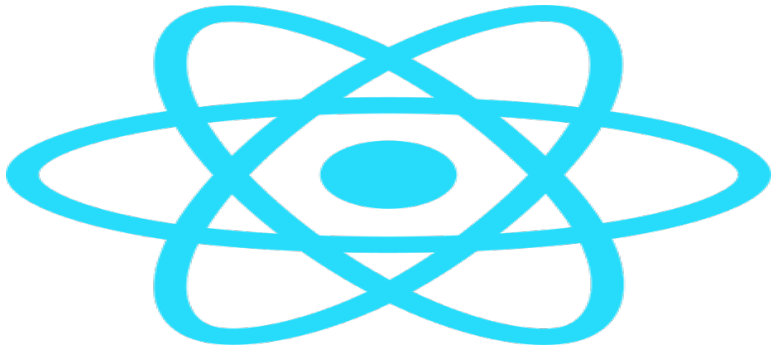
React : formulaire

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

Plan

React.js

Formulaire

- Outil graphique que nous créons avec le langage **HTML**
- Permettant à l'utilisateur d'interagir avec l'application en
  - saisissant de données
  - cochant des cases
  - sélectionnant des options
- Solution pour soumettre les données vers
  - une autre page/composant,
  - une ressource externe (base de données...)

React.js

Apport de **React.js** ?

- Récupération de données saisies
- Validation et contrôle de valeurs saisies
- Gestion d'erreurs
- ...

React.js

Dans la suite

Nous allons

1. créer le composant `PersonneAdd` dans `components`,
2. définir `PersonneAdd` comme composant enfant de `Personne`,
3. préparer `PersonneAdd` pour ajouter des nouvelles personnes (dans la base de données).

React.js

Exercice

- Dans `PersonneAdd`, ajouter un `label`, `input:text` et un bouton `Afficher`,
- En cliquant sur le bouton `Afficher`, le texte saisi dans l'`input` s'affiche à côté.

React.js

Commençons par définir `PersonneAdd` comme enfant de `Personne`

```
import { Link } from "react-router-dom";
import PersonneAdd from "../components/PersonneAdd";

export default function Personne() {
    const personnes = [
        { id: 1, nom: 'Wick', prenom: 'John', age: 45 },
        { id: 2, nom: 'Dalton', prenom: 'Jack', age: 40 },
        { id: 3, nom: 'Dupont', prenom: 'Sophie', age: 30 }
    ];
    return (
        <div>
            <h1>Gestion de personnes</h1>
            <PersonneAdd />
            <ul>
                {
                    personnes.map((elt) =>
                        <li key={elt.id}>
                            <Link to={`/personne/${elt.id}`}>

                                {elt.nom} {elt.prenom}
                            </Link>
                        </li>
                    )
                }
            </ul>
        </div>
    );
}
```

React.js

Contenu de `PersonneAdd`

```
import { useState } from "react";

const PersonneAdd = () => {

    const [nomForm, setNomForm] = useState('Doe');

    const afficherNom = (e) => {
        console.log(nomForm);
    }
    const traiterValeur = (e) => {
        setNomForm(e.target.value)
    }

    return (
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" type="text" name="nom" value={nomForm} onChange={
              traiterValeur} />
            <button onClick={afficherNom}>Ajouter</button>{nomForm}
        </div>
    );
};

export default PersonneAdd;
```

React.js

Pour la suite, nous allons

- créer un formulaire avec trois champs : `Nom`, `Prénom` et `Age`
- récupérer les valeurs saisies dans le formulaire
- valider les données du formulaire
- afficher les messages d'erreur

