Portfolio - Part **3**

1. Define user stories and mock-ups

1.1. User stories

1.1.1. Must Have (indispensable)

*As a **guest**, I would like to **explore the interactive** map to discover **the locations.***

*As a **guest**, I would like to consult the detailed information sheets on a place or character in order to learn more about the universe.*

*As a **registered user**, I wish **to create an account** and log in to **access the contribution features.***

*As a **registered user**, I would like **to publish an artistic creation** (image + description) in order to **share my work with the community**.*

*As an **administrator**, I wish to **moderate posts** in order to ensure **quality and compliance with the rules.***

1.1.2. Should Have (important but not vital for the MVP)

*As a **guest**, I want to **search for content by filter** (places, characters, creations) in order to **quickly access the information I want**.*

*As an **administrator**, I want to **manage users** (block/delete) in order to **maintain a secure space.***

1.1.3. Could Have (useful but optional)

*As a **registered user**, I wish to **use real-time messaging** to **chat with the community.***

*As a **registered user**, I wish to **edit or delete my creations** in order to **maintain control over my content.***

1.1.4. Won’t Have (not planned for this MVP)

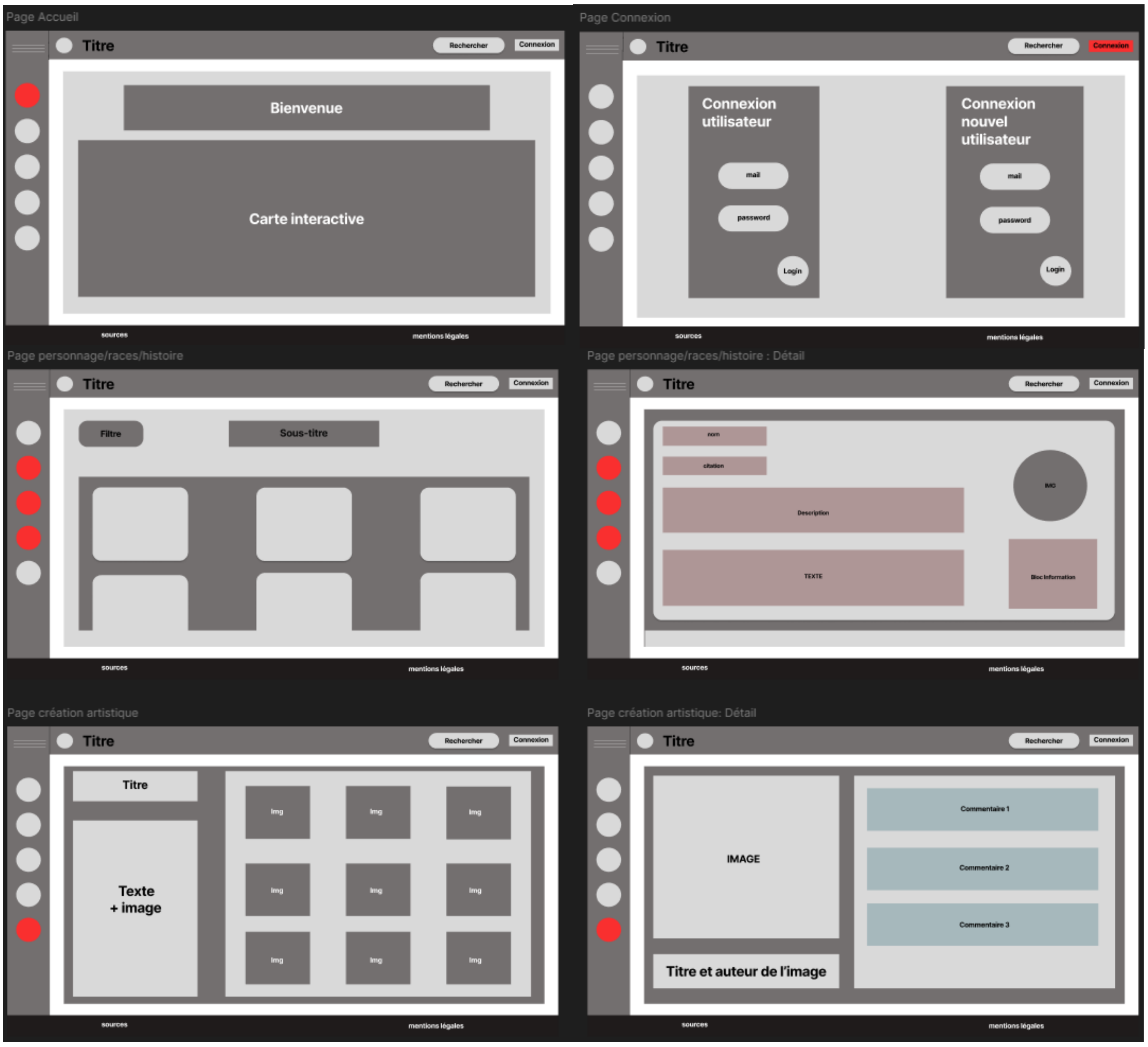
*As a **user**, I want **to purchase creations or derivative items** in **order to own them.***

*As a **user**, I want to **navigate an interactive family** tree in **order to see the links between characters.***

