# Notes

# PhishVision - Technical Guide for Team Members

## 📚 Table of Contents



---

## 🎯 Project Overview

**PhishVision** is a full-stack phishing simulation and email analysis platform built for security awareness training. It has two main features:

1. **Phishing Campaign Generator** - Create and track phishing simulation campaigns
2. **Email Analyzer** - Analyze emails for phishing indicators using NLP

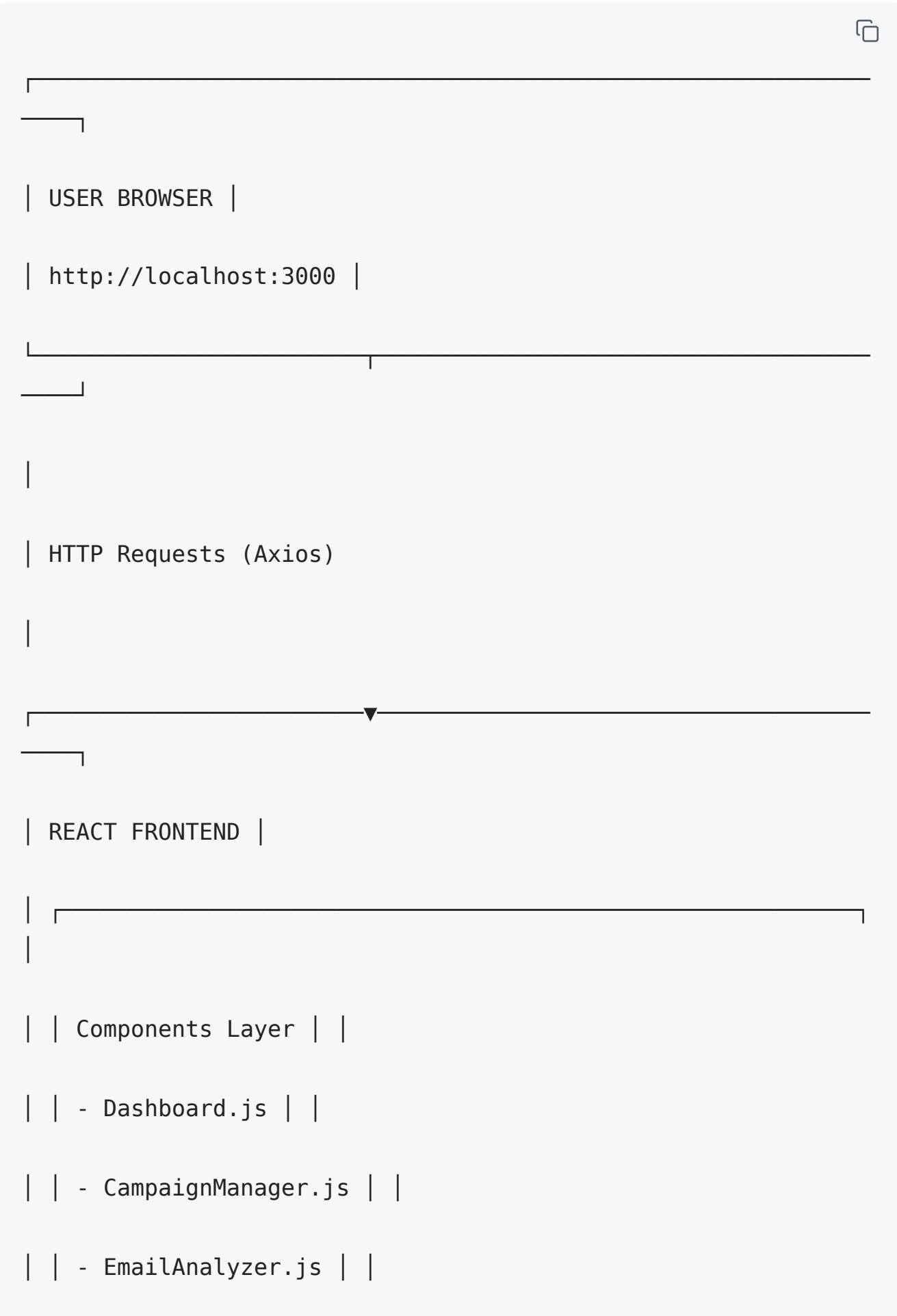
### Tech Stack

- **Backend**: Python 3.10+ with Flask
- **Frontend**: React 18 with Tailwind CSS
- **Database**: SQLite (dev) / PostgreSQL-ready (production)
- **AI**: Google Gemini API for template generation
- **Email**: SMTP (Gmail App Passwords)

