ReadRSS – Documentation technique et utilisateur

Lecteur RSS/Atom local, rapide et sécurisé.
Version 0.12 — 29/10/2025.

Projet Rust par Anthony MITTICA, Dan SPACEY, Léo ROBINAND, et Valentin ROLLY

Table des matières

1. Introduction

1.1. Contexte du projet

ReadRSS est un projet open source développé en Rust, visant à offrir un lecteur RSS simple, rapide et sécurisé. Il a été conçu pour répondre aux besoins des utilisateurs souhaitant suivre leurs flux d'actualités et blogs préférés sans les complexités des solutions existantes.

1.2. Objectifs et promesse utilisateur

L'objectif principal de ReadRSS est de fournir une expérience utilisateur fluide, avec une interface intuitive et des performances optimales. L'application promet une lecture rapide des flux RSS, une gestion efficace des articles, et une sécurité renforcée pour protéger les données des utilisateurs.

1.3. Public visé (utilisateurs / développeurs)

ReadRSS s'adresse à deux types de public : les **utilisateurs finaux** qui cherchent une solution légère et efficace pour lire leurs flux RSS, et les **développeurs** souhaitant contribuer à un projet open source moderne écrit en Rust.

1.4. Technologies utilisées (Rust, egui, tokio, reqwest…)

Le projet utilise plusieurs technologies clés : **Rust** pour la robustesse et la performance, **egui** pour l'interface graphique, **tokio** pour la gestion asynchrone des tâches, et **reqwest** pour les requêtes HTTP. D'autres bibliothèques comme **serde** pour la sérialisation et **tracing** pour le logging complètent l'écosystème technique.

---

2. Vue d’ensemble du projet

2.1. Description fonctionnelle

ReadRSS permet aux utilisateurs d'ajouter, parcourir et lire des flux RSS. L'application se distingue par sa simplicité d'utilisation et son interface épurée, tout en offrant des fonctionnalités avancées comme la déduplication des articles et la personnalisation des thèmes.

2.2. Philosophie de conception (simplicité, rapidité, sécurité)

La philosophie de ReadRSS repose sur trois piliers : **simplicité** pour une prise en main immédiate, **rapidité** pour un affichage fluide des articles, et **sécurité** pour protéger les données des utilisateurs contre les flux malveillants.

2.3. Schéma d’architecture générale

L'architecture de ReadRSS est modulaire, séparant clairement la logique métier (`rss-core`) de l'interface utilisateur (`rss-gui`). Cette séparation facilite la maintenance et l'évolution du projet.

2.4. Flux logique du traitement RSS

Le flux de traitement commence par l'ajout d'un flux RSS par l'utilisateur. L'application télécharge

périodiquement les nouveaux articles, les stocke localement, et les affiche dans l'interface. Un mécanisme de déduplication évite les doublons, et les articles sont marqués comme "lus" ou "non lus" pour un suivi optimal.

---

3. Installation et utilisation

3.1. Prérequis techniques

Pour installer ReadRSS, il est nécessaire d'avoir **Rust** (stable (>= 1.80)) et **Cargo** installés sur votre système. Ces outils sont essentiels pour compiler le projet depuis les sources.

3.2. Installation depuis les sources

Build local :

```
./scripts/build_release.sh
```

Paquet Debian :

```
./scripts/build_deb.sh
```

Releases GitHub disponibles : binaires Linux (.tar.gz + .deb) et Windows (.zip).

3.3. Lancement de l’application

Une fois compilé, lancez l'application avec la commande `cargo run -p rss-gui`. Une interface graphique s'ouvrira, permettant d'ajouter et de consulter les flux RSS.

3.4. Structure des dossiers et fichiers de configuration

Le projet est organisé en deux crates principales : `rss-core` pour la logique métier et `rss-gui` pour l'interface. Les fichiers de configuration sont stockés au format JSON, facilitant leur modification par l'utilisateur.

---

4. Architecture logicielle

4.1. Organisation du workspace Cargo

Le projet est structuré en un workspace Cargo, avec une séparation claire entre les modules. Cela permet une meilleure organisation du code et une gestion simplifiée des dépendances.

4.2. Description des deux crates : rss-core : logique interne & rss-gui : interface graphique

rss-core : Contient la logique métier, comme la gestion des flux, le téléchargement des articles, et la persistance des données.

rss-gui : Gère l'interface utilisateur, construite avec **egui**, et interagit avec `rss-core` via des canaux asynchrones.