1.2. Summary MoSCoW

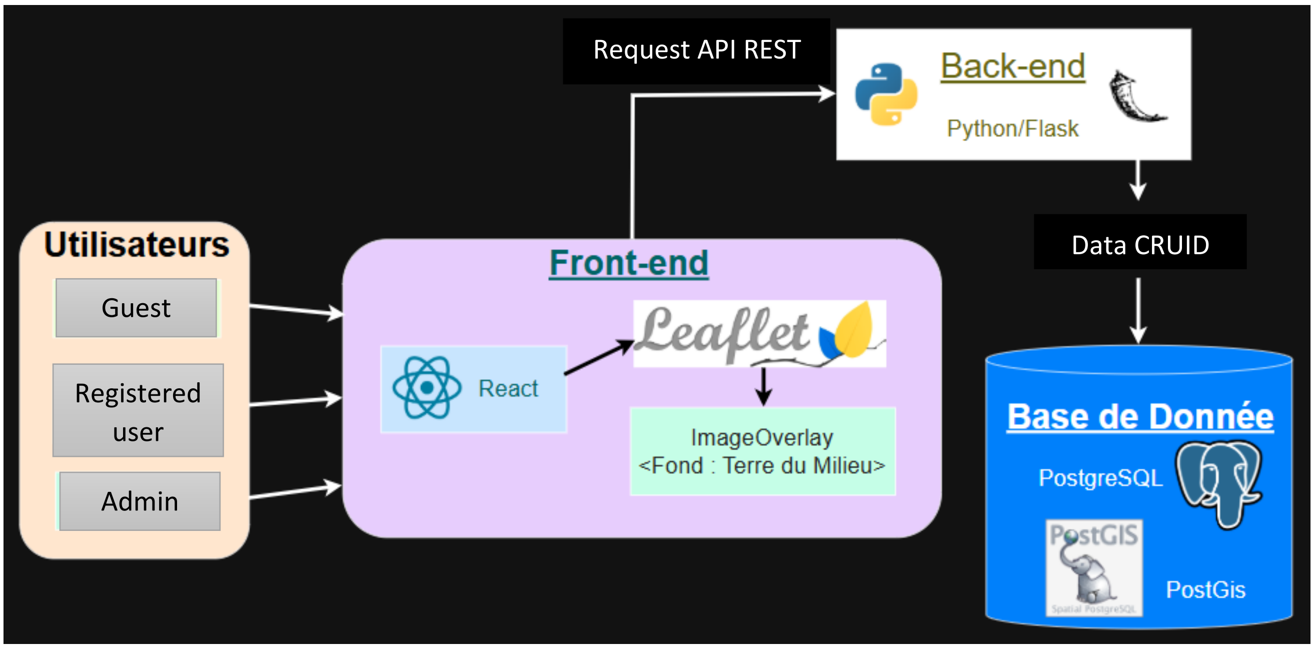
Priority	Selected features
Must Have **(Indispensable)**	- Interactive map to explore locations - Detailed information sheets on locations/characters - Account creation and login - Publication of artistic creations (image + description) - Moderation of publications by the administrator
Should Have **(Important but not vital)**	- Search and filters (places, characters, creations) - User management by the administrator (block/delete)
Could Have **(Optional)**	- Real-time messaging between registered users - Editing/deleting your own creations
Won’t Have **(not planned for this MVP)**	- Purchase of creations or derivative items (e-commerce) - Interactive family tree of characters

1.3. Wireframe



2. Design system architecture

2.1. High-Level Architecture Diagram



2.2. Explications

2.2.1. Front-end (React + Leaflet)

- **Language**: JavaScript (with JSX to combine HTML and JavaScript) + CSS for styling.

- **Main technologies :**
  - **React**: structures the user interface (UI) and manages the application state.
  - **Leaflet**: interactive mapping JavaScript library.
  - **ImageOverlay**: serves as a background image (e.g. the map of Middle Earth).

- **Key components :**
  - `App` (point d'entrée React)
  - `App` (React entry point)
  - `Header`
  - `Sidebar` (navigation between sections)
  - `Map` (integrates Leaflet to display areas and locations in GeoJSON format)
  - `InfoGroup` (displays characters, stories, creatures)
  - `InfoSolo` (displays a character/creature profile)
  - `ArtWorkGallery` (gallery of artistic creations)
  - `ArtWorkSolo` (displays a creation with commentary)

2.2.2. Back-end (Flask API in Python)

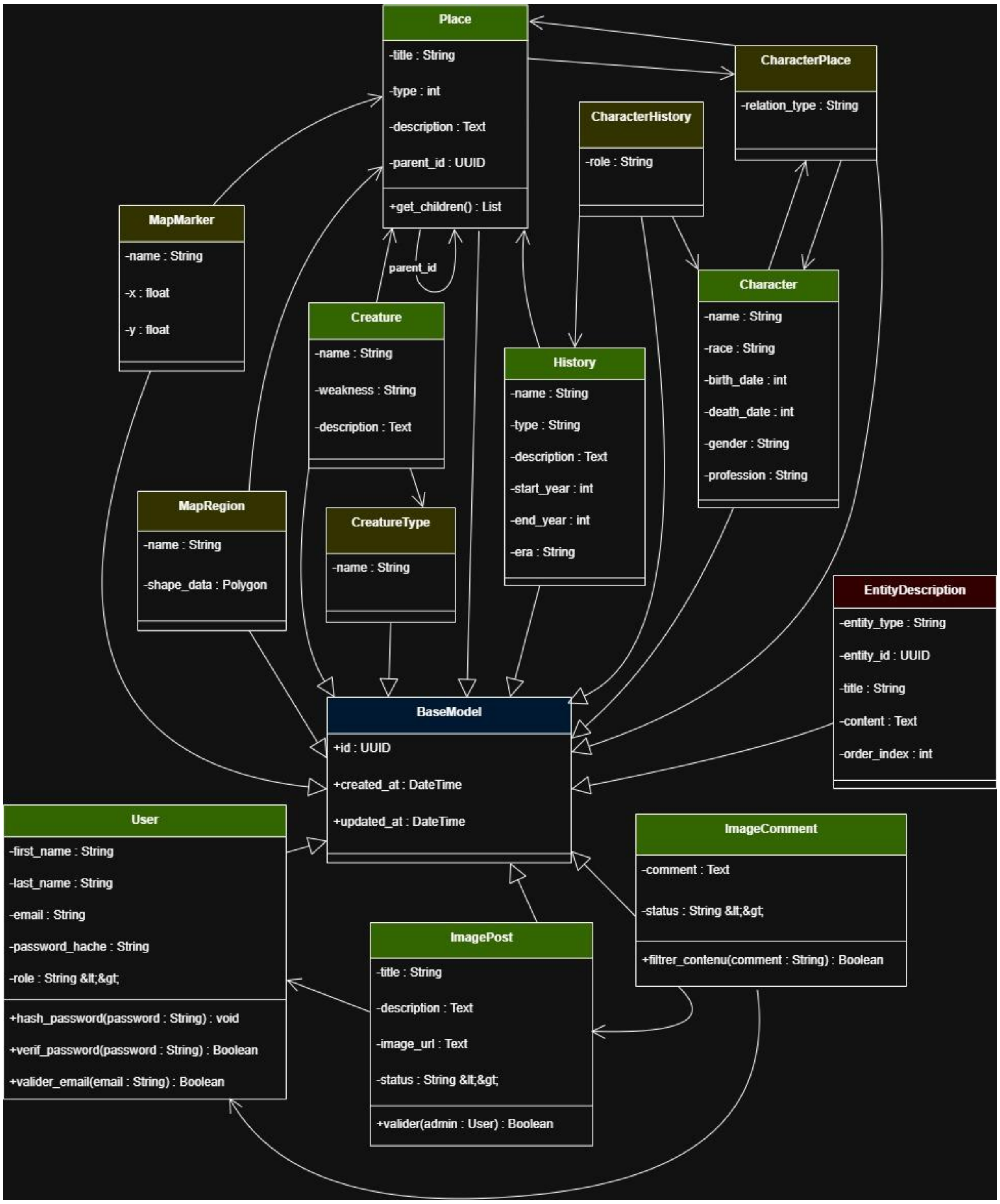
- **Language :** Python.
- **Framework :** Flask (micro-framework web).
- **Rôle :**
  - Serves as a **single interface** between the front end and the database.
  - Provides a **REST API** (`endpoints /characters, /stories, /regions, /locations, /artworkgallery, /artworksolo, etc.).`
  - Returns data in **JSON** (text content) or **GeoJSON** (geographic data) format.
  - Contains **business logic**: validation, CRUD controllers (Create, Read, Update, Delete), application rules.