React.js

Et un formulaire dans `PersonneAdd` avec la fonction `ajouterPersonne()`

```
const PersonneAdd = () => {
    const ajouterPersonne = (e) => {
        e.preventDefault();
        console.log("Formulaire soumis");
    }

    return (
        <form onSubmit={ajouterPersonne}>
            <div>
                <label htmlFor="nom">Nom :</label>
                <input id="nom" type="text" name="nom" />
            </div>
            <div>
                <label htmlFor="prenom">Prénom :</label>
                <input id="prenom" type="text" name="prenom" />
            </div>
            <div>
                <label htmlFor="age">Âge :</label>
                <input id="age" type="number" name="age" />
            </div>
            <button type="submit">Ajouter</button>
        </form>
    );
};

export default PersonneAdd;
```

React.js

Pour assurer le binding, on utilise `useState`

```
const [formData, setFormData] = useState({
    nom: '',
    prenom: '',
    age: ''
});
```

React.js

Pour assurer le binding, on utilise `useState`

```
const [formData, setFormData] = useState({
    nom: '',
    prenom: '',
    age: ''
});
```


Sans oublier d'importer `useState`

```
import { useState } from "react";
```

React.js

Pour mettre à jour `formData` après chaque changement

```
const traiterValeur = (e) => {
    const { name, value } = e.target;
    setFormData({
        ...formData,
        [name]: value
    });
};
return (
    <form onSubmit={ajouterPersonne}>
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" type="text" name="nom" value={formData.nom} onChange={traiterValeur
              } />
        </div>
        <div>
            <label htmlFor="prenom">Prénom :</label>
            <input id="prenom" type="text" name="prenom" value={formData.prenom} onChange={
              traiterValeur} />
        </div>
        <div>
            <label htmlFor="age">Âge :</label>
            <input id="age" type="number" name="age" value={formData.age} onChange={
              traiterValeur} />
        </div>
        <button type="submit">Ajouter</button>
    </form>
);
```

React.js

Modifions `ajouterPersonne` pour afficher l'objet soumis

```
const ajouterPersonne = (e) => {
    e.preventDefault();
    console.log(formData);
}
```

React.js

Pour soumettre le formulaire, il faut qu'il soit valide

- le nom est obligatoire et ne doit pas dépasser 30 caractères,
- le prénom est obligatoire et doit commencer par une majuscule,
- l'âge est obligatoire et doit contenir une valeur comprise entre 18 et 120.

Commençons par définir un état pour les erreurs (de type objet) et la fonction `validate`

```
const [errors, setErrors] = useState({});

const validate = () => {
    const newErrors = {};
    if (!formData.nom) {
        newErrors.nom = 'Le nom est requis';
    } else if (formData.nom.length > 30) {
        newErrors.prenom = 'Le nom ne doit pas dépasser 30 caractères'
    }
    if (!formData.prenom) {
        newErrors.prenom = 'Le prénom est requis';
    } else if (!/^[A-Z]{1}.*/.test(formData.prenom)) {
        newErrors.prenom = 'Le prénom doit commencer par une majuscule'
    }
    if (!formData.age) {
        newErrors.age = 'L\'âge est requis';
    } else if (formData.age > 120 || formData.age < 18) {
        newErrors.age = 'L\'âge doit être compris entre 18 et 120'
    }
    return newErrors;
};
```

React.js

Utilisons `validate` à la soumission du formulaire

```
const ajouterPersonne = (e) => {
    e.preventDefault();
    const formErrors = validate();
    if (Object.keys(formErrors).length === 0) {
        console.log('Formulaire soumis avec succès', formData);
        setFormData({ nom: '', prenom: '', age: '' });
        setErrors({});
    } else {
        setErrors(formErrors);
    }
};
```

