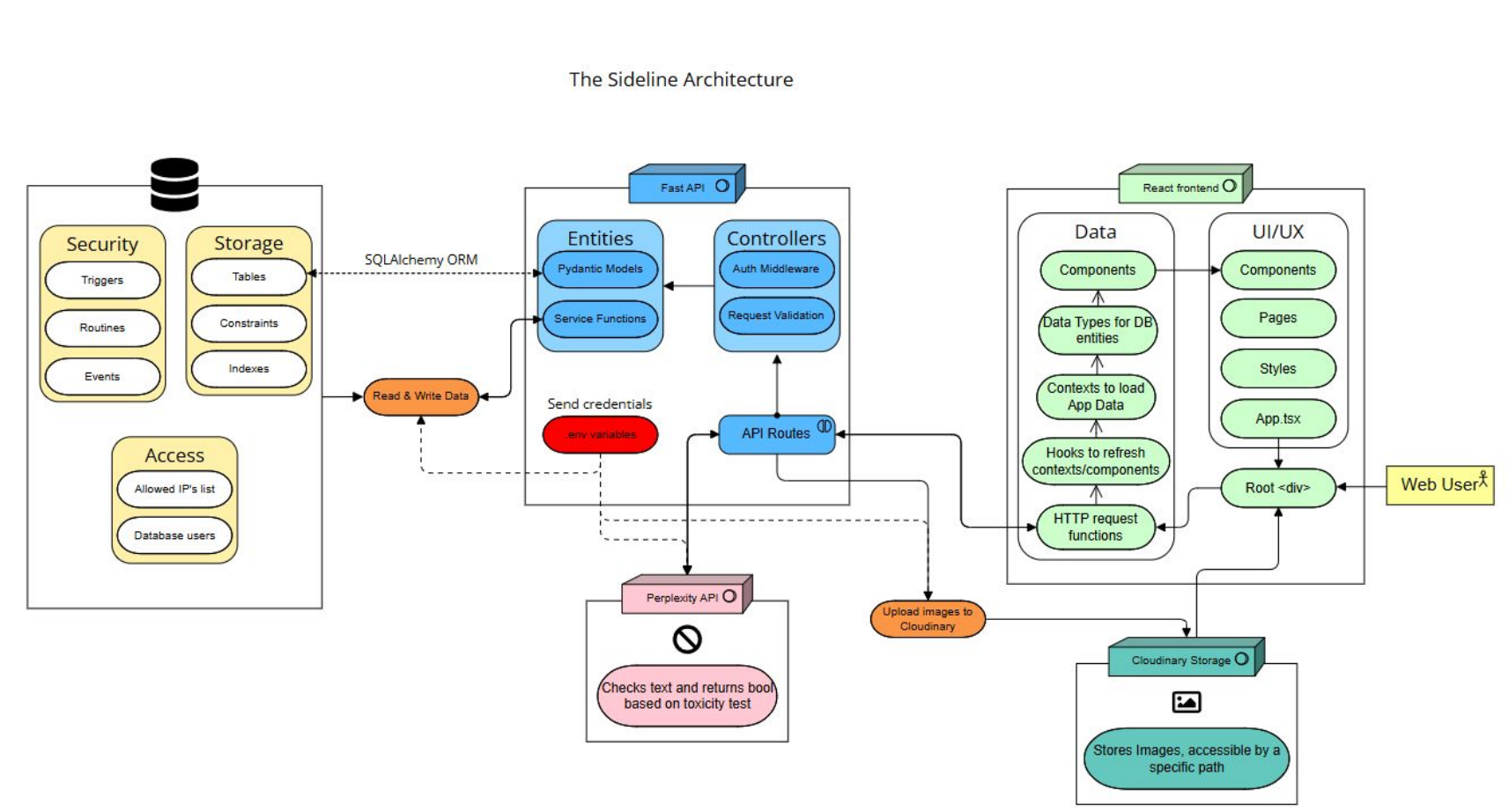
