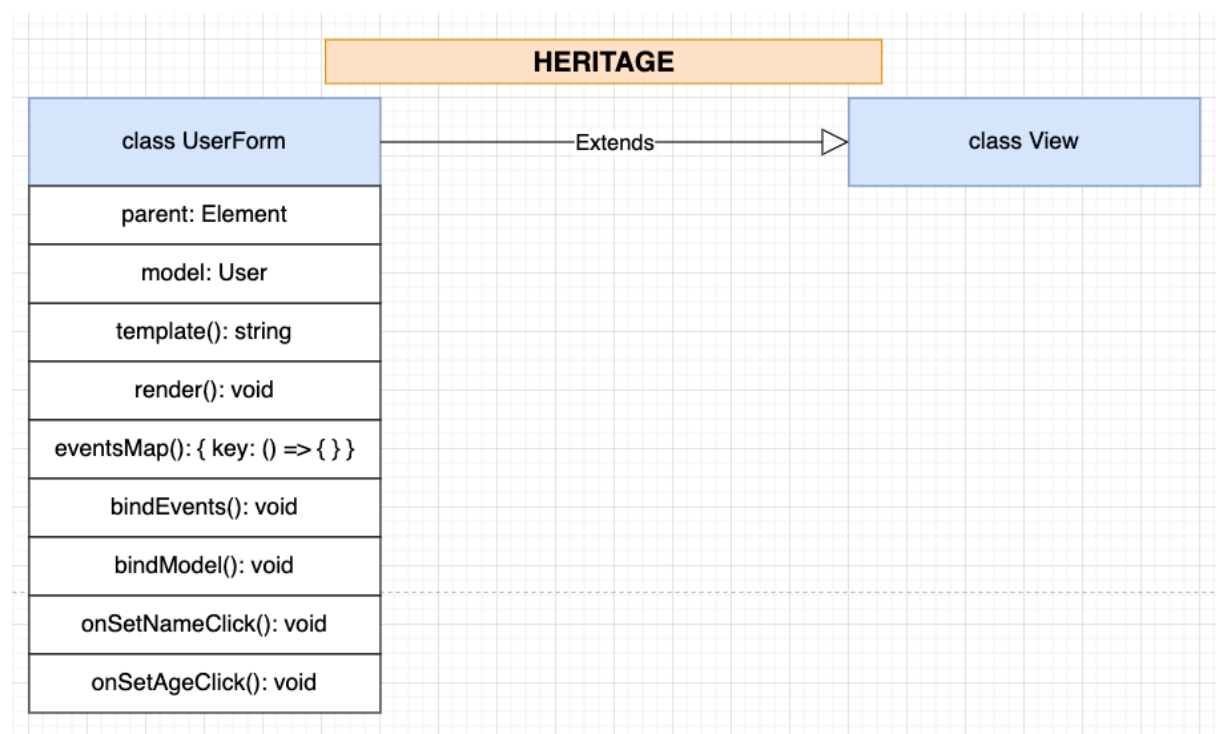


Mais gardez bien à l'esprit que le but premier depuis le début, c'est de créer un framework web, généralement avec la création d'un framework web on veut rendre la création de vues et la création de modèles aussi simple que possible, comme on l'a fait avec `User` et `buildUser`