2.2.3. Databases (PostgreSQL + PostGIS)

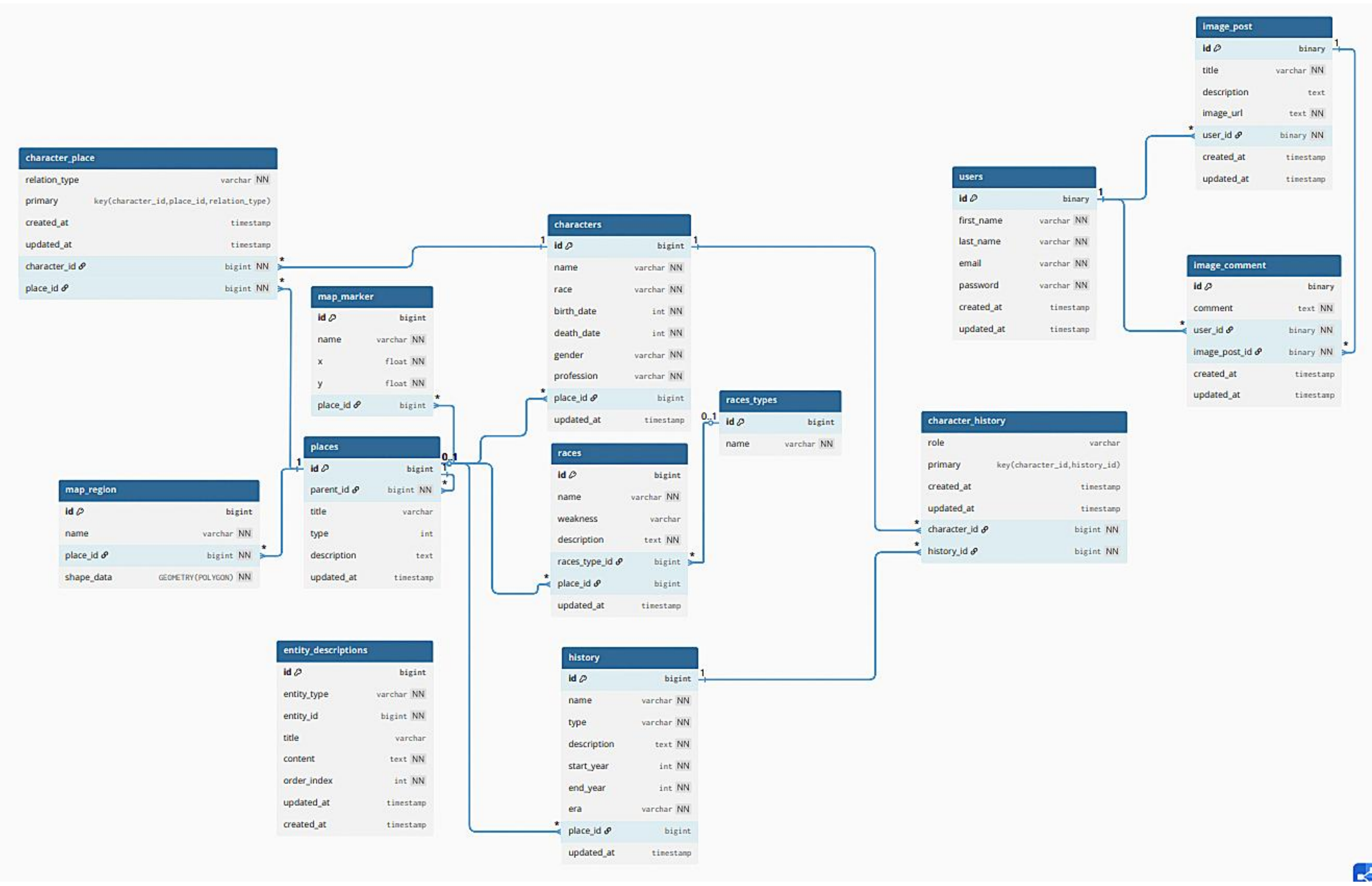
- **Language :** SQL.
- **Technology :** PostgreSQL (relational database) with the **PostGIS** extension for managing spatial data.
- **Storage :**
  - **Structured** data: users, characters, stories, creatures, artistic creations, comments.
  - **Geographical** data :
    - Polygons (areas: Gondor, Mordor, etc.).
    - Points (specific locations: cities, mountains, fortresses).
- **Relationship managed** via primary and foreign keys

3. Define the components, classes, and design of the DB

3.1. Back-end, classes, attributes, and methods



3.2. Database, ER (entity-relationship) diagram



3.3 Front-end, interface components and interactions

App

- **Role** : Application entry point, manages **routing** and global context (authentication, theme, etc.).

- **Interactions** :
  - Contains the **header**, **navigation menu**, **main content**, and **footer**.
  - Passes the necessary props (parameters) to child components (user, data loaded from the back end).

Header

- **Role** : Navigation bar and user information.

- **Content**: Logo, login/registration menu, (profile button), search button.

- **Interactions** :
  - Allows users to log in to their account
  - Searches the site

Sidebar

- **Role** : Tabs to access the different sections
  - **Interactive map**
  - **Characters / History / Creatures**
  - **Artistic creations (photos and comments)**

- **Interactions** :

  - Changes the component displayed in **the main content**.
  - Allows you to navigate between the main tabs: Map, Characters/Story/Creatures, Artistic Creations.
  - Allows you to enlarge or reduce the sidebar.