React.js

Modifions le `render` pour afficher les messages d'erreur

```
return (
    <form onSubmit={ajouterPersonne}>
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" type="text" name="nom" value={formData.nom} onChange={traiterValeur
              } />
            {errors.nom && <span style={{ color: 'red' }}>{errors.nom}</span>}

        </div>
        <div>
            <label htmlFor="prenom">Prénom :</label>
            <input id="prenom" type="text" name="prenom" value={formData.prenom} onChange={
              traiterValeur} />
            {errors.prenom && <span style={{ color: 'red' }}>{errors.prenom}</span>}

        </div>
        <div>
            <label htmlFor="age">Âge :</label>
            <input id="age" type="number" name="age" value={formData.age} onChange={
              traiterValeur} />
            {errors.age && <span style={{ color: 'red' }}>{errors.age}</span>}

        </div>
        <button type="submit">Ajouter</button>
    </form>
);
```

React.js

Pour valider les formulaires, on peut utiliser **Yup**

- Librairie **JavaScript** pour la validation de valeurs
- Écrite en **JavaScript** et **TypeScript**
- Page **GitHub** : `https://github.com/jquense/yup`

React.js

Pour valider les formulaires, on peut utiliser **Yup**

- Librairie **JavaScript** pour la validation de valeurs
- Écrite en **JavaScript** et **TypeScript**
- Page **GitHub** : `https://github.com/jquense/yup`

Pour installer `yup`

```
npm install yup
```

React.js

Démarche

1. Importer **Yup**
2. Définir un schéma de validation
3. Appeler une méthode `validate` pour vérifier si le formulaire est valide

React.js

Démarche

1. Importer **Yup**
2. Définir un schéma de validation
3. Appeler une méthode `validate` pour vérifier si le formulaire est valide

Pour importer `yup`

```
const yup = require('yup')
```

React.js

Commençons par définir un schéma de validation : chaque élément doit porter le nom d'un champ du formulaire

```
const validationSchema = yup.object().shape({
    nom: yup.string().required().max(30),
    prenom: yup.string().required().matches(/^[A-Z]{1}.*/),
    age: yup.number().min(18).max(150),
});
```

React.js

Autres validateurs pour `string`

- `length`
- `url`
- `uuid`
- `lowercase`
- `uppercase`
- `trim`
- ...

React.js

Autres validateurs pour `number`

- `lessThan`
- `moreThan`
- `positive`
- `negative`
- `integer`
- ...

React.js

Utilisons la méthode `validate` pour vérifier si le formulaire est valide

```
const validate = async () => {
    try {
        await validationSchema.validate(formData, { abortEarly: false });
        return {};
    } catch (validationErrors) {
        const newErrors = {};
        validationErrors.inner.forEach((error) => {
            newErrors[error.path] = error.message;
        });
        return newErrors;
    }
};
```

React.js

Utilisons la méthode `validate` pour vérifier si le formulaire est valide

```
const validate = async () => {
    try {
        await validationSchema.validate(formData, { abortEarly: false });
        return {};
    } catch (validationErrors) {
        const newErrors = {};
        validationErrors.inner.forEach((error) => {
            newErrors[error.path] = error.message;
        });
        return newErrors;
    }
};
```


`abortEarly` accepte deux valeurs

- `true` (par défaut) : la méthode `validate()` arrête la validation dès qu'elle rencontre la première erreur. C'est-à-dire, si plusieurs champs sont invalides, seule la première erreur est affichée.
- `false` : la validation continue même après avoir rencontré une erreur, permettant ainsi de récupérer toutes les erreurs de validation à la fois.

React.js

Et la fonction `ajouterPersonne`

```
const ajouterPersonne = async (e) => {
    e.preventDefault();
    const formErrors = await validate();

    if (Object.keys(formErrors).length === 0) {
        console.log('Formulaire soumis avec succès', formData);
        setFormData({ nom: '', prenom: '', age: '' });
        setErrors({});
    } else {
        setErrors(formErrors);
    }
};
```