4.3. Dépendances principales (tokio, reqwest, serde, egui, tracing…)

Les dépendances incluent **tokio** pour l'asynchrone, **reqwest** pour les requêtes HTTP, **serde** pour la sérialisation, et **egui** pour l'interface graphique. **tracing** est utilisé pour le logging et le débogage.

4.4. Communication interne (canaux async, tâches concurrentes)

La communication entre les modules se fait via des canaux asynchrones, permettant un traitement concurrent des tâches sans bloquer l'interface utilisateur.

---

5. Modules du cœur (rss-core)

Pour chaque module :

- **Rôle**
- **Structures clés**
- **Fonctionnement interne**
- **Exemple d'utilisation**

5.1. `config.rs` — Configuration et AppConfig

Rôle

Ce module est responsable de la gestion de la configuration globale de l'application ReadRSS. Il permet de définir les paramètres par défaut pour le thème, les flux RSS et l'interface utilisateur. Il assure également le chargement et la sauvegarde de la configuration dans un fichier JSON, tout en fournissant des méthodes pour convertir les couleurs en un format compatible avec la bibliothèque `egui`.

Structures Clés

`AppConfig` La structure `AppConfig` est la racine de la configuration de l'application. Elle regroupe trois sections principales :

- `theme` : Configuration des couleurs et du thème visuel.
- `feeds` : Paramètres liés à la récupération des flux RSS, tels que l'intervalle de mise à jour, le timeout des requêtes, et le nombre de tentatives de reconnexion.
- `ui` : Paramètres d'interface utilisateur, comme la taille de la police, la largeur des panneaux, et l'affichage des aperçus d'articles.

Les méthodes associées à `AppConfig` incluent :

- `config_file_path()` : Détermine le chemin du fichier de configuration dans le dossier de configuration de l'utilisateur.
- `load()` : Charge la configuration depuis le fichier. En cas d'erreur, utilise une configuration par défaut et la sauvegarde.
- `save()` : Sérialise et sauvegarde la configuration dans le fichier JSON.
- `update_theme()`, `update_feeds()`, `update_ui()` : Met à jour une section spécifique de la configuration et persiste les modifications.

`ThemeConfig` La structure `ThemeConfig` définit les couleurs de l'interface, telles que la

couleur de fond, la couleur des panneaux, la couleur du texte, etc. Elle propose des valeurs par défaut pour un thème sombre. Les méthodes utilitaires comme `background_color32()` convertissent les couleurs en `egui::Color32` pour une utilisation dans l'interface graphique.

`FeedConfig` La structure `FeedConfig` contient les paramètres liés à la récupération des flux RSS :

- `update_interval_minutes` : Intervalle de mise à jour des flux, défini à 30 minutes par défaut.
- `max_articles_per_feed` : Nombre maximal d'articles à conserver par flux, défini à 100 par défaut.
- `request_timeout_seconds` : Timeout pour les requêtes HTTP, défini à 10 secondes par défaut.
- `retry_attempts` : Nombre de tentatives en cas d'échec de téléchargement, défini à 3 par défaut.

`UiConfig` La structure `UiConfig` gère les paramètres d'interface utilisateur :

- `font_size` : Taille de la police, définie à 14.0 par défaut.
- `left_panel_width` : Largeur du panneau latéral, définie à 300.0 pixels par défaut.
- `show_article_preview` : Affiche un aperçu des articles, activé par défaut.
- `articles_per_page` : Nombre d'articles affichés par page, défini à 20 par défaut.

Fonctionnement Interne

Le chargement de la configuration se fait en vérifiant d'abord l'existence du fichier `config.json`. Si le fichier n'existe pas, une configuration par défaut est créée et sauvegardée. Les modifications apportées à la configuration sont sauvegardées immédiatement dans le fichier JSON. Les couleurs sont converties en `egui::Color32` pour être utilisées par l'interface graphique.

Exemple d'Utilisation

```
let mut config = AppConfig::load();
config.update_theme(ThemeConfig {
    background_color: [40, 40, 40],
    ..Default::default()
}).unwrap();
config.save().unwrap();
```

5.2. `data.rs` — Persistance (DataApi)

Rôle

Ce module gère la persistance des données de l'application, notamment la liste des flux RSS, l'état de lecture des articles, et le cache des articles téléchargés. Il utilise des verrous asynchrones pour garantir la sécurité des accès concurrents.