---

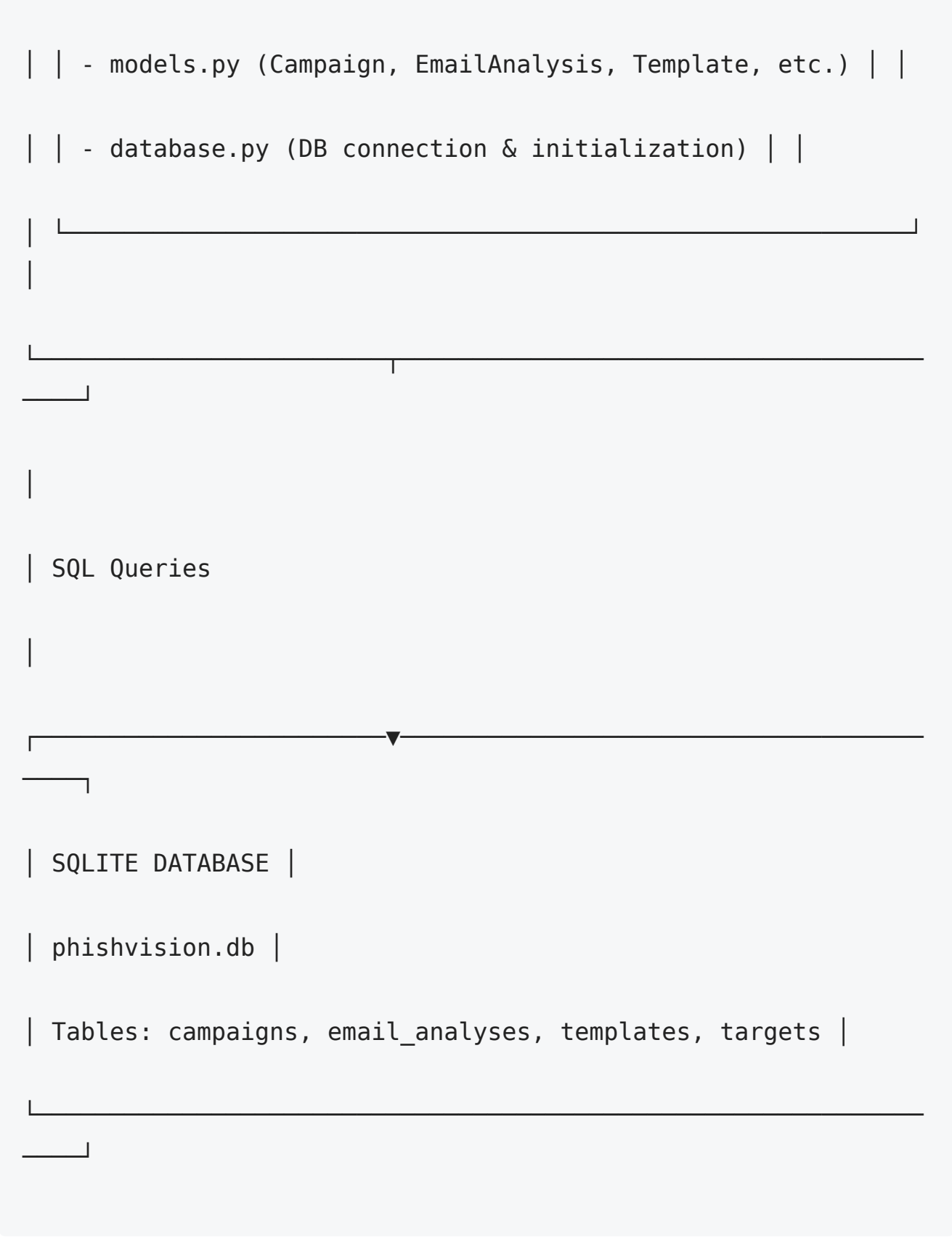
# 🏗️ Architecture

## High-Level System Architecture




| | - TemplateManager.js | |
| └──────────────────────────────────────────────────────────────────────┘
|
| ┌──────────────────────────────────────────────────────────────────────┐
|
| | API Client Layer (api/api.js) | |
| | - Axios HTTP client | |
| | - API endpoint mappings | |
| └──────────────────────────────────────────────────────────────────────┘
|
└──────────────────────────────┬──────────────────────────────────────────
────┘
|
| REST API (JSON)
|
┌──────────────────────────────▼──────────────────────────────────────────
────┐
| FLASK BACKEND |
| http://localhost:5000 |
| ┌──────────────────────────────────────────────────────────────────────┐
|

```
| | Routes Layer (routes/) | |
| | - campaign_routes.py → /api/campaigns/ | |
| | - analyzer_routes.py → /api/analyzer/ | |
| | - template_routes.py → /api/templates/ | |
| | - dashboard_routes.py → /api/dashboard/ | |
| | - tracking_routes.py → /track/ | |
| └──────────────────────────────────────────────────────────────┘
|
| ┌──────────────────────────────────────────────────────────────┐
|
| | Services Layer (services/) | |
| | - nlp_analyzer.py (Email analysis engine) | |
| | - email_service.py (SMTP email sending) | |
| | - ai_template_generator.py (Gemini AI integration) | |
| | - header_validator.py (SPF/DKIM validation) | |
| └──────────────────────────────────────────────────────────────┘
|
| ┌──────────────────────────────────────────────────────────────┐
|
| | Database Layer (SQLAlchemy ORM) | |
```




## 🔧 Backend Deep Dive

### Directory Structure

```
backend/
├── app.py # Main Flask application entry point
├── database.py # Database configuration & initialization
├── models.py # SQLAlchemy database models
│
├── routes/ # API endpoint handlers
│ ├── __init__.py
│ ├── campaign_routes.py # Campaign CRUD operations
│ ├── analyzer_routes.py # Email analysis endpoints
│ ├── template_routes.py # Template management
│ ├── dashboard_routes.py # Statistics & metrics
│ └── tracking_routes.py # Email open/click tracking
│
├── services/ # Business logic layer
│ ├── __init__.py
│ ├── nlp_analyzer.py # Core email analysis engine
│ ├── email_service.py # SMTP email sending service
```

```
| ├── ai_template_generator.py # AI-powered template
generation

| ├── header_validator.py # Email header validation

| └── email_parser.py # Email parsing utilities

|

└── templates/ # Email template system

├── __init__.py

└── phishing_templates.py # Built-in phishing templates
```

## Backend Process Flow

### 1. Application Startup (`app.py`)

```
# app.py - Simplified flow



from flask import Flask

from flask_cors import CORS

from database import init_db



app = Flask(__name__)

CORS(app) # Enable cross-origin requests from React frontend
```

```
# Initialize database (creates tables if they don't exist)
with app.app_context():
init_db()


# Register route blueprints
from routes import campaign_routes, analyzer_routes,
dashboard_routes
app.register_blueprint(campaign_routes.bp)
app.register_blueprint(analyzer_routes.bp)
app.register_blueprint(dashboard_routes.bp)


if __name__ == '__main__':
app.run(debug=True, port=5000)
```