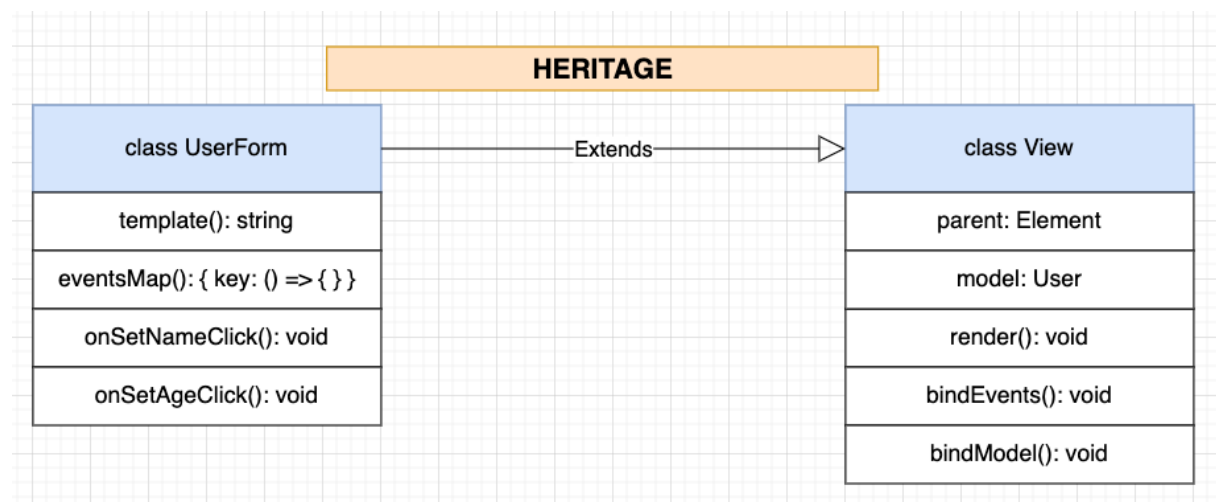
Si on choisit donc cette option, on attends de nos utilisateurs (ceux qui vont utiliser le framework) qu'ils implementent ces methodes pour nous (`render()` et `bindEvents()`)

Donc en vrai, la composition n'est pas la meilleure solution ici, l'heritage pourrait etre la solution a notre probleme

Mise en place de l'heritage



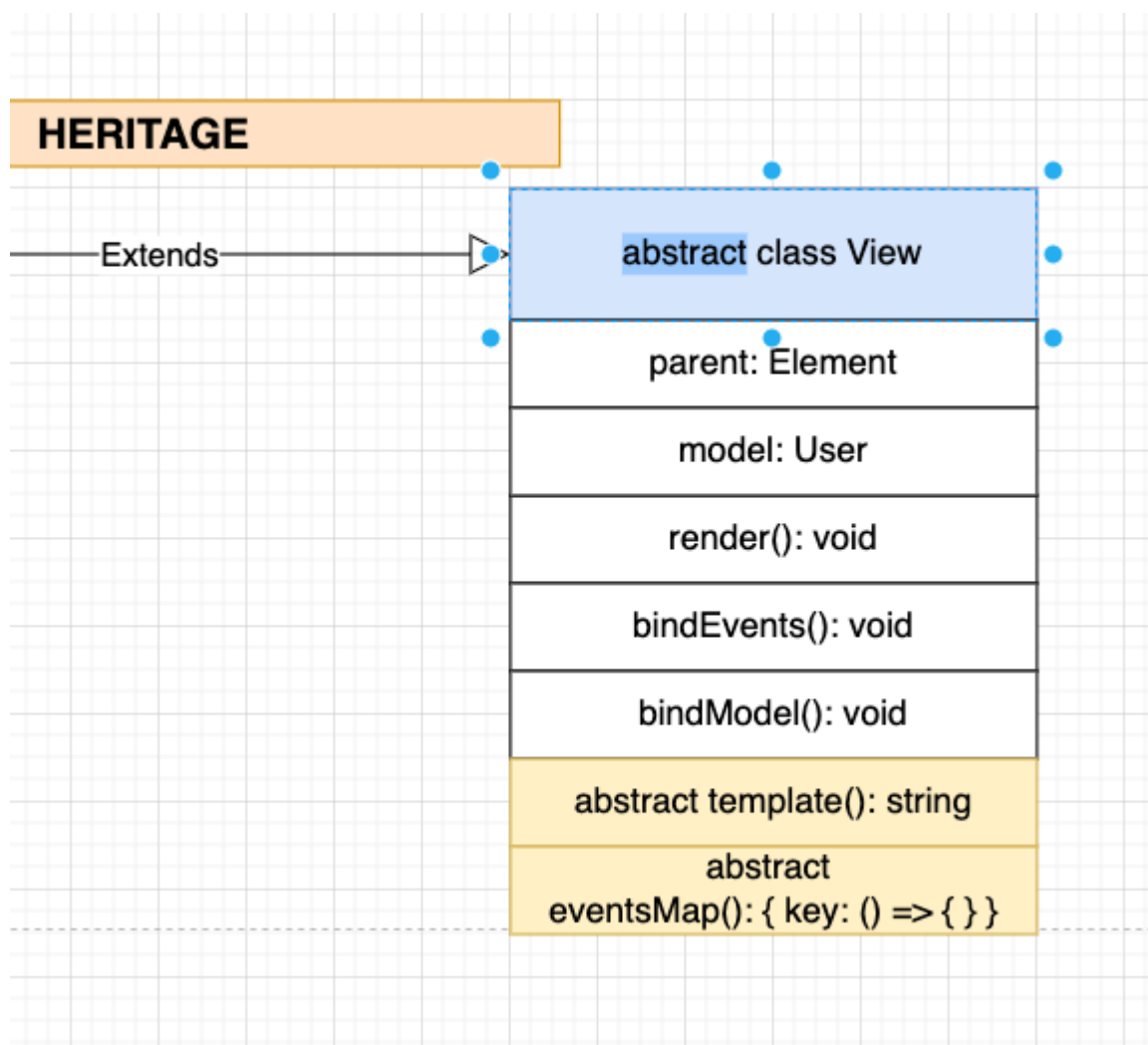
`UserForm` va donc heriter de `View` et maintenant c'est de se dire, quels sont les methodes qui vont aller dans la classe `View`