Structures Clés

`DataApi` La structure `DataApi` est l'interface principale pour interagir avec les données persistées. Elle contient :

- `feeds` : Liste partagée des flux RSS.
- `read_inner` : État de lecture des articles.
- `articles_inner` : Cache des articles téléchargés.

- `feeds_path`, `read_path`, `articles_path` : Chemins vers les fichiers JSON.

`ReadData` La structure `ReadData` stocke les articles lus sous forme de `HashMap<String, HashSet<String>>`, où la clé est l'ID du flux et la valeur est un ensemble d'IDs d'articles lus.

Fonctionnalités Principales

La méthode `load_from_dir()` initialise `DataApi` en chargeant les fichiers JSON depuis un dossier. Elle utilise une stratégie de fallback en cas d'erreur de lecture, en tentant de lire un fichier temporaire. Les méthodes `persist_feeds()`, `persist_read()`, et `persist_articles()` sauvegardent les données de manière atomique en utilisant un fichier temporaire.

Les méthodes `add_feed()`, `remove_feed()`, et `list_feeds()` gèrent les opérations sur la liste des flux. Les méthodes `is_read()`, `mark_read()`, et `upsert_articles()` gèrent les articles, en évitant les doublons et en persistant les modifications.

Exemple d'Utilisation

```
let feeds = shared_feed_list(vec![]);
let data_api = DataApi::load_from_dir(feeds, "./config").await;
data_api.add_feed(FeedDescriptor {
    id: "feed1".to_string(),
    title: "Mon Flux".to_string(),
    url: "https://example.com/feed.xml".to_string(),
}).await;
```

5.3. `feed.rs` — Modélisation des flux et articles

Rôle

Ce module définit les structures de données pour représenter les flux RSS et leurs articles, ainsi que les fonctions pour les manipuler.

Structures Clés

`FeedDescriptor` La structure `FeedDescriptor` décrit un flux RSS avec les champs suivants :

- `id` : Identifiant unique du flux.
- `title` : Titre du flux.
- `url` : URL du flux.

`FeedEntry` La structure `FeedEntry` représente un article dans un flux RSS. Elle contient des champs tels que :

- `feed_id` : ID du flux parent.
- `title`, `summary`, `url` : Métadonnées de l'article.
- `published_at` : Date de publication.
- `guid` : Identifiant unique de l'article.
- `content_html` : Contenu HTML de l'article.
- `image_url` : URL d'une image associée.

Fonctionnalités Principales

Les méthodes `from_rss_item()` et `from_atom_entry()` convertissent respectivement un `rss::Item` et un `atom::Entry` en `FeedEntry`. La méthode `identity()` génère une clé unique pour chaque article, utilisée pour éviter les doublons.

Les fonctions `shared_feed_list()`, `add_feed()`, `remove_feed()`, et `list_feeds()` gèrent les opérations sur la liste des flux.

Exemple d'Utilisation

```
let feeds = shared_feed_list(vec![]);
add_feed(&feeds, FeedDescriptor {
    id: "feed1".to_string(),
    title: "Mon Flux".to_string(),
    url: "https://example.com/feed.xml".to_string(),
}).await;
```

5.4. `poller.rs` — Téléchargement périodique et retries

Rôle

Ce module gère le téléchargement périodique des flux RSS. Il récupère les flux à intervalle régulier, gère les erreurs et les nouvelles tentatives, et émet des événements lorsque de nouveaux articles sont disponibles.

Structures Clés

`PollConfig` La structure `PollConfig` définit la configuration du poller avec les champs suivants :

- `interval` : Intervalle entre deux téléchargements.
- `request_timeout` : Timeout pour les requêtes HTTP.
- `max_retries` : Nombre maximal de tentatives en cas d'échec.
- `retry_backoff_ms` : Délai entre les tentatives.

`PollerHandle` La structure `PollerHandle` gère le cycle de vie du poller avec la méthode `stop()` pour arrêter proprement la tâche.

`Event` L'énumération `Event` représente les événements émis par le poller, comme `NewArticles(String, Vec<FeedEntry>)`.