Map

- **Role** : Displays the map of Middle Earth.

- **Features** :
  - Clickable points for regions
  - Outline around region with possible hover effect

- **Interactions** :
  - Click on a marker (location) → displays a new page with the associated location.
  - Ability to filter markers by location type (forest, fortress, city, etc.)

InfoGroupe

- **Role** : Displays all items in the selected group (character, creature, or story)

- **Contenu** :
  - Name + images (illustration).
  - Links to other related characters or events.

- **Interactions** :
  - Receives data from the back-end via REST API.
  - Can pass filters or selection criteria to child components.
  - Reusable React component for displaying each group.

InfoSolo

- **Role** : Displays details of the item selected in infoGroup
- **Content** :
    - Name, description, image (illustration).
    - Information block
    - Possible links to other related characters or events.
- **Interactions** :
    - Receives data from the back-end via REST API.
    - Reusable React component for displaying each selected item

ArtWorkGallery

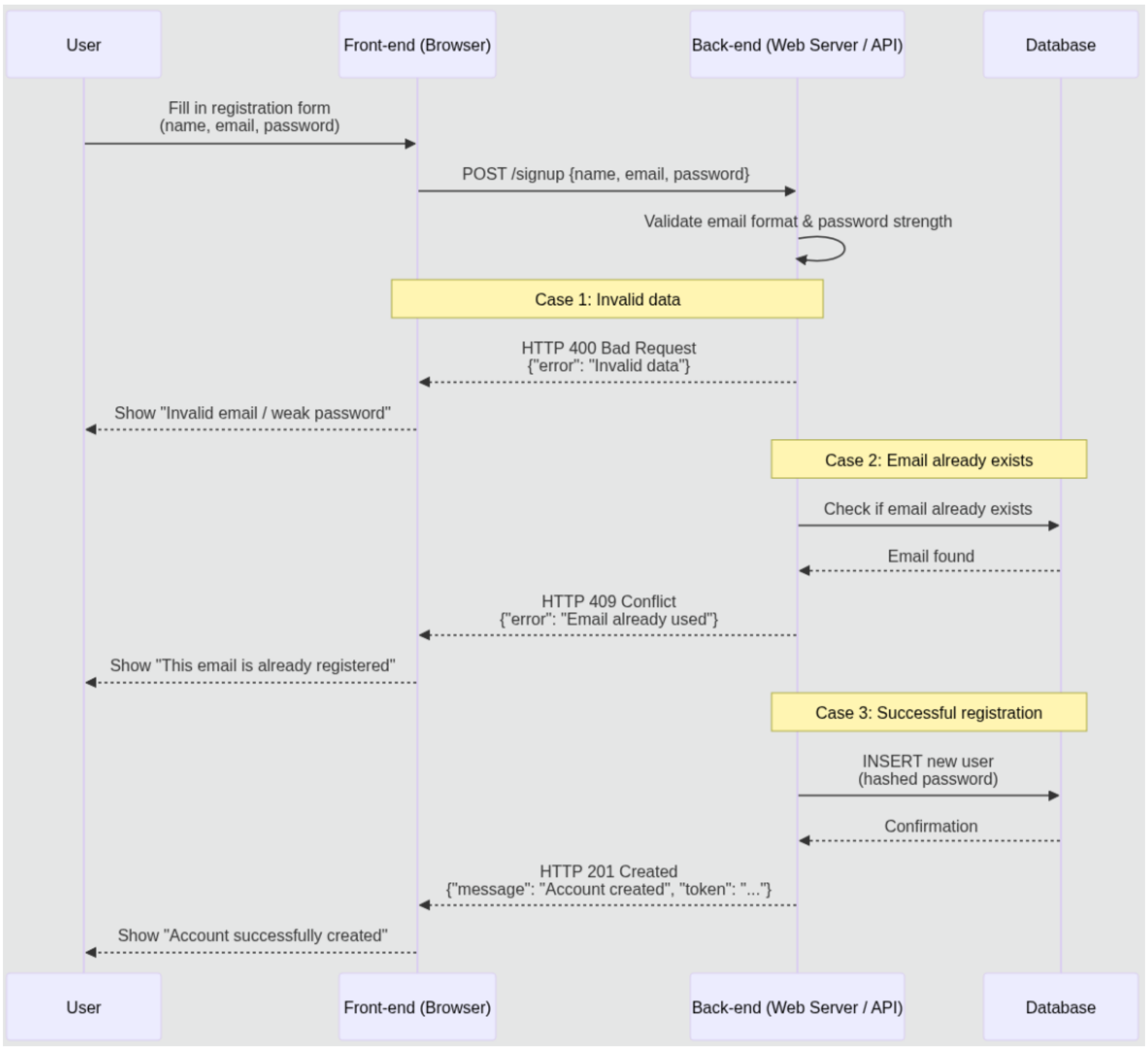
- **Role** : Displays user creations (images, illustrations, fan art).
- **Content** :
    - Thumbnails, title, author.
    - Option to click to view the image in full size and comments.
- **Interactions** :
    - Data loading from the back end (images and comments).