React.js

Le `render` ne change pas

```
return (
    <form onSubmit={ajouterPersonne}>
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" type="text" name="nom" value={formData.nom} onChange={traiterValeur
              } />
            {errors.nom && <span style={{ color: 'red' }}>{errors.nom}</span>}

        </div>
        <div>
            <label htmlFor="prenom">Prénom :</label>
            <input id="prenom" type="text" name="prenom" value={formData.prenom} onChange={
              traiterValeur} />
            {errors.prenom && <span style={{ color: 'red' }}>{errors.prenom}</span>}

        </div>
        <div>
            <label htmlFor="age">Âge :</label>
            <input id="age" type="number" name="age" value={formData.age} onChange={
              traiterValeur} />
            {errors.age && <span style={{ color: 'red' }}>{errors.age}</span>}

        </div>
        <button type="submit">Ajouter</button>
    </form>
);
```

React.js

Remarque

- Par défaut, les messages d'erreur s'affichent en anglais
- Cependant, il est possible d'utiliser `yup-locales` pour les afficher en français

React.js

Remarque

- Par défaut, les messages d'erreur s'affichent en anglais
- Cependant, il est possible d'utiliser `yup-locales` pour les afficher en français

Pour installer

```
npm install yup-locales
```

React.js

Dans `index.js`, commençons par importer `yup-locales`

```
import { fr } from 'yup-locales'
```

React.js

Dans `index.js`**, commençons par importer** `yup-locales`

```
import { fr } from 'yup-locales'
```

Importons également `setLocale`

```
import { setLocale } from 'yup';
```

React.js

Dans `index.js`, commençons par importer `yup-locales`

```
import { fr } from 'yup-locales'
```

Importons également `setLocale`

```
import { setLocale } from 'yup';
```


Modifions ensuite la `Locale` et vérifions que les messages d'erreur s'affichent désormais en français

```
setLocale(fr)
```

React.js

Question

Comment peut-on afficher des messages d'erreur personnalisés ?

React.js

Pour personnaliser les messages d'erreur

```
const validationSchema = yup.object().shape({
    nom: yup
        .string()
        .required()
        .max(30),
    prenom: yup
        .string()
        .required()
        .matches(/^[A-Z]{1}.*/, "Le prénom doit commencer par une majuscule"),
    age: yup
        .number()
        .min(18)
        .max(150, "L'âge max autorisé : 150"),
});
```

React.js

Pour récupérer le seuil indiqué dans certaines fonctions comme `min`, `max`...

```
const validationSchema = yup.object().shape({
    nom: yup
        .string()
        .required()
        .max(30),
    prenom: yup
        .string()
        .required()
        .matches(/^[A-Z]{1}.*/, "Le prénom doit commencer par une majuscule"),
    age: yup
        .number()
        .min(18)
        .max(150, (args) => `L'âge max autorisé : ${args.max}, valeur saisie : ${args.value}`),
});
```

React.js

Ajoutons le `console.log` **suivant dans** `PersonneAdd`

```
const PersonneAdd = () => {
    const [formData, setFormData] = useState({
        nom: '',
        prenom: '',
        age: ''
    });
    const [errors, setErrors] = useState({});

    const traiterValeur = (e) => {
        const { name, value } = e.target;
        setFormData({
            ...formData,
            [name]: value
        });
    };

    console.log('called');

    // + le code précédent
}
```

React.js

Ajoutons le `console.log` suivant dans `PersonneAdd`

```
const PersonneAdd = () => {
    const [formData, setFormData] = useState({
        nom: '',
        prenom: '',
        age: ''
    });
    const [errors, setErrors] = useState({});

    const traiterValeur = (e) => {
        const { name, value } = e.target;
        setFormData({
            ...formData,
            [name]: value
        });
    };

    console.log('called');

    // + le code précédent
}
```

Constat

Vérifiez que `called` s'affiche dans la console du navigateur après chaque modification de valeur d'un champ du formulaire.

React.js

Explication

Chaque changement de valeur d'un champ nécessite de déclencher un `setState` et donc un re-rendu du composant, ce qui peut entraîner une baisse des performances, surtout dans les formulaires avec de nombreux champs ou des formulaires dynamiques.

