Rusti Framework

Documentation du Système de Formulaires

Version 1.0

Décembre 2025

■ Table des matières

1. Introduction

Le système de formulaires Rusti est inspiré de Django et fournit une validation typée, une sanitisation automatique et une gestion des erreurs intégrée. Il est conçu pour être sûr, performant et facile à utiliser.

Caractéristiques principales

■	Validation typée	Chaque champ a un type Rust natif
■	Sanitisation automatique	Protection XSS intégrée
■	Gestion des erreurs	Erreurs par champ personnalisables
■	Repopulation	Valeurs conservées en cas d'erreur
■	Type-safe	Erreurs de compilation au lieu de runtime
■	Extensible	Créez vos propres champs facilement

2. Installation

Le système de formulaires est inclus dans Rusti. Aucune installation supplémentaire nécessaire.

```
# Cargo.toml
[dependencies]
rusti = "0.1.0"
```

3. Concepts de base

Trait RustiField

Tous les champs implémentent le trait RustiField :

```
pub trait RustiField {
    type Output;
    fn process(&self, raw_value: &str) -> Result<Self::Output, String>;
}
```

• **Output** : Le type Rust natif retourné (String, i64, bool, etc.)

• **process()** : Valide et transforme la valeur brute

Struct Forms

Gère la validation et stocke les résultats :

```
pub struct Forms {
    pub errors: HashMap<String, String>,      // Erreurs par champ
    pub cleaned_data: HashMap<String, Value>, // Données validées
}
```

4. Guide rapide

Étape 1 : Créer un formulaire

```
use rusti::formulaire::{Forms, CharField, EmailField};

let mut form = Forms::new();
```

Étape 2 : Valider les champs

```
form.field("username", &CharField, &raw_username);
form.field("email", &EmailField, &raw_email);
```

Étape 3 : Vérifier la validité

```
if form.is_valid() {
    // Traiter les données
    let username: String = form.cleaned_data.get("username").unwrap();
} else {
    // Afficher les erreurs
    println!("Erreurs : {:?}", form.errors);
}
```

5. Types de champs disponibles

Champ	Type de sortie	Description
CharField	String	Texte court avec sanitisation
TextField	String	Texte long (textarea)
PasswordField	String	Mot de passe avec hash Argon2
EmailField	String	Email avec validation
IntegerField	i64	Nombre entier
FloatField	f64	Nombre décimal
BooleanField	bool	Checkbox/Booléen
DateField	NaiveDate	Date (JJ-MM-AAAA)
DateTimeField	NaiveDateTime	Date et heure
IPAddressField	IpAddr	Adresse IP (v4 ou v6)
URLField	String	URL (http/https)
JSONField	Value	Contenu JSON

5.1 CharField

Champ texte court avec sanitisation automatique.

```
use rusti::formulaire::CharField;

let result = form.field("username", &CharField, "john_doe");
// Type de retour : Option<String>
```

Comportement :

• Supprime les balises HTML

• Trim les espaces

• Sanitise contre XSS

Exemple :

Input	Output
`<script>alert('XSS')</script>john`	`john`
`  hello world`	`hello world`
`<b>Bold</b> text`	`Bold text`

5.2 PasswordField

Champ mot de passe avec hachage Argon2 automatique.

```
use rusti::formulaire::PasswordField;

let result = form.field("password", &PasswordField, "MyP@ssw0rd");
// Type de retour : Option<String> (hash Argon2)
```

■■ Important :

• Le résultat est un **hash**, pas le mot de passe en clair

• Format PHC : $argon2id$v=19$m=4096,t=3,p=1$...

• Validation longueur minimum : 8 caractères

• Ne sanitise PAS (champ sensible)

6. Création d'un formulaire

Méthode recommandée : Wrapper personnalisé

```
use rusti::formulaire::{Forms, CharField, EmailField, IntegerField};

pub struct ContactForm(Forms);

impl ContactForm {
    pub fn new() -> Self {
        Self(Forms::new())
    }

    pub fn name(&mut self, raw_value: &str) -> Option<String> {
        self.0.field("name", &CharField, raw_value)
    }

    pub fn email(&mut self, raw_value: &str) -> Option<String> {
        self.0.field("email", &EmailField, raw_value)
    }

    pub fn age(&mut self, raw_value: &str) -> Option<i64> {
        self.0.field("age", &IntegerField, raw_value)
    }

    pub fn is_valid(&self) -> bool {
        self.0.is_valid()
    }
}
```