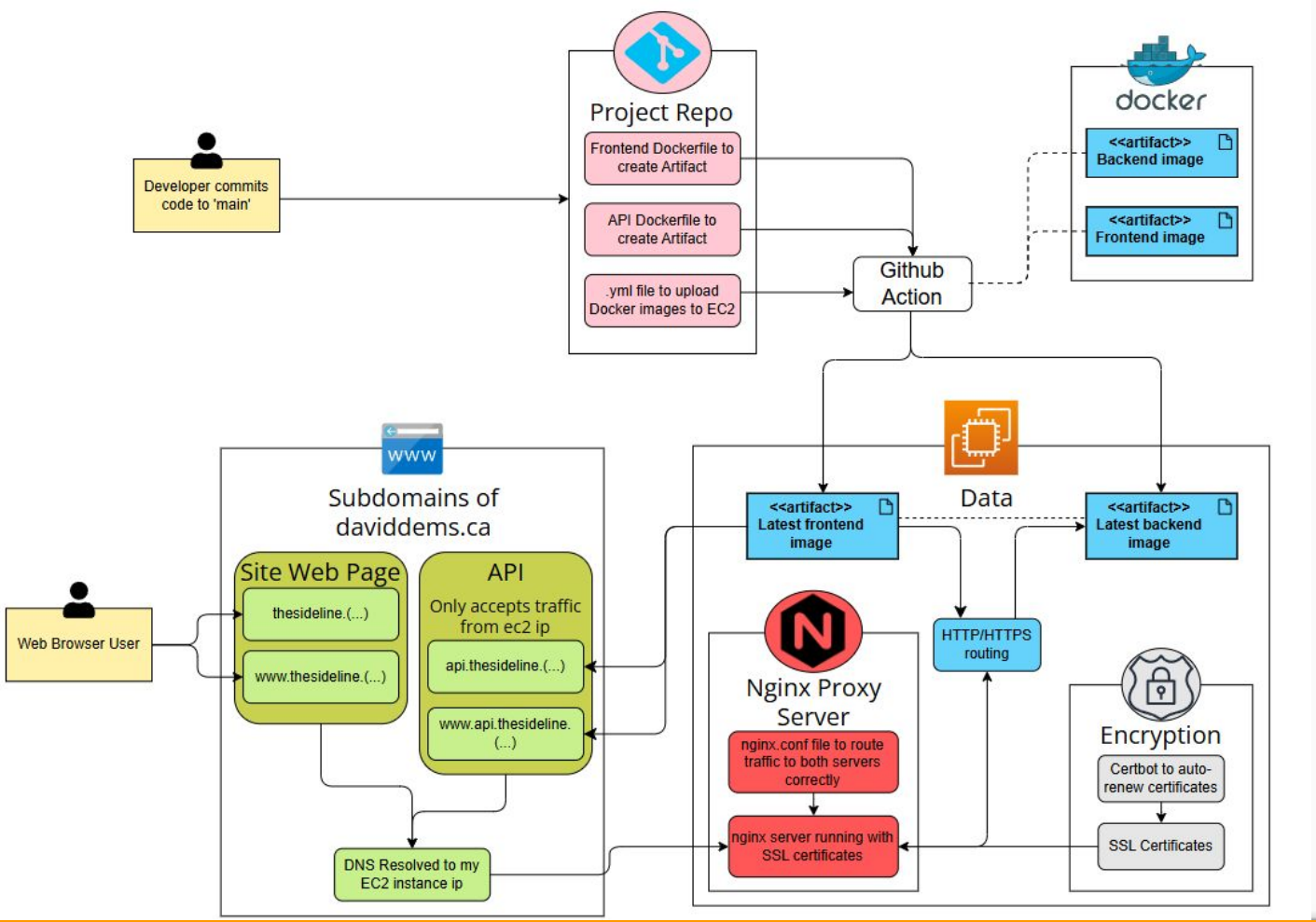
TheSideline

