Rusti Forms

Système de Formulaires Django-like en Rust

Un système de validation puissant et sûr

Documentation v2.0 Complète

Table des matières

1. Introduction

Rusti Forms est un système de validation de formulaires pour le framework web Rusti, inspiré du système de formulaires de Django. Il combine la simplicité d'utilisation de Django avec la sécurité et les performances de Rust.

Validation automatique avec types de champs prédéfinis
Sanitization XSS intégrée
Trim automatique des espaces
Hash sécurisé des mots de passe (Argon2)
API Django-like familière et intuitive
Type-safe grâce à Rust

2. Quick Start

Installation et premier formulaire en 30 secondes

```
// Cargo.toml
[dependencies]
rusti = "0.1"

// main.rs
use rusti::rusti_form;
use rusti::formulaire::formsrusti::{Forms, FormulaireTrait};
use rusti::formulaire::field::{CharField, EmailField};
use std::collections::HashMap;

#[rusti_form]
pub struct ContactForm {
    pub form: Forms,
}

impl FormulaireTrait for ContactForm {
    fn new() -> Self {
        Self { form: Forms::new() }
    }

    fn validate(&mut self, raw_data: &HashMap<String, String>) -> bool {
        self.require("name", &CharField { allow_blank: false }, raw_data);
        self.require("email", &EmailField, raw_data);
        self.is_valid()
    }
}

// Utilisation
```

```
let mut form = ContactForm::new();
if form.validate(&raw_data) {
    let name: String = form.get_value("name").unwrap();
    // Traiter les données...
}
```

3. Types de champs disponibles

Type	Description	Validation
CharField	Champ texte court	Sanitization XSS, trim
TextField	Champ texte long	Sanitization XSS, trim
EmailField	Email	Format RFC 5322
PasswordField	Mot de passe	Hash Argon2id
IntegerField	Nombre entier	Parse vers i64
FloatField	Nombre décimal	Parse vers f64
BooleanField	Booléen	true/false, 1/0, on/off
DateField	Date	Format YYYY-MM-DD
SlugField	Slug URL-friendly	Lettres, chiffres, tirets
URLField	URL	Format URL valide
IPAddressField	Adresse IP	IPv4 ou IPv6
JSONField	Données JSON	Parse JSON valide

4. Exemples pratiques

4.1 Formulaire simple

```
use rusti::rusti_form;
use rusti::formulaire::formsrusti::{Forms, FormulaireTrait};
use rusti::formulaire::field::{CharField, EmailField, TextField};
use std::collections::HashMap;

#[rusti_form]
pub struct ContactForm {
    pub form: Forms,
}

impl FormulaireTrait for ContactForm {
    fn new() -> Self {
        Self { form: Forms::new() }
    }

    fn validate(&mut self, raw_data: &HashMap<String, String>) -> bool {
        self.require("name", &CharField { allow_blank: false }, raw_data);
        self.require("email", &EmailField, raw_data);
        self.require("message", &TextField, raw_data);
```

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

Utilisation dans un handler Axum

```
pub async fn contact_submit(
    RustiForm(form): ExtractForm<ContactForm>,
) -> Response {
    if form.is_valid() {
        let name: String = form.get_value("name").unwrap();
        let email: String = form.get_value("email").unwrap();
        // Traiter les données...
    } else {
        // Afficher les erreurs
        let mut context = Context::new();
        context.insert("form", &form);
        template.render("contact.html", &context)
    }
}
```

4.2 Validation personnalisée

```
#[rusti_form]
pub struct RegisterForm {
    pub form: Forms,
}

impl FormulaireTrait for RegisterForm {
    fn new() -> Self {
        Self { form: Forms::new() }
    }

    fn validate(&mut self, raw_data: &HashMap<String, String>) -> bool {
        self.require("username", &CharField { allow_blank: false }, raw_data);
        self.require("email", &EmailField, raw_data);

        if let Some(password) = raw_data.get("password") {
            self.validate_password(password);
        } else {
            self.errors.insert("password".to_string(), "Requis".to_string());
        }

        self.is_valid()
    }
}

impl RegisterForm {
    fn validate_password(&mut self, raw_value: &str) -> Option<String> {
        use fancy_regex::Regex;

        let regex = Regex::new(
            r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[A-Za-z\d]{8,}$"
        ).unwrap();

        if raw_value.len() < 8 {
            self.errors.insert(
                "password".to_string(),
                "Minimum 8 caractères".to_string()
            );
            return None;
        }

        if !regex.is_match(raw_value).unwrap_or(false) {
            self.errors.insert(
                "password".to_string(),
                "Doit contenir maj, min et chiffre".to_string()
            );
            return None;
        }

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

```
    }
}
```

4.3 Validation inter-champs

Vérification de cohérence entre plusieurs champs (style Django clean())