Fonctionnalités Principales

La fonction `spawn_poller()` lance une tâche asynchrone qui télécharge les flux à intervalle régulier. La fonction `fetch_feed()` télécharge et parse un flux RSS ou Atom, en vérifiant que l'URL utilise HTTPS. La fonction `fetch_feed_with_retries()` implémente un mécanisme de retries avec un backoff exponentiel.

Exemple d'Utilisation

```
let config = PollConfig::default();
let client = reqwest::Client::new();
let (update_tx, _update_rx) = mpsc::channel(32);
let seen = SeenStore::in_memory();
let feeds = shared_feed_list(vec![]);
let handle = spawn_poller(feeds, config, client, update_tx, seen);
```

```
handle.stop().await.unwrap();
```

5.5. `storage.rs` — Déduplication persistée (SeenStore)

Rôle

Ce module gère un magasin persistant pour suivre les articles déjà vus et éviter les doublons.

Structures Clés

`SeenData` La structure `SeenData` stocke les articles vus sous forme de `HashMap<String, HashSet<String>>`.

`SeenStore` La structure `SeenStore` est l'interface pour interagir avec le magasin. Elle peut être en mémoire ou persistée dans un fichier JSON.

Fonctionnalités Principales

La méthode `is_new_and_mark()` vérifie si un article est nouveau et le marque comme vu. La méthode `persist()` sauvegarde l'état dans un fichier JSON si un chemin est configuré.

Exemple d'Utilisation

```
rust
 Copier
let seen = SeenStore::load_from("./config/seen.json").await;
let entry = FeedEntry { ... };
if seen.is_new_and_mark(&entry).await {
    println!("Nouvel article !");
}
```

5.6. `error.rs` — Gestion centralisée des erreurs

Rôle

Ce module définit les erreurs possibles dans le moteur de polling et de parsing, en utilisant `thiserror` pour des messages clairs.

Structure Clé

`PollError` L'énumération `PollError` contient les variantes suivantes :

- `Network` : Erreur réseau.
- `Parse` : Erreur de parsing.
- `Task` : Erreur dans une tâche asynchrone.
- `UpdateChannelClosed` : Canal de mise à jour fermé.
- `UnsupportedScheme` : URL non-HTTPS.
- `InvalidUrl` : URL invalide.
- `TooLarge` : Flux trop volumineux.

Exemple d'Utilisation

```
match fetch_feed(&client, &feed, Duration::from_secs(10)).await {
    Ok(entries) => println!("{} articles téléchargés", entries.len()),
    Err(PollError::Network(e)) => eprintln!("Erreur réseau : {}", e),
```

```
    Err(PollError::UnsupportedScheme) => eprintln!("Seules les URLs HTTPS sont
autorisées"),
}
```

---

6. Interface graphique (rss-gui)

6.1. Technologies utilisées (eframe / egui)

L'interface est construite avec **eframe** et **egui**, des bibliothèques Rust permettant de créer des interfaces graphiques modernes et réactives.

6.2. Organisation du code (main.rs, app.rs, vues)

Le code est organisé en plusieurs fichiers : `main.rs` pour le point d'entrée, `app.rs` pour la logique principale, et des fichiers dédiés pour chaque vue (liste d'articles, détail d'article, etc.).

6.3. Principales vues :

L'interface propose plusieurs vues :

- Une **liste d'articles** pour parcourir les flux.
- Un **détail d'article** pour lire le contenu.
- Une section **Discover** pour ajouter de nouveaux flux.
- Un menu **Paramètres** pour personnaliser l'application.

6.4. Gestion du thème et de la personnalisation

Les utilisateurs peuvent personnaliser l'apparence de l'application (thème clair/sombre, taille de police, etc.) via un fichier de configuration JSON.

6.5. Interaction avec le cœur (`DataApi, PollerHandle, Event`)

L'interface interagit avec `rss-core` via des interfaces comme `DataApi` pour accéder aux données, et `PollerHandle` pour contrôler le téléchargement des flux.

---

7. Sécurité et robustesse

7.1. Politique de sécurité (HTTPS obligatoire, taille max, timeout)

ReadRSS impose l'utilisation de **HTTPS** pour tous les flux, limite la taille des données téléchargées, et utilise des *timeouts* pour éviter les blocages.

7.2. Protection contre les flux malveillants

Les données entrantes sont validées pour éviter les attaques (ex. : injection de code). Le HTML n'est jamais interprété, éliminant les risques de **XSS**.

