

# WEB APP DASHBOARD

## DOCUMENTATION

### BACKEND

The project consists of backend FastAPI, frontend React, and database MongoDB. Aka FARM stack, you can check on google for more info.

The Backend provides endpoints for user authentication, model predictions, and data management. The backend interacts with a **MongoDB** for data storage and uses **Rate Limiting** to prevent abuse. Here's the backend structure:

```
Back-End/
├── main.py           # Entry point for the FastAPI application
├── requirements.txt  # Python dependencies for the Backend
├── rateLimiter.py    # Rate limiting configuration
├── config/
│   └── db.py         # MongoDB connection and configuration
├── models/
│   └── TrainingRound.py # Validation for rounds
├── routes/ # Routes are where endpoints
│   ├── auth.py       # Authentication routes (register, login, logout, etc.)
│   ├── predict.py    # Prediction route
│   └── route.py       # Routes for client and round data
├── tests/
│   ├── test_auth_endpoints.py # Tests for authentication endpoints
│   ├── test_interaction.py    # Tests for frontend-backend interaction
│   └── test_authentication.py # Tests for frontend authentication
```

There's also .env file (Environment Variables) in the root of Back-End folder which contains sensitive information, it is not uploaded in the GitHub repository, and it is required for running the project locally:

```
MONGO_PUBLIC_URL=mongodb://mongo:HiwDMYxRRpgqkefLILYZynRVwRWqImpy@autorack.proxy.rlwy.net:44467
DB_NAME=f1_all
```

### MAIN.PY

**main.py** initializes and configures the **FastAPI** application. It sets up middleware, routes, database connections, rate limiting, and static file serving. It serves the frontend application for deployment. The comments are in the code, but I'll try to explain some parts here in the documentation.

### Application Lifecycle

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    await open_connection()
    yield
    await close_connection()
```

It uses simple functions from db.py and it ensures the database connection is opened when the app starts and closed when it shuts down.

## Rate Limiting

```
app.state.limiter = limiter
app.add_middleware(SlowAPIMiddleware)

@app.exception_handler(RateLimitExceeded)
async def rate_limit_exceeded_handler(request, exc):
    return JSONResponse(
        status_code=429,
        content={"detail": "Rate limit exceeded. Please try again later."},
    )
```

This adds a rate limiter using slowapi library which prevents abuse by limiting the number of requests per user by IP address. It also has an exception handler that returns an error message when it exceeds the number of requests.

There are remaining code left but it's just including routes and serving static and frontend. You can check out main.py for that.

## DB.PY

**db.py** provides functionality to initialize, access, and close the database connection, ensuring that the application can interact with the database efficiently and securely.

```
mongodb = MongoDB()

async def open_connection():
    mongo_uri = os.getenv("MONGO_PUBLIC_URL")
    db_name = os.getenv("DB_NAME")

    mongodb.client = AsyncIOMotorClient(mongo_uri)
    mongodb.set_db(mongodb.client[db_name])

async def close_connection():
    if mongodb.client:
        mongodb.client.close()
```

We create a global instance of MongoDB class. The instance is used throughout the application to access the database. The first function initializes the connection to the MongoDB database, and it gets the connection string from .env file locally or from railway added variables. The second function closes the connection when the application shuts down.

## TRAININGROUND.PY

**TrainingRound.py** represents and validates the structure of the data stored in and retrieved from the database. It provides validation to prevent invalid data from being stored in MongoDB when the rounds are posted from an endpoint. This is how each document should look like:

```
{
  "_id": {
    "$oid": "67edd9240ee6d6aca2faa2c6"
  },
  "round": "1",
  "created_at": "2025-04-03T03:41:08.097731+00:00",
  "Global": {
    "accuracy": 75.53516819571865,
    "precision": 76.17001706675619,
  }
}
```

