Wildlife Detection System - Technical Documentation

Table of Contents

1. Project Overview
2. System Architecture
3. Data Models
4. Backend Components
    - API Routes
    - Services
    - Utils
5. Frontend Components
    - Web Interfaces
    - JavaScript Clients
6. Machine Learning Components
7. Analysis Tools
8. Core Features
9. Installation & Setup
10. Usage Guide

Project Overview

The Wildlife Detection System is a comprehensive platform for managing, annotating, and analyzing camera trap images for wildlife research. Designed to meet the requirements specified by Prof. Peeva, the system focuses on:

- Wildlife species detection and classification
- Environmental and behavioral data collection
- Diurnal activity analysis
- Predator-prey relationship tracking
- Machine learning model training for automated detection

The system provides a complete workflow from image collection to annotation, analysis, and machine learning model training, allowing researchers to process large volumes of camera trap images efficiently and extract valuable ecological insights.

System Architecture

The Wildlife Detection System follows a modern web application architecture with a Flask-based backend, SQLite database, and browser-based frontend. The system is structured into several key components:

```
WildlifeDetectionSystem/
├── api/                    # API server and backend logic
│   ├── app/                # Main application code
│   │   ├── models/         # Database models
│   │   ├── routes/         # API endpoints
│   │   ├── services/       # Business logic
│   │   ├── static/         # Static assets (JS, CSS)
│   │   ├── templates/      # HTML templates
│   │   ├── utils/          # Utility functions
│   │   └── __init__.py     # Application factory
│   ├── debug/              # Debugging tools
│   ├── instance/           # Instance-specific data (database)
│   ├── scripts/            # Utility scripts
│   ├── .env                # Environment variables
│   ├── config.py           # Configuration
│   └── run.py              # Application entry point
├── data/                   # Data storage
│   ├── export/             # Exported datasets (COCO, YOLO)
│   ├── processed_images/   # Processed images
│   └── raw_images/         # Original camera trap images
├── models/                 # Machine learning models
│   ├── configs/            # Model configurations
│   └── trained/            # Trained model weights
├── notebooks/              # Jupyter notebooks
│   └── training/           # Training notebooks
├── output/                 # Output data and results
│   └── inference_results/  # Inference results
├── reports/                # Generated reports
└── scripts/                # Top-level scripts
    └── setup.sh            # Setup script
```

The architecture follows a layered design pattern:

1. **Presentation Layer**: HTML templates and JavaScript for user interfaces
2. **API Layer**: Flask routes that handle HTTP requests and responses
3. **Service Layer**: Business logic that implements core functionality
4. **Data Access Layer**: Database models and queries
5. **Persistence Layer**: SQLite database for data storage

Data Models

The system uses several interrelated data models to represent wildlife observations and classifications:

Image Model

The `Image` model represents camera trap images uploaded to the system:

```python
class Image(db.Model):
    """Model representing an uploaded camera trap image."""
    id = db.Column(db.Integer, primary_key=True)
    filename = db.Column(db.String(255), nullable=False, unique=True)
    original_path = db.Column(db.String(512), nullable=False)
    processed_path = db.Column(db.String(512), nullable=True)
    upload_date = db.Column(db.DateTime, default=datetime.utcnow)
    width = db.Column(db.Integer, nullable=True)
    height = db.Column(db.Integer, nullable=True)

    # Metadata
    location = db.Column(db.String(255), nullable=True)
    timestamp = db.Column(db.DateTime, nullable=True)
    camera_id = db.Column(db.String(100), nullable=True)

    # Relationships
    annotations = db.relationship('Annotation', backref='image', lazy=True, cascade="a
    environmental_data = db.relationship('EnvironmentalData', backref='image', lazy=Tr
```

Species Model

The `Species` model represents wildlife species for classification:

```python
class Species(db.Model):
    """Model representing wildlife species for classification."""
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False, unique=True)
    scientific_name = db.Column(db.String(255), nullable=True)
    description = db.Column(db.Text, nullable=True)

    # Relationships
    annotations = db.relationship('Annotation', backref='species', lazy=True)
    sequence_events = db.relationship('SequenceEvent', backref='species', lazy=True)
```