Social Media platform for sports fanatics

Made by: David Demers, Rima Dagher, Louis Chartier


The Sideline Architecture
Security
Triggers
Routines
Events
Storage
Tables
Constraints
Indexes
Access
Allowed IP's list
Database users
SQLAlchemy ORM
Read & Write Data
Fast API
Entities
Pydantic Models
Service Functions
Controllers
Auth Middleware
Request Validation
Send credentials
.env variables
API Routes
Perplexity API
Checks text and returns bool based on toxicity test
Upload images to Cloudinary
Cloudinary Storage
Stores Images, accessible by a specific path
React frontend
Data
Components
Data Types for DB entities
Contexts to load App Data
Hooks to refresh contexts/components
HTTP request functions
UI/UX
Components
Pages
Styles
App.tsx
Root
Web User



Deployment Architecture

Architecture & Design Patterns

Architecture Style:

Pattern: Layered Architecture

Structure: Controller -> Service -> Model

Framework: FastAPI with SQLAlchemy ORM

Database: MySQL with connection pooling

Architecture & Design Patterns

Key Designs Pattern:

Separation of Concerns: Controller, service, and Model for each feature

DTO with Pydatinc: Pydantic Schemas for request response validation

Dependency injection: FastAPI's build-in DI for database session and auth

Strategy Pattern: Different image transformations based on image type

Observer Pattern: Notification system for user interactions

```python
def get_current_user(db: Session = Depends(get_db), token: str = Depends(oauth2_scheme)) -> User
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[JWT_ALGORITHM]) # decode token
        user_id = int(payload.get("sub"))  #  decode user_id
    except (JWTError, ValueError):
        logger.error(f"Token decode error: {JWTError}")
        raise HTTPException(status_code=401, detail="Invalid token")

    user = db.query(User).filter(User.id == user_id).first()
    if not user:
        logger.error(f"User not found: {user_id}")
        raise HTTPException(status_code=404, detail="User not found")
    logger.info(f"Current user retrieved: {user.email}")
    return user
```

```python
@router.post("/change-password")
def change_password_endpoint(
    change_data: ChangePasswordRequest,
    db: Session = Depends(get_db),
    current_user: User = Depends(get_current_user),   # auto resolves from token
):
```

Backend Structure

```
backend/src/
├── main.py              # Application entry point
├── api.py               # Route registration
├── database/
│   └── core.py          # Database configuration & session management
├── auth/                # Authentication & authorization
├── users/               # User management
├── posts/               # Post creation & management
├── images/              # Image upload & storage (Cloudinary)
├── likes/               # Like functionality
├── reposts/             # Repost/share functionality
├── replies/             # Comment/reply system
├── followers/           # Social following system
├── notifications/       # Real-time notifications
├── feed/                # Timeline & content aggregation
├── filtering/           # Content moderation (toxicity detection)
├── preferences/         # User settings
├── admins/              # Admin functionality
└── health.py            # Health check endpoints
```

🔒 Authentication & Security Architecture

JWT Implementation

```
# Token-based authentication with configurable expiry
JWT_SECRET = os.getenv("JWT_SECRET")
JWT_ALGORITHM = "HS256"
JWT_EXPIRE_MINUTES = 30

# OAuth2 scheme for automatic token extraction
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/login")
```

🔒 Authentication & Security Architecture

Security Features:

- Password Hashing: bcrypt with salt
- Token Validation: JWT with automatic expiry
- Route Protection: Dependency injection for auth requirements
- Input validation: Comprehensive Pydatinc validation
- SQL injection Prevention: ORM-based query building

Authorization Levels:

- Public: Guest access (feed viewing)
- Authenticated: Basic user operations
- Active User: Excludes soft-deleted accounts

Image Management System

Cloudinary Integration

Smart Transformations: Type-specific image processing:

- Avatars: 200*200 crop with face detection
- Posts: 800*600 limit maintaining aspect ratio

Optimization: Automatic quality/format optimization

CDN Delivery: Global content delivery network

Image Storage Pattern

```
# Organized folder structure
avatars: "chirpsocial_v2/avatars/"
posts: "chirpsocial_v2/posts/"

# Database references
public_id: Cloudinary identifier
image_path: Direct CDN URL
```