```

    "recall": 69.21844246280298,
    "f1": 67.45199995026546,
    "avg_loss": 0.9376450266156878
  },
  "client_0": {
    "accuracy": 43.883792048929664,
    "precision": 48.6834537554834,
    "recall": 48.761899385814864,
    "f1": 37.546744583753444,
    "avg_loss": 0.9635852397907347
  },
  "client_1": {
    "accuracy": 71.7125382262997,
    "precision": 73.88304374833203,
    "recall": 66.57821093628861,
    "f1": 66.2880742039191,
    "avg_loss": 0.6869678493766558
  },
  "client_2": {
    "accuracy": 54.89296636085626,
    "precision": 42.23581426321249,
    "recall": 50.35736276771687,
    "f1": 40.91039997856779,
    "avg_loss": 1.1392560609605251
  },
  "client_3": {
    "accuracy": 72.47706422018348,
    "precision": 60.678118445172004,
    "recall": 65.30573160408825,
    "f1": 61.2506676595467,
    "avg_loss": 0.7106136600098883
  }
}

```

## ROUTE.PY

**route.py** has endpoints for interacting with the training rounds of client/global data stored in the MongoDB database. The endpoints handle tasks such as posting new rounds, fetching unique client IDs, retrieving all rounds for a specific client/global ID, and retrieving the best global model by F1 score. These endpoints only allow an admin to retrieve and update training data.

### Detailed Explanation of the Endpoints

#### 1. /get - Retrieve All Training Rounds

- Purpose: Fetches all training rounds from the database.
- Access Control: Only accessible to users with the admin role.
- Use Case: Primarily used for debugging or retrieving all training data for analysis.

#### 2. /post - Post Training Rounds

- Purpose: Replaces the existing training rounds in the database with new data.
- Access Control: Requires a valid API key ([x\\_api\\_key](#)) for authentication.

- Use Case: Used to update the database with new training data.
- 

### 3. /client - Retrieve Unique Client IDs

- Purpose: Dynamically fetches all unique client IDs and global metrics from the database.
  - Access Control: Only accessible to users with the admin role.
  - Use Case: Used to populate dropdowns or filters in the frontend for selecting specific clients.
- 

### 4. /rounds/{client\_id} - Retrieve Training Rounds for a Specific Client

- Purpose: Retrieves all training rounds for a specific client, with optional sorting (ascending or descending).
  - Access Control: Only accessible to users with the admin role.
  - Use Case: Used to display client-specific training data in tables or charts.
- 

### 5. /best-f1-global - Retrieve Best F1 Score for Global Model Summary

- Purpose: Finds and retrieves the training round with the highest F1 score for global model.
- Access Control: Only accessible to users with the admin role.
- Use Case: Used to highlight the best global model in the frontend.

## AUTH.PY

**auth.py** provides endpoints and utilities for user registration, login, session management, and access control. This ensures that only authenticated and authorized users can access protected resources in the application. It limits the number of requests to authentication endpoints to prevent abuse (rate limiting).

**Note:** We're using session cookies for authentication and session management.

- Session cookies rely on a server-side session store (in this case, MongoDB) to manage session data.
- The cookie only contains a reference (the session token) to the session data stored on the server.
- Validation requires checking the session token against the session store.

**Server-Side Control:** Sessions can be invalidated/revoked server-side (e.g., by deleting the session token from the database).

**Security:** The session token is stored as an HTTP-only cookie, which prevents client-side JavaScript from accessing it, reducing the risk of XSS attacks.

The main problem with JWT (JSON Web Token) is once a JWT is issued, it cannot be revoked or invalidated which means that it can be reused by attackers if intercepted (e.g., via a man-in-the-middle attack) because it does not track usage.

## **Detailed Explanation of the code**

### **1. Session Management**

- **Purpose:**
    - The session is created with 32-byte token.
    - Manages user sessions by creating, validating, and deleting session tokens.
    - Ensure sessions expire after 1 hour unless the user login again.
  - **Security:**
    - Expired tokens are automatically removed to prevent unauthorized access.
- 

### **2. Authentication Middleware**

- **Purpose:**
    - Validates the user's session by checking the session\_token stored in the browser's cookies and retrieves their details from the database.
    - Enforces role-based access control for protected routes.
- 

### **3. /register - User Registration**

- **Purpose:**
    - Register a new user by validating their email and password.
    - Hashes the password and stores the user in the database with a default role of clinic (Only have access to model trial).
  - **Security:**
    - Prevents duplicate registrations by checking if the email already exists in the database.
    - Uses strong password hashing (argon2id) to protect user credentials.
- 

### **4. /login - User Login**

- **Purpose:**
    - Authenticates the user by verifying their credentials.
    - Creates a new session token and sets it as an HTTP-only secure cookie for the duration of the session.
  - **Security:**
    - **httponly=True:** Protects against XSS (Cross-Site Scripting)
      - Prevents JavaScript from accessing the cookie (document.cookie), making it harder for attackers to steal the session token.
    - **secure=True:** Protects against MITM (Man-in-the-Middle)
      - Ensures the cookie is only sent over HTTPS connections, preventing exposure over insecure HTTP.
    - **samesite="strict":** Protects against CSRF (Cross-Site Request Forgery)
      - The cookie will not be sent with cross-site requests
- 

## 5. /logout – User Logout

- **Purpose:**
    - Deletes the user's session token from the database.
    - Clear the session token cookie in the browser.
  - **Security:**
    - Ensure the session is invalidated immediately upon logout.
    - Prevents unauthorized access by removing the session token from both the database and the browser.
- 

## 6. Protected Routes

- **Purpose:**
  - Ensures only authenticated users with the appropriate role can access protected resources.
  - Prevents unauthorized access to sensitive resources.
  - Examples:

- /dashboard: Accessible only to users with the admin role.
- /clients: Accessible only to users with the admin role.
- /model-trial: Accessible to users with the clinic or admin role.

## PREDICT.PY

**predict.py** is to handle image classification predictions using a pre-trained machine learning model. It provides an API endpoint (/predict) that allows users to upload images, processes the images, and returns predictions along with metadata such as confidence scores, class probabilities, and model details.

Note: When we're sending the image to process it, we don't need to encrypt it because Railway provides TLS (Transport Layer Security) for applications deployed on their platform. It automatically serves traffic over HTTPS with TLS encryption. So, all of GET/POST data queries are encrypted during transmission.

### Purpose

#### 1. Image Classification:

- Accepts an image file as input.
- Preprocesses the image to make it compatible with the model.
- Uses a pre-trained neural network to classify the image into one of the predefined classes.

#### 2. Model Management:

- Loads the latest version of the model from MongoDB's GridFS storage.
- Ensures the model is ready for inference by loading its state and setting it to evaluation mode.

#### 3. API Response:

- Returns the classification result, including:
  - Predicted class.
  - Confidence score.
  - Probabilities for all classes.
  - Metadata such as image dimensions, file type, and model upload date.

#### 4. Rate Limiting:

- Limits the number of requests to the /predict endpoint to prevent abuse (e.g., 10 requests per minute).

## TESTS FOLDER

We used Playwright and Pytest for testing. The folder contains multiple test files, each focusing on specific aspects of the backend:

- `test_auth_endpoints.py`:
  - Tests the authentication-related endpoints, such as `/register`, `/login`, `/logout`, and `/verify-token`.
  - Includes tests for:
    - User registration.
    - Login with valid and invalid credentials.
    - Session token validation and expiration.
    - Logout functionality.
- `test_authentication.py`:
  - Focuses on end-to-end authentication workflows, such as:
    - Registering and logging in users.
    - Handling session expiry.
    - Role-based access control for protected routes.
- `test_interaction.py`:
  - Tests the interaction between different backend components and the frontend.
  - Includes tests for:
    - Admin access to the dashboard and clients page.
    - Uploading files and verifying classification results on the Model Trial page.

You can open the `report.html` to see the logs and the result.



## DATABASE

We're using Railway's database service MongoDB.

- **Database Name:** fl\_all
  - **Purpose:**
    - Stores user authentication data, session tokens, training rounds, and machine learning model metadata.
    - Supports the backend API for the application.
- 

### Collections

The database consists of the following collections:

#### 1. Users

- **Purpose:** Stores user authentication details.
- **Schema:**

```
{
  "_id": "ObjectId", // Unique identifier for the user
  "username": "String", // User's email address
  "password": "String", // Hashed password using argon2id
  "role": "String", // User role (e.g., admin, clinic)
  "created_at": "Date" // Timestamp of user registration
}
```

- **Relationships:**
  - One-to-One: A user can only have one session at a time in the Sessions collection. (Prevents session sharing or session hijacking.)

- **Example Document:**

```
{
  "_id": "64f8c9e2b5a1c2d3e4f5g6h7",
  "username": "testuser@example.com",
  "password": "$argon2id$v=19$m=65536,t=3,p=4$...",
  "role": "clinic",
  "created_at": "2023-09-01T12:00:00Z"
}
```

---

#### 2. Sessions

- **Purpose:** Tracks active user sessions.

- **Schema:**

```
{
  "_id": "ObjectId", // Unique identifier for the session
  "user_id": "ObjectId", // Reference to the user in the Users collection
  "token": "String", // Session token
  "expires": "Date" // Expiration timestamp for the session
}
```

- **Relationships:**

- One-to-One: Only one session can belong to a single user.

- **Example Document:**

