# CartAlex Project: Technical Documentation

**Version:** 1.0

**Date:** September 24, 2025

**Author:** Ezz Eldin

**Status:** Active Development

# 1. Project Overview

### 1.1. Purpose of the Project

CartAlex is an enterprise-grade Web Geographic Information System (WebSIG) engineered for the   
**Centre d'Études Alexandrines (CEAlex)**. Its primary objective is to provide a robust, interactive, and high-performance platform for the visualization, analysis, and management of archaeological and historical data specific to Alexandria, Egypt. The system is designed to serve as a central hub for researchers, historians, and archaeologists, enabling them to explore complex spatial and relational datasets through an intuitive web interface.

### 1.2. Key Features and Functionalities

* **Interactive Map Interface:** A fluid, responsive map powered by MapLibre GL JS, offering seamless panning, zooming, and interaction.
* **Multi-Source Data Rendering:** The system is architected to render multiple data types from various sources concurrently:
  + **Vector Data:** Points of interest (excavation sites) and complex polygons (regional parcels, public spaces).
  + **Raster Data:** Historical scanned maps and modern satellite imagery.
* **Advanced Data Filtering:** A dynamic query panel allows users to perform complex, multi-criteria filtering on vector layers. The initial implementation focuses on the "Sites de fouilles"   
  (excavation sites) layer, with filters for vestiges, discoveries, and bibliographical references,  
  including date-range selections.
* **Dynamic Layer Management:** A "Choix des couches" panel provides users with granular control over layer visibility, allowing them to toggle individual data layers on or off.
* **Opacity Control:** A built-in opacity slider for raster and key polygon layers enables users to perform visual comparisons between historical and contemporary map data.
* **Containerized Architecture:** The entire application stack is **containerized using Docker** and managed via Docker Compose, ensuring consistent, reproducible deployments and simplifying the development workflow.

### 1.3. High-Level Architecture Description

CartAlex is built on a modern, decoupled microservices-oriented architecture. The system is composed of four primary, containerized services: a web application server (Node.js/Express), a geospatial database (PostgreSQL/PostGIS), and two specialized vector tile servers (Tegola and pg\_tileserv). This separation of concerns ensures that each component is optimized for its specific task, enhancing performance, scalability, and maintainability.

# 2. Technologies & Tools

The technology stack for CartAlex was selected to prioritize performance, scalability, and adherence to open standards.

|  |  |  |
| --- | --- | --- |
| **Component** | **Technology/Tool** | **Role & Justification** |
| **Frontend** | **MapLibre GL JS** | The core rendering library for the interactive map. Chosen for its high performance, open-source nature, and compatibility with the Mapbox Vector Tile (MVT) specification. |
| **JavaScript (ES6)** | The primary language for client-side logic, managing the user interface, state, and interactions with the map and backend API. |
| **HTML5 / CSS3** | Standard for structuring and styling the web interface. CSS variables are used for theme management and maintainability. |
| **Backend** | **Node.js** | The runtime environment for the backend server. Its asynchronous, non-blocking I/O model is well-suited for handling API requests and serving static files. |
| **Express.js** | A minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications, serving as our API and static file server. |
| **Database** | **PostgreSQL 13** | A powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance. |
| **PostGIS 3.1** | A critical extension for PostgreSQL that adds support for geographic objects, allowing for location queries to be run in SQL. It is the foundation of our geospatial data storage. |
| **Tile Servers** | **Tegola** | A high-performance vector tile server written in Go. Chosen specifically for its efficiency in serving point data. It connects directly to PostGIS and serves sites\_fouilles tiles. |
| **pg\_tileserv** | A lightweight, high-performance tile server that directly serves PostGIS tables as vector tiles. Chosen for its simplicity and efficiency in serving polygon and line data. |
| **Deployment** | **Docker & Docker Compose** | The core of our deployment and development strategy. Docker provides containerization for each service, and Docker Compose orchestrates the multi-container application, ensuring a consistent and isolated environment. |
| **Build Tools** | **Webpack** | A static module bundler for modern JavaScript applications. Used to bundle all frontend assets (JS, CSS) into optimized files for production. |
| **Babel** | A JavaScript compiler used to transpile modern ES6+ code into a backwards-compatible version of JavaScript that can be run by older browsers if needed. |