ArtWorkSolo

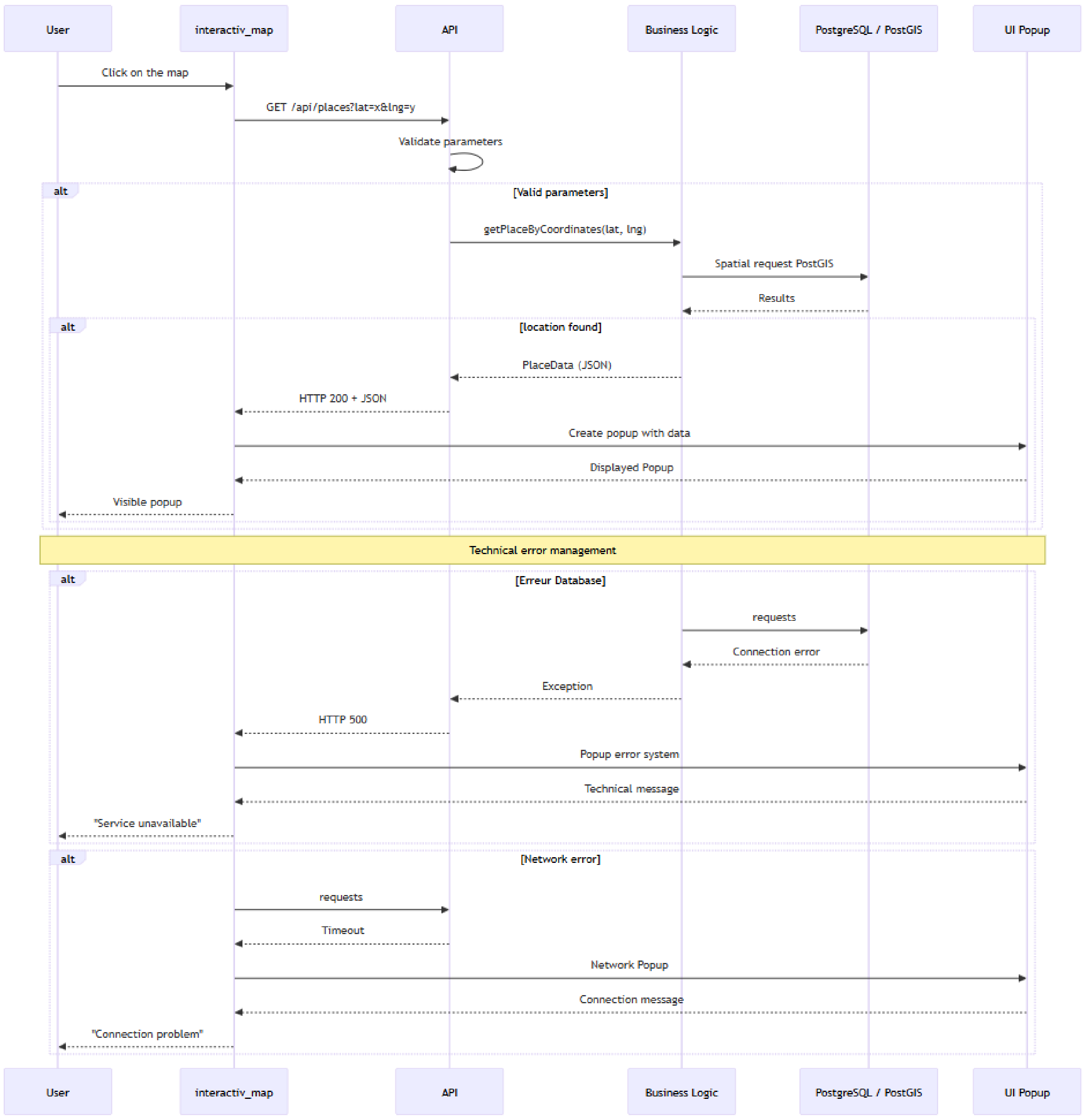
- **Role** : Comment management for each artistic creation.
- **Contenu** : List of comments, field to add a new one.
- **Interactions** :
    - POST to add a comment, GET to retrieve the list.
    - Real-time updates possible via WebSocket or refresh.
    - React component for comments.