📡 API Design & Patterns

RESTful Design

- Resource-Based URLs
- HTTP Methods
- Status Codes
- Pagination

Error Handling

- HTTPException
- Logging
- Validation

```python
class UserPublic(BaseModel): # Public profile vie
    id: int
    username: str
    email: EmailStr
    first_name: str | None = None
    last_name: str | None = None
    private: bool
    bio: str | None = None
    date_of_birth: date | None = None
    joined: datetime | None = None
    avatar_image_id: int | None = None
    avatar_url: Optional[str] = None  # URL to ac
    deleted: bool = False
    deleted_at: Optional[datetime] = None

    class Config:
        from_attributes = True
```

```python
class UserUpdate(BaseModel):
    username: Optional[str] = Field(None, min_length=3,
    bio: Optional[str] = Field(None, max_length=160)
```

```python
class UserCreate(BaseModel): # For user registration
    username: str = Field(..., min_length=3, max_length=255)
    email: EmailStr
    password: str = Field(..., min_length=8)
    first_name: str | None = Field(None, max_length=255)
    last_name: str | None = Field(None, max_length=255)
    private: bool = True
    date_of_birth: date | None = None
    bio: str | None = Field(None, max_length=160)
```

API Advanced Features

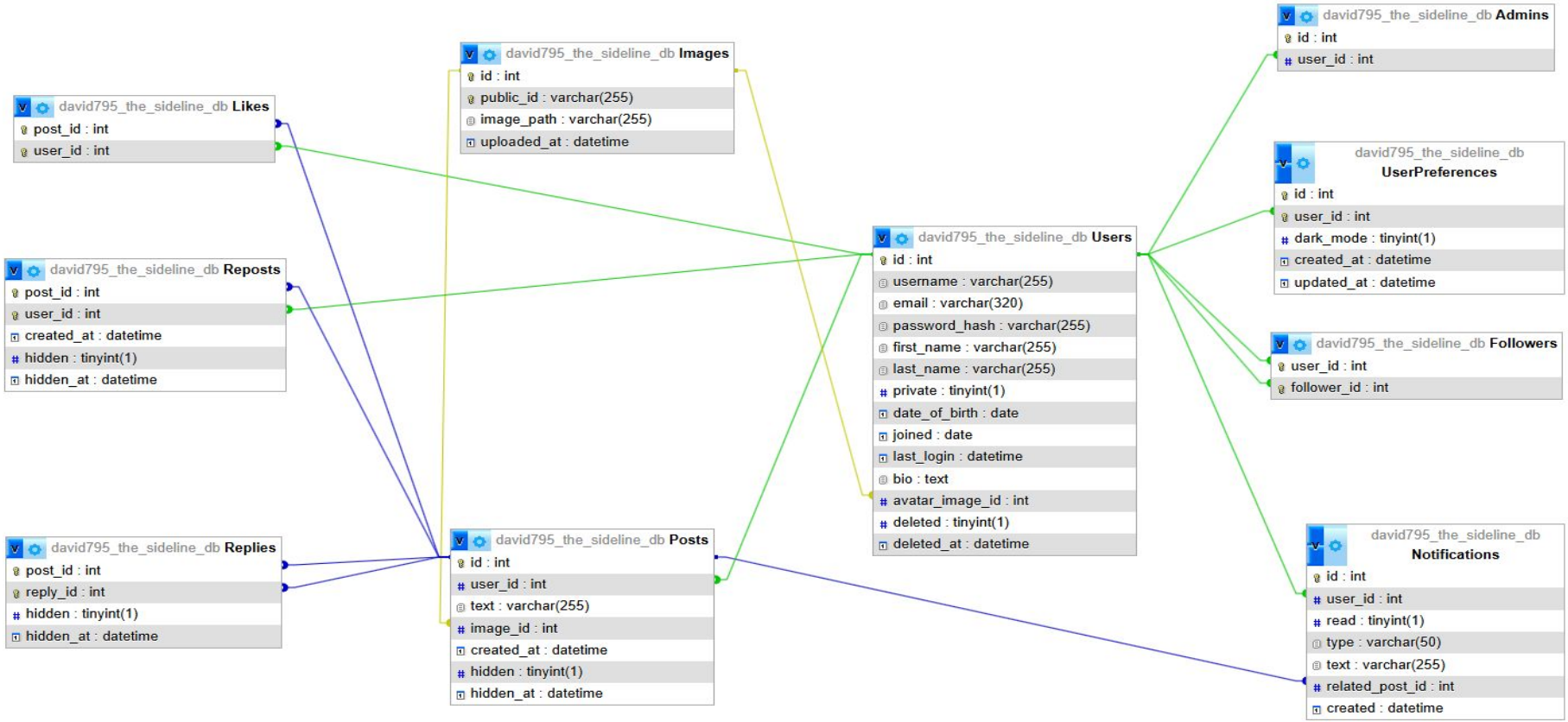
Content Moderation

- Toxicity Detection: External API integration for content filtering (Toxicity score)
- Admin Controls: Hidden content management
- Soft Deletes: Recoverable data deletion

Social Features

- Feed Algorithm: Mixed content(posts + reposts) with timestamp sorting
- Notification System: Event driven notifications for user interactions
- Reply Threading: Hierarchical comment system

Entity Relationship Diagram



Frontend Structure

```
src/
├── api/                    # Axios instance for backend API
├── assets/                 # Static assets (images, icons, etc.)
│
├── components/
│   ├── auth/               # Login, Register, Change Password
│   ├── posts/              # PostCard, ThreadModal, CreatePostModal