Il persiste un léger problème, c'est que `render()` et `bindEvents()` peuvent essayer d'appeler des méthodes qui ne seront peut-être pas la (`template()` et `eventsMap()`)

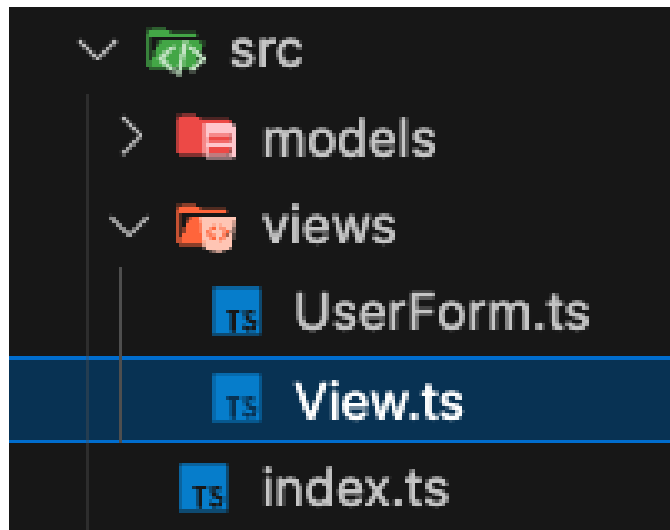
La classe `View` ne devrait pas être en charge de l'implémentation du modèle (via `template` et de `eventsMap`) car ce sont 2 méthodes très personnalisées qui sont définies par `UserForm`

Pour ça, la solution est de déclarer la classe `View` comme abstraite avec 2 méthodes abstraites dont les enfants seront dans l'obligation de les implémenter



De plus ça en fera une classe qu'on n'instanciera jamais

Donc on va mettre en place cette solution



```
import { User } from "../models/User";

export abstract class View {
  constructor(public parent: Element, public model: User)
  { // en voyant User on sait que notre classe abstraite sera
    generique, on s'en occupe juste apres
    this.bindModel();
  }

  bindModel(): void {
    this.model.on('change', () => {
      this.render();
    });
  }

  bindEvents(fragment: DocumentFragment): void {
    const eventsMap = this.eventsMap();

    for (let eventKey in eventsMap) {
      const [eventName, selector] = eventKey.split
('::');
      fragment.querySelectorAll(selector).forEach(ele
```

```

ment => {
    element.addEventListener(eventName, eventsMap[eventKey]);
});
}
}

render(): void {
    this.parent.innerHTML = '';
    const templateElement = document.createElement('template');
    templateElement.innerHTML = this.template();
    this.bindEvents(templateElement.content);
    this.parent.append(templateElement.content);
}

abstract eventsMap(): { [key: string]: () => void };
abstract template(): string;
}

```

Et `UserForm` devient:

```

import { View } from './View';

export class UserForm extends View {

    eventsMap() : { [key: string]: () => void } {
        return {
            'click:.set-age': this.onSetAgeClick,
            'click:.set-name': this.onSetNameClick
        };
    }
}