4. High-level sequence diagrams

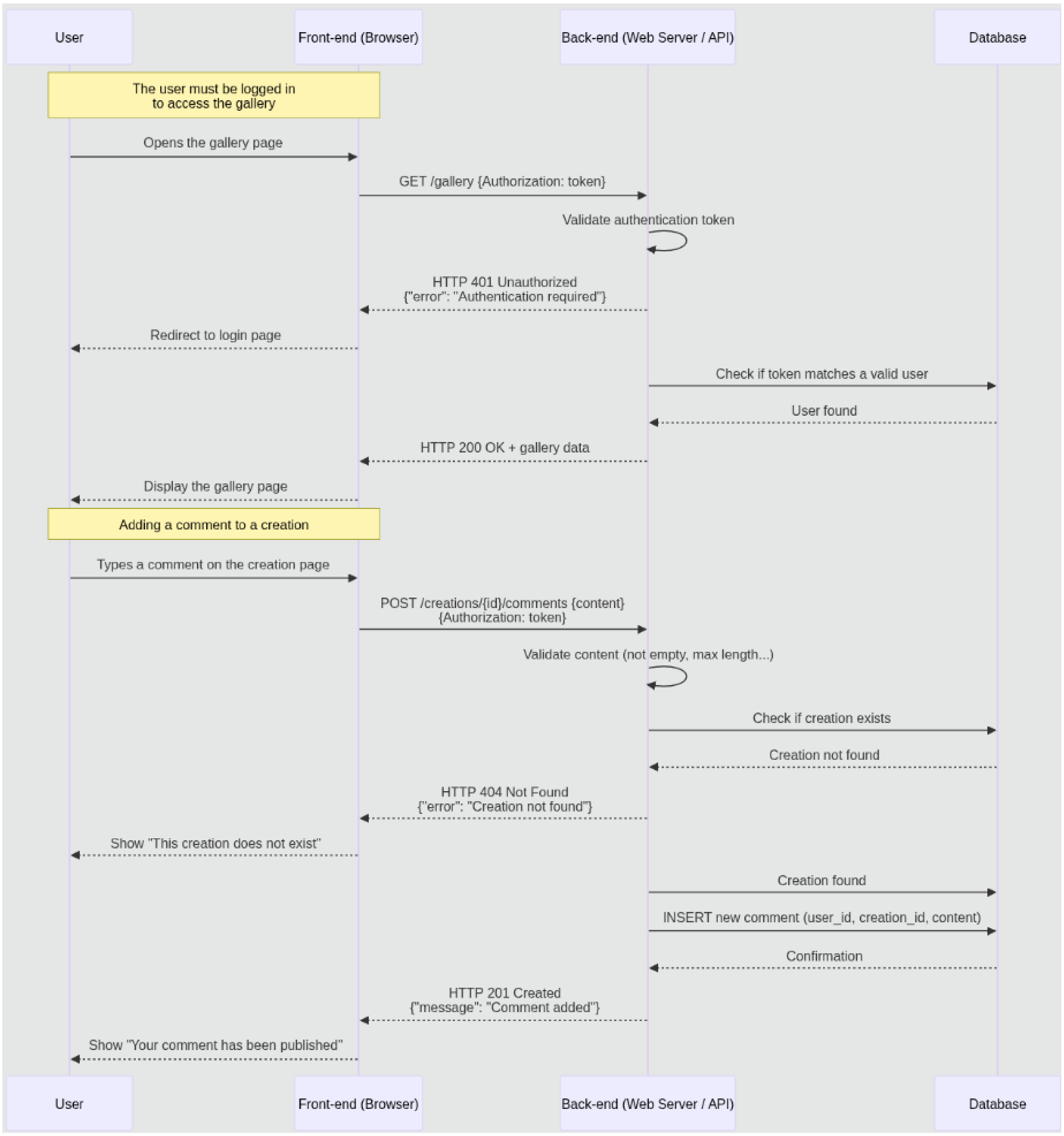
4.1. User registration


User
Front-end (Browser)
Back-end (Web Server / API)
Database
Fill in registration form (name, email, password)
POST /signup {name, email, password}
Validate email format & password strength
Case 1: Invalid data
HTTP 400 Bad Request {"error": "Invalid data"}
Show "Invalid email / weak password"
Case 2: Email already exists
Check if email already exists
Email found
HTTP 409 Conflict {"error": "Email already used"}
Show "This email is already registered"
Case 3: Successful registration
INSERT new user (hashed password)
Confirmation
HTTP 201 Created {"message": "Account created", "token": "..."}
Show "Account successfully created"
User
Front-end (Browser)
Back-end (Web Server / API)
Database

4.2. Click on the map

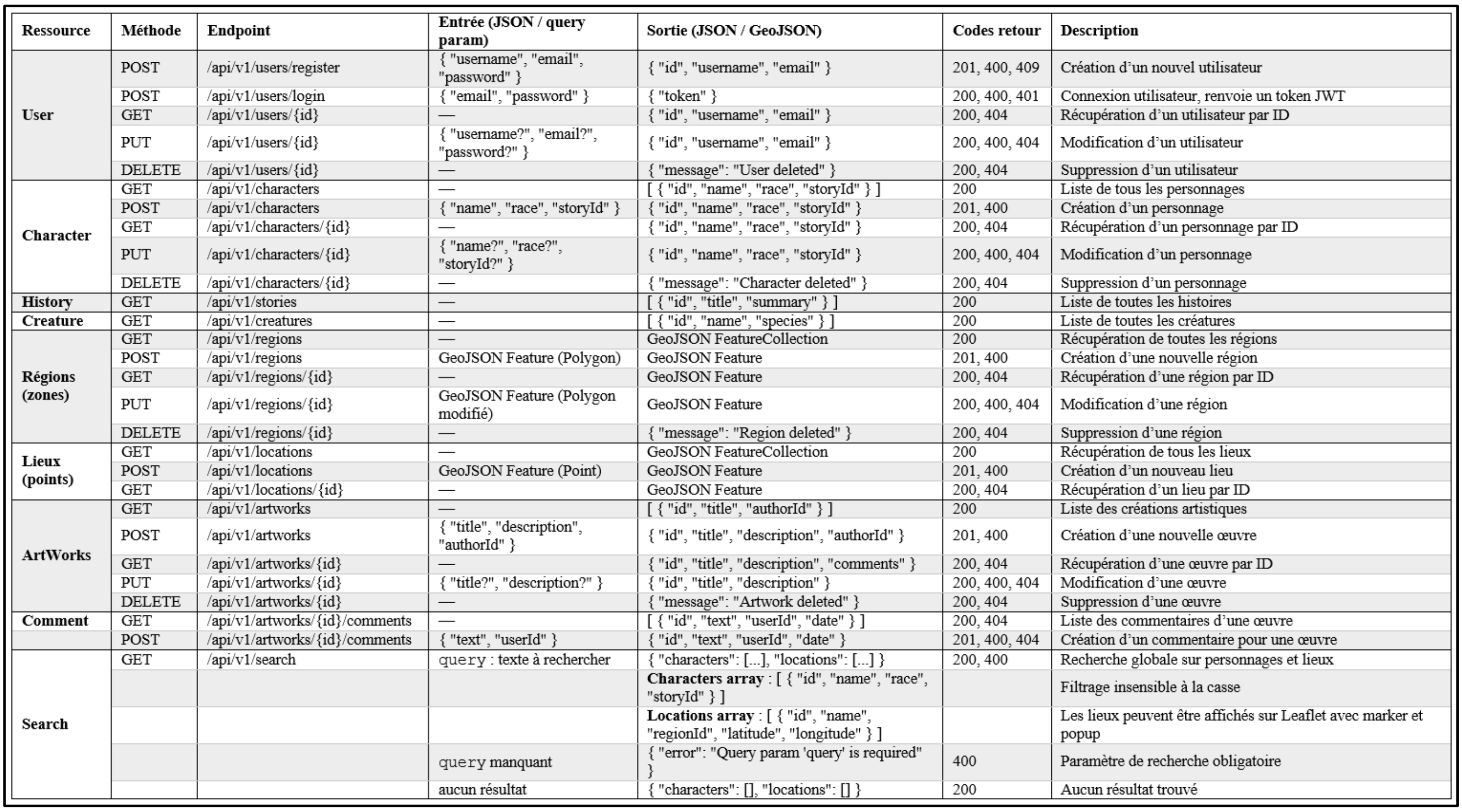


4.3. Adding a comment


User
Front-end (Browser)
Back-end (Web Server / API)
Database
The user must be logged in to access the gallery
Opens the gallery page
GET /gallery {Authorization: token}
Validate authentication token
HTTP 401 Unauthorized {"error": "Authentication required"}
Redirect to login page
Check if token matches a valid user
User found
HTTP 200 OK + gallery data
Display the gallery page
Adding a comment to a creation
Types a comment on the creation page
POST /creations/{id}/comments {content} {Authorization: token}
Validate content (not empty, max length...)
Check if creation exists
Creation not found
HTTP 404 Not Found {"error": "Creation not found"}
Show "This creation does not exist"
Creation found
INSERT new comment (user_id, creation_id, content)
Confirmation
HTTP 201 Created {"message": "Comment added"}
Show "Your comment has been published"
User
Front-end (Browser)
Back-end (Web Server / API)
Database

