Fonctionnement de la personnalisation du widget de signalement d'anomalie

Liste de Ioc du widget

Voici une explication du fonctionnement des IoC (Inversion of Control) de type Input, Form, Drawing et Service dans la classe Reporting :

IoC de type Input

L'IoC de type Input (ex : InputActionByDefaut) permet de gérer la saisie d'informations directement sur la carte, comme la sélection d'un point géographique par un clic. Il encapsule la logique d'interaction avec la carte (écoute des événements, récupération des coordonnées, affichage d'un marqueur) et expose des méthodes pour activer/désactiver cette saisie, récupérer ou réinitialiser les données.

Dans Reporting, il est instancié et branché à la carte via setMap() et utilisé lors de la première étape du processus de signalement.

Implementation générique

Voici un exemple d'implémentation personnalisée d'un IoC de type Input pour la classe Reporting.

Vous pouvez créer votre propre classe héritant de InputActionByDefaut (ou sans heritage avec toutes les fonctions de base) et l'injecter dans le composant Reporting :

```
import InputActionByDefaut from "./Reporting.js";

class CustomInputAction extends InputActionByDefaut {
    constructor(map) {
        super(map);
    }

    // Redéfinir la méthode _handler pour personnaliser la saisie sur la
carte
    _handler(e) {
        super._handler(e);
        // Ajoutez ici votre logique personnalisée, par exemple :
        console.log("Coordonnée saisie :", this.coordinate);
        // Vous pouvez aussi stocker des infos supplémentaires dans
this.data
    }
}

export default CustomInputAction;
```

Utilisation dans Reporting

```
// ...existing code...
import CustomInputAction from "./CustomInputAction";

// ...après avoir instancié Reporting...
const reporting = new Reporting();
const customInput = new CustomInputAction();
reporting.setComponentInput(customInput);
// ...puis ajoutez le contrôle à la carte...
map.addControl(reporting);
// ...existing code...
```

Explication :

> Vous pouvez ainsi remplacer dynamiquement la logique de saisie sur la carte (IoC Input) sans modifier le cœur de la classe Reporting. Cela permet d'adapter le comportement à vos besoins spécifiques (ex : gestion d'un clic droit, ajout de popups, etc.).

Exemple concret

(code JS directement intégrable sur une page HTML)

> Le deplacement sur la carte enregistre la saisie du centre de la carte

cf. Exemple fonctionnel

API

le principe d'IoC (Inversion of Control) permet à la classe Reporting d'utiliser des composants externes (Input, Form, Drawing, Service) qui implémentent une API précise.

Chaque IoC (par exemple InputActionByDefaut) expose des méthodes standardisées comme active(), disable(), clear(), getData(), et des méthodes de configuration (setMap(), setForm(), etc.).

La classe Reporting utilise uniquement cette API commune pour piloter chaque étape du workflow, sans dépendre de l'implémentation interne de chaque action. Cela permet de remplacer facilement un composant IoC par un autre (par exemple, un Input personnalisé) tant qu'il respecte l'API attendue.

Exemple d'API attendue pour un IoC Input :

- setMap(map)
- setIcon(icon)
- active()
- disable()
- clear()
- getData()

Avantage :

> La classe Reporting reste générique, modulaire et facilement extensible, car elle interagit avec ses IoC uniquement via leur API publique, sans connaître leur fonctionnement interne.

Limites

L'utilisation du IoC Input permet de personnaliser la logique de saisie sur la carte sans modifier le cœur de la classe Reporting. Cependant, certaines limites existent :

- API contractuelle : Le composant injecté doit impérativement respecter l'API attendue (méthodes comme setMap, active, getData, etc.). Si une méthode manque ou ne respecte pas la signature, l'intégration peut échouer ou provoquer des erreurs.
- Intégration avec le workflow : Le IoC Input ne peut agir que sur la partie du workflow qui lui est déléguée (ex : saisie de coordonnées). Les autres étapes (formulaire, dessin, service) restent indépendantes et ne peuvent pas être modifiées via ce composant.
- Dépendances internes : Certaines interactions ou dépendances internes à Reporting (par exemple, la gestion de l'état global ou des transitions entre étapes) ne sont pas accessibles ou modifiables via le IoC Input.
- nteropérabilité : Si le IoC Input personnalisé introduit des comportements très spécifiques, il peut devenir difficile à réutiliser ou à maintenir dans d'autres contextes ou avec d'autres IoC.
- Sécurité et validation : Toute la logique de validation ou de filtrage des données saisies doit être gérée dans le IoC Input lui-même. Le cœur de Reporting ne fait pas de vérification supplémentaire.