### What happens on startup:

1. Flask app is created
2. CORS is enabled (allows React frontend to make requests)
3. Database is initialized (SQLite file is created if needed)
4. All API routes are registered
5. Server starts on `http://localhost:5000`

## 2. Database Models (`models.py`)

```
# models.py - Key models


class Campaign(db.Model):
"""Stores phishing campaign information"""
id = db.Column(db.String(36), primary_key=True) # UUID
name = db.Column(db.String(200), nullable=False)
template_type = db.Column(db.String(100))
status = db.Column(db.String(20)) # draft, active, completed
created_at = db.Column(db.DateTime)


# Relationships
targets = db.relationship('Target', backref='campaign')



class Target(db.Model):
"""Individual target in a campaign"""
id = db.Column(db.String(36), primary_key=True)
```

```
campaign_id = db.Column(db.String(36),
db.ForeignKey('campaign.id'))

email = db.Column(db.String(255))

tracking_token = db.Column(db.String(100)) # Unique token for
tracking



# Tracking metrics

email_sent = db.Column(db.Boolean, default=False)

email_opened = db.Column(db.Boolean, default=False)

link_clicked = db.Column(db.Boolean, default=False)

opened_at = db.Column(db.DateTime)

clicked_at = db.Column(db.DateTime)




class EmailAnalysis(db.Model):

"""Stores results of email analysis"""

id = db.Column(db.String(36), primary_key=True)

email_from = db.Column(db.String(255))

email_subject = db.Column(db.Text)
```

```
risk_score = db.Column(db.Float) # 0-100

classification = db.Column(db.String(20)) # safe, suspicious,
malicious

suspicious_keywords = db.Column(db.Text) # JSON

explanation = db.Column(db.Text)

created_at = db.Column(db.DateTime)
```

**Database Relationships:**

- One Campaign → Many Targets (one-to-many)
- Each Target has unique tracking token
- EmailAnalysis records are independent (no relations)

---

### 3. Email Analysis Engine (`services/nlp_analyzer.py`)

**This is the core of the email analyzer. Here's how it works:**

```
# nlp_analyzer.py - Simplified analysis flow



class NLPAnalyzer:

def analyze_email(self, subject, body, sender, attachments,
headers):

"""

Main analysis function - called when user submits email
```

```
Process:
1. Combine subject + body for full text analysis
2. Run multiple detection checks (in parallel)
3. Calculate risk score from all checks
4. Generate human-readable explanation
5. Return classification + recommendations
"""

full_text = f"{subject} {body}".lower()

# STEP 1: Find suspicious keywords
keywords = self._find_suspicious_keywords(full_text)
# Returns: {'urgent': ['urgent', 'immediate'],
# 'financial': ['bank', 'account'], ...}

# STEP 2: Calculate urgency score (0-100)
urgency = self._calculate_urgency_score(full_text)
```

```
# Checks: urgent keywords, time pressure, ALL CAPS,
punctuation



# STEP 3: Analyze URLs

urls = self._extract_urls(body)

url_analysis = self._analyze_urls(urls)

# Checks: IP addresses, suspicious TLDs, URL shorteners, @
symbols



# STEP 4: Check sender

sender_analysis = self._analyze_sender(sender)

# Checks: display name mismatch, suspicious patterns



# STEP 5: Analyze attachments (CRITICAL for malware)

attachment_analysis = self._analyze_attachments(attachments)

# Checks: .exe, .scr, .bat (malicious), .html, .zip
(suspicious)



# STEP 6: Encoding obfuscation detection
```

```
encoding_analysis = self._analyze_encoding_obfuscation(sender,
subject, headers)

# Checks: Base64 encoding, high-entropy domains



# STEP 7: Image-heavy spam detection

image_analysis = self._analyze_image_heavy_ratio(body)

# Checks: Many images + little text = spam



# STEP 8: Spam/scam keyword detection

spam_analysis = self._calculate_spam_score(full_text)

# Checks: 'miracle', 'weird trick', 'doctors hate', etc.



# STEP 9: Calculate final risk score (0-100)

risk_score = self._calculate_risk_score(

keywords, urgency, url_analysis, sender_analysis,

attachment_analysis, encoding_analysis,

image_analysis, spam_analysis

)
```

```
# STEP 10: Classify based on score

classification = self._classify_risk(risk_score)

# 0-29: safe, 30-59: suspicious, 60-100: malicious



# STEP 11: Generate explanation for user

explanation = self._generate_explanation(...)



# STEP 12: Generate security recommendations

recommendations =
self._generate_recommendations(classification, risk_score)



return {

'risk_score': risk_score,

'classification': classification,

'explanation': explanation,

'recommendations': recommendations,

# ... all other analysis data
```

```
}
```

## Risk Score Calculation:

```
def _calculate_risk_score(self, keywords, urgency,
url_analysis,

sender_analysis, attachment_analysis,

encoding_analysis, image_analysis, spam_analysis):

score = 0



# Keywords (max 40 points)

keyword_counts = {

'urgent': len(keywords['urgent']) * 8,

'financial': len(keywords['financial']) * 10,

'credentials': len(keywords['credentials']) * 15, # Highest
weight

'rewards': len(keywords['rewards']) * 12,

'threats': len(keywords['threats']) * 15,

'spam_scam': len(keywords['spam_scam']) * 12

}
```

```
score += min(sum(keyword_counts.values()), 40)



# Urgency (max 25 points)

score += urgency * 0.25



# URLs (max 25 points)

if url_analysis['suspicious_urls']:

score += min(len(url_analysis['suspicious_urls']) * 15, 25)



# Sender (max 10 points)

if sender_analysis['is_suspicious']:

score += 10



# Attachments (max 40 points - CRITICAL)

if attachment_analysis['risk_level'] == 'critical':

score += 40 # Malware detected!

elif attachment_analysis['risk_level'] == 'high':

score += 30
```

```
elif attachment_analysis['risk_level'] == 'medium':

score += 15



# Encoding (max 25 points)

if encoding_analysis['risk_level'] == 'high':

score += 25

elif encoding_analysis['risk_level'] == 'medium':

score += 15



# Image-heavy (max 20 points)

if image_analysis['is_image_heavy']:

score += 20



# Spam keywords (max 40 points)

if spam_analysis['is_spam']:

score += spam_analysis['spam_score']
```

```
return min(score, 100) # Cap at 100
```