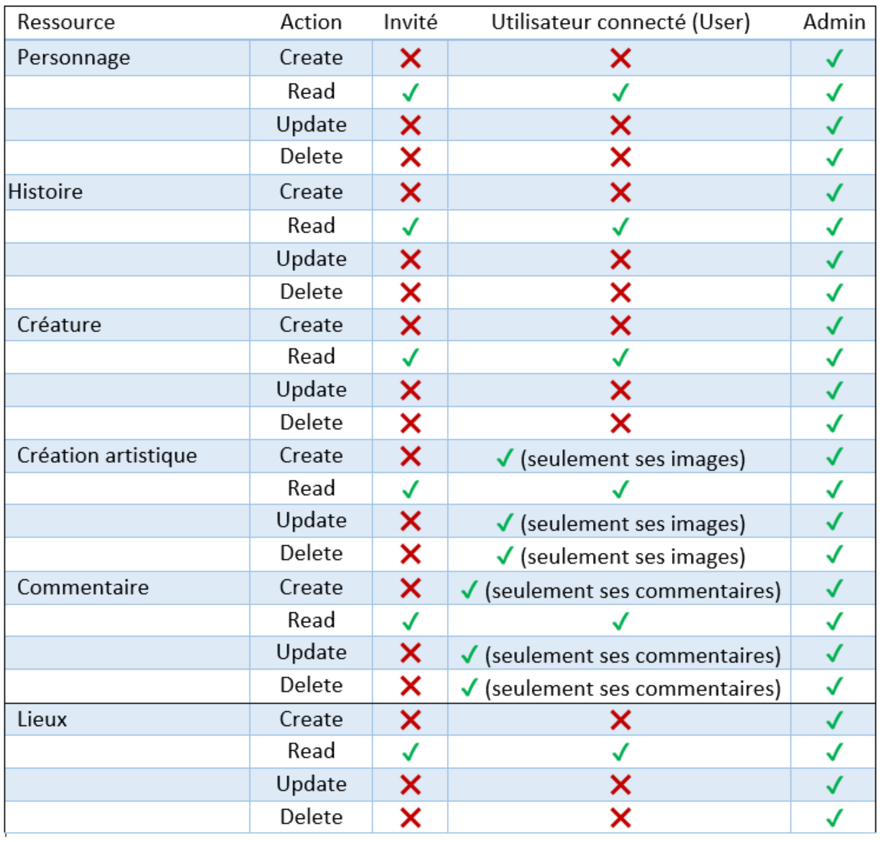
5. Documenting the internal API

5.1. Internal endpoint table



Ressource	Méthode	Endpoint	Entrée (JSON / query param)	Sortie (JSON / GeoJSON)	Codes retour	Description
User	POST	/api/v1/users/register	{ "username", "email", "password" }	{ "id", "username", "email" }	201, 400, 409	Création d'un nouvel utilisateur
	POST	/api/v1/users/login	{ "email", "password" }	{ "token" }	200, 400, 401	Connexion utilisateur, renvoie un token JWT
	GET	/api/v1/users/{id}	—	{ "id", "username", "email" }	200, 404	Récupération d'un utilisateur par ID
	PUT	/api/v1/users/{id}	{ "username?", "email?", "password?" }	{ "id", "username", "email" }	200, 400, 404	Modification d'un utilisateur
	DELETE	/api/v1/users/{id}	—	{ "message": "User deleted" }	200, 404	Suppression d'un utilisateur
Character	GET	/api/v1/characters	—	[{ "id", "name", "race", "storyId" }]	200	Liste de tous les personnages
	POST	/api/v1/characters	{ "name", "race", "storyId" }	{ "id", "name", "race", "storyId" }	201, 400	Création d'un personnage
	GET	/api/v1/characters/{id}	—	{ "id", "name", "race", "storyId" }	200, 404	Récupération d'un personnage par ID
	PUT	/api/v1/characters/{id}	{ "name?", "race?", "storyId?" }	{ "id", "name", "race", "storyId" }	200, 400, 404	Modification d'un personnage
	DELETE	/api/v1/characters/{id}	—	{ "message": "Character deleted" }	200, 404	Suppression d'un personnage
History	GET	/api/v1/stories	—	[{ "id", "title", "summary" }]	200	Liste de toutes les histoires
Creature	GET	/api/v1/creatures	—	[{ "id", "name", "species" }]	200	Liste de toutes les créatures
Régions (zones)	GET	/api/v1/regions	—	GeoJSON FeatureCollection	200	Récupération de toutes les régions
	POST	/api/v1/regions	GeoJSON Feature (Polygon)	GeoJSON Feature	201, 400	Création d'une nouvelle région
	GET	/api/v1/regions/{id}	—	GeoJSON Feature	200, 404	Récupération d'une région par ID
	PUT	/api/v1/regions/{id}	GeoJSON Feature (Polygon modifié)	GeoJSON Feature	200, 400, 404	Modification d'une région
	DELETE	/api/v1/regions/{id}	—	{ "message": "Region deleted" }	200, 404	Suppression d'une région
Lieux (points)	GET	/api/v1/locations	—	GeoJSON FeatureCollection	200	Récupération de tous les lieux
	POST	/api/v1/locations	GeoJSON Feature (Point)	GeoJSON Feature	201, 400	Création d'un nouveau lieu
	GET	/api/v1/locations/{id}	—	GeoJSON Feature	200, 404	Récupération d'un lieu par ID
ArtWorks	GET	/api/v1/artworks	—	[{ "id", "title", "authorId" }]	200	Liste des créations artistiques
	POST	/api/v1/artworks	{ "title", "description", "authorId" }	{ "id", "title", "description", "authorId" }	201, 400	Création d'une nouvelle œuvre
	GET	/api/v1/artworks/{id}	—	{ "id", "title", "description", "comments" }	200, 404	Récupération d'une œuvre par ID
	PUT	/api/v1/artworks/{id}	{ "title?", "description?" }	{ "id", "title", "description" }	200, 400, 404	Modification d'une œuvre
	DELETE	/api/v1/artworks/{id}	—	{ "message": "Artwork deleted" }	200, 404	Suppression d'une œuvre
Comment	GET	/api/v1/artworks/{id}/comments	—	[{ "id", "text", "userId", "date" }]	200, 404	Liste des commentaires d'une œuvre
	POST	/api/v1/artworks/{id}/comments	{ "text", "userId" }	{ "id", "text", "userId", "date" }	201, 400, 404	Création d'un commentaire pour une œuvre
Search	GET	/api/v1/search	`query` : texte à rechercher	{ "characters": [...], "locations": [...] }	200, 400	Recherche globale sur personnages et lieux
				Characters array : [{ "id", "name", "race", "storyId" }]		Filtrage insensible à la casse
				Locations array : [{ "id", "name", "regionId", "latitude", "longitude" }]		Les lieux peuvent être affichés sur Leaflet avec marker et popup
			`query` manquant	{ "error": "Query param 'query' is required" }	400	Paramètre de recherche obligatoire
			aucun résultat	{ "characters": [], "locations": [] }	200	Aucun résultat trouvé