```

```

onSetAgeClick = (): void => {
    this.model.setRandomAge();
}

onSetNameClick = (): void => {
    const input = this.parent.querySelector('input');

    if(input) {
        const name = input.value;
        this.model.set({ name });
    }
}

template(): string {
    return `
        <div>
            <h1>User Form</h1>
            <div>User Name: ${this.model.get('name')}</div>
            <div>User Age: ${this.model.get('age')}</div>
            <input />
            <button class="set-name">Save</button>
            <button class="set-age">Random Age</button>
        </div>
    `;
}

```

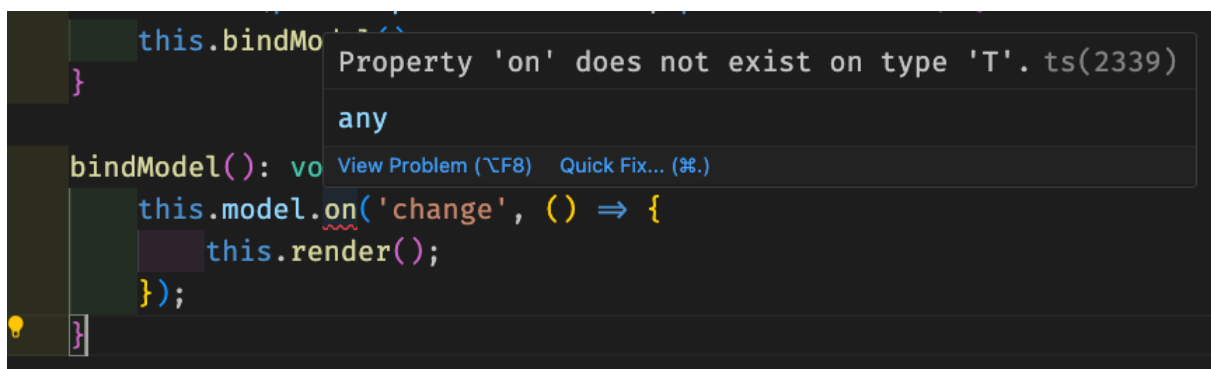
On teste dans notre navigateur et on peut voir que ça fonctionne toujours

On va faire en sorte maintenant que la classe `View` n'ait pas un couplage fort avec `User`, pour cela on va transformer `View` en classe generique

```
export abstract class View<T> {
    constructor(public parent: Element, public model: T) {
        this.bindModel();
    }

    ...
}
```

Des qu'on fait ça, une erreur apparaît



Logique, car TS ne peut pas savoir à l'avance de quel type `T` on parle. On a déjà vu cette erreur, il faut ajouter une contrainte de type

On va ajouter dans le fichier `View.ts` une interface

```
interface ModelForView {
    on(eventName: string, callback: () => void): void;
}
```

```
export abstract class View<T extends ModelForView> {
    constructor(public parent: Element, public model: T) {
        this.bindModel();
    }
}
```

Une autre chose c'est que techniquement on peut également utiliser la classe `Model` directement comme une contrainte générique, car il répond aux critères attendus, donc on peut supprimer l'interface `ModelForView` et mettre:

```
src > views > View.ts > View
2 import { Model } from "../models/Model";
1
3 export abstract class View<T extends Model> {
1     constructor(public parent: Element, public model: T) {
2         this.bindModel();
3     }
4 }
```

Car `Model` est elle meme une classe generique, ce qui nous met dans une impasse

La meilleure solution est de dire que chaque fois que nous creons une vue ou specifions une vue de quelque maniere que ce soit, nous allons passer deux types generiques, `T` et `K`, et que `K` sera le type dans `Model`

```
export abstract class View<T extends Model<K>, K> {
    ...
}
```

Une erreur qu'on connait deja

```
export abstract class View<T extends Model<K>, K> {
    constructor(public parent: Element, public model: T) {
        this.bindModel();
    }
    bindModel(): void {
        this.model.on('change', () => {
            // ...
        });
    }
}
```

Type 'K' does not satisfy the constraint 'HasId'. ts(2344)
View.ts(3, 48): This type parameter might need an `extends`
(type parameter) K in View<T extends Model<K>, K>
View Problem (⌘F8) Quick Fix... (⌘.)