# 3. System Architecture

### 3.1. Architecture Diagram

The system operates as a cohesive unit of four containerized services:

1. **Client (Browser):** The user interacts with the application through a web browser. The frontend logic, built with HTML, CSS, and JavaScript, is responsible for all rendering and user interactions.
2. **Web Application Server (app):**
   * **Static Server:** Serves the initial map.html, bundled JavaScript, and CSS files to the client.
   * **API Server:** Provides RESTful endpoints that the client consumes to populate the filter panel (e.g., /getValues/vestiges) and to retrieve lists of feature IDs based on user-selected filters.
3. **Database Server (db):** The PostgreSQL/PostGIS database is the single source of truth for all geospatial and relational data.
4. **Tile Servers (tegola & pg\_tileserv):** These services act as a high-performance read-only cache between the client and the database for geospatial data. They dynamically generate vector tiles from the database in response to requests from the client's map.

### 3.2. Component Interaction Flow

* **Map Data:** The client's map (MapLibre) directly requests vector tiles from the **tegola** server (for points) and the **pg\_tileserv** server (for polygons and lines). These servers, in turn, query the **db** service to generate the tiles.
* **Filter Data:** The client's filter panel logic makes API calls to the **app** server's backend. The backend then queries the **db** service for non-geospatial data (e.g., lists of authors) or for filtered lists of feature IDs.

# 4. Development Workflow

### 4.1. Branching Strategy & Git Workflow

The project follows a standard **GitFlow** branching model:

* **main:** This branch contains production-ready, stable code. Direct commits are forbidden. Merges only occur from the develop branch.
* **develop:** The primary development branch. All feature branches are created from develop and merged back into it.
* **Feature Branches (feature/...):** For new features or bug fixes. Each task is developed in its own branch (e.g., feature/add-opacity-slider).

### 4.2. Environments

* **Development:** The local environment is managed entirely by Docker Compose, ensuring it perfectly mirrors the production setup.
* **Staging/Production:** (To be defined) A similar Docker Compose setup would be deployed to a cloud server, with environment variables managed through a secure system.

### 4.3. Testing Strategy

(To be implemented)

* **Unit Tests:** Jest or a similar framework should be implemented to test individual backend functions (e.g., API route handlers) and frontend components.
* **Integration Tests:** A strategy should be developed to test the interaction between the services (e.g., ensuring the backend can connect to the database).

# 5. Detailed Modules Explanation

### 5.1. Map Rendering Module (src/index.js)

* **Description:** This is the core of the frontend. It initializes the MapLibre GL JS map instance and defines the map's style, which includes all data sources and layer appearances.
* **Data Flow:** It fetches vector tiles from the Tegola and pg\_tileserv endpoints and renders them as visual layers.
* **Dependencies:** maplibre-gl, app.js.

### 5.2. Application Logic Module (src/js/app.js)

* **Description:** This module contains the main App class that orchestrates the entire client-side application. It initializes the filters, manages the map state, and handles user events.
* **Data Flow:** It communicates with the UI module to build the interface and with the backend API to fetch filter data.
* **Dependencies:** FilterCollection.js, ui.js, server\_config.js.

### 5.3. UI Module (src/js/ui.js)

* **Description:** This module is responsible for dynamically generating the HTML for the filter panels and attaching all necessary event listeners (e.g., for opening/closing panels, clicking checkboxes, and using sliders).
* **Data Flow:** It receives data from the app.js module and translates it into DOM elements. It then listens for user interactions and calls back to the app.js module to update the application state.
* **Dependencies:** None.

