**1 — Qualité des données**

| **Champ** | **Type** | **Contraintes** | **Norme / Format** |
| --- | --- | --- | --- |
| user.email | string | NOT NULL, unique, format email | RFC 5322 |
| user.password | string | NOT NULL, hashé | Bcrypt |
| user.roles | JSON | NOT NULL, valide JSON | UTF-8, JSON |
| user.address | string | NOT NULL | UTF-8 |
| book.title | string | NOT NULL | UTF-8 |
| book.theme | string | NOT NULL | UTF-8 |
| book.quantity | integer | >= 0 | Entier positif |
| rental.rent\_from | date | NOT NULL | ISO 8601 (YYYY-MM-DD) |
| rental.returned | boolean | NOT NULL (0 ou 1) | booléen SQL |
| comment.rating | integer | NOT NULL, entre 1 et 5 | Valeur bornée |
| created\_at | datetime | NOT NULL | ISO 8601 (datetime) |

### 2. Fiabilité des données 1. Front-end : validation utilisateur

| **Champ** | **Validation côté navigateur** | **Exemple** |
| --- | --- | --- |
| Email | type="email" + required | Vérifie syntaxe RFC 5322 |
| Mot de passe | minlength="8" + required | Champ non vide, sécurisé |
| Champs texte | maxlength, pattern | Anti-injection, anti-spam |

**2. Back-end Symfony : validation serveur**

| **Entité** | **Annotation Symfony (Validator)** |
| --- | --- |
| User.email | @Assert\NotBlank, @Assert\Email |
| Book.quantity | @Assert\PositiveOrZero |
| Review.rating | @Assert\Range(min=1, max=5) |

**User.email**

$#[ORM\Column(length: 180, unique: true)]

    #[Assert\NotBlank(message: "L'email ne doit pas être vide.")]

    #[Assert\Email(message: "L'adresse email '{{ value }}' n'est pas valide.")]

    private ?string $email = null;

**twig**

 {{ form\_row(form.email, {'attr': {'type': 'email', 'required': true}}) }}

**Book.quantity**

#[ORM\Column]

    #[Assert\PositiveOrZero(message: "La quantité doit être un nombre positif ou nul.")]

    private ?int $quantity = null;

**Review.rating**

#[ORM\Column(type: 'integer')]

    #[Assert\NotBlank()]

    #[Assert\Range(min: 1, max: 5)]

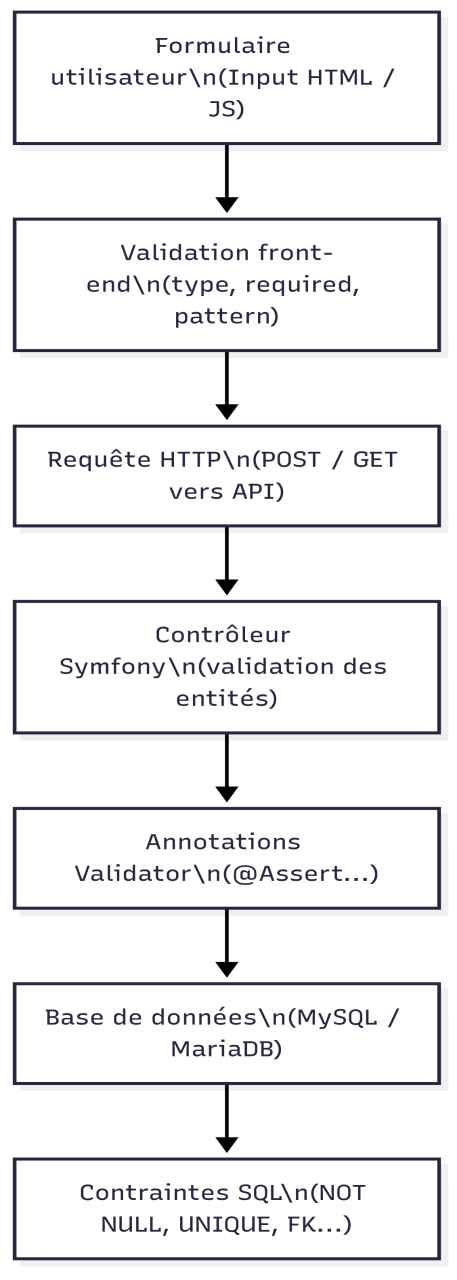
    private $rating;