Pour ce faire, dans `Model.ts` on va exporter l'interface `HasId`

```
export interface HasId {
    id?: number;
}
```

Et dans `View.ts`

```
export abstract class View<T extends Model<K>, K extends HasId> { // ici
```

```
    ...  
}
```

On corrige `UserForm`

```
import { User, UserProps } from '../models/User';  
import { View } from './View';  
  
export class UserForm extends View<User, UserProps> {  
    ...  
}
```

Separation des vues

Dans la methode `template()` de `UserForm` on retire pour ne garder que les lignes suivantes:

```
template(): string {  
    return `  
        <div>  
            <input placeholder="${this.model.get('name')}" /> <== on ajoute un placeholder  
            <button class="set-name">Save</button>  
            <button class="set-age">Random Age</button>  
        >  
        </div>  
    `;  
}
```

Le bouton `save` on va le remplacer par `Change name` , et on ajoute un bouton `Save User`

```

template(): string {
  return `
    <div>
      <input placeholder="${this.model.get('name')}}" />
      <button class="set-name">Change name</button>
      <button class="set-age">Random Age</button>
      <button class="save-model">Save User</button>
    </div>
  `;
}

```

Ce bouton va nous permettre de sauvegarder les données dans notre serveur json

Dans `eventsMap()` on ajoute la clé suivante:

```

eventsMap() : { [key: string]: () => void } {
  return {
    'click:.set-age': this.onSetAgeClick,
    'click:.set-name': this.onSetNameClick,
    'click:.save-model': this.onSaveClick
  };
}

```

Et on implémente la méthode `onSaveClick()` :

```

onSaveClick = (): void => {
  this.model.save();
}

```

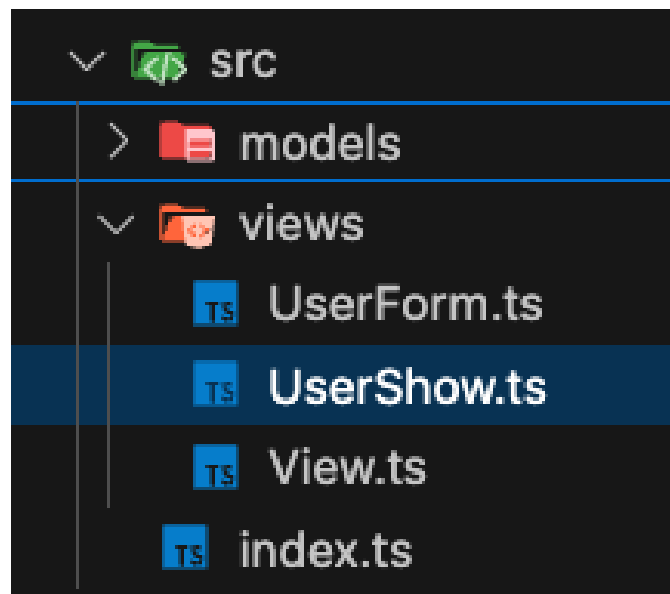
On teste


```

1      },
13     {
1         "name": "TOTO",
2         "age": 20,
3         "id": 3
4     }
5 ]
6 }

```

UserEdit et UserShow



```

import { User, UserProps } from "../models/User";
import { View } from "../View";

export class UserShow extends View<User, UserProps> {
  template(): string {
    return `
      <div>
        <h1>User Details</h1>
        <div>User Name: ${this.model.get('name')}</div>
      </div>
    `;
  }
}

```

```
div>
    <div>User Age: ${this.model.get('age')}</div>
v>
    </div>
;
}
}
```

Pour `eventsMap` je n'ai rien de particulier, donc j'ai 2 options, soit renvoyer un objet vide

```
export class UserShow extends View<User, UserProps> {

    eventsMap(): { [key: string]: () => void; } {
        return {} // ici
    }

    template(): string {
        ...
    }
}
```

Ou bien je pars du principe que je pourrais avoir plusieurs vues qui n'ont pas de gestionnaire d'evenements, et donc dans la classe abstraite, je definis un comportement par default de `eventsMap()`

```
1
33  eventsMap(): { [key: string]: () => void; } {
1   ... return {};
2   };
3
4   abstract template(): string;
5 }
```

et je retire cette methode de la classe `UserShow`

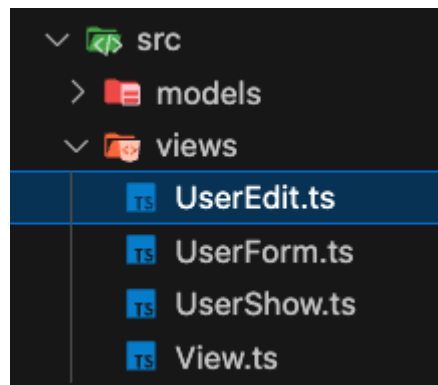
```

export class UserShow extends View<User, UserProps> {

    template(): string {
        return `
            <div>
                <h1>User Details</h1>
                <div>User Name: ${this.model.get('name')}</div>
            </div>
            <div>User Age: ${this.model.get('age')}</div>
        `;
    }
}