Annotation Model

The `Annotation` model represents bounding box annotations on images:

python

```python
class Annotation(db.Model):
    """Model representing a bounding box annotation for an image."""
    id = db.Column(db.Integer, primary_key=True)
    image_id = db.Column(db.Integer, db.ForeignKey('image.id'), nullable=False)
    species_id = db.Column(db.Integer, db.ForeignKey('species.id'), nullable=False)

    # Bounding box coordinates (normalized 0-1)
    x_min = db.Column(db.Float, nullable=False)
    y_min = db.Column(db.Float, nullable=False)
    x_max = db.Column(db.Float, nullable=False)
    y_max = db.Column(db.Float, nullable=False)

    # Metadata
    confidence = db.Column(db.Float, nullable=True)  # For model predictions
    is_verified = db.Column(db.Boolean, default=False)  # Human verification flag
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utc

    # Relationships
    behavioral_notes = db.relationship('BehavioralNote', backref='annotation', lazy=Tr
```

EnvironmentalData Model

The `EnvironmentalData` model captures environmental context for diurnal activity analysis:

python

```python
class EnvironmentalData(db.Model):
    """Environmental data for diurnal activity analysis as per Prof. Peeva's requireme
    id = db.Column(db.Integer, primary_key=True)
    image_id = db.Column(db.Integer, db.ForeignKey('image.id'), nullable=False, unique

    # Light conditions
    light_condition = db.Column(db.String(50), nullable=True)  # Full darkness, Early

    # Environmental factors
    temperature = db.Column(db.Float, nullable=True)  # Temperature in Celsius
    moon_phase = db.Column(db.String(50), nullable=True)  # Moon phase (Full, New, etc
    snow_cover = db.Column(db.Boolean, nullable=True)  # Snow cover present

    # Habitat data
    vegetation_type = db.Column(db.String(100), nullable=True)  # Type of vegetation
    habitat_type = db.Column(db.String(100), nullable=True)  # Plains or mountains (St
```

BehavioralNote Model

The `BehavioralNote` model stores observations about animal behavior:

```python
class BehavioralNote(db.Model):
    """Notes about animal behavior for behavioral tracking."""
    id = db.Column(db.Integer, primary_key=True)
    annotation_id = db.Column(db.Integer, db.ForeignKey('annotation.id'), nullable=Fal

    behavior_type = db.Column(db.String(100), nullable=False)  # Behavior category
    notes = db.Column(db.Text, nullable=True)  # Detailed notes
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

SequenceEvent Model

The `SequenceEvent` model tracks chronological appearances for predator-prey analysis:

```python
class SequenceEvent(db.Model):
    """Tracks chronological appearances of animals for predator-prey analysis."""
    id = db.Column(db.Integer, primary_key=True)
    location = db.Column(db.String(255), nullable=False)  # Camera location
    species_id = db.Column(db.Integer, db.ForeignKey('species.id'), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False)  # When animal was detected

    # Sequence tracking
    previous_event_id = db.Column(db.Integer, db.ForeignKey('sequence_event.id'), null
    time_since_previous = db.Column(db.Integer, nullable=True)  # Time in seconds