React.js

Explication

Chaque changement de valeur d'un champ nécessite de déclencher un `setState` et donc un re-rendu du composant, ce qui peut entraîner une baisse des performances, surtout dans les formulaires avec de nombreux champs ou des formulaires dynamiques.

Question

Comment faire pour éviter les re-rendu du composant ?

React.js

Explication

Chaque changement de valeur d'un champ nécessite de déclencher un `setState` et donc un re-rendu du composant, ce qui peut entraîner une baisse des performances, surtout dans les formulaires avec de nombreux champs ou des formulaires dynamiques.

Question

Comment faire pour éviter les re-rendu du composant ?

Réponse

Utiliser **React Hook Form** (RHF)

React.js

React Hook Form

- Bibliothèque puissante pour gérer les formulaires dans les applications **React**
- Conçu pour optimiser les performances et simplifier la gestion des formulaires.
- Ayant plusieurs avantages :
  - Optimisation des performances
  - Gestion simplifiée des erreurs de validation
  - Simplification du code

React.js

React Hook Form

- Bibliothèque puissante pour gérer les formulaires dans les applications **React**
- Conçu pour optimiser les performances et simplifier la gestion des formulaires.
- Ayant plusieurs avantages :
  - Optimisation des performances
  - Gestion simplifiée des erreurs de validation
  - Simplification du code

Pour installer `RHF`

```
npm install react-hook-form
```

React.js

Commençons par supprimer tout le code suivant

```
const [formData, setFormData] = useState({
    nom: '',
    prenom: '',
    age: ''
});

const [errors, setErrors] = useState({});

const traiterValeur = (e) => {

    const { name, value } = e.target;
    setFormData({
        ...formData,
        [name]: value
    });
};

const validate = async () => {
    try {
        await validationSchema.validate(formData, { abortEarly: false });
        return {};
    } catch (validationErrors) {
        const newErrors = {};
        validationErrors.inner.forEach((error) => {
            newErrors[error.path] = error.message;
        });
        return newErrors;
    }
};
```

React.js

Pour utiliser `Yup` avec `RHF`, il faut installer

```
npm install @hookform/resolvers
```

React.js

Pour utiliser `Yup` avec `RHF`, il faut installer

```
npm install @hookform/resolvers
```

Utilisons le hook `useForm` pour la validation du formulaire tout en spécifiant l'utilisation de `Yup`

```
const { register, handleSubmit, formState: { errors } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

React.js

Pour utiliser `Yup` avec `RHF`, il faut installer

```
npm install @hookform/resolvers
```

Utilisons le hook `useForm` pour la validation du formulaire tout en spécifiant l'utilisation de `Yup`

```
const { register, handleSubmit, formState: { errors } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

Les imports nécessaires

```
import { yupResolver } from '@hookform/resolvers/yup';
import { useForm } from 'react-hook-form';
```

React.js

Explication

- `resolver: yupResolver(validationSchema)` : permet d'indiquer à **React Hook Form** d'utiliser **Yup** pour la validation.
- `register` de **React Hook Form** sera utilisé pour connecter les champs du formulaire.
- `handleSubmit` est la fonction qui gère la soumission du formulaire.
- `formState.errors` est un objet contenant les erreurs de validation : les messages d'erreur seront accessibles via `errors.nom.message`, `errors.prenom.message`, etc.