```

Donc les modifications se font sur `UserForm` et se repercutent dans `UserShow`, on va voir comment coordonner tout ca avec `UserEdit`



```

import { User, UserProps } from "../models/User";
import { View } from "../View";

export class UserEdit extends View<User, UserProps> {

```

```
}
```

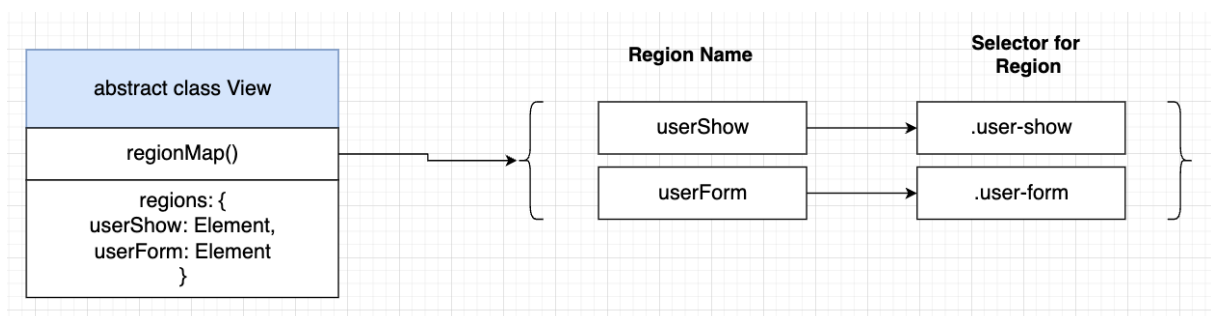
On doit implementer la methode `template()`

```
export class UserEdit extends View<User, UserProps> {  
  template(): string {  
    return `  
      <div>  
        <div class="user-show"></div>  
        <div class="user-form"></div>  
      </div>  
    `;  
  }  
}
```

Vous pouvez voir qu'on a utiliser comme noms de classes html, des noms speciaux qui vous laisse devinez comment on va imbriquer les vues

On va creer une instance de `UserShow` et transmettre a cette div en tant que `parent` de `UserShow`

Rappelez vous que l'element `parent` est la facon dont on decide ou une vue doit faire son rendu



dans la classe View, on va ajouter une nouvelle méthode `regionMap` ainsi qu'une nouvelle propriété `regions`

`regionMap` aura le meme comportement que `eventsMap`

Les devs qui vont utiliser notre framework vont implementer `regionMap()`, qui va retourner un objet de paires cle-valeur, la cle sera la vue (ou le nom de la region) et la valeur sera la classe html ou inserer la vue