│   ├── search/             # Search bar & result components
│   ├── ui/                 # Reusable UI (buttons, modals, etc.)
│   └── users/              # Profile, EditProfile, Avatar handling
│
├── context/
│   ├── AppDataProvider.tsx        # Global app state
│   └── NotificationsProvider.tsx  # Notifications state mgmt
│
├── hooks/
│   ├── useAuthQuery.ts     # Auth state (token validation)
│   ├── useAvatar.ts        # Fetch user avatars
│   ├── useImageUpload.ts   # Cloudinary upload handling
│   ├── useNotifications.ts# Notifications (local + API)
│   ├── useReplyCount.ts    # Reply counts per post
│   ├── useUser.ts          # Fetch single user
│   └── useUserPosts.ts     # Fetch user posts
│
├── pages/
│   ├── HomePage.tsx        # Feed + threads
│   ├── LoginPage.tsx       # Login form
│   ├── RegisterPage.tsx    # Register form
│   ├── ProfilePage.tsx     # User profiles
│   └── NotificationsPage.tsx # Notification center
│
├── types/                  # Shared TypeScript interfaces (Post, User, Notification)
│
├── App.tsx                 # App shell, routes, layout
├── index.css               # Global styles
└── main.tsx                # React entry point
```

Architecture Principles

- **Component driven development** (small, reusable building blocks)
- **Custom hooks:** abstract backend call + local state)
- **Context providers:** global app state (auth, notifications)
- **Page level components:** connect hooks + UI
- **Modal popup components:** quick and easy interactivity
- **Strong typing:** with TypeScript for safer API usage

Core Features

- **Feed (Homepage):** lists all posts from API
- **ProfilePage:** self + other users profiles with tabs(Posts, Reposts, Likes, Replies)
- **Post system:** (create, view, like, repost, reply with threaded modal)
- **NotificationsPage:** like/reply/repost/follow alerts with icons & read/unread
- **Auth (Login / Register):** token storage + context-based auth
- **Search:** find users
- **Image upload:** Cloudinary integration
- **Avatars:** dynamic per user