React.js

Utilisons `register` pour la construction des champs et `handleSubmit` qui prend comme paramètre une callback qui sera exécutée si le formulaire est valide

```
return (
    <form onSubmit={handleSubmit(ajouterPersonne)}>
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" {...register('nom')} />
            {errors.nom && <span style={{ color: 'red' }}>{errors.nom.message}</span>}

        </div>
        <div>
            <label htmlFor="prenom">Prénom :</label>
            <input id="prenom" {...register('prenom')} />
            {errors.prenom && <span style={{ color: 'red' }}>{errors.prenom.message}</span>}

        </div>
        <div>
            <label htmlFor="age">Âge :</label>
            <input id="age" type="number" {...register('age')} />
            {errors.age && <span style={{ color: 'red' }}>{errors.age.message}</span>}

        </div>
        <button type="submit">Ajouter</button>
    </form>
);
```

React.js

Et enfin la fonction `ajouterPersonne`

```
const ajouterPersonne = (formData) => {
    console.log('Formulaire soumis avec succès', formData);
};
```

React.js

Et enfin la fonction `ajouterPersonne`

```
const ajouterPersonne = (formData) => {
    console.log('Formulaire soumis avec succès', formData);
};
```


Constat

Vérifiez que `called` ne s'affiche plus après chaque modification de valeur d'un champ du formulaire, il s'affiche uniquement à la soumission du formulaire.

React.js

Question

Comment désactiver le bouton et le réactiver lorsqu'il est valide ?

React.js

Question

Comment désactiver le bouton et le réactiver lorsqu'il est valide ?

Récupérons `isValid` **de** `formState`

```
const { register, handleSubmit, formState: { errors, isValid } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

React.js

Question

Comment désactiver le bouton et le réactiver lorsqu'il est valide ?

Récupérons `isValid` de `formState`

```
const { register, handleSubmit, formState: { errors, isValid } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

Utilisons `isValid` pour activer/désactiver le bouton

```
<button type="submit" disabled={!isValid}>Ajouter</button>
```

React.js

Autres propriétés de `formState`

- `isDirty` : indique si au moins un champ du formulaire a été modifié depuis le rendu initial.
- `isSubmitting` : indique si le formulaire est en cours de soumission.
- `isSubmitted` : indique si le formulaire a été soumis au moins une fois.
- `isSubmitSuccessful` : indique si le formulaire a été soumis avec succès.
- `touchedFields` : objet contenant les champs touchés (interagis par l'utilisateur).
- `isValidating` : indique si la validation est en cours d'exécution.
- `submitCount` : le nombre de fois que le formulaire a été soumis.
- `dirtyFields` : objet contenant tous les champs modifiés par rapport à leur valeur initiale.

React.js

Récupérons `isSubmitSuccessful` **de** `formState`

```
const { register, handleSubmit, formState: { errors, isValid, isSubmitSuccessful } } = useForm
   ({
    resolver: yupResolver(validationSchema)
});
```

React.js

Récupérons `isSubmitSuccessful` **de** `formState`

```
const { register, handleSubmit, formState: { errors, isValid, isSubmitSuccessful } } = useForm
   ({
    resolver: yupResolver(validationSchema)
});
```

Utilisons `isSubmitSuccessful` **pour afficher un message de confirmation**

```
{
    isSubmitSuccessful &&
    <span style={{ color: 'white', backgroundColor: 'dodgerblue' }}>
        Ajout effectué avec succès
    </span>
}
```

React.js

Question

Comment vider les champs d'un formulaire avec une soumission avec succès ?

React.js

Question

Comment vider les champs d'un formulaire avec une soumission avec succès ?

Récupérons `isValid` **de** `useForm`

```
const { register, handleSubmit, reset, formState: { errors, isValid } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

React.js

Question

Comment vider les champs d'un formulaire avec une soumission avec succès ?

Récupérons `isValid` de `useForm`

```
const { register, handleSubmit, reset, formState: { errors, isValid } } = useForm({
    resolver: yupResolver(validationSchema)
});
```

Utilisons `reset` dans `ajouterPersonne` pour vider les champs du formulaire

```
const ajouterPersonne = (formData) => {
    console.log('Formulaire soumis avec succès', formData);
    reset()
};
```

React.js

`reset` **peut également être utilisée pour initialiser les champs du formulaire avec des valeurs par défaur**