    # Self-referential relationship
    next_events = db.relationship('SequenceEvent',
                                  backref=db.backref('previous_event', remote_side=[id]
                                  foreign_keys=[previous_event_id])
```

Entity Relationship Diagram

The relationships between these models are organized as follows:

1. An `Image` can have multiple `Annotation`s (one-to-many)
2. An `Image` has one `EnvironmentalData` record (one-to-one)
3. A `Species` can be referenced by multiple `Annotation`s (one-to-many)
4. An `Annotation` can have multiple `BehavioralNote`s (one-to-many)

5. A `Species` can be associated with multiple `SequenceEvent`s (one-to-many)
6. A `SequenceEvent` can reference another `SequenceEvent` as its predecessor (self-referential)

Backend Components

API Routes

The system implements several Flask blueprints to organize API routes:

Main Routes (`main.py`)

General routes and system information:

- `GET /`: Main dashboard
- `GET /api`: API root endpoint
- `GET /health`: Health check endpoint
- `GET /system-info`: System diagnostic information

Image Routes (`images.py`)

Routes for image management:

- `GET /api/images/`: List all images
- `POST /api/images/`: Upload a new image
- `GET /api/images/<id>`: Get a specific image metadata
- `GET /api/images/<id>/file`: Get the image file
- `GET /api/images/folders`: Get list of image folders
- `GET /api/images/annotated`: Get annotated images
- `GET /api/images/unannotated`: Get unannotated images
- `POST /api/images/<id>/no-animals`: Mark an image as having no animals
- `POST /api/images/index-existing`: Index existing images from raw_images directory

Species Routes (`species.py`)

Routes for species management:

- `GET /api/species/`: List all species
- `GET /api/species/<id>`: Get a specific species
- `POST /api/species/`: Create a new species
- `PUT /api/species/<id>`: Update a species

- `DELETE /api/species/<id>`: Delete a species
- `POST /api/species/initialize`: Initialize default species

Annotation Routes (`annotations.py`)

Routes for annotation management:

- `GET /api/annotations/image/<id>`: Get annotations for an image
- `GET /api/annotations/<id>`: Get a specific annotation
- `POST /api/annotations/`: Create a new annotation
- `POST /api/annotations/batch`: Create multiple annotations
- `PUT /api/annotations/<id>`: Update an annotation
- `DELETE /api/annotations/<id>`: Delete an annotation
- `GET /api/annotations/export`: Export annotations in COCO format
- `GET /api/annotations/export/yolo`: Export annotations in YOLO format
- `GET /api/annotations/export/both`: Export in both formats
- `GET /api/annotations/datasets`: Get list of annotated datasets
- `GET /api/annotations/exports`: Get list of existing exports

Environmental Routes (`environmental_routes.py`)

Routes for environmental data:

- `GET /api/environmental/image/<id>`: Get environmental data for an image
- `POST/PUT /api/environmental/image/<id>`: Create or update environmental data
- `DELETE /api/environmental/image/<id>`: Delete environmental data
- `POST /api/environmental/behavior/annotation/<id>`: Add behavioral note
- `GET /api/environmental/behavior/annotation/<id>`: Get behavioral notes
- `DELETE /api/environmental/behavior/note/<id>`: Delete behavioral note
- `POST /api/environmental/sequence`: Record sequence event
- `GET /api/environmental/sequence`: Get sequence events
- `GET /api/environmental/sequence/analyze/<location>`: Analyze predator-prey patterns

System Routes (`system.py`)

Routes for system statistics:

- `GET /api/system/stats`: Get comprehensive system statistics

Jupyter Routes (`jupyter_routes.py`)

Routes for Jupyter notebook integration:

- `GET /api/jupyter/start`: Start Jupyter Lab
- `GET /api/jupyter/status`: Check Jupyter status

Report Routes (`report_routes.py`)

Routes for report generation:

- `POST /api/generate-report/`: Generate a report for an export

Static Routes (`static_routes.py`)

Routes for serving static files and pages:

- `GET /advanced-annotator`: Serve advanced annotator page
- `GET /environmental-editor`: Serve environmental editor page
- `GET /static/<path>`: Serve static files

Services

The system implements several service classes to encapsulate business logic:

AnnotationService (`annotation_service.py`)

Services for annotation manipulation:

- `get_annotations_by_image_id(image_id)`: Get annotations for an image
- `get_annotation_by_id(annotation_id)`: Get an annotation by ID
- `create_annotation(...)`: Create a new annotation
- `create_batch_annotations(...)`: Create multiple annotations
- `update_annotation(...)`: Update an annotation
- `delete_annotation(annotation_id)`: Delete an annotation
- `get_annotated_datasets()`: Get list of annotated datasets
- `export_coco_format(output_dir)`: Export annotations in COCO format
- `export_yolo_format(output_dir, val_split)`: Export annotations in YOLO format
- `get_dataset_stats(dataset_name)`: Get statistics for a dataset

EnvironmentalService (`environmental_service.py`)

Services for environmental data:

- `get_environmental_data(image_id)`: Get environmental data for an image
- `create_or_update_environmental_data(...)`: Create or update environmental data
- `delete_environmental_data(image_id)`: Delete environmental data
- `analyze_light_conditions(image_path)`: Analyze image light conditions

BehavioralTrackingService (`environmental_service.py`)

Services for behavioral tracking:

- `add_behavioral_note(...)`: Add a behavioral note
- `get_behavioral_notes(annotation_id)`: Get behavioral notes
- `delete_behavioral_note(note_id)`: Delete a behavioral note
- `record_sequence_event(...)`: Record a sequence event
- `get_sequence_events(...)`: Get sequence events
- `analyze_predator_prey_patterns(...)`: Analyze predator-prey patterns

ImageService (`image_service.py`)

Services for image management:

- `allowed_file(filename)`: Check if file has allowed extension
- `save_uploaded_file(file, location, camera_id)`: Save uploaded file
- `get_all_images(...)`: Get all images with filtering
- `get_annotated_images(...)`: Get annotated images
- `get_unannotated_images(...)`: Get unannotated images
- `get_image_by_id(image_id)`: Get an image by ID

SpeciesService (`species_service.py`)

Services for species management:

- `get_all_species()`: Get all species
- `get_species_by_id(species_id)`: Get a species by ID
- `get_species_by_name(name)`: Get a species by name
- `create_species(...)`: Create a new species
- `update_species(...)`: Update a species
- `delete_species(species_id)`: Delete a species

- `initialize_default_species()`: Initialize default species