---

## 4. Campaign Creation Process (`services/email_service.py`)

### What happens when a user creates a campaign:

```
# campaign_routes.py - Campaign creation endpoint



@bp.route('/api/campaigns/', methods=['POST'])

def create_campaign():

data = request.json



# STEP 1: Create campaign in database

campaign = Campaign(

id=str(uuid.uuid4()),

name=data['name'],

template_type=data['template_type'],

status='active',

created_at=datetime.utcnow()

)
```

```
db.session.add(campaign)



# STEP 2: Parse target emails (comma-separated string)

target_emails = [e.strip() for e in
data['targets'].split(',')]



# STEP 3: Create target records with unique tracking tokens

for email in target_emails:

target = Target(

id=str(uuid.uuid4()),

campaign_id=campaign.id,

email=email,

tracking_token=str(uuid.uuid4()), # Unique per target

email_sent=False

)

db.session.add(target)



db.session.commit()
```

```
# STEP 4: Send emails to all targets
email_service = EmailService()
email_service.send_campaign_emails(campaign)


return jsonify(campaign.to_dict()), 201
```

## Email Sending Process:

```
# services/email_service.py


class EmailService:
def send_campaign_emails(self, campaign):
"""Send phishing emails to all targets in campaign"""


# Get email template
template = get_template_by_type(campaign.template_type)


# Send to each target
```

```
for target in campaign.targets:

# STEP 1: Personalize email content

html_content = self._personalize_template(

template.html_content,

target

)



# STEP 2: Inject tracking pixel and link

html_content = self._inject_tracking(html_content,
target.tracking_token)



# STEP 3: Send email via SMTP

self._send_email(

to=target.email,

subject=template.subject,

html_body=html_content

)



# STEP 4: Mark as sent
```

```
target.email_sent = True

target.sent_at = datetime.utcnow()



db.session.commit()



def _inject_tracking(self, html, token):

"""Add tracking pixel and modify links"""



# Tracking pixel (1x1 transparent image)

pixel = f'<img src="http://localhost:5000/track/open/{token}"
width="1" height="1" />'

html = html.replace('</body>', f'{pixel}</body>')



# Replace placeholder link with tracking link

tracking_link = f'http://localhost:5000/track/click/{token}'

html = html.replace('{{tracking_link}}', tracking_link)
```

```
return html
```

---

## 5. Email Tracking Process (`routes/tracking_routes.py`)

### How tracking works:

```
# tracking_routes.py



@bp.route('/track/open/<token>')

def track_open(token):

"""

Called when recipient opens email (loads tracking pixel)

"""

# Find target by token

target = Target.query.filter_by(tracking_token=token).first()



if target and not target.email_opened:

# Mark as opened (only first time)

target.email_opened = True

target.opened_at = datetime.utcnow()
```

```
db.session.commit()



# Return 1x1 transparent pixel

return send_file('pixel.gif', mimetype='image/gif')




@bp.route('/track/click/<token>')

def track_click(token):

"""

Called when recipient clicks link in email

"""

# Find target by token

target = Target.query.filter_by(tracking_token=token).first()



if target and not target.link_clicked:

# Mark as clicked (only first time)

target.link_clicked = True

target.clicked_at = datetime.utcnow()
```

```
db.session.commit()



# Redirect to education page

return redirect('http://localhost:3000/education')
```

### Tracking Flow:

1. Email sent with unique token in pixel URL and link URL
2. User opens email → Browser loads pixel → `GET /track/open/{token}` → Mark opened
3. User clicks link → Browser requests link → `GET /track/click/{token}` → Mark clicked
4. Frontend can query campaign stats to see who opened/clicked

---

## 6. AI Template Generation (`services/ai_template_generator.py`)

```
# ai_template_generator.py



class AITemplateGenerator:

def __init__(self):

api_key = os.getenv('GEMINI_API_KEY')

genai.configure(api_key=api_key)

self.model = genai.GenerativeModel('gemini-1.5-pro-latest')
```

```
def generate_template(self, description, category, difficulty,
company_name):

"""

Generate phishing template using Google Gemini AI



Process:

1. Get current date/events context for realism

2. Build detailed prompt with difficulty guidelines

3. Send to Gemini API

4. Parse JSON response

5. Validate required fields

6. Return template or fallback on error

"""



# STEP 1: Get current context (month, events, etc.)

current_context = self._get_current_events_context()

# Example: "Current date: November 29, 2024 | Day: Friday |

# Relevant events: Thanksgiving, Black Friday"
```

```
# STEP 2: Build AI prompt

prompt = f"""You are creating a phishing simulation template
for

authorized security training.



Category: {category}

Difficulty: {difficulty}

Scenario: {description}

Company: {company_name}

Context: {current_context}



Generate a complete HTML email template with:

- Realistic corporate styling

- Psychological triggers matching difficulty

- Template variables: {{{{full_name}}}}, {{{{tracking_link}}}}

- Educational red flags for training
```

```
Return ONLY valid JSON:

{{

"name": "template name",

"subject": "email subject",

"html_content": "full HTML email",

"metadata": {{

"difficulty": "{difficulty}",

"primary_trigger": "urgency + authority",

"red_flags": ["list", "of", "red flags"],

"training_notes": "why this is effective"

}}

}}

"""



# STEP 3: Call Gemini API

try:

response = self.model.generate_content(prompt)

text = response.text.strip()
```

```
# STEP 4: Clean and parse JSON

text = re.sub(r'```json\s*', '', text)

text = re.sub(r'\s*```', '', text)

template_data = json.loads(text)



# STEP 5: Validate required fields

if not all(k in template_data for k in ['name', 'subject',
'html_content']):

raise ValueError("Missing required fields")



return template_data



except Exception as e:

# STEP 6: Return fallback template on error

return self._get_fallback_template(description, category,
company_name)
```