```
const ajouterPersonne = (formData) => {
    console.log('Formulaire soumis avec succès', formData);
    reset({nom: "Doe", prenom: "John", age: 0})
};
```

React.js

Autres propriétés de `useForm`

- `getValues` : récupère les valeurs actuelles des champs du formulaire.
- `setError` : définit manuellement une erreur sur un champ spécifique.
- `unregister` : permet de dés-enregistrer un champ, ce qui signifie que ce champ ne sera plus pris en compte dans la validation ni dans la collecte des données du formulaire.
- `handleReset` : similaire à `handleSubmit`, mais pour la gestion des événements de réinitialisation des formulaires.
- `watch` : permet de surveiller en temps réel les valeurs d'un ou plusieurs champs d'un formulaire.
- ...

React.js

Pour observer un champ

```
const nom = watch("nom");
```

React.js

Pour observer un champ

```
const nom = watch("nom");
```

Pour observer plusieurs champs

```
const [nom, prenom] = watch(["nom", "prenom"]);
```

React.js

Pour observer un champ

```
const nom = watch("nom");
```

Pour observer plusieurs champs

```
const [nom, prenom] = watch(["nom", "prenom"]);
```

Pour observer tous les champs

```
const allFields = watch();
```

React.js

Pour observer un champ

```
const nom = watch("nom");
```

Pour observer plusieurs champs

```
const [nom, prenom] = watch(["nom", "prenom"]);
```

Pour observer tous les champs

```
const allFields = watch();
```

Ainsi on peut les afficher dans la vue

```
<p>Nom complet : {nom} {prenom}</p>
```

React.js

Question

Comment utiliser RHF sans Yup ?

React.js

Question

Comment utiliser RHF sans Yup ?

Commençons par supprimer le resolver de Yup dans useForm

```
const { register, handleSubmit, watch, reset, formState: { errors, isValid, isSubmitSuccessful
  } } = useForm();
```

React.js

Question

Comment utiliser RHF sans Yup ?

Commençons par supprimer le resolver de Yup dans useForm

```
const { register, handleSubmit, watch, reset, formState: { errors, isValid, isSubmitSuccessful
  } } = useForm();
```

Supprimons également le schéma de validation

```
const validationSchema = yup.object().shape({
    nom: yup.string().required().max(30),
    prenom: yup.string().required().matches(/^[A-Z]{1}.*/),
    age: yup.number().min(18).max(150),
});
```

React.js

Question

Comment utiliser RHF sans Yup ?

Commençons par supprimer le resolver de Yup dans useForm

```
const { register, handleSubmit, watch, reset, formState: { errors, isValid, isSubmitSuccessful
  } } = useForm();
```

Supprimons également le schéma de validation

```
const validationSchema = yup.object().shape({
    nom: yup.string().required().max(30),
    prenom: yup.string().required().matches(/^[A-Z]{1}.*/),
    age: yup.number().min(18).max(150),
});
```

Et l'import suivant

```
import { yupResolver } from '@hookform/resolvers/yup';
```

React.js

Définissons les validateurs directement dans `register` et supprimons la condition de désactivation du bouton

```
return (
    <form onSubmit={handleSubmit(ajouterPersonne)}>
        {isSubmitSuccessful && <span style={{ color: 'white', backgroundColor: 'dodgerblue' }}>
          Ajout effectué avec succès</span>}
        <div>
            <label htmlFor="nom">Nom :</label>
            <input id="nom" {...register('nom', { required: true, maxLength: 30 })} />
            {errors.nom && <span style={{ color: 'red' }}>{errors.nom.message}</span>}

        </div>
        <div>
            <label htmlFor="prenom">Prénom :</label>
            <input id="prenom" {...register('prenom', { required: true, pattern: /^[A-Z]{1}.*/
              })} />
            {errors.prenom && <span style={{ color: 'red' }}>{errors.prenom.message}</span>}