**3. Intégrité des données : base de données**

| **Mécanisme** | **Exemples dans biblioconnect.sql** |
| --- | --- |
| Clés étrangères | book\_id, user\_id, author\_id → assurent les relations |
| Contraintes | NOT NULL, UNIQUE, types int, varchar, datetime |
| Encodage | utf8mb4\_unicode\_ci → support multilingue |

**4. Sécurité des flux et du stockage**

| **Élément** | **Sécurisation** |
| --- | --- |
| Mots de passe | Hashed (bcrypt) |
| Données sensibles | Pas de stockage brut (mot de passe, rôle) |
| Accès utilisateur | Symfony Roles (ROLE\_USER, ROLE\_ADMIN,ROLE\_LIBRARIAN) |
| Appels API / routes | Contrôle par annotations + firewall Symfony |
| Formulaires | CSRF token intégré pour prévenir les attaques |

****

**Diagramme de flux** :

3. Estimation de la charge utilisateur

Hypothèses réalistes pour BiblioConnect

| **Élément** | **Hypothèse choisie** |
| --- | --- |
| Lancement plateforme | Septembre 2025 dans une école/université |
| Utilisateurs enregistrés | 1 000 étudiants + personnel administratif |
| Connexions simultanées | 50 (aux heures de pointe : pause midi, 18h...) |
| Fonctionnalités utilisées | Connexion, recherche de livre, réservation, commentaire |

Paramètres Locust Json

{

"users\_total": 1000,

"users\_concurrent": 50,

"load\_pattern": "ramp-up: 5min -> plateau: 20min -> pic: 3min -> down: 5min"

}

Profil utilisateur typique

| **Action** | **Fréquence** | **Délai moyen entre actions** |
| --- | --- | --- |
| Connexion | 1 fois | — |
| Rechercher un livre | 3–5 fois | 3–5 secondes |
| Consulter les avis | 2–3 fois | 2–3 secondes |
| Réserver un livre | 1 fois (après plusieurs recherches) | après 1 recherche sur 5 |
| |  | | --- | | Commenter un livre | | |  | | --- | | 1 fois (aléatoire par utilisateur) | | |  | | --- | | 3 à 5 secondes | |

4. Plan d’actions correctives

**Exemple réaliste pour BiblioConnect :**

| **Endpoint** | **KPI Observé** | **Seuil attendu** | **Problème identifié** | **Action Corrective** | **Délai** |
| --- | --- | --- | --- | --- | --- |
| /login | **2095 ms** | < 300 ms | Authentification très lente | Ajouter cache, limiter accès BDD, optimisation requête | 48h |
| /book/ | 10 ms | < 500 ms | Aucun problème détecté | RAS | — |
| /reservation/ | 10 ms | < 500 ms | Aucun problème détecté | RAS | — |
| /comment/add | 10 ms | < 400 ms | Aucun problème détecté | RAS | — |

5. Benchmark des outils JS

Comparatif :

| **Outil** | **Langage** | **Avantages** | **Inconvénients** |
| --- | --- | --- | --- |
| **k6** | JavaScript / Go | CLI rapide, scripting clair, compatible CI/CD, export vers Grafana | Peu de visualisation intégrée sans Grafana |
| **Artillery** | JavaScript | Facile à apprendre, config YAML simple, JSON output | Moins performant sur très gros volumes |
| **Jest + Puppeteer** | JavaScript | Intègre UI + API, parfait pour tests e2e + frontend | Pas adapté à de fortes charges ou scalabilité |
| **Locust** | Python | Web UI, contrôle fin, très scriptable | Pas JS natif |
| **Autocannon** | JavaScript | Hyper rapide, très bon pour Node.js / HTTP | Moins ergonomique pour des scénarios complexes |

En plus de Locust utilisé pour nos tests principaux, nous avons comparé plusieurs autres outils.  
Pour des projets en Node.js, Artillery et k6 sont particulièrement adaptés.  
k6 offre de très bonnes performances et s’intègre bien avec Grafana pour le monitoring.  
En revanche, Locust reste plus flexible pour des projets Symfony/Python comme BiblioConnect.