## ⚛️ Frontend Deep Dive

## Directory Structure

```
frontend/src/
├── index.js # React app entry point
├── App.js # Main app component with routing
├── ThemeContext.js # Dark/light theme provider
│
├── components/ # React components
│ ├── LandingPage.js # Homepage
│ ├── Dashboard.js # Statistics dashboard
│ ├── CampaignManager.js # Campaign list & creation
│ ├── CampaignDetails.js # Single campaign view
│ ├── TemplateManager.js # Template browser
│ ├── EmailAnalyzer.js # Email analysis tool
│ └── ConfirmDialog.js # Reusable confirmation dialog
│
├── api/ # API client layer
│ └── api.js # Axios HTTP client functions
│
```

```
├── utils/ # Utility functions
│ └── emailParser.js # Email parsing logic
│
└── config/ # Configuration
└── colors.js # Theme color definitions
```

## Frontend Process Flow

### 1. Application Bootstrap (`index.js` → `App.js`)

```
// index.js - Entry point
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

```
// App.js - Main application
```

```
import { BrowserRouter as Router, Routes, Route } from 'react-
router-dom';

import { ThemeProvider } from './ThemeContext';



function App() {

return (

<ThemeProvider> {/* Provides dark/light theme to all
components */}

<Router> {/* Enables client-side routing */}

<AppContent />

</Router>

</ThemeProvider>

);

}



function AppContent() {

return (

<div>

<Navigation /> {/* Top navigation bar */}
```

```
<Routes>

<Route path="/" element={<LandingPage />} />

<Route path="/dashboard" element={<Dashboard />} />

<Route path="/campaigns" element={<CampaignManager />} />

<Route path="/campaigns/:id" element={<CampaignDetails />} />

<Route path="/templates" element={<TemplateManager />} />

<Route path="/analyzer" element={<EmailAnalyzer />} />

</Routes>

</div>

);

}
```

---

### 2. API Client Layer (`api/api.js`)

**All backend communication goes through this layer:**

```
// api/api.js

import axios from 'axios';



const API_BASE_URL = 'http://localhost:5000/api';
```

```
// Campaign API calls
export const getCampaigns = () =>
axios.get(`${API_BASE_URL}/campaigns/`);

export const createCampaign = (data) =>
axios.post(`${API_BASE_URL}/campaigns/`, data);

export const deleteCampaign = (id) =>
axios.delete(`${API_BASE_URL}/campaigns/${id}`);

// Email Analyzer API calls
export const analyzeEmail = (data) =>
axios.post(`${API_BASE_URL}/analyzer/analyze`, data);

export const getAnalysisHistory = () =>
axios.get(`${API_BASE_URL}/analyzer/history`);
```

```
// Dashboard API calls

export const getDashboardStats = () =>

axios.get(`${API_BASE_URL}/dashboard/stats`);



export const getCampaignPerformance = () =>

axios.get(`${API_BASE_URL}/dashboard/campaign-performance`);
```

### Why this pattern?

- Centralized API endpoint management
- Easy to update URLs
- Consistent error handling
- Reusable across components

---

## 3. Email Analyzer Component (`components/EmailAnalyzer.js`)

### Complete user flow:

```
// EmailAnalyzer.js - User flow



function EmailAnalyzer() {

// STATE MANAGEMENT

const [rawEmail, setRawEmail] = useState(''); // User's pasted
```

```
email

const [parsedData, setParsedData] = useState(null); //
Extracted fields

const [formData, setFormData] = useState({}); // Data to send
to backend

const [result, setResult] = useState(null); // Analysis result

const [loading, setLoading] = useState(false);



// STEP 1: User pastes email

const handleEmailPaste = (e) => {

const pastedText = e.target.value;

setRawEmail(pastedText);



// Auto-parse email

const parsed = parseEmail(pastedText); // From
utils/emailParser.js

setParsedData(parsed);



// Populate form data

setFormData({
```

```
email_from: parsed.email_from,

email_subject: parsed.email_subject,

email_body: parsed.email_body,

headers: parsed.headers

});

};



// STEP 2: User clicks "Analyze Email"

const handleSubmit = async (e) => {

e.preventDefault();

setLoading(true);



try {

// STEP 3: Send to backend

const response = await analyzeEmail(formData);



// STEP 4: Display results

setResult(response.data);
```

```
} catch (error) {
console.error('Analysis failed:', error);
} finally {
setLoading(false);
}
};

return (
<div>
{/* STEP 1: Input Section */}
<form onSubmit={handleSubmit}>
<textarea
value={rawEmail}
onChange={handleEmailPaste}
placeholder="Paste full email here..."
/>

{/* STEP 2: Preview parsed data */}
```

```
{parsedData && (
<div className="preview">
<p>From: {parsedData.email_from}</p>
<p>Subject: {parsedData.email_subject}</p>
<p>Body preview: {parsedData.email_body.substring(0, 100)}...
</p>
</div>
)}



<button type="submit">Analyze Email</button>
</form>



{/* STEP 3: Display results */}
{result && (
<div className="results">
<h2>Risk Score: {result.risk_score}/100</h2>
<p>Classification: {result.classification}</p>
<p>{result.explanation}</p>
```