        </div>
        <div>
            <label htmlFor="age">Âge :</label>
            <input id="age" type="number" {...register('age', { min: 18, max: 150 })} />
            {errors.age && <span style={{ color: 'red' }}>{errors.age.message}</span>}

        </div>
        <button type="submit">Ajouter</button>
    </form>
);
```

React.js

Explication

En cas de données invalides, les messages d'erreur ne s'affichent pas malgré la soumission du formulaire.

React.js

Explication

En cas de données invalides, les messages d'erreur ne s'affichent pas malgré la soumission du formulaire.

Question

Comment faire pour afficher des messages d'erreur ?

React.js

Explication

En cas de données invalides, les messages d'erreur ne s'affichent pas malgré la soumission du formulaire.

Question

Comment faire pour afficher des messages d'erreur ?

Réponse

Il faut définir un message d'erreur pour chaque validateur ?

React.js

Ajoutons les messages d'erreur suivant pour le champ `nom` par exemple

```
<div>
    <label htmlFor="nom">Nom :</label>
    <input
        id="nom"
        {...register('nom', {
            required: 'Le nom est obligatoire',
            maxLength: { value: 30, message: 'Vous ne devez pas dépasser 30 caractères'
        }})}
     />
    {errors.nom && <span style={{ color: 'red' }}>{errors.nom.message}</span>}

</div>
```

React.js

Ajoutons les messages d'erreur suivant pour le champ `nom` par exemple

```
<div>
    <label htmlFor="nom">Nom :</label>
    <input
        id="nom"
        {...register('nom', {
            required: 'Le nom est obligatoire',
            maxLength: { value: 30, message: 'Vous ne devez pas dépasser 30 caractères'
        }})}
     />
    {errors.nom && <span style={{ color: 'red' }}>{errors.nom.message}</span>}

</div>
```


Testez et vérifiez que les messages d'erreur s'affichent à la soumission du formulaire.

React.js

Validateurs supportés par **RHF**

- `required`
- `minLength`
- `maxLength`
- `min`
- `max`
- `pattern`
- `validate`

React.js

Nous pouvons utiliser `validate` pour définir un ou plusieurs validateurs personnalisés

```
<div>
    <label htmlFor="age">Âge :</label>
    <input id="age" type="number" {...register('age', {
        validate: {
            notEmpty: (value) => value.trim() !== '' || "L'âge ne peut pas être vide",
            between: (value) => (value > 18 && value < 150) || "L'âge doit être compris entre
               18 et 150",
        },
    })} />
    {errors.age && <span style={{ color: 'red' }}>{errors.age.message}</span>}
</div>
```

React.js

Nous pouvons utiliser `validate` pour définir un ou plusieurs validateurs personnalisés

```
<div>
    <label htmlFor="age">Âge :</label>
    <input id="age" type="number" {...register('age', {
        validate: {
            notEmpty: (value) => value.trim() !== '' || "L'âge ne peut pas être vide",
            between: (value) => (value > 18 && value < 150) || "L'âge doit être compris entre
               18 et 150",
        },
    })} />
    {errors.age && <span style={{ color: 'red' }}>{errors.age.message}</span>}
</div>
```


Testez et vérifiez que les messages d'erreur relatif à l'âge s'affichent à la soumission du formulaire.

React.js

Pour afficher les messages d'erreurs à chaque changement de valeur (et non à la soumission du formulaire)

```
const { register, handleSubmit, reset, formState: { errors, isSubmitSuccessful } } = useForm({
    mode: 'onChange'
});
```

React.js

Pour afficher les messages d'erreurs à chaque changement de valeur (et non à la soumission du formulaire)

```
const { register, handleSubmit, reset, formState: { errors, isSubmitSuccessful } } = useForm({
    mode: 'onChange'
});
```

Testez et vérifiez que les messages d'erreur s'affichent au changement de valeurs.

React.js

Les modes de validation disponibles

- `onSubmit` (par défaut)
- `onChange`
- `onBlur`
- `all`
- `onTouched`