7.3. Gestion des erreurs (`PollError`)

Les erreurs sont gérées de manière centralisée, avec des messages explicites pour aider l'utilisateur à résoudre les problèmes (ex. : flux inaccessible).

7.4. Écriture atomique et récupération en cas d’interruption

Les données sont sauvegardées de manière atomique, garantissant leur intégrité même en cas d'interruption soudaine.

7.5. Absence de HTML interprété (pas de XSS)

Pour des raisons de sécurité, le contenu HTML des articles n'est pas interprété, mais affiché en texte brut.

---

8. Persistance et données

8.1. Fichiers JSON utilisés

Les données (flux, articles, configuration) sont stockées dans des fichiers JSON, faciles à lire et à modifier.

8.2. Structure des données (`FeedDescriptor, FeedEntry`)

Les flux sont représentés par `FeedDescriptor` (URL, titre), et les articles par `FeedEntry` (titre, contenu, date, etc.).

8.3. Déduplication et SeenStore

Un mécanisme de déduplication (`SeenStore`) évite l'affichage multiple du même article.

8.4. États “lus” / “non lus”

Les articles sont marqués comme "lus" ou "non lus" pour un suivi optimal de la lecture.

8.5. Mécanisme de sauvegarde immédiate

Les modifications (ex. : marquer un article comme lu) sont sauvegardées immédiatement pour éviter toute perte de données.

---

9. Performance et fiabilité

9.1. Async/await et non-blocage de l’interface

L'utilisation de **async/await** garantit que l'interface reste réactive, même pendant le téléchargement des flux.

9.2. Téléchargement en streaming (BytesMut → freeze)

Les flux sont téléchargés en *streaming* pour optimiser la mémoire et les performances.

9.3. Backoff exponentiel pour les retries

En cas d'échec, le téléchargement est relancé avec un délai croissant (*backoff exponentiel*), améliorant la fiabilité.

9.4. Tests unitaires et intégration (`poll_once`)

Des tests unitaires et d'intégration couvrent les fonctionnalités clés, comme le téléchargement d'un flux (`poll_once`).

9.5. Mesure de performance (profilage avec flamegraph)

Des outils comme **flamegraph** sont utilisés pour analyser et optimiser les performances.

---

10. Développement et packaging

10.1. Structure CI/CD (GitHub Actions)

Le projet utilise **GitHub Actions** pour automatiser les tests et le déploiement.

10.2. Lignes de commande utiles (RUST_LOG, cargo deb, etc.)

`RUST_LOG=debug cargo run` pour activer les logs détaillés.

`cargo test` pour lancer les tests.

---

11. Problèmes courants et dépannage

11.1. Aucun article téléchargé

Vérifiez votre connexion Internet et l'URL du flux. Certains flux peuvent nécessiter une authentification.

11.2. Erreurs de flux (HTTP, timeout, parsing)

Ces erreurs sont généralement liées à un problème réseau ou à un flux mal formé. Vérifiez l'URL et réessayez plus tard.

11.3. Emojis absents sous Linux

Installez les polices emoji pour afficher correctement les caractères spéciaux.

11.4. JSON corrompu / restauration

Supprimez le fichier de configuration corrompu et relancez l'application pour le régénérer.

11.5. Lecture lente ou interface bloquée

Vérifiez les performances de votre système et réduisez le nombre de flux actifs si nécessaire.

---

12. Roadmap et perspectives

12.1. Fonctionnalités prévues

À venir : import/export **OPML**, recherche plein texte, et support multi-langues (**i18n**).

12.2. Extensions possibles (OPML, recherche plein texte, i18n)

Ces fonctionnalités sont en cours de développement pour enrichir l'expérience utilisateur.

12.3. Améliorations techniques envisagées

Optimisation des performances, ajout de nouveaux protocoles (ex. : Atom), et amélioration de l'interface.

---

13. Glossaire

Définitions des termes clés (RSS, Atom, async, Arc, RwLock, serde, etc.)

- **RSS** : Format de syndication de contenu (ex. : blogs, actualités).
- **Atom** : Alternative à RSS, également supportée.
- **Async/await** : Mécanisme pour écrire du code asynchrone en Rust.
- **Arc** : *Atomic Reference Counting*, pour gérer la mémoire partagée.
- **RwLock** : Verrou pour synchroniser l'accès aux données en lecture/écriture.
- **Serde** : Bibliothèque pour sérialiser/désérialiser des données.