```
<ul>

{result.recommendations.map((rec, i) => (

<li key={i}>{rec}</li>

))}

</ul>

</div>

)}

</div>

);

}
```

---

### 4. Email Parser Utility (`utils/emailParser.js`)

### How email parsing works:

```
// emailParser.js



export function parseEmail(rawEmail) {

const lines = rawEmail.split('\n');

let email_from = '';
```

```
let email_subject = '';

let email_body = '';

let headers = '';

let headerEndIndex = -1;



// STEP 1: Parse headers line by line

for (let i = 0; i < lines.length; i++) {

const line = lines[i];



// Extract "From:" header

if (line.match(/^(From|from|FROM):\s*(.+)/i)) {

const fromValue = line.split(':')[1].trim();

// Handle "Name <email@domain.com>" format

const emailMatch = fromValue.match(/<([^>]+)>/) ||

fromValue.match(/([^\s]+@[^\s]+)/);

email_from = emailMatch ? emailMatch[1] : fromValue;

}
```

```
// Extract "Subject:" header
if (line.match(/^(Subject|subject|SUBJECT):\s*(.+)/i)) {
email_subject = line.split(':')[1].trim();
}


// Detect end of headers (blank line)
if (line.trim() === '' && i > 0 && headerEndIndex === -1) {
headerEndIndex = i;
headers = lines.slice(0, i).join('\n');
}
}


// STEP 2: Extract body (everything after headers)
if (headerEndIndex !== -1) {
email_body = lines.slice(headerEndIndex +
1).join('\n').trim();
}
```

```
// STEP 3: Return parsed data

return {

email_from,

email_subject,

email_body,

headers,

success: !!(email_from && email_subject && email_body)

};

}
```

---

## 🔄 How Everything Works Together

### Complete Flow: Creating a Campaign

```
USER ACTION FRONTEND BACKEND DATABASE

| | | |

| 1. User fills form | | |

| (name, template, targets) | | |

|----------------------------->| | |

| | 2. Validate form | |
```

```
| | | |

| | 3. POST /api/campaigns/ | |

| |---------------------------->| |

| | | 4. Create Campaign |

| | |------------------------->|

| | | 5. Create Targets |

| | |------------------------->|

| | | 6. Generate tokens |

| | | |

| | | 7. Send emails (SMTP) |

| | | |

| | 8. Return campaign data | |

| |<----------------------------| |

| 9. Show success + redirect | | |

|<----------------------------| | |
```

## Complete Flow: Analyzing an Email

```
USER ACTION FRONTEND BACKEND
```

```
| | |

| 1. Paste full email | |

|----------------------------->| |

| | 2. Auto-parse email |

| | (extract from/subject/body) |

| | |

| | 3. Show preview |

|<-----------------------------| |

| | |

| 4. Click "Analyze" | |

|----------------------------->| |

| | 5. POST /api/analyzer/analyze

| |--------------------------->|

| | | 6. NLPAnalyzer.analyze_email()

| | | - Find keywords (40pts max)

| | | - Urgency score (25pts max)

| | | - URL analysis (25pts max)

| | | - Sender check (10pts max)
```

```
| | | - Attachments (40pts max)

| | | - Encoding (25pts max)

| | | - Images (20pts max)

| | | - Spam (40pts max)

| | | 7. Calculate risk (0-100)

| | | 8. Classify (safe/suspicious/malicious)