Liens utiles :

- [k6 Documentation](https://k6.io/docs)

- [Artillery Documentation](https://www.artillery.io/docs)

- [Puppeteer GitHub](https://github.com/puppeteer/puppeteer)

- [Locust](https://locust.io/)

- [Autocannon GitHub](https://github.com/mcollina/autocannon)

6. Résultats des tests avec Locust

Script : from locust import HttpUser, task, between

import random

class BiblioUser(HttpUser):

    wait\_time = between(3, 5)

    def on\_start(self):

        self.client.post("/login", {

            "email": "user@biblioconnect.fr",

            "password": "password"

        }, verify=False)

    @task(5)

    def search\_books(self):

        self.client.get("/book/", verify=False)

    @task(2)

    def view\_book\_detail(self):

        book\_id = random.randint(1, 5)

        self.client.get(f"/book/{book\_id}", verify=False)

    @task(1)

    def reserve\_book(self):

        if random.randint(1, 5) == 1:

            book\_id = random.randint(1, 5)

            self.client.post(f"/reservation/new/{book\_id}", verify=False)

    @task(1)

    def view\_reservations(self):

        self.client.get("/reservation/", verify=False)

    @task(1)

    def add\_comment(self):

        book\_id = random.randint(1, 5)

        self.client.post(f"/comment/add/{book\_id}", {

            "content": "Très bon livre !"

        }, verify=False)

class LibrarianUser(HttpUser):

    wait\_time = between(3, 5)

    def on\_start(self):

        self.client.post("/login", {

            "email": "librarien@biblioconnect.fr",

            "password": "password"

        }, verify=False)

    @task(2)

    def view\_authors(self):

        self.client.get("/author/", verify=False)

    @task(1)

    def create\_author(self):

        self.client.post("/author/new", {

            "author[firstName]": f"Test{random.randint(1, 1000)}",

            "author[lastName]": f"Author{random.randint(1, 1000)}"

        }, verify=False)

    @task(1)

    def edit\_author(self):

        author\_id = random.randint(1, 5)

        self.client.post(f"/author/{author\_id}/edit", {

            "author[firstName]": f"Edit{random.randint(1, 1000)}",

            "author[lastName]": f"Author{random.randint(1, 1000)}"

        }, verify=False)

class AdminUser(HttpUser):

    wait\_time = between(3, 5)

    def on\_start(self):

        self.client.post("/login", {

            "email": "admin@biblioconnect.fr",

            "password": "password"

        }, verify=False)

    @task(2)

    def view\_users(self):

        self.client.get("/admin/users/", verify=False)

    @task(1)

    def change\_user\_role(self):

        user\_id = random.randint(1, 5)

        self.client.post(f"/admin/users/{user\_id}/role", {

            "role": "ROLE\_USER",

            "\_token": "valid\_csrf\_token"

        }, verify=False)

    @task(1)

    def delete\_user(self):