### 5.4. Filter Logic Modules (src/js/FilterCollection.js, Filter.js, SubFilter.js)

* **Description:** A set of classes that model the filter hierarchy. FilterCollection manages a set of Filters (e.g., "Vestiges"), which in turn manage a set of SubFilters (e.g., "Caractérisation"). This object-oriented approach keeps the filtering logic clean and maintainable.
* **Data Flow:** These modules make API calls to the backend to initialize their values and to get lists of filtered feature IDs.
* **Dependencies:** server\_api.js.

### 5.5. Backend API (src/server/)

* **Description:** An Express.js application that serves the frontend and provides a REST API.
* **server.js:** The main entry point that configures the Express server, middleware, and routes.
* **routes.js:** Defines all API endpoints, such as /getValues/:tableName for populating filters and /sitesFouilles/:filter for executing filter queries.
* **db.js:** Manages the connection to the PostgreSQL database.
* **Data Flow:** Receives requests from the frontend, queries the database, and returns JSON responses.

# 6. Setup & Installation Guide

### 6.1. Local Setup

1. **Prerequisites:** Ensure Docker and Docker Compose are installed.
2. **Clone Repository:** git clone <your-repository-url>
3. **Navigate to Root:** cd cartalex-tiles
4. **Environment Configuration:**
   * Copy the .env.example file to a new file named .env.
   * Verify that the default values in .env are correct for your local Docker setup (they typically are).
5. **Build and Launch:** ( Run docker-compose up --build -d. )

### 6.2. Deployment

The application is deployed as a set of Docker containers. The same docker-compose.yml file can be used on a production server, with the following considerations:

* **Environment Variables:** Production database credentials and other secrets should be managed securely (e.g., with Docker Secrets or a cloud provider's secret manager) and not hardcoded in the .env file.
* **Networking:** The application's port (3000) should be exposed to the internet, likely through a reverse proxy like Nginx for security and performance.

# 7. How to Launch the Project

### 7.1. Step-by-Step Launch Guide

1. **Open a terminal** and navigate to the project's root directory.
2. **Run the launch command:**  
   docker-compose up --build -d
3. **Wait for services to start.** You can check the status of the containers with docker-compose ps.
4. **Access the application** in your browser at http://localhost:3000/map.html.
5. **Access Tile Server Admin Panels (for debugging):**
   * **pg\_tileserv:** http://localhost:7800/
   * **Tegola:** http://localhost:8080/

### 7.2. Common Issues

* **Containers fail to start:** Check the logs for a specific container with docker-compose logs <service-name> (e.g., docker-compose logs app). This often points to a database connection issue or a misconfigured environment variable.

# 8. Team & Roles

* **Lead Developer / Architect:**   
  - kamal Mohsen (Head of IT Department )  
  - Ezz Eldin , Mustafa Morsi (Responsible for overall architecture, backend development, and deployment).
* **Collaboration Tools:**
  + **Version Control:** Git ( <https://github.com/EzzEldinx/Cartalex-tiles.git> ).

# 9. Future Improvements

### 9.1. Possible Optimizations

* **Database Indexing:** Ensure all frequently queried columns in the database (especially foreign keys and geometry columns) are indexed to improve query performance.
* **Code Splitting:** The frontend JavaScript can be split into smaller chunks using Webpack, so users only load the code they need for the initial view.
* **Image Optimization:** The historical map raster images in the public folder should be optimized (e.g., converted to WebP format) to reduce their file size.

### 9.2. Scalability Considerations

* The current architecture is highly scalable. If the application experiences high traffic, the tile servers (which are stateless) can be easily replicated behind a load balancer.

### 9.3. Security Notes

* **Database Credentials:** The .env file containing database credentials should **never** be committed to version control.
* **Input Sanitization:** All API endpoints should have robust input validation and sanitization to prevent SQL injection and other attacks. The current implementation relies on parameterized queries, which is a strong defense.
* **Production Deployment:** For a public-facing deployment, a reverse proxy like Nginx should be placed in front of the Node.js application to handle HTTPS, rate limiting, and other security concerns.