| | | 9. Generate explanation

| | | 10. Save to database

| | 11. Return analysis |

| |<----------------------------|

| 12. Display results | |

|<----------------------------| |
```

## Complete Flow: Email Tracking

```
RECIPIENT ACTION TRACKING SYSTEM BACKEND DATABASE

| | | |

| 1. Opens email | | |

| (email client loads images) | | |
```

```
|----------------------------->| | |
| | 2. GET /track/open/{token} | |
| |----------------------------->| |
| | | 3. Find target by token |
| | |------------------------->|
| | | 4. Mark opened=True |
| | |------------------------->|
| | 5. Return 1x1 pixel | |
| |<-----------------------------| |
| 6. Image rendered | | |
|<-----------------------------| | |
| | | |
| 7. Clicks link in email | | |
|----------------------------->| | |
| | 8. GET /track/click/{token} | |
| |----------------------------->| |
| | | 9. Find target by token |
| | |------------------------->|
```

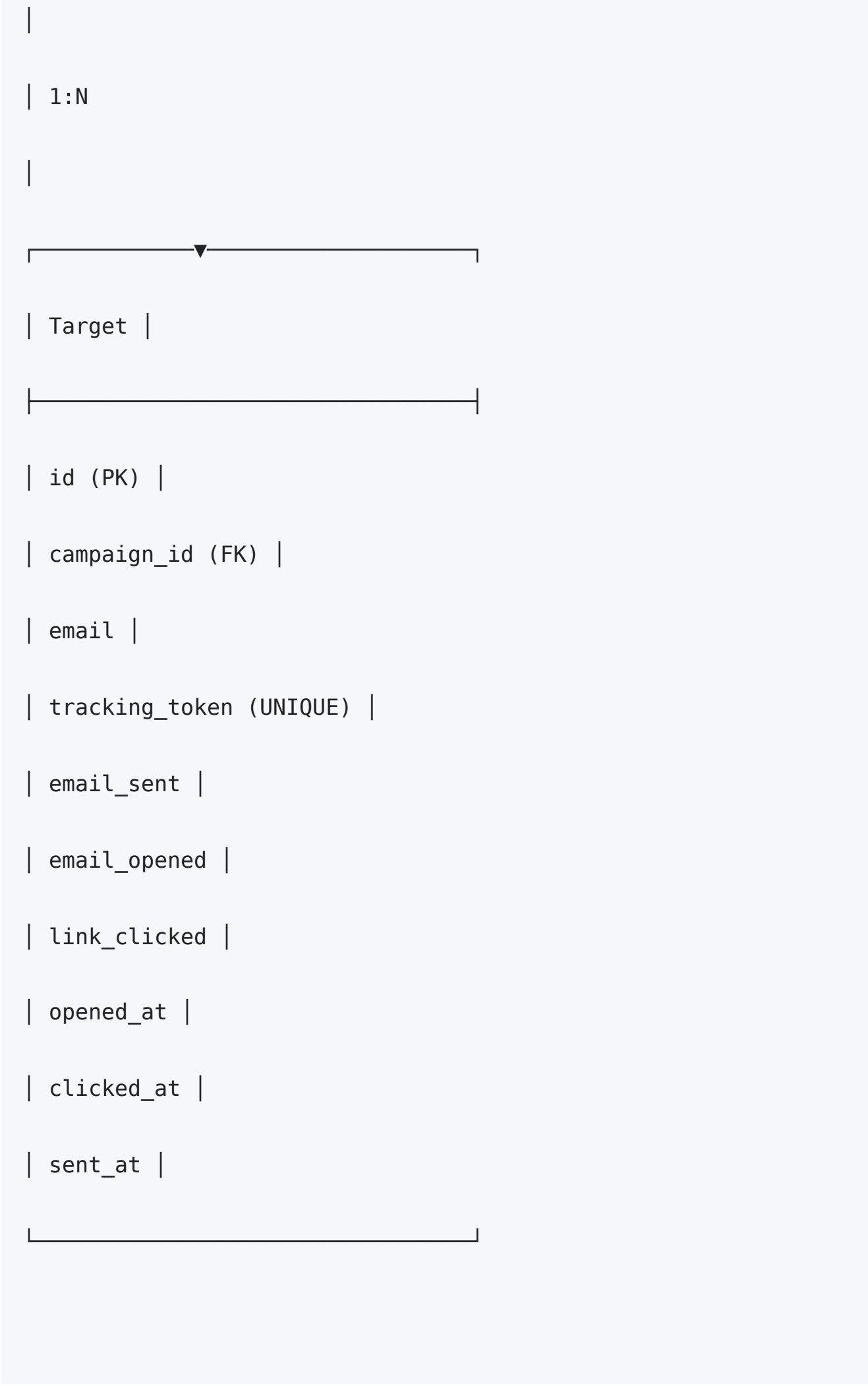
```
| | | 10. Mark clicked=True |

| | |------------------------->|

| | 11. Redirect to education | |

| 12. Education page shown | | |

|<-----------------------------| | |
```

---

# 💾 Database Schema

## Entity Relationship Diagram

```
┌──────────────────────────────────────────┐

│ Campaign │

├──────────────────────────────────────────┤

│ id (PK) │

│ name │

│ template_type │

│ status (draft/active/completed) │

│ created_at │

└──────────────┬───────────────────────────┘
```


1:N
Target
id (PK)
campaign_id (FK)
email
tracking_token (UNIQUE)
email_sent
email_opened
link_clicked
opened_at
clicked_at
sent_at

```
┌─────────────────────────────────────┐
│ EmailAnalysis                       │
├─────────────────────────────────────┤
│ id (PK)                             │
│ email_from                          │
│ email_subject                       │
│ email_body                          │
│ headers                             │
│ risk_score (0-100)                  │
│ classification                      │
│ suspicious_keywords (JSON)          │
│ url_analysis (JSON)                 │
│ urgency_score                       │
│ attachment_analysis (JSON)          │
│ encoding_analysis (JSON)            │
│ explanation (TEXT)                  │
│ recommendations (JSON)              │
│ created_at                          │
```

```
└──────────────────────────────────────┘



┌──────────────────────────────────────┐
│ Template │
├──────────────────────────────────────┤
│ id (PK) │
│ name │
│ category │
│ subject │
│ html_content (TEXT) │
│ is_builtin │
│ created_at │
└──────────────────────────────────────┘
```

---

# 📡 API Reference

## Campaigns

### GET /api/campaigns/

- Returns: List of all campaigns with stats
- Response:

```
[
{
"id": "uuid",
"name": "Q4 Security Training",
"template_type": "bank_alert",
"status": "active",
"total_targets": 50,
"emails_sent": 50,
"opened_count": 32,
"clicked_count": 15,
"open_rate": 64.0,
"click_rate": 30.0,
"created_at": "2024-11-29T10:00:00"
}
]
```


## POST /api/campaigns/

- Body:

```
{
"name": "Campaign Name",
"template_type": "bank_alert",
"targets": "user1@example.com, user2@example.com"
}
```

- Returns: Created campaign object

### GET /api/campaigns/:id

- Returns: Campaign details with all targets

### DELETE /api/campaigns/:id

- Returns: 204 No Content

---

## Email Analyzer

### POST /api/analyzer/analyze

- Body:

```
{
"email_from": "sender@example.com",
"email_subject": "Urgent: Verify your account",
"email_body": "Click here to verify...",
```

```
"headers": "Received: from...",
"attachments": ["invoice.exe", "document.pdf"]
}
```

- Returns:

```
{
"id": "uuid",
"risk_score": 75.5,
"classification": "malicious",
"suspicious_keywords": {
"urgent": ["urgent"],
"credentials": ["verify", "account"],
"financial": []
},
"url_analysis": {
"total_urls": 1,
"suspicious_urls": [
{
```

```
"url": "http://192.168.1.1/verify",

"reasons": ["Uses IP address instead of domain"]

}

]

},

"urgency_score": 65.0,

"attachment_analysis": {

"total_attachments": 2,

"malicious_files": ["invoice.exe"],

"risk_level": "critical"

},

"explanation": "This email has been classified as 'MALICIOUS'
with a risk score of 75.5/100...",