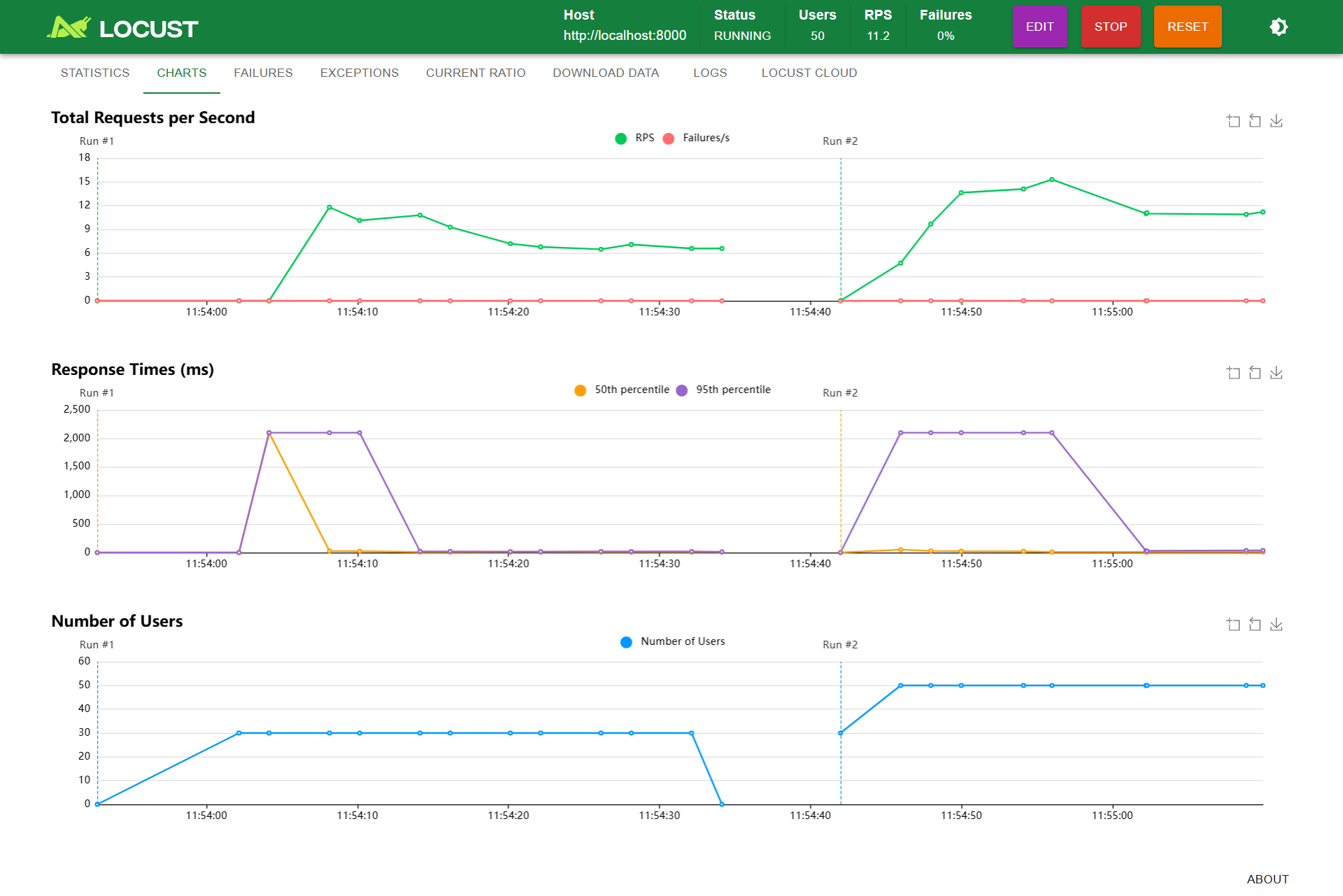
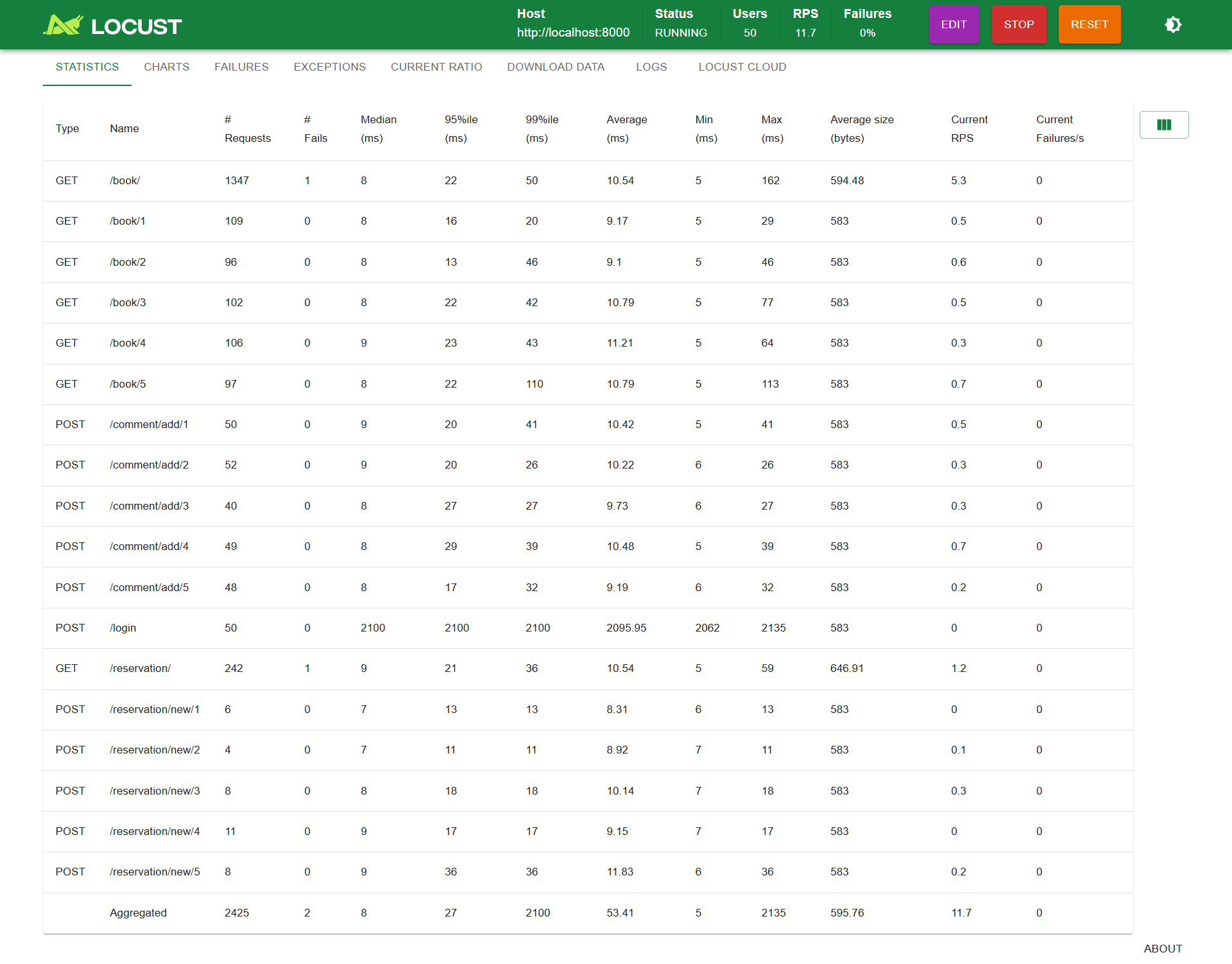
        user\_id = random.randint(1, 5)