Utils

Utility functions for common operations:

ImageUtils (`image_utils.py`)

Utility functions for image processing:

- `read_image(file_path)`: Read an image file
- `save_image(image, output_path)`: Save an image file
- `get_image_dimensions(file_path)`: Get image dimensions
- `extract_image_metadata(file_path)`: Extract metadata from image EXIF tags

Frontend Components

Web Interfaces

The system provides several web interfaces for user interaction:

Dashboard (`dashboard.html`)

The main dashboard provides an overview of the system and access to its features:

- System statistics (images, annotations, species)
- Links to annotation tools, export tools, and ML workbench
- Progress indicators for annotation completion
- Admin tools access

Advanced Annotator (`advanced-annotator.html`)

A sophisticated interface for annotating wildlife in camera trap images:

- Drawing bounding boxes around animals
- Selecting species for annotations
- Navigating through images
- Filtering images by folder and annotation status
- Batch operations
- Keyboard shortcuts for efficient annotation

Environmental Editor (`environmental-editor.html`)

Interface for editing environmental data for images:

- Setting light conditions (darkness, twilight, daylight)
- Recording temperature, moon phase, and snow cover
- Setting habitat and vegetation types
- Auto-detection of environmental factors based on filenames
- Batch operations

JavaScript Clients

Annotator API Client (`annotator-api-client.js`)

Client for interacting with the annotation API:

- `fetchSpecies()`: Fetch all species
- `fetchFolders()`: Fetch all folders
- `fetchImages(...)`: Fetch images with pagination
- `fetchAnnotations(imageId)`: Fetch annotations for an image
- `saveAnnotations(imageId, annotations)`: Save batch annotations
- `markNoAnimals(imageId)`: Mark an image as having no animals
- `exportAnnotations(format)`: Export annotations

Machine Learning Components

The system includes machine learning components for automated wildlife detection:

Training Notebooks

Jupyter notebooks for model training:

- `wildlife_model.ipynb`: Notebook for training wildlife detection models
- `training.ipynb`: General training and evaluation notebook

Model Structure

The system uses YOLOv8 models for object detection:

- `yolov8n.pt`: Nano model (smallest, fastest)
- `yolov8s.pt`: Small model (balanced)
- `yolov8m.pt`: Medium model (more accurate)

Training Process

The training process follows these steps:

1. Export annotations in YOLO format
2. Split data into training and validation sets
3. Configure training parameters
4. Train YOLOv8 model
5. Evaluate model performance
6. Save trained weights

Inference

The trained models can be used for automated detection:

1. Load trained weights
2. Process new images
3. Filter detections by confidence threshold
4. Convert detections to annotations
5. Save annotations to database

Analysis Tools

The system provides several scripts for analyzing wildlife data:

Seasonal Analysis (`seasonal_analysis.py`)

Analyzes seasonal activity patterns of wildlife:

- Extracting seasonal information from image metadata
- Analyzing species activity by season
- Analyzing activity by light conditions
- Generating charts and visualizations

Animal Similarity Analysis (`animal_similarity_analysis.py`)

Analyzes similarities between species like wolf and jackal:

- Extracting features from annotations
- Calculating color histograms
- Extracting shape features
- Comparing feature differences
- Identifying distinguishing features

Bulk Annotation Helper (`bulk_annotation_helper.py`)

Helps with efficiently annotating large batches of images:

- Listing unannotated images
- Creating batch annotations
- Marking images with no animals
- Analyzing annotation progress

Export Tool (`export_tool.py`)

Monitors and manages export processes:

- Checking server connection
- Getting datasets and exports info
- Analyzing export details
- Generating reports

Export Report Generator (`export_report.py`)

Generates detailed reports for exports:

- Creating text reports with dataset statistics
- Generating PDF reports
- Analyzing class distribution
- Providing format-specific information

Update Environmental Data (`update_environmental_data.py`)

Updates environmental data for existing annotations:

- Extracting date information from filenames
- Determining light conditions
- Determining habitat and vegetation types
- Estimating temperature and snow cover

Core Features

The Wildlife Detection System provides several core features:

Image Management

- Upload and index camera trap images
- Organize images into folders
- View image metadata

- Filter images by annotation status
- Search images by filename

Annotation

- Draw bounding boxes around wildlife
- Assign species to annotations
- Verify model predictions
- Batch annotation of similar images
- Export annotations in COCO and YOLO formats

Environmental Data Collection

- Record light conditions (darkness, twilight, daylight)
- Record habitat information (mountains, plains)
- Track seasonal factors (temperature, snow cover)
- Auto-detect environmental factors from metadata

Behavioral Analysis

- Track animal behaviors through notes
- Analyze predator-prey relationships
- Analyze seasonal activity patterns
- Compare species similarities

Machine Learning

- Train wildlife detection models
- Evaluate model performance
- Use models for automated detection
- Validate model predictions

Reporting

- Generate detailed export reports
- Analyze dataset statistics
- Visualize species distribution
- Track annotation progress

Installation & Setup

Prerequisites

- Python 3.8 or higher
- SQLite
- OpenCV
- Flask
- SQLAlchemy
- Jupyter (optional, for notebooks)
- CUDA-capable GPU (optional, for faster training)

Installation Steps

1. Clone the repository:

   bash

   ```
   git clone <repository-url>
   cd WildlifeDetectionSystem
   ```

2. Run the setup script:

   bash

   ```
   bash scripts/setup.sh
   ```

3. Activate the virtual environment:

   bash

   ```
   source api/venv/bin/activate
   ```

4. Verify the installation:

   bash

   ```
   cd api
   python debug/app_connectivity.py
   ```

Configuration

The system configuration is defined in `api/config.py`:

- `UPLOAD_FOLDER`: Path for raw image storage
- `PROCESSED_FOLDER`: Path for processed images
- `ANNOTATIONS_FOLDER`: Path for annotation files
- `EXPORT_DIR`: Path for dataset exports
- `MODEL_FOLDER`: Path for trained models
- `CONFIG_FOLDER`: Path for model configurations

You can also create a `.env` file in the `api` directory to set environment variables:

```
FLASK_APP=run.py
FLASK_ENV=development
FLASK_DEBUG=1
SECRET_KEY=your-secret-key
```

Running the Application

1. Start the Flask server:

   bash

   ```
   cd api
   python run.py
   ```

2. Access the web interface at `http://localhost:5000`

Initializing the Database

1. Start the server to create tables automatically
2. Initialize default species:

   bash

   ```
   curl -X POST http://localhost:5000/api/species/initialize
   ```

3. Index existing images:

   bash

   ```
   curl -X POST http://localhost:5000/api/images/index-existing
   ```

Usage Guide

Annotating Images

1. Open the Advanced Annotator interface at `/advanced-annotator`
2. Select a folder from the dropdown
3. Draw bounding boxes around animals using the Box tool
4. Select the species for each annotation
5. Save annotations using the "Submit & Next" button
6. Use keyboard shortcuts for faster annotation:
   - B: Box tool
   - S: Select tool

- `Delete`: Delete annotation
- `F`: Fit to screen
- `1-9`: Select species
- `Left/Right arrows`: Navigate images

Adding Environmental Data

1. Open the Environmental Editor interface at `/environmental-editor`
2. Select an image from the list
3. Fill in environmental data fields:
   - Light Condition
   - Temperature
   - Snow Cover
   - Moon Phase
   - Habitat Type
   - Vegetation Type
4. Use the "Auto-Detect" button for automatic detection
5. Save data using the "Save Data" button

Exporting Datasets

1. Open the Dashboard interface at `/`
2. Click "Export COCO" or "Export YOLO" under Export Tools
3. The system will create a dataset in the specified format
4. A report will be generated with dataset statistics

Training Models

1. Start the Jupyter server from the Dashboard
2. Open the training notebook
3. Configure training parameters
4. Select the exported dataset
5. Run the training cells
6. Evaluate the model performance
7. Save the trained weights

Running Analysis

Various analysis scripts can be run from the command line:

```bash
# Seasonal analysis
python api/scripts/seasonal_analysis.py --charts --output ./charts

# Animal similarity analysis
python api/scripts/animal_similarity_analysis.py extract --species Wolf Jackal
python api/scripts/animal_similarity_analysis.py analyze --data ./similarity_analysis/

# Bulk annotation
python api/scripts/bulk_annotation_helper.py progress
python api/scripts/bulk_annotation_helper.py sequence test_01 100 120 7 0.2 0.3 0.5 0.

# Update environmental data
python api/scripts/update_environmental_data.py
```

Conclusion

The Wildlife Detection System provides a comprehensive solution for wildlife researchers using camera traps. It streamlines the process from image collection to annotation, analysis, and model training, allowing researchers to efficiently process large volumes of camera trap images and extract valuable ecological insights.

With its modular architecture, the system can be extended to include additional features and adapt to specific research requirements. The integration of machine learning capabilities enables automated detection, reducing the manual annotation workload for researchers.

The environmental and behavioral tracking features specifically address Prof. Peeva's requirements for diurnal activity analysis and predator-prey relationship tracking, making this system a valuable tool for wildlife research in Bulgaria and beyond.