```
{
  "_id": "64f8c9e2b5a1c2d3e4f5g6h8",
  "user_id": "64f8c9e2b5a1c2d3e4f5g6h7",
  "token": "a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6",
  "expires": "2023-09-01T13:00:00Z"
}
```

---

### 3. Rounds

- **Purpose:** Stores training round data, including metrics for global and client-specific performance.

- **Schema:**

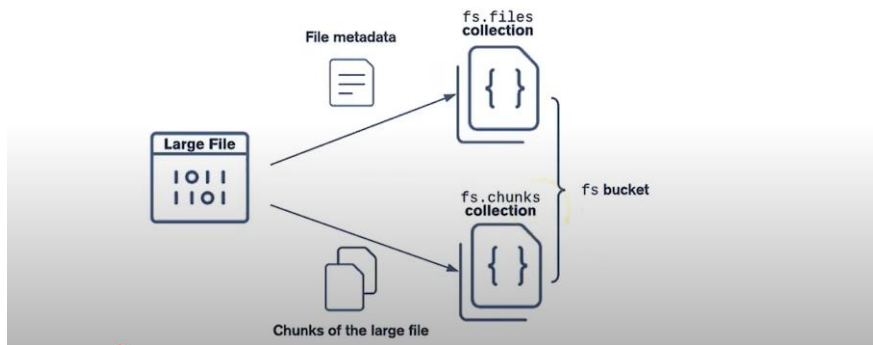
```
{
  "_id": "ObjectId", // Unique identifier for the round
  "round": "String", // Training round number
  "created_at": "Date", // Timestamp of when the round was created
  "Global": { // Global metrics for the round
    "accuracy": "Float",
    "precision": "Float",
    "recall": "Float",
    "f1": "Float",
    "avg_loss": "Float"
  },
  "client_1": { // Metrics for client 1
    "accuracy": "Float",
    "precision": "Float",
    "recall": "Float",
    "f1": "Float",
    "avg_loss": "Float"
  },
  "client_2": { // Metrics for client 2
    "accuracy": "Float",
    "precision": "Float",
    "recall": "Float",
    "f1": "Float",
    "avg_loss": "Float"
  },
  // And so on...
}
```

- **Relationships:**
  - None (self-contained collection).
- **Example Document:** You can see the example in TrainingRound.py part.

#### 4. Models/FS.Files

- Note: When a new model is uploaded, the file is stored in GridFS, creating a document in the fs.files collection. The model's name, including the file\_id of the uploaded file, is stored in the models collection.

GridFS by default uses two collections `fs.files` and `fs.chunks` to store the file's metadata and the chunks



- **Purpose:** Stores metadata and files for machine learning models.
- **Schema:**

```
{
  "_id": "ObjectId", // Unique identifier for the file in GridFS
  "filename": "String", // Name of the file
  "contentType": "String", // MIME type of the file (e.g., "application/octet-stream")
  "md5": "String", // MD5 hash of the file for integrity verification
  "chunkSize": "Number", // Size of each chunk in bytes (used internally by GridFS)
  "length": "Number", // Total size of the file in bytes
  "uploadDate": "Date" // Timestamp of when the file was uploaded to GridFS
}
```

- **Relationships:**
  - One-to-One: Each model metadata document references a file in GridFS.

- **Example Document:**

```
{
  "_id": {
    "$oid": "67edd92449a40a8d7c9413d6"
  },
  "filename": "best_federated_model.pth",
  "contentType": "application/octet-stream",
  "md5": "2b710c2127d8ed185b4e239f4a8eb419",
  "chunkSize": 261120,
  "length": 51760110,
  "uploadDate": {
    "$date": "2025-04-03T00:41:26.457Z"
  }
}
```

## FRONTEND

The frontend is a React-based web application that serves as the user interface for the FL-ALL. It allows the admin to interact with the backend, visualize data, and perform tasks such as authentication, viewing performance metrics, and clinics uploading images for analysis. Here's an overview of the frontend structure:

Front-End/	
├─ App.jsx	# Main React component that defines routes and layout.
├─ index.css	# Tailwind CSS styles for the application.
├─ main.jsx	# Entry point for the React application.
├─ components/	# Contains React components.
│   └─ auth/	# Components for authentication (e.g., Login, Register, AuthContext).
│   └─ dashboard/	# Components for the dashboard (e.g., DashboardStats, PerformanceChart).
│   └─ modeltrial/	# Components for the Model Trial page (e.g., PredictionResult, ModelTrial)
│   └─ layout/	# Layout components like Sidebar and Header.
│   └─ common/	# Shared components like tables and selectors.
│   └─ clients/	# Component for the Clients page.
├─ hooks/	# Custom React hooks for reusable logic.
│   └─ useTheme.js	# Manages light/dark theme settings.
│   └─ usePerformanceData.js	# Fetches performance data for clients.
│   └─ useClients.js	# Fetches unique client IDs.
├─ services/	# API service functions for interacting with the backend.
└─ api.js	# Contains functions for API calls (e.g., login, fetch metrics).

## MAIN.JSX

The **main.jsx** file is the entry point of the React application. It wraps the entire app with multiple providers to manage global states and functionalities.

```
createRoot(document.getElementById('root')).render(
  <HelmetProvider>
    <BrowserRouter>
      <AuthProvider>
        <App />
      </AuthProvider>
    </BrowserRouter>
  </HelmetProvider>
);
```

## 1. HelmetProvider

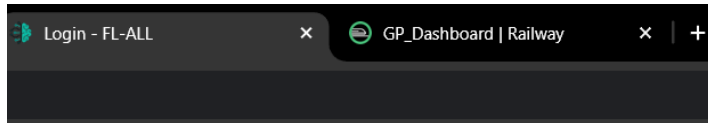
- **Purpose:**

- Manages the <head> section of the HTML document dynamically.
- Allows you to set page titles for each route.

- **How It Works:**

- Wraps the app to enable the use of the Helmet component in child components.
- Example in **App.jsx**:

```
<Helmet>
  <title>{getPageTitle()} - FL-ALL</title>
</Helmet>
```



## 2. BrowserRouter

- **Purpose:**

- Provides routing functionality for the application.
- Enables navigation between different pages (e.g., /login, /dashboard, /model-trial).

- **How It Works:**

- Wraps the app to enable the use of Route, Navigate, and useNavigate for client-side routing.
- Example in **App.jsx**:

```
<Routes>
  <Route path="/" element={<Navigate to="/login" />} />
  <Route path="/login" element={<Login setActiveTab={setActiveTab} />} />
  <Route path="/register" element={<Register setActiveTab={setActiveTab} />} />
</Routes>
```

## 3. AuthProvider

- **Purpose:**

- Manages authentication state (e.g., isAuthenticated, userRole, username).
- Provides functions like refreshAuthState (to validate sessions) and logout (to log out users and remove session\_token and session storage).

- **How It Works:**

- Wraps the app to make authentication-related state and functions available to all components via the AuthContext.
- Example in AuthContext.jsx:

```
<AuthContext.Provider value={{ isAuthenticated, userRole, logoutt, etc... }}>
  {children}
</AuthContext.Provider>
```

- Example in Sidebar.jsx:

```
const { userRole, logoutt } = useContext(AuthContext);
```

We can use these consts or functions in Sidebar.jsx now.

---

## How These Providers Work Together

### 1. HelmetProvider:

- Ensures each page has a dynamic title and meta tags for better user experience and SEO.

### 2. BrowserRouter:

- Handles navigation between pages and ensures the app is a single-page application (SPA).

### 3. AuthProvider:

- Manages user authentication and ensures only authorized users can access certain routes.

Together, these providers create a robust foundation for the frontend, enabling dynamic routing, authentication, and page management.

## APP.JSX

**App.jsx** is the **main component** of the frontend application. It defines the structure, layout, and routing for the entire app. It handles navigation, page titles, and rendering components like the sidebar and header.

### Key Features

#### 1. Role-Based Access:

- Uses ProtectedRoute to ensure only users with the correct roles can access certain pages (e.g., only admin can access /dashboard and /clients).
- Example:
  - 1. Admin-only routes:

```
<Route path="/dashboard" element={<ProtectedRoute element={<DashboardStats />} allowedRoles={['admin']} fetchAuth={authDashboard} />} />
```

## 2. Admin and clinic routes:

```
<Route path="/model-trial" element={<ProtectedRoute element={<ModelTrial />} allowedRoles={['admin', 'clinic']} fetchAuth={authModelTrial} />} />
```

## 2. Dynamic Navigation:

- Tracks the active tab and updates it in localStorage for persistence.

```
useEffect(() => {  
  localStorage.setItem('activeTab', activeTab);  
}, [activeTab]);
```

## 3. Responsive Layout:

- Manages the sidebar state for mobile and desktop views.

```
const [isSidebarOpen, setSidebarOpen] = useState(false);
```

## 4. Page Titles:

- Dynamically updates the browser tab title using react-helmet-async.

```
<Helmet>  
  <title>{getPageTitle() || ""} - FL-ALL</title>  
</Helmet>
```

## SERVICE/API.JS

**api.js** is a **service module** that basically centralizes all API calls to the backend. It provides functions for interacting with the backend endpoints, such as fetching data, handling authentication, and sending predictions.

### Purpose

#### 1. Centralized API Management:

- Keeps all API-related logic in one place, making it easier to maintain and update.

#### 2. Simplifies Frontend Code:

- Allows components to call these functions instead of writing fetch logic.

Here's a simple one-line explanation for each function in **api.js**:

- **fetchUniqueClientIds**: Fetches a list of unique client IDs from the backend.
- **fetchTrainingMetrics**: Retrieves training metrics for a specific client and round order.
- **fetchBestF1Global**: Gets the best F1 score for global model performance.

- login: Sends user credentials to the backend to log in and start a session.
- logout: Logs the user out by invalidating their session on the backend.
- register: Registers a new user by sending their credentials to the backend.
- verify\_token: Verifies if the user's session token is valid.
- predict: Sends an image file to the backend for classification and returns the prediction result.
- authDashboard: Checks if the user is authorized to access the dashboard.
- authModelTrial: Checks if the user is authorized to access the model trial page.
- authClients: Checks if the user is authorized to access the clients page.

## COMPONENTS/AUTH FOLDER

The **/auth** folder contains components and context related to **user authentication and authorization**. These components handle tasks like logging in, registering, protecting routes, and managing authentication state.

### Purpose of the /auth Folder

1. **Authentication:**
  - Handles user login, registration, and logout.
  - Manages the authentication state based on the session token provided by the backend.
2. **Authorization:**
  - Ensures only authorized users can access specific routes or pages.
3. **State Management:**
  - Provides a centralized (global variables) way to manage authentication state (e.g., isAuthenticated, userRole) using AuthContext.
4. **Error Handling:**
  - Displays appropriate messages when users try to access unauthorized pages.

---

### Key Components in the /auth Folder

#### 1. AuthContext.jsx

- **Purpose:**
  - Manages global authentication state and provides it to the entire app.



- **Key Features:**

- Function refreshAuthState:

- Calls the backend's **/verify-token** endpoint to check if the session token of the user is valid.
    - Updates the frontend's authentication state (isAuthenticated, userRole, username) based on the response.

```
const refreshAuthState = async () => {  
  const response = await verify_token(); // Backend verifies the session token  
  if (response.ok) {  
    const data = await response.json();  
    setIsAuthenticated(true);  
    setUserRole(data.role);  
    setUsername(data.username);  
  } else {  
    setIsAuthenticated(false);  
  }  
};
```

- Function logoutt:

- Calls the backend's /logout endpoint to invalidate the session token.
    - Clears session-related data from the frontend (e.g., sessionStorage).

```
const logoutt = async () => {  
  await logout(); // Backend invalidates the session token  
  setIsAuthenticated(false);  
  setUserRole(null);  
  setUsername(null);  
  sessionStorage.clear();  
  navigate("/login");  
};
```

---

## 2. Login.jsx

- **Purpose:**

- Provides a form for users to log in.

- **Key Features:**

- Sends login credentials to the backend using the **login** function from **api.js**.
  - Redirects users to the appropriate page based on their role (admin → Dashboard, clinic → Model Trial).

---

## 3. Register.jsx

- **Purpose:**
    - Provides a form for users to register a new account.
  - **Key Features:**
    - Sends registration data to the backend using the **register** function from **api.js**.
    - Redirects users to the login page after successful registration.
- 

#### 4. ProtectRoute.jsx

- **Purpose:**
  - Protects routes by ensuring only authenticated and authorized users can access them.
- **Key Features:**
  - Checks if the user is authenticated
  - Verifies the user's role and backend authorization.
  - Redirects unauthorized users to **/unauthorized** or unauthenticated users to **/login**.
- **Usage:**
  - Wraps protected routes in **App.jsx**.
  - Example:

```
<Route path="/clients" element={ <ProtectedRoute element={<ClientsPage />} allowedRoles={['admin']}  
fetchAuth={authClients} />} />
```

---

#### 5. Unauthorized.jsx

- **Purpose:**
    - Displays a message when a user tries to access a page they are not authorized to view.
  - **Key Features:**
    - Simple UI with a message indicating the user lacks permission.
-

## How These Components Work Together

### 1. Authentication Flow:

- **Login.jsx** and **Register.jsx** handle user login and registration.
- **AuthContext.jsx** manages the authentication state and provides it to all components.

### 2. Route Protection:

- **ProtectRoute.jsx** ensures only authenticated and authorized users can access specific routes.
- Redirects unauthorized users to /unauthorized or /login.

### 3. Error Handling:

- **Unauthorized.jsx** displays a message when users try to access restricted pages.

## COMPONENTS/MODEL-TRIAL FOLDER

The **/model-trial** folder in your project contains components related to the **Model Trial feature**, which allows users to upload images for classification and view the results. This feature is designed to provide an interactive interface for testing the global model's predictions.

### Purpose of the /model-trial Folder

#### 1. Image Classification:

- Allows users to upload microscopic cell images for automated classification by the model.

#### 2. Prediction Results:

- Displays the classification results, including the predicted class, confidence, and probabilities for each class.

#### 3. Prediction History:

- Maintains a history of uploaded images and their predictions for easy reference.

## Key Components in the model-trial Folder

### 1. ModelTrial.jsx

- **Purpose:**

- The main component for the Model Trial feature.

- Manages the overall workflow, including image upload, prediction handling, and displaying results.
  - **Key Features:**
    - **Image Upload:**
      - Allows users to upload images for classification.
      - Validates the file type and size (supports PNG, JPG, JPEG up to 5MB).
    - **Prediction Handling:**
      - Calls the **predict** function from the API to get classification results.
      - Stores predictions in sessionStorage for persistence across page reloads.
    - **Prediction Display:**
      - Displays the current prediction using the **PredictionResult.jsx** component.
      - Shows a history of predictions using the **PredictionHistory.jsx** component.
    - **Error Handling:**
      - Displays error messages for invalid files or failed predictions.
    - **Clear History:**
      - Allows users to clear the prediction history.
- 

## 2. PredictionResult.jsx

- **Purpose:**
  - Displays the detailed results of the current prediction.
- **Key Features:**
  - **Image Display:**
    - Shows the uploaded image with a modal for viewing it in full size.
  - **Image Details:**
    - Displays metadata like file name, dimensions, type, and size.
  - **Classification Results:**
    - Shows the predicted class, confidence percentage, and class probabilities.

- **Confidence Bar:**
  - Visualizes the confidence level as a progress bar.
- **Usage:**
  - Used in **ModelTrial.jsx** to display the current prediction.
  - Example:

```
<PredictionResult prediction={currentPrediction} />
```

---

### 3. PredictionHistory.jsx

- **Purpose:**
  - Displays a list of previous predictions for easy reference.
- **Key Features:**
  - **Thumbnail View:**
    - Shows a thumbnail of each uploaded image.
  - **Prediction Details:**
    - Displays the file name, predicted class, confidence, and image dimensions.
  - **Selection:**
    - Allows users to select a prediction to view its details in **PredictionResult.jsx**.
  - **Confidence Bar:**
    - Visualizes the confidence level for each prediction.
- **Usage:**
  - Used in **ModelTrial.jsx** to display the prediction history.
  - Example:

```
<PredictionHistory predictions={predictions} selectedId={selectedPrediction} onSelect={setSelectedPrediction} />
```

---

## How These Components Work Together

1. **ModelTrial.jsx:**
  - Acts as the parent component, managing the state and workflow for the Model Trial feature.

- Handles image uploads, predictions, and history management.

## 2. **PredictionResult.jsx:**

- Displays the details of the currently selected prediction, including the image and classification results.

## 3. **PredictionHistory.jsx:**

- Provides a list of past predictions, allowing users to select and view details for any prediction.

---

### Workflow

#### 1. **Image Upload:**

- Users upload an image in **ModelTrial.jsx**.
- The file is validated and sent to the backend for prediction.

#### 2. **Prediction Handling:**

- The backend returns the predicted class, confidence, and probabilities.
- The prediction is stored in the state and displayed using **PredictionResult.jsx**.

#### 3. **Prediction History:**

- Each prediction is added to the history and displayed in **PredictionHistory.jsx**.
- Users can select a prediction from the history to view its details.

#### 4. **Clear History:**

- Users can clear all predictions from the history and sessionStorage.

---

### HOOKS FOLDER

A hook is function that gives you access to React's internal Memory (useState, useEffect, useContext, etc...). The purpose of the hooks folder is to keep data-fetching, state management, formatting, and other logic separate from UI components.

Note: localStorage is scoped data stored in local storage is accessible across all tabs and windows from the same origin. sessionStorage, on the other hand, is scoped to the individual browser tab or window. When you close the tab, the session storage is cleared.

## Hooks in the Folder

### 1. useClients.js

- **Purpose:**
  - Fetches a list of unique client IDs from the backend and formats them for display.
  - Manages the selected client using sessionStorage to persist the selection across page reloads.
- **How It Works:**
  - useClients: Fetches and formats client IDs (e.g., client\_1 → Client 1).
  - useSelectedClient: Manages the currently selected client, loading it from sessionStorage or defaulting to the first client.
- **Usage:**
  - Used in components like ClientSelector, ClientOverview, and ClientsPage to populate dropdowns or lists of clients and manage the selected client.
- **Example:**

```
const clients = useClients();  
const [selectedClient, setSelectedClient] = useSelectedClient(clients);
```

---

### 2. usePerformanceData.js

- **Purpose:**
  - Fetches performance data (e.g., training metrics) for a selected client and prepares it for charts and tables.
- **How It Works:**
  - Calls the fetchTrainingMetrics function from api.js to get data for a specific client.
  - Stores the data in state and provides it to components.
  - Handles loading and error states.
- **Usage:**
  - Used in components like ClientOverview, ClientsPage, and PerformanceTable to display client performance data.
- **Example:**

```
const { data, chartData } = usePerformanceData(selectedClient);
```

---

### 3. useTheme.js

- **Purpose:**
  - Manages the application's theme (light or dark mode).
- **How It Works:**
  - Reads the current theme from localStorage or defaults to light mode.
  - Updates the theme in localStorage and toggles the dark class on the <html> element.
- **Usage:**
  - Used in components like Header.jsx to allow users to toggle between light and dark themes.
- **Example:**

```
const { theme, setTheme } = useTheme();
```

---

### How These Hooks Fit into the Application

1. useClients.js:
  - Simplifies fetching and formatting client data.
  - Manages the selected client persistently across page reloads.
  - Used in dropdowns or selectors like ClientSelector.jsx.
2. usePerformanceData.js:
  - Centralizes logic for fetching and preparing performance data.
  - Used in tables and chart to display client-specific metrics.
3. useTheme.js:
  - Provides a consistent way to manage themes across the app.
  - Used in layout components like Header.jsx to toggle themes.



## COMPONENTS/LAYOUT FOLDER

**/layout** folder contains 2 components:

### 1. Sidebar.jsx

- **Purpose:** Provides a navigation menu for the app.
- **Features:**
  - Displays navigation links based on the user's role (admin or clinic).
  - Highlights the active tab.
  - Includes a logout button.
- **Usage:** Rendered in App.jsx to allow navigation between pages.

### 2. Header.jsx

- **Purpose:** Displays the top header of the app.
- **Features:**
  - Shows the current page title dynamically.
  - Includes a theme toggle (light/dark mode).
  - Displays the logged-in user's profile (initials and username).
- **Usage:** Rendered in App.jsx to provide page-specific title.

## COMPONENTS/COMMON FOLDER

The **/common** folder contains **shared components** that are reusable across multiple parts of the application.

### Key Components in the common Folder

#### 1. ClientSelector.jsx

- **Purpose:**
  - A dropdown component that allows users to select a client (e.g., hospital).
- **Key Features:**
  - Fetches a list of clients using the **useClients** hook.
  - Displays the list of clients in a dropdown menu.

- Triggers a function `setSelectedClient` when the selected client changes.
- **Usage:**
  - Used in components like **ClientOverview.jsx** and **ClientsPage.jsx** to allow users to select a client.
- **Example:**