```
impl RegisterForm {
    fn clean(&mut self, raw_data: &HashMap<String, String>) {
        // Skip si déjà des erreurs
        if self.is_not_valid() {
            return;
        }

        let username: Option<String> = self.get_value("username");
        let password = raw_data.get("password");

        if let (Some(user), Some(pass)) = (username, password) {
            if pass.to_lowercase().contains(&user.to_lowercase()) {
                self.errors.insert(
                    "password".to_string(),
                    "Le mot de passe ne peut pas contenir le username".to_string()
                );
            }
        }
    }
}

// Utilisation dans validate()
impl FormulaireTrait for RegisterForm {
    fn validate(&mut self, raw_data: &HashMap<String, String>) -> bool {
        // 1. Validation des champs individuels
        self.require("username", &CharField { allow_blank: false }, raw_data);
        self.require("email", &EmailField, raw_data);

        // 2. Validation personnalisée
        if let Some(password) = raw_data.get("password") {
            self.validate_password(password);
        }

        // 3. Validation inter-champs
        self.clean(raw_data);

        // 4. Retour du résultat
        self.is_valid()
    }
}
```

5. Comparaison Django - Rusti

Pour faciliter la transition depuis Django, voici les équivalences entre les deux frameworks :

Django	Rusti	Notes
`forms.CharField(max_length=100)`	`CharField { allow_blank: false }`	Pas de max_length en Rust
`forms.EmailField()`	`EmailField`	Validation RFC 5322
`form.is_valid()`	`self.is_valid()`	Via Deref
`form.cleaned_data["email"]`	`self.get_value::<String>("email")`	Type-safe
`form.errors["email"]`	`self.errors.get("email")`	HashMap standard
`def clean(self):`	`fn clean(&mut self, data: &HashMap)`	Validation inter-champs
`raise ValidationError("msg")`	`self.errors.insert(field, msg)`	Ajout manuel erreur

6. API Référence

Méthode	Description
`require(name, field, data)`	Valide un champ obligatoire
`optional(name, field, data)`	Valide un champ optionnel
`field(name, field, value)`	Valide une valeur brute directement
`is_valid()`	Retourne true si aucune erreur
`is_not_valid()`	Retourne true s'il y a des erreurs
`get_value<T>(name)`	Récupère une valeur validée typée
`errors.insert(name, msg)`	Ajoute une erreur personnalisée
`errors.get(name)`	Récupère l'erreur d'un champ
`cleaned_data.get(name)`	Récupère la valeur brute validée
`clear()`	Réinitialise errors et cleaned_data

7. Pièges courants

7.1 Utilisation redondante de self.form

Ne pas faire :

```
// INCORRECT - Redondant avec Deref
self.form.require("email", &EmailField, raw_data);
self.form.is_valid()
```

Faire :

```
// CORRECT - Direct grâce à la macro #[rusti_form]
self.require("email", &EmailField, raw_data);
self.is_valid()
```

7.2 Type mismatch sur get_value

Ne pas faire :

```
// INCORRECT - Type mismatch
let age: String = self.get_value("age").unwrap();  // age est un IntegerField !
```

Faire :

```
// CORRECT - Type correspondant au champ
let age: i64 = self.get_value("age").unwrap();     // IntegerField -> i64
let email: String = self.get_value("email").unwrap();  // EmailField -> String
```

7.3 Oublier de vérifier is_valid() avant get_value()

Ne pas faire :

```
// INCORRECT - Peut paniquer si validation échouée
let email: String = form.get_value("email").unwrap();  // unwrap() peut échouer !
```

Faire :

```
// CORRECT - Toujours vérifier avant
if form.is_valid() {
    let email: String = form.get_value("email").unwrap();  // Safe ici
} else {
    // Gérer les erreurs
}
```

7.4 Oublier la macro #[rusti_form]

Ne pas faire :

```
// INCORRECT - Pas de macro
#[derive(Serialize, Deserialize, Debug)]
```

```
pub struct UserForm {
    #[serde(flatten)]  // Doit être ajouté manuellement
    pub form: Forms,
}

// Et il faut implémenter Deref manuellement...
```

Faire :

```
// CORRECT - La macro fait tout automatiquement
#[rusti_form]
pub struct UserForm {
    pub form: Forms,
}

// Deref + DerefMut + #[serde(flatten)] ajoutés automatiquement !
```

8. Sécurité intégrée

Fonctionnalité	Description	Champs concernés
Protection XSS	Suppression automatique des balises script, iframe, object, et attributs JavaScript (onclick, onerror	CharField, TextField
Hash Argon2id	Hash sécurisé avec salt unique, résistant aux attaques GPU/ASIC. Format PHC standard.	PasswordField
Trim automatique	Suppression des espaces en début et fin de chaîne	Tous les champs texte
Validation de format	Vérification stricte des formats (email RFC 5322, URL, IP, JSON)	EmailField, URLField, IPAddressField, JSONField

9. Templates HTML

Affichage des erreurs dans les templates Tera