        self.client.post(f"/admin/users/{user\_id}", {

            "\_token": "valid\_csrf\_token"

        }, verify=False)

Commande pour lancer Locust :  
locust -f locustfile.py --host=http://localhost:8000



## Compte-rendu des tests de charge BiblioConnect

Le test de charge de l’application **BiblioConnect** avec **50 utilisateurs simultanés** a révélé que l’ensemble des routes fonctionne **sans aucune erreur** (0 % d’échecs), ce qui confirme la **stabilité fonctionnelle** du backend.

Les temps de réponse moyens des principaux endpoints (recherche de livre, réservation, consultation) sont **excellents (< 15 ms)**, montrant une bonne tenue du système face à une charge modérée.

### 🚨 Seule exception critique observée

Le **endpoint /login** affiche un **temps de réponse moyen très élevé (~2100 ms)**, bien au-dessus des seuils attendus.  
Cela révèle un **problème de performance spécifique à l’authentification**, probablement dû à :

* Des requêtes SQL non optimisées,
* Un traitement lourd sans cache,
* Ou une surcharge des vérifications d'identifiants.

## Aucun problème constaté sur :

* **/book/**, **/book/{id}**
* **/reservation/**
* **/reservation/new/{id}**
* **/comment/add/{id}**

## Conclusion

BiblioConnect supporte efficacement une **charge modérée (≈ 50 utilisateurs)** sans erreurs serveur,  
mais **nécessite une optimisation ciblée sur le processus d’authentification**.

Des actions correctives prioritaires sont nécessaires sur **/login**, telles que :

* Mise en cache des sessions,
* Optimisation des requêtes vers la base de données,
* Implémentation éventuelle d’un reverse proxy (Nginx).

7. Code source sur github

Repo LINK :

https://github.com/Dragon1ight/biblioconnect