```
<ClientSelector selectedClient={selectedClient} onClientChange={setSelectedClient} />
```

---

## 2. Table.jsx

- **Purpose:**
  - Provides a generic table structure that can be reused across multiple components.
- **Key Features:**
  - Handles empty data (happens during fetching or loading).
  - Displays tabular data in both desktop and mobile view.

## COMPONENTS/DASHBOARD FOLDER

The **/dashboard** folder contains components that are used to display dashboard-related stuff. These components provide an overview of global and client-specific performance metrics, visualizations, and summaries.

---

### Purpose of the /dashboard Folder

1. **Global and Client Performance Overview:**
    - Displays key metrics and performance data for both global and client-specific models.
  2. **Data Visualization:**
    - Provides charts and tables to visualize performance metrics like accuracy, F1 score, loss, precision, and recall.
- 

### Key Components in the /dashboard Folder

#### 1. DashboardStats.jsx

- **Purpose:**
  - Displays a summary of global model performance metrics.

- **Key Features:**
    - Fetches the best global F1 score and related metrics using the **fetchBestF1Global** API function.
- 

## 2. ClientOverview.jsx

- **Purpose:**
    - Displays performance data for a selected client.
  - **Key Features:**
    - Uses **ClientSelector.jsx** to allow users to select a client.
    - Displays performance data in a table (**PerformanceTable.jsx**) and a chart (**PerformanceChart.jsx**).
- 

## 3. PerformanceChart.jsx

- **Purpose:**
    - Visualizes client performance metrics in a line chart.
  - **Key Features:**
    - Uses the recharts library to render a responsive line chart.
    - Allows users to select multiple metrics (e.g., accuracy, F1 score) to display on the chart.
    - Includes a summary of the best round using the **ClientSummary.jsx** component.
- 

## 4. PerformanceTable.jsx

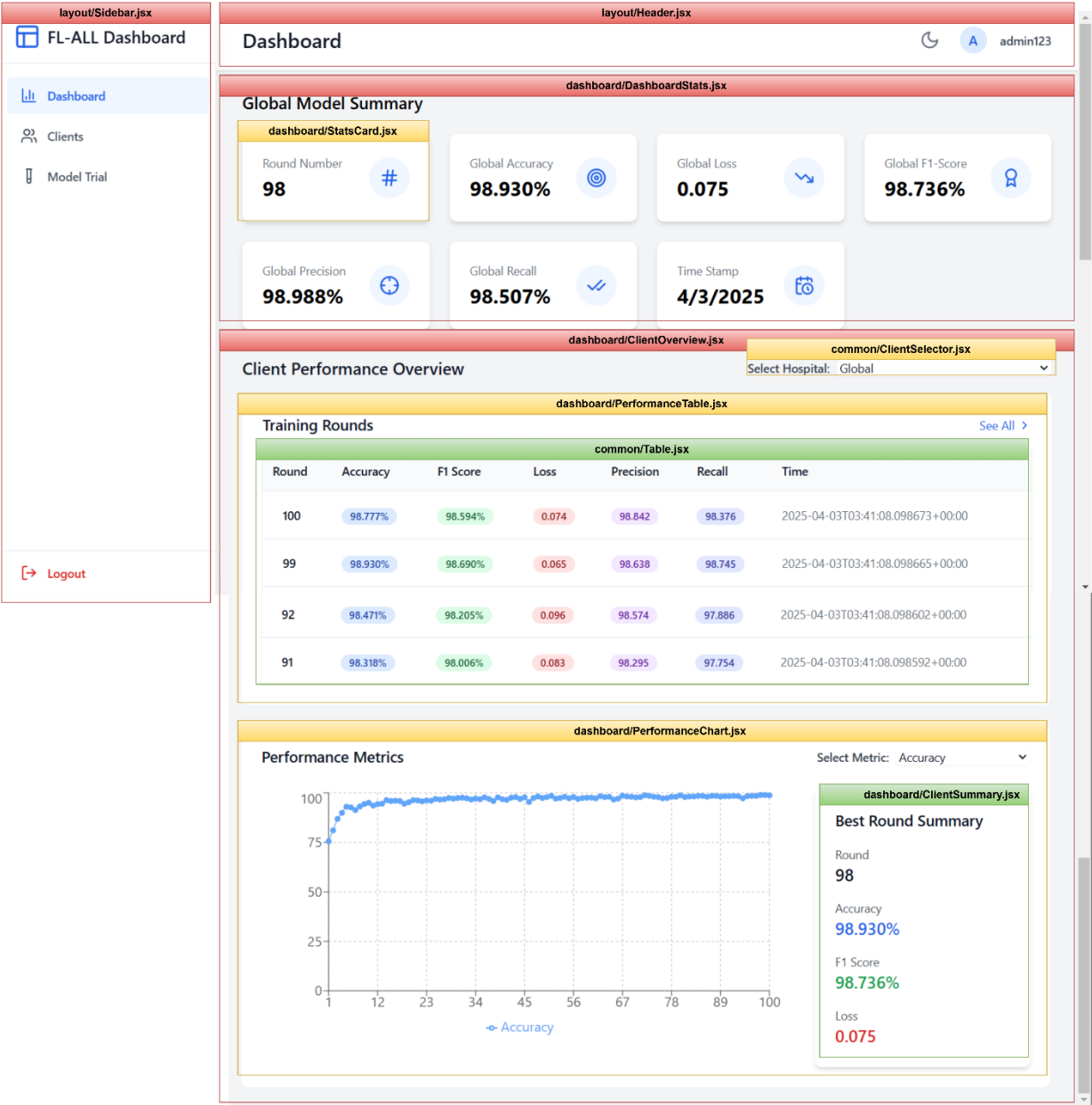
- **Purpose:**
    - Displays client performance data in a tabular format.
  - **Key Features:**
    - Uses the **ResponsiveTable.jsx** component from the common folder to render the table.
- 

## 5. StatsCard.jsx

- **Purpose:**
    - Displays individual performance metrics in a card format.
  - **Key Features:**
    - Shows a title, value, and an icon for each metric. It is used for **DashboardStats.jsx**.
- 

## 6. ClientSummary.jsx

- **Purpose:**
  - Displays a summary of the best round for a selected client.
- **Key Features:**
  - Finds the round with the best F1 score and displays its metrics.



dashboard/PerformanceChart.jsx

Performance Metrics

Select Metric: Accuracy