8. Validation personnalisée

Validation avec regex

```
use fancy_regex::Regex;

impl UserForm {
    pub fn password(&mut self, raw_value: &str) -> Option<String> {
        let regex = Regex::new(
            r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{10,}$"
        ).unwrap();

        if !regex.is_match(raw_value).unwrap_or(false) {
            self.0.errors.insert(
                "password".to_string(),
                "Le mot de passe doit contenir au moins 10 caractères,
                 une majuscule, une minuscule, un chiffre et un caractère spécial.".to_
            );
            return None;
        }

        self.0.field("password", &PasswordField, raw_value)
    }
}
```

Validation croisée (cross-field)

```
impl RegistrationForm {
    pub fn validate(&mut self, raw: &RegistrationFormRaw) -> bool {
        self.password(&raw.password);
        self.password_confirm(&raw.password_confirm);

        // Validation croisée
        if self.0.is_valid() {
            if raw.password != raw.password_confirm {
                self.0.errors.insert(
                    "password_confirm".to_string(),
                    "Les mots de passe ne correspondent pas.".to_string()
                );
            }
        }

        self.0.is_valid()
    }
}
```

9. Gestion des erreurs

Structure des erreurs

```
pub struct Forms {
    pub errors: HashMap<String, String>,  // Clé = nom du champ
}
```

Accéder aux erreurs

```
// Vérifier si le formulaire est valide
if form.is_valid() {
    // Pas d'erreurs
}

// Accéder aux erreurs
let errors = form.errors();

// Erreur d'un champ spécifique
if let Some(error) = form.errors().get("email") {
    println!("Erreur email: {}", error);
}
```

Messages d'erreur par défaut

Champ	Message d'erreur
EmailField	Format d'email invalide.
IntegerField	Ce champ doit être un nombre entier.
FloatField	Nombre décimal invalide.
PasswordField	Le mot de passe doit contenir au moins 8 caractères.
DateField	Format de date invalide (JJ-MM-AAAA).
URLField	L'URL doit commencer par http:// ou https://

10. Sanitisation automatique

Protection XSS intégrée

Tous les champs texte sont automatiquement sanitisés contre les attaques XSS.

```
let input = "<script>alert('XSS')</script>Hello";
form.field("message", &CharField, input);
// Résultat: "Hello"
```

Ce qui est supprimé

■	Balises <script>
■	Balises <iframe>, <embed>, <object>
■	Événements JavaScript (onclick, onerror, etc.)
■	Protocole javascript:
■	Toutes les balises HTML

Champs sensibles NON sanitisés

Les champs suivants ne sont pas sanitisés pour préserver les caractères spéciaux :

• PasswordField

• Champs contenant 'password', 'token', 'secret', 'key' dans leur nom

11. Utilisation dans les templates

Template HTML avec gestion des erreurs

```
<form method="post">
    {% csrf %}

    <div>
        <label>Email :</label>
        <input name="email"
               value="{{ form.cleaned_data.email | default(value='') }}">
        {% if form.errors.email %}
            <span class="error">{{ form.errors.email }}</span>
        {% endif %}
    </div>

    <button type="submit">Envoyer</button>
</form>
```

Repopulation automatique

Les valeurs valides sont automatiquement conservées en cas d'erreur :

```
// Vue
let context = json!({
    "errors": form.errors,
    "cleaned_data": form.cleaned_data,  // Données valides conservées
});
```

13. Bonnes pratiques

1. Toujours utiliser un wrapper

Créez une struct wrapper pour chaque formulaire au lieu d'utiliser Forms directement.

2. Valider tous les champs avant de vérifier

Appelez toutes les méthodes de validation avant de vérifier is_valid().

3. Ne pas hasher deux fois le mot de passe

PasswordField hash automatiquement. Ne pas hasher à nouveau.

4. Toujours retourner le formulaire en cas d'erreur

Permet la repopulation des champs valides.

5. Créer des méthodes de validation custom

Ajoutez de la logique métier dans vos méthodes de champs.

6. Utiliser FormContext pour les templates

Facilite la sérialisation pour Tera.

14. FAQ

Q : Comment créer un champ personnalisé ?

R : Implémentez le trait RustiField avec votre logique de validation.

Q : Comment gérer les champs optionnels ?

R : Vérifiez si la valeur est vide avant de valider, retournez Some(String::new()) si optionnel.

Q : Comment valider que deux champs sont identiques ?

R : Utilisez la validation croisée dans une méthode validate_all().

Q : Comment désactiver la sanitisation ?

R : Nommez le champ avec 'password', 'token', 'secret' ou 'key', ou créez un champ custom sans sanitisation.

Q : Les données sont-elles automatiquement sanitisées ?

R : Oui, double protection : middleware + RustiField.

Q : Comment gérer les fichiers uploadés ?

R : Non supporté en V1. Utilisez axum::extract::Multipart directement. Prévu pour V2.

Rusti Forms V1.0

Sécurisé • Typé • Performant

Documentation complète

Décembre 2025