Feed

- Feed from Posts in cache loaded with AppDataProvider.
- Loads recent posts for home page and profile posts if logged in.
- Guest & Auth feeds.
- Allows new users to get a feel for the website.

```
// Preload main feed data when user logs in (but not for deleted users)
const { data: feedData, isLoading: feedLoading, error: feedError } = useQuery({
  queryKey: ["feed", isAuthenticated ? "user" : "guest"],
  queryFn: async () => {
    return isAuthenticated
      ? await postsApi.getUserFeed()
      : await postsApi.getGuestFeed();
  },
  staleTime: 0,
  enabled: !authLoading && !isDeleted, // Don't fetch feed for deleted users
  refetchOnWindowFocus: false, // Prevent unnecessary refetches
  refetchOnMount: true, // Always refetch when component mounts
  refetchOnReconnect: true, // Refetch when reconnecting
});

// Force refetch feed when authentication state changes
useEffect(() => {
  if (!authLoading) {
    // Force refetch the feed when auth state changes
    queryClient.refetchQueries({ queryKey: ["feed"] });
  }
}, [isAuthenticated, authLoading, queryClient]);
```

Notification System

- **Stored & managed via NotificationsProvider**
- **Custom hook useNotifications:**
  - Fetch all notifications
  - Mark as read / mark all as read
  - Delete single or clear all
- **UI:**
  - Sidebar badge with unread count
  - Icons + color coded types (likes, reply, repost, follow)
  - Clicking opens post thread or user profile

```typescript
export function useNotifications() {

    const [readIds, setReadIds] = useState<Set<number>>(new Set());

    const { data, isLoading, error } = useQuery<Notification[]>({
        queryKey: ["notifications"],
        queryFn: async () => {
        const res = await api.get("/notifications/");
        return res.data as Notification[];
        },
    });

    const markAsReadLocal = (id: number) => {
        setReadIds((prev) => new Set(prev).add(id));
    };

    const markAllAsReadLocal = () => {
        if (data) {
        const allIds = new Set(data.map((n) => n.id));
        setReadIds(allIds);
        }
    };
```

Posts & Threads

- **PostCard:** username, avatar, timestamp, actions (like, comment, repost)
- **ThreadModal:** open post in a modal with replies
- **Reply flow:** reply creates nested posts inside the thread with a link back to the replied post
- **Reusable PostCard type:** ensures consistency across Feed, Profile, Notifications

UI/UX Design

- **TailwindCSS:** utility first responsive design
- **Lucide icons:** modern UI feedback
- **Sticky sidebars:** nav & logout always visible
- **Color coded notifications:**
  - ❤️ **Like = red**
  - 💬 **Reply = orange**
  - 🔁 **Repost = blue**
  - ➕ **Follower = yellow**
- **Modal based flows:** no hard reloads, smoother UX

Tasks & Roles

- Database design and creation: David
- Backend scaffold & layered architecture(FastAPI setup, Logging setup, separation of concerns): Rima
- FastAPI Users and Authorization: Rima
- FastAPI Cloudinary configuration(image storage & API endpoints): Rima
- FastAPI Posts related entities (posts, replies, likes, ...): David
- UI/UX designs (react pages): Louis
- React components: Everyone
- Hooks: Louis
- Data contexts: Louis & Louis
- Frontend centralized image upload feat (avatar & posts via hooks): Rima
- Initial deployment and CI/CD with Github actions: David

Problems Encountered

- Designing too many triggers, actions should be done by API
- Did not perfectly normalize database on initial design
- Deploying at the end instead of from the start with CD
- Configuring EC2’s accepted ips for Github Actions
- Centralized Image upload architecture (Eliminated code duplication, provided type-safe image uploads, and created a maintainable system that scales across the application)

Average Time Spent

David: 6-7 Hours

Rima:5-6 Hours

Louis: 5-6 Hours

Future

Posts Features

- Create a sports results database
- Create a sports objects API to serve and interact to the frontend
- Create components to view sports data in the website
- Change existing components to be able to interact with sports data, like reference a stat line from a game and have its link in the post.
- Hashtags to help search filters and feed

User Features

- Posts can be reported
- Users can edit posts
- Users can go private and require follow confirmation

System Features

- Run google ads every 10 posts on the feed.
- Setup ip specific API request limitations.
- Setup elastic load-balancing for our AWS EC2 instances