dashboard/ClientSummary.jsx

Best Round Summary

Round

98

Accuracy

98.930%

F1 Score

98.736%

Loss

0.075

Dashboard

Clients

Model Trial

Logout

clients/ClientsPage.jsx

Client Training Logs

Select Hospital: Global

common/Table.jsx						
Round	Accuracy	F1 Score	Loss	Precision	Recall	Time
100	98.777%	98.594%	0.074	98.842	98.376	2025-04-03T03:41:08.098673+00:00
99	98.930%	98.690%	0.065	98.638	98.745	2025-04-03T03:41:08.098665+00:00
98	98.930%	98.736%	0.075	98.988	98.507	2025-04-03T03:41:08.098658+00:00
97	98.471%	98.290%	0.095	98.235	98.358	2025-04-03T03:41:08.098646+00:00
96	98.471%	98.092%	0.081	98.469	97.765	2025-04-03T03:41:08.098636+00:00
95	98.318%	98.053%	0.102	97.894	98.232	2025-04-03T03:41:08.098625+00:00
94	97.248%	96.639%	0.098	97.502	96.019	2025-04-03T03:41:08.098617+00:00
93	98.318%	98.108%	0.065	98.000	98.232	2025-04-03T03:41:08.098609+00:00
92	98.471%	98.205%	0.096	98.574	97.886	2025-04-03T03:41:08.098602+00:00
91	98.318%	98.006%	0.083	98.295	97.754	2025-04-03T03:41:08.098592+00:00

## Cell Classification Model

Upload microscopic cell images for automated classification



[Browse](#) to choose a file

Supported formats: PNG, JPG, JPEG (max 5MB)

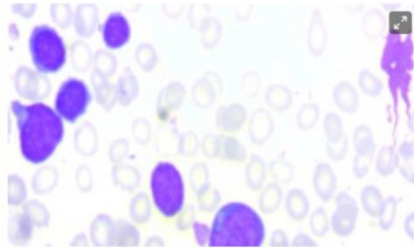
### Analysis History

No analyses performed yet. Upload an image to get started.

Supported formats: PNG, JPG, JPEG (max 5MB)

## model-trial/PredictionResult.jsx

## Analysis Results



### Image Details

File Name: WBC-Benign-001\_test.jpg  
Dimensions: 224 x 224px  
Image Type: JPG  
File Size: 18.78 KB

### Classification

Class

Benign

Confidence Score

100.0%

Class Probabilities	
Benign	100.00%
Early Stage Malignant	0.00%
Premalignant	0.00%
Progressive Malignant	0.00%

Model Date: 2025-04-03 00:41:26.457000

## model-trial/PredictionHistory.jsx

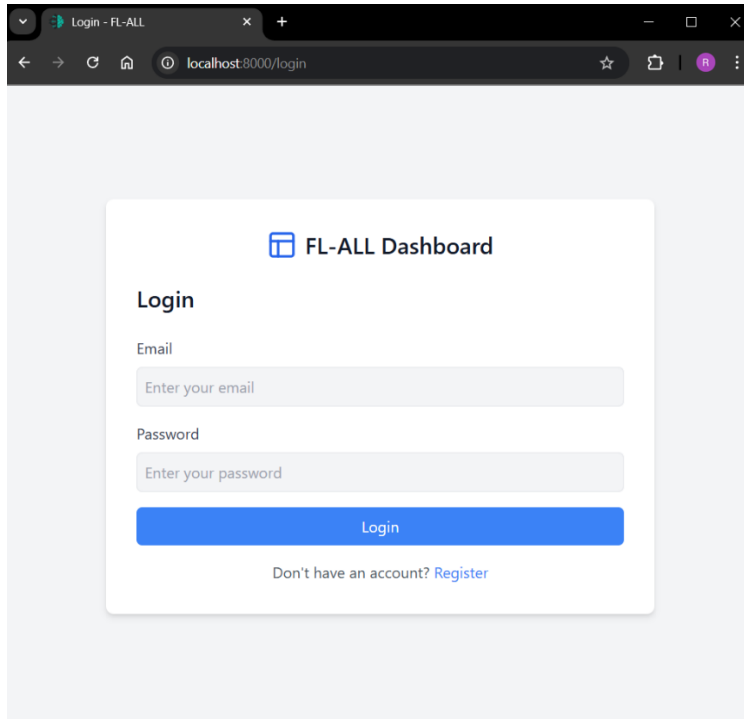
### Analysis History

 Clear History

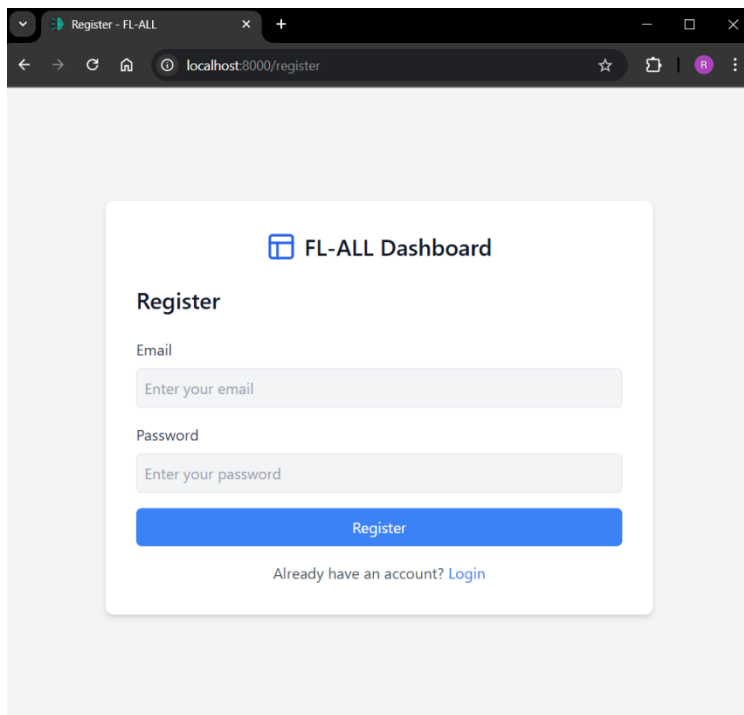


WBC-Benign-001\_test.jpg #1  
Benign  
100.0%  
JPG • 224x224

The login and register pages manage user authentication. The login interface enables existing users to sign in using their email and password, with proper error handling for invalid credentials. Upon successful login, users are redirected based on their role—admins to the Dashboard and clinic users to the Model Trial page. The register interface allows new users to create an account, validating inputs such as duplicate emails or weak passwords. After successful registration, users are redirected to the login page.

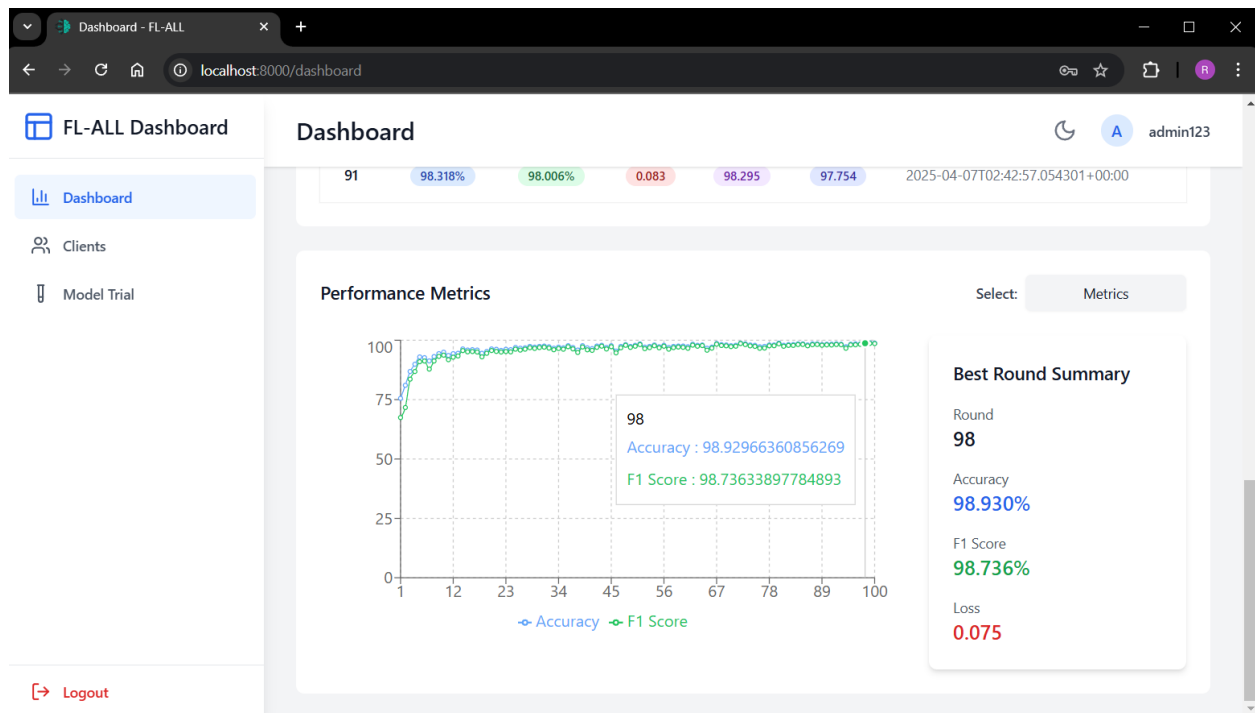