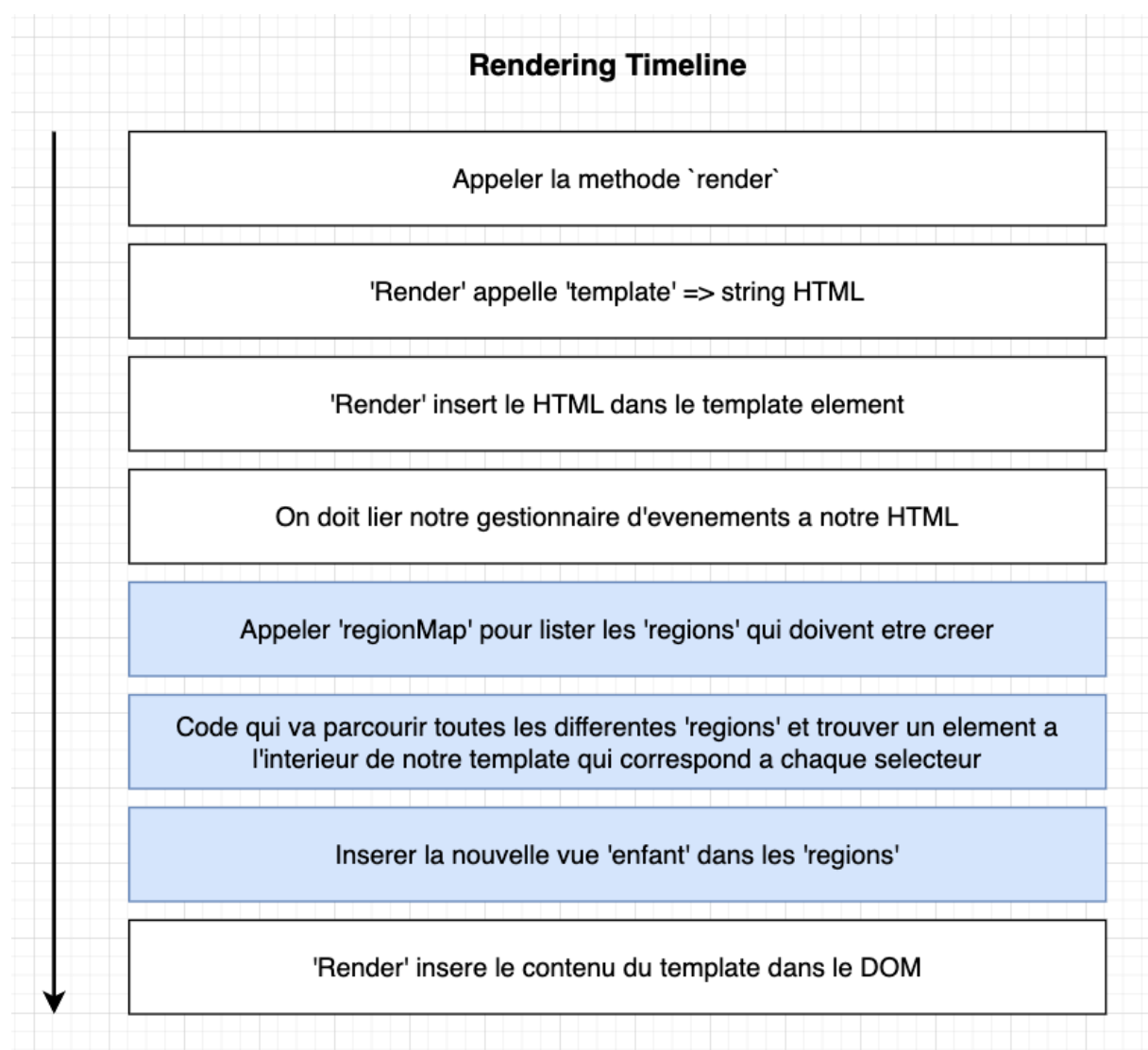
Donc dans la classe `View`

```
3
2 export abstract class View<T extends Model<K>, K extends HasId> {
1
5   regions: {[key: string]: Element} = {};
```

Comme pour `eventsMap` on va definir une methode par default de `regionsMap()`

```
regionsMap(): { [key: string]: string } {
  return {};
}
```

De nouvelles actions seront requises pour creer nos templates et les inserer dans le DOM



```
// View.ts

render(): void {
  this.parent.innerHTML = '';
  const templateElement = document.createElement('template');
  templateElement.innerHTML = this.template();

  this.bindEvents(templateElement.content);
  this.mapRegions(templateElement.content); // ici

  this.parent.append(templateElement.content);
}
```

juste au dessus on definit `mapRegions`

```
mapRegions(fragment: DocumentFragment): void {
  const regionsMap = this.regionsMap();

  for (let key in regionsMap) {
    const selector = regionsMap[key];
    const element = fragment.querySelector(selector);

    if (element) {
      this.regions[key] = element;
    }
  }
}
```

Maintenant a l'interieur de `UserEdit` on va redefinir `regionsMap()`

```
regionsMap(): { [key: string]: string } {
  return {
```

```

        userShow: '.user-show',
        userForm: '.user-form'
    };
}

```

Dans `index.ts`

```

const rootElement = document.getElementById('root');
const user = User.buildUser({ name: 'HENRY', age: 20 });

const userEdit = new UserEdit(rootElement!, user);
userEdit.render();
console.log(userEdit)