5.2. CRUD matrix with roles and permissions



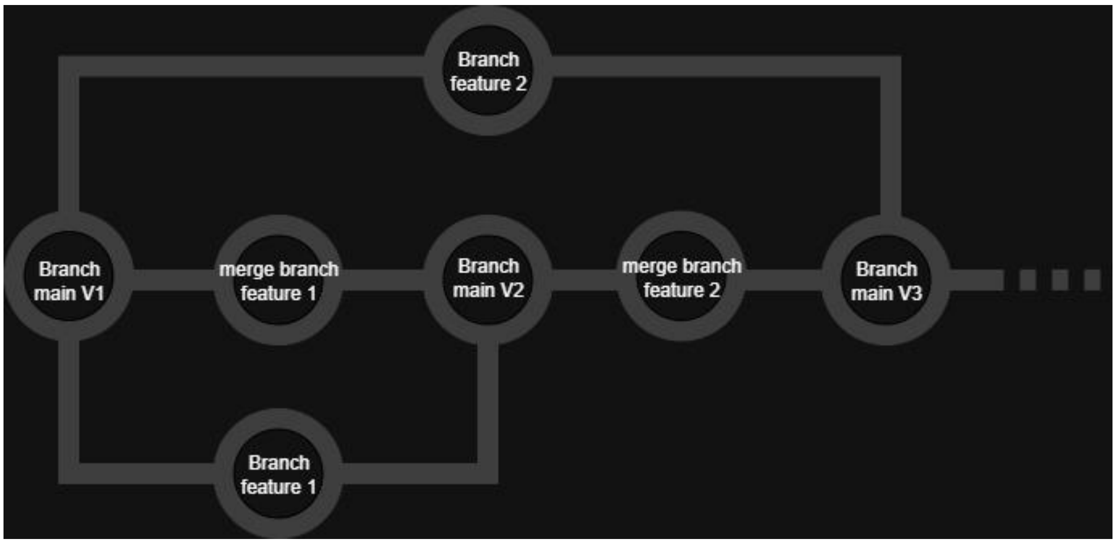
Ressource	Action	Invité	Utilisateur connecté (User)	Admin
Personnage	Create	✗	✗	✓
	Read	✓	✓	✓
	Update	✗	✗	✓
	Delete	✗	✗	✓
Histoire	Create	✗	✗	✓
	Read	✓	✓	✓
	Update	✗	✗	✓
	Delete	✗	✗	✓
Créature	Create	✗	✗	✓
	Read	✓	✓	✓
	Update	✗	✗	✓
	Delete	✗	✗	✓
Création artistique	Create	✗	✓ (seulement ses images)	✓
	Read	✓	✓	✓
	Update	✗	✓ (seulement ses images)	✓
	Delete	✗	✓ (seulement ses images)	✓
Commentaire	Create	✗	✓ (seulement ses commentaires)	✓
	Read	✓	✓	✓
	Update	✗	✓ (seulement ses commentaires)	✓
	Delete	✗	✓ (seulement ses commentaires)	✓
Lieux	Create	✗	✗	✓
	Read	✓	✓	✓
	Update	✗	✗	✓
	Delete	✗	✗	✓

6. Planning SCM and QA strategies

6.1. SCM Strategy - Version control with the GIT tool

- **Template of GIT commit:**
    - Use a professional format for **commits** (e.g., docs: update README with installation instructions or fix: resolve login button).
    - Commit titles should not **exceed 50 characters** to ensure they are clear and concise.
    - Commits that require a detailed description may contain more characters, but should **be segmented for each feature added**.
    - Type to define the **type** of commit:
        - *Docs*: Documentation.
        - *Feat*: Addition of a feature.
        - *Fix*: Bug fix.
        - *Refactor*: Change to the code that does not affect functionality.
        - *Test*: Modification of tests.
        - *Style*: Change to the style of the code (without changing the logic).

- **Establish a branch strategy :**



- **Git collaboration process:**
    - Regular commits
        - After each significant task (adding a feature, fix, etc.)
    - Pull requests + Reviews code
        - A merge request is made before sending it to the main branch. The student's objective is then to analyse the code submitted by another student in order to improve its quality. They can suggest optimisations, report any errors, or suggest other improvements if necessary.

6.2. QA (quality assurance) strategy

6.2.1. Define a testing strategy

- Unit tests: Check the behaviour of isolated functions, methods and classes.
  - Tool
    - Pytest

- Integration tests: Check that modules work together, endpoints, API ⇔ DB
  - Tool
    - Postman
    - Newman : for automation in CI/CD pipeline

6.2.2. Plan a deployment pipeline

A deployment pipeline is an automated process that takes your code from development to production through various validation stages. Its purpose is to test the application before it goes live.

Tools used:

Github actions : to automatically launch actions when code is pushed via GitHub

Docker : The application is stored in a container to enable an identical environment on the machine (local) + staging + production.

Pipeline map

What is it?

- Local: development on my machine

- Staging: testing before production

- Production: final version accessible to users

Feasible part for the portfolio:

- Writing the code
- Pushing the code to GitHub.
- GitHub actions with tests launch automatically.

Optional part (if time permits) :

- If no problems are encountered, then automatic deployment to staging.
- Manual testing on staging.
- If staging is validated, manual deployment to production.
- Verification that everything is working.
  - If a problem occurs, it is possible to roll back.

Possible services for free accommodation :

- Heroku
- Railway
- Vercel
- Others…

Objectives:

• **Main**: Dockerfile + GitHub Actions with automatic testing

• **Bonus**: Test environment + production deployment

7.Technical justification

React

❖ Use of React because it will save time by using reusable components for HTML + CSS

PostgreSQL / Postgis

❖ Interesting use because it allows for better management of large volumes of data and provides a geographic information system thanks to PostGIS, which will enable the use of geospatial shapes with coordinates to place a marker on a specific area

Python / Flask

❖ Use of Python (and Flask) because we have a better knowledge of this language and have already had the opportunity to work on it in a group project. Furthermore, given that we are learning many new technologies in this project, we prefer to avoid overloading ourselves by learning too many new languages simultaneously, so that we have time to release a functional project that we will be able to explain and master.

Javascript / Leaflet

❖ Use of JavaScript for dynamic display of web pages. Leaflet, a JavaScript library, will be useful for visualising and styling the geospatial data stored in PostGIS