The screenshot shows a web browser window with the address bar displaying 'localhost:8000/login'. The page features a white card with a blue icon and the text 'FL-ALL Dashboard'. Below this, the heading 'Login' is displayed. There are two input fields: 'Email' with the placeholder 'Enter your email' and 'Password' with the placeholder 'Enter your password'. A blue 'Login' button is positioned below the password field. At the bottom of the card, there is a link that says 'Don't have an account? Register'.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/register'. The page features a white card with a blue icon and the text 'FL-ALL Dashboard'. Below this, the heading 'Register' is displayed. There are two input fields: 'Email' with the placeholder 'Enter your email' and 'Password' with the placeholder 'Enter your password'. A blue 'Register' button is positioned below the password field. At the bottom of the card, there is a link that says 'Already have an account? Login'.





The clients page provides in-depth insights into the performance of individual clients. Users can select a client from a dropdown menu and view their training data in a detailed, paginated table.

FL-ALL Dashboard

Dashboard

Clients

Model Trial

Logout

Clients

Client Training Logs

Select Hospital: Global

Round	Accuracy	F1 Score	Loss	Precision	Recall	Time
100	98.777%	98.594%	0.074	98.842	98.376	2025-04-07T02:42:57.054427+00:00
99	98.930%	98.690%	0.065	98.638	98.745	2025-04-07T02:42:57.054421+00:00
98	98.930%	98.736%	0.075	98.988	98.507	2025-04-07T02:42:57.054352+00:00
97	98.471%	98.290%	0.095	98.235	98.358	2025-04-07T02:42:57.054346+00:00
96	98.471%	98.092%	0.081	98.469	97.765	2025-04-07T02:42:57.054340+00:00
95	98.318%	98.053%	0.102	97.894	98.232	2025-04-07T02:42:57.054335+00:00
94	97.248%	96.639%	0.098	97.502	96.019	2025-04-07T02:42:57.054326+00:00
93	98.318%	98.108%	0.065	98.000	98.232	2025-04-07T02:42:57.054318+00:00
92	98.471%	98.205%	0.096	98.574	97.886	2025-04-07T02:42:57.054312+00:00
91	98.318%	98.006%	0.083	98.295	97.754	2025-04-07T02:42:57.054301+00:00

Page 1 of 10

Model Trial - FL-ALL

localhost:8000/model-trial

admin123

FL-ALL Dashboard

Dashboard

Clients

Model Trial

Logout

Model Trial

Cell Classification Model

Upload microscopic cell images for automated classification

Upload icon

Browse to choose a file

Supported formats: PNG, JPG, JPEG (max 5MB)

Analysis Results

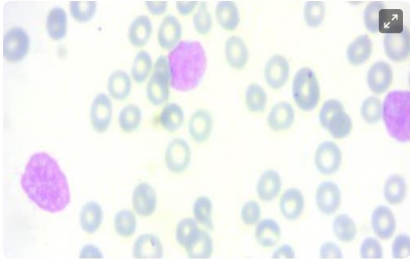


Image Details

File Name: WBC-Malignant-Pro-013.jpg

Dimensions: 224 x 224px

Image Type: JPG

File Size: 16.31 KB

Classification

Class

Progressive Malignant

Confidence Score

100.0%

Class Probabilities

Benign	0.00%
Early Stage Malignant	0.00%
Premalignant	0.00%
Progressive Malignant	100.00%

Model Date: 2025-04-06 23:43:31.594000