```
<!-- Formulaire avec erreurs -->
<form method="post">
    <div class="form-group">
        <label for="email">Email :</label>
        <input type="email" name="email" id="email"
               value="{{ form.cleaned_data.email | default(value="") }}">

        {% if form.errors.email %}
            <span class="error">{{ form.errors.email }}</span>
        {% endif %}
    </div>

    <div class="form-group">
        <label for="password">Mot de passe :</label>
        <input type="password" name="password" id="password">

        {% if form.errors.password %}
            <span class="error">{{ form.errors.password }}</span>
        {% endif %}
    </div>

    <!-- Afficher toutes les erreurs en haut -->
    {% if form.errors %}
        <div class="alert alert-danger">
            <ul>
            {% for field, message in form.errors %}
                <li><strong>{{ field }}</strong>: {{ message }}</li>
            {% endfor %}
            </ul>
        </div>
    {% endif %}

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

```
</form>
```

10. FAQ (Foire Aux Questions)

Q1 : Comment valider plusieurs champs ensemble ?

```
// Utilisez une méthode clean() comme dans Django
impl MyForm {
    fn clean(&mut self, raw_data: &HashMap<String, String>) {
        if self.is_not_valid() {
            return;  // Skip si déjà des erreurs
        }

        let password: Option<String> = self.get_value("password");
        let confirm: Option<String> = self.get_value("password_confirm");

        if password != confirm {
            self.errors.insert(
                "password_confirm".to_string(),
                "Les mots de passe ne correspondent pas".to_string()
            );
        }
    }
}
```

Q2 : Différence entre require() et optional() ?

require() : Le champ DOIT être présent et non vide. Ajoute une erreur 'Requis' si absent.
optional() : Le champ peut être absent ou vide. Pas d'erreur si absent, mais validation appliquée si présent.

```
// require() - Champ obligatoire
self.require("email", &EmailField, raw_data);
// Si absent ou vide -> Erreur "Requis"

// optional() - Champ facultatif
self.optional("phone", &CharField::new(), raw_data);
// Si absent -> Pas d'erreur
// Si présent -> Validation appliquée
```

Q3 : Comment réutiliser des validateurs personnalisés ?

```
// Créez une fonction helper réutilisable
fn validate_strong_password(form: &mut Forms, field_name: &str, value: &str) {
    let regex = Regex::new(r"^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$").unwrap();

    if !regex.is_match(value).unwrap_or(false) {
        form.errors.insert(
            field_name.to_string(),
            "Mot de passe faible".to_string()
```

```
        );
    }
}

// Utilisation dans plusieurs formulaires
impl RegisterForm {
    fn validate(&mut self, raw_data: &HashMap<String, String>) -> bool {
        // ...
        if let Some(pwd) = raw_data.get("password") {
            validate_strong_password(self, "password", pwd);
        }
        self.is_valid()
    }
}
```

Q4 : Que fait exactement la macro #[rusti_form] ?

La macro génère automatiquement :
1. **#[derive(Serialize, Deserialize, Debug)]** si pas déjà présent
2. **#[serde(flatten)]** sur le champ Forms
3. **impl Deref** pour accéder directement aux méthodes de Forms
4. **impl DerefMut** pour les méthodes mutables

Q5 : Peut-on avoir plusieurs champs Forms dans une struct ?

Non. La macro #[rusti_form] nécessite exactement UN champ de type Forms. Si vous avez besoin de formulaires imbriqués, créez des structs séparées et combinez-les manuellement.

11. Index alphabétique des méthodes

Méthode	Page	Catégorie
`clean()`	5, 11	Validation inter-champs
`clear()`	8	Réinitialisation
`field()`	8	Validation directe
`get_value<T>()`	4, 8	Récupération données
`is_not_valid()`	8	Vérification
`is_valid()`	4, 8	Vérification
`new()`	4	Construction
`optional()`	8, 11	Validation champs
`require()`	4, 8, 11	Validation champs
`validate()`	4	Trait FormulaireTrait

Conclusion

Rusti Forms offre un système de validation puissant et sûr, combinant la familiarité de Django avec la robustesse de Rust. La macro **#[rusti_form]** réduit considérablement le code boilerplate, permettant aux développeurs de se concentrer sur la logique métier. La sécurité est intégrée par défaut avec sanitization XSS, hash Argon2, et validation stricte des formats.

Pour aller plus loin, consultez la documentation complète du framework Rusti et les exemples dans le repository GitHub.