```

```

▼ Object { parent: div#root , model: {...}, regions: {...} }
  ► model: Object { attributes: {...}, events: {...}, sync: {...}, ... }
  ► parent: <div id="root"> 
    ▼ regions: Object { userShow: div.user-show , userForm: div.user-form }
      ► userForm: <div class="user-form"> 
      ► userShow: <div class="user-show"> 
      ► <prototype>: Object { ... }
    ► <prototype>: Object { ... }

```

Il nous reste a mettre en place un systeme qui va creer une instance de `UserForm` et une instance de `UserShow` et de les transmettre aux elements parents

En fait, dans `View.ts` , dans la methode `render()` apres avoir lier les evenements et mapper les 'regions', on va imbriquer les vues avant de les ajouter au DOM

```

render(): void {
  this.parent.innerHTML = '';
  const templateElement = document.createElement('template');
  templateElement.innerHTML = this.template();

  this.bindEvents(templateElement.content);
  this.mapRegions(templateElement.content);

  this.onRender();

  this.parent.append(templateElement.content);
}

```

Toujours dans `View.ts` on va definir une methode par default de `onRender()` et les enfants devront la redefinir

NB: On ne l'a met pas en `abstract` car certaines vues ne vont pas imbriquer d'autres vues (comme `UserForm`)

```

1
41  onRender(): void { }
1
2  render(): void {
3    this.parent.innerHTML = '';

```

On finit du coup dans `UserEdit` qui va redefinir `onRender()` de son cote

```

onRender(): void {
  new UserShow(this.regions.userShow, this.model).render();
  new UserForm(this.regions.userForm, this.model).render();
}

```

Notre framework touche a sa fin

Nous avons mis en place la classe `Collection` qui nous permettait de recuperer l'ensemble des utilisateurs

On peut l'utiliser pour afficher la liste des utilisateurs

Vous pouvez vous challenger à l'ajouter à notre DOM, et en choisissant un utilisateur, il doit mettre à jour `UserShow`

Corrigé

On va créer la classe `UserList`

```
import { Collection } from "../framework/models/Collection";
import { View } from "../framework/views/View";
import { User } from "../User";
import { UserProps } from "../UserProps.interface";

export class UserList extends View<User, UserProps> {
  collection: Collection<User, UserProps>;

  constructor(parent: Element, user: User) {
    super(parent, user);
    this.collection = User.buildUserCollection();
    this.initCollection()
  }

  initCollection() {
    this.collection.on('change', () => {
      this.render();
    });
    this.collection.fetch();
  }

  eventsMap(): { [key: string]: () => void } {
    return {
      'change:select': this.onUserSelect
    };
  }
}
```

```

    }

    onUserSelect = (): void => {
        const selectElement = this.parent.querySelector('se
lect');
        if (selectElement) {
            const userId = selectElement.value;
            const selectedUser = this.collection.models.find(
user => user.get('id') === userId)

            if(selectedUser) {
                this.model.set({name: selectedUser.get('nam
e'), age: selectedUser.get('age')})
            }
        }
    }

    template(): string {
        return `
            <div>
                <h1>User List</h1>
                <select>
                    ${this.renderUserOptions()}
                </select>
            </div>
        `;
    }

    renderUserOptions(): string {
        if (this.collection.models.length === 0) {
            return '<option>Loading...</option>';
        }

        return this.collection.models.map((user: User) => {
            return `<option value="${user.get('id')}">${use
r.get('name')}</option>`;
        }).join('');
    }

```

```
}  
}
```

Et `UserEdit` devient

```
export class UserEdit extends View<User, UserProps> {  
  
  regionMap(): { [key: string]: string; } {  
    return {  
      userList: '.user-list',  
      userShow: '.user-show',  
      userForm: '.user-form'  
    }  
  }  
  
  template(): string {  
    return `  
    <div>  
      <div class="user-list"></div>  
      <div class="user-show"></div>  
      <div class="user-form"></div>  
    </div>  
    `;  
  }  
  
  onRender(): void {  
    new UserShow(this.regions.userShow, this.model).render()  
    new UserForm(this.regions.userForm, this.model).render()  
    new UserList(this.regions.userList, this.model).render()  
  }  
}
```