En résumé :

> Le IoC Input offre une grande souplesse pour la saisie sur la carte, mais il doit respecter l'API définie et ne peut pas modifier le fonctionnement global du widget au-delà de son périmètre d'action.

IoC de type Form

L'IoC de type Form (ex : FormActionByDefaut) gère la logique du formulaire HTML de signalement (récupération des champs, validation, gestion des erreurs, etc.). Il encapsule l'écoute de la soumission du formulaire, la transformation des données en objet JS, et la gestion de l'état du formulaire (erreur, succès, reset).

Dans Reporting, il est instancié et branché au formulaire DOM via setForm() et utilisé lors de la deuxième étape du processus.

Utilisation dans Reporting

```
// ...existing code...
import CustomFormAction from "./CustomFormAction";

// ...après avoir instancié Reporting...
const reporting = new Reporting();
const customForm = new CustomFormAction();
reporting.setComponentForm(customForm);
// ...puis ajoutez le contrôle à la carte...
map.addControl(reporting);
// ...existing code...
```

Exemple concret

(code JS directement intégrable sur une page HTML)

> Modification des champs du formulaire

cf. Exemple fonctionnel

IoC de type Drawing

L'IoC de type Drawing (ex : DrawingActionByDefaut) gère l'outil de dessin sur la carte (polygones, lignes, etc.). Il encapsule l'initialisation du widget de dessin, la gestion de son affichage, l'export des dessins au format souhaité, et la réinitialisation des données.

Dans Reporting, il est instancié et branché à la carte et au DOM via setMap() et setTarget(), et utilisé lors de l'étape de dessin.

Utilisation dans Reporting

```
// ...existing code...
import CustomDrawingAction from "./CustomDrawingAction";

// ...après avoir instancié Reporting...
const reporting = new Reporting();
const customDrawing = new CustomDrawingAction();
reporting.setComponentDrawing(customDrawing);
// ...puis ajoutez le contrôle à la carte...
map.addControl(reporting);
// ...existing code...
```

IoC de type Service

L'IoC de type Service (ex : ServiceActionByDefaut) gère l'envoi des données de signalement vers un service externe (API, serveur, etc.). Il encapsule la logique d'appel réseau (méthode send()), la gestion des erreurs et le nettoyage des données après envoi.

Dans Reporting, il est instancié et utilisé lors de l'étape d'envoi du signalement.

Les étapes pour mettre en place une classe IoC de type Service

Ceci afin de personnaliser l'envoi des signalements dans le widget Reporting.

1. Créer une classe Service personnalisée

Créez un nouveau fichier, par exemple CustomServiceAction.js, et implémentez une classe qui respecte l'API attendue (au minimum : send(data), clear(), etc.).

```
class CustomServiceAction {
    constructor() {
        // Initialisation éventuelle
    }

    // Méthode appelée par Reporting pour envoyer les données
    async send(data) {
        // Exemple d'appel à une API REST
        try {
            const response = await fetch("https://mon-api/endpoint", {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(data)
            });
            if (!response.ok) throw new Error("Erreur lors de l'envoi");
            return await response.json();
        } catch (err) {
            // Gestion des erreurs
            console.error("Erreur ServiceAction :", err);
            throw err;
        }
    }

    // Méthode pour réinitialiser l'état si besoin
    clear() {
        // Réinitialisation éventuelle
    }
}

export default CustomServiceAction;
```

2. Injecter la classe Service dans Reporting

Dans votre code principal, importez votre classe et injectez-la dans le widget Reporting :

```
import Reporting from "chemin/vers/Reporting";
import CustomServiceAction from "chemin/vers/CustomServiceAction";

const reporting = new Reporting();
const customService = new CustomServiceAction();
reporting.setComponentService(customService);
map.addControl(reporting);
```

3. Respecter l'API attendue

Assurez-vous que votre classe Service implémente bien toutes les méthodes attendues par Reporting (ex : send(data), clear()). Vous pouvez vous inspirer de la classe ServiceActionByDefaut fournie dans le framework.

4. Tester l'intégratio

Déclenchez un signalement depuis l'interface pour vérifier que votre service personnalisé est bien appelé et que les données sont transmises à votre API.

Voir aussi l'exemple fonctionnel pour un cas concret ci-dessous.

Exemple concret

(code JS directement intégrable sur une page HTML)

> Branchement au service d'anomalie du Geoportail

cf. Exemple fonctionnel

Résumé

Chaque IoC est une brique indépendante, injectée dans le workflow du composant Reporting pour gérer une étape précise (saisie sur carte, formulaire, dessin, envoi). Cela permet de remplacer ou personnaliser facilement chaque étape sans modifier la logique globale du composant.