# 10. Appendix

### 10.1. Special Scripts

* **final\_fix.sql:** A critical SQL script used to correct data inconsistencies in the references\_biblio table. It should be run if the "Bibliographies" filter shows incorrect data points.
  + **How to run:**  
    docker exec -i cartalex\_db psql -U postgres -d cartalex\_basileia\_3857 < final\_fix.sql

# 11. Updating and Extending the Application

This section provides a guide for common updates and content additions to the CartAlex platform.

### **11.1.** How **to Add a New Historical Map**

The application is designed to easily accommodate new historical raster maps. The process involves preparing the map tiles and updating the frontend configuration.

# Step 1: Prepare Map Tiles

The application expects raster tiles to be in a standard {z}/{x}/{y}.png format.

1. **Georeference the Map:** Use a GIS software like QGIS to georeference your historical map image against a modern map (like OpenStreetMap).
2. **Generate Tiles:** Use the gdal2tiles.py script (part of the GDAL library) or a QGIS plugin to export the georeferenced map as a directory of tiles.
3. **Place Tiles in public Folder:**

* Create a new directory inside the project's public/ folder (e.g., public/new\_map/).
* Place the generated tile directory (containing the {z}, {x}, and {y} subdirectories) inside this new folder.

# Step 2: Update the Frontend Configuration

1. **Open src/index.js**.
2. **Add a New Source:** In the sources object of the map style, add a new entry for your map. The key should be a unique identifier, and the tiles URL must point to the new directory in the public folder.  
     
   // ... inside the sources object  
   "new\_historical\_map": {  
       "type": "raster",  
       "tiles": ["/new\_map/{z}/{x}/{y}.png"],  
       "tileSize": 256,  
       "attribution": "Attribution for the New Map, Year"  
   }
3. **Add a New Layer:** In the layers array of the map style, add a new layer that references the source you just created. Give it a user-friendly ID, which will be its display name.  
     
   // ... inside the layers array  
   {  
       "id": "Name of New Map (Year)",  
       "type": "raster",  
       "source": "new\_historical\_map",  
       "layout": { "visibility": "none" } // Off by default  
   }
4. **Rebuild the Application:** Run docker-compose up --build -d to apply the changes. The new map will now appear as a toggleable layer in the "Choix des couches" panel.

### **11.2. How to Add New Points or Polygons**

Adding new vector data (points, lines, or polygons) involves updating the database and potentially the tile server configuration.

# Step 1: Import Data into PostGIS

1. **Prepare Data:** Ensure your new vector data is in a standard format (e.g., Shapefile, GeoJSON, GeoPackage).
2. **Import to Database:** Use a GIS tool like QGIS or the shp2pgsql command-line utility to import your data into a new table in the cartalex\_basileia\_3\_857 database.

* **Crucially, ensure the new data is in the correct projection (EPSG:3857)** to match the rest of the project's data.

# Step 2: Update Tile Server and Frontend

The process depends on the type of data:

* **For Points (sites\_fouilles):**

1. **Database:** Add the new points directly to the public.sites\_fouilles table.
2. **Tegola:** No changes are needed in tegola.toml, as it is already configured to serve this entire table. The new points will appear automatically.

* **For New Polygons or Lines:**

1. **pg\_tileserv:** This server automatically detects and serves new tables. No configuration changes are needed.
2. **Frontend (src/index.js):** You must add a new **source** and **layer** for the new table, just like in Step 2 of adding a historical map.

* The source URL will point to the new table (e.g., http://localhost:7800/public.new\_polygon\_table/{z}/{x}/{y}.pbf).
* The layer's source will be the new source name, and its source-layer will be the table name (e.g., public.new\_polygon\_table).

1. **Rebuild:** Run docker-compose up --build -d. The new layer will now be available in the   
    "Choix des couches" panel.