"recommendations": [

"DO NOT click any links or download attachments",

"Report this email to your IT security team immediately"

],

"created_at": "2024-11-29T10:30:00"
```

```
}
```

### GET /api/analyzer/history

- Query params: `?limit=50`
- Returns: List of recent analyses

### GET /api/analyzer/stats

- Returns: Analyzer statistics

```
{
"total_analyzed": 150,
"safe_count": 80,
"suspicious_count": 45,
"malicious_count": 25,
"avg_risk_score": 42.5
}
```

---

## Dashboard

### GET /api/dashboard/stats

- Returns:

```
{
```

```
"total_campaigns": 10,

"total_emails_sent": 500,

"total_opened": 325,

"total_clicked": 150,

"avg_open_rate": 65.0,

"avg_click_rate": 30.0,

"total_analyzed": 150,

"safe_count": 80,

"suspicious_count": 45,

"malicious_count": 25

}
```


## GET /api/dashboard/campaign-performance

- Returns: Top 5 campaigns with open/click rates


```
[

{

"name": "Q4 Security Training",

"open_rate": 65.0,
```

```
"click_rate": 30.0

}

]
```

### GET /api/dashboard/threat-distribution

- Returns: Email analysis distribution

```
[

{"name": "Safe", "value": 80},

{"name": "Suspicious", "value": 45},

{"name": "Malicious", "value": 25}

]
```

---

## Templates

### GET /api/templates

- Query params: `?category=financial`
- Returns: List of available templates

### POST /api/templates/generate-ai

- Body:

```
{
"description": "Urgent VPN certificate renewal",
"category": "IT",
"difficulty": "expert",
"company_name": "Acme Corp"
}
```

- Returns: AI-generated template

---

### Tracking

#### GET /track/open/:token

- Returns: 1×1 transparent GIF
- Side effect: Marks target as opened

#### GET /track/click/:token

- Returns: HTTP 302 redirect to education page
- Side effect: Marks target as clicked

---

## 🧪 Code Flow Examples

### Example 1: User Creates Campaign

1. User fills form on `/campaigns` page
2. Frontend calls `createCampaign(data)` from `api/api.js`
3. Axios sends `POST http://localhost:5000/api/campaigns/`

4. Backend route `campaign_routes.py` receives request
5. Creates `Campaign` and `Target` records in database
6. `EmailService` sends emails to all targets via SMTP
7. Each email has unique tracking token in pixel and link
8. Backend returns campaign JSON
9. Frontend redirects to campaign details page
10. User sees campaign with 0% open rate (nobody opened yet)

### Example 2: Recipient Opens Email

1. Recipient opens email in Gmail
2. Gmail loads all images including tracking pixel
3. Browser requests `GET http://localhost:5000/track/open/abc-123-def`
4. Backend finds target with token `abc-123-def`
5. Sets `target.email_opened = True` and `target.opened_at = now()`
6. Returns 1×1 transparent GIF (invisible to user)
7. Frontend dashboard now shows 1 open (when user refreshes)

### Example 3: User Analyzes Email

1. User pastes raw email on `/analyzer` page
2. `emailParser.js` extracts From, Subject, Body, Headers
3. Frontend shows preview of parsed data
4. User clicks "Analyze Email"
5. Frontend calls `analyzeEmail(formData)`
6. Backend `NLPAnalyzer` runs 8 different checks
7. Calculates risk score from all checks (e.g., 75/100)
8. Classifies as "malicious" (>= 60)
9. Generates explanation and recommendations
10. Saves analysis to database
11. Returns JSON result
12. Frontend displays risk score, classification, explanation

---

# 🛠️ Development Workflow

## Setting Up Development Environment

### Terminal 1 - Backend:

```
cd backend

source venv/bin/activate

python app.py

# Server runs on http://localhost:5000
```

### Terminal 2 - Frontend:

```
cd frontend

npm start

# Server runs on http://localhost:3000
```

## Making Changes

### Backend Changes:

1. Edit Python files in `backend/`
2. Flask auto-reloads on file save (debug mode)
3. Test with browser or Postman

### Frontend Changes:

1. Edit React files in `frontend/src/`
2. Webpack auto-recompiles on save

3. Browser auto-refreshes (hot reload)

## Database Changes

### Adding a new model field:

```
# models.py

class Campaign(db.Model):

# ... existing fields ...

new_field = db.Column(db.String(100)) # Add this
```

### Reset database:

```
cd backend

rm instance/phishvision.db # Delete database

python app.py # Recreates with new schema
```

## Adding a New API Endpoint

### 1. Backend Route:

```
# routes/campaign_routes.py

@bp.route('/api/campaigns/<id>/pause', methods=['PUT'])

def pause_campaign(id):
```

```
campaign = Campaign.query.get_or_404(id)

campaign.status = 'paused'

db.session.commit()

return jsonify(campaign.to_dict())
```

### 2. Frontend API Client:

```
// api/api.js

export const pauseCampaign = (id) =>

axios.put(`${API_BASE_URL}/campaigns/${id}/pause`);
```

### 3. Frontend Component:

```
// components/CampaignManager.js

const handlePause = async (campaignId) => {

await pauseCampaign(campaignId);

loadCampaigns(); // Refresh list

};
```

---

## 🚀 Deployment Checklist

Before deploying to production:

## Security

- [ ] Add authentication/authorization
- [ ] Enable rate limiting
- [ ] Add input validation everywhere
- [ ] Remove debug mode
- [ ] Restrict CORS to specific domains
- [ ] Add CSRF protection
- [ ] Rotate any committed secrets
- [ ] Add HTTPS/SSL

## Performance

- [ ] Switch from SQLite to PostgreSQL
- [ ] Add database indexes
- [ ] Enable query caching
- [ ] Optimize images
- [ ] Minify frontend assets
- [ ] Enable gzip compression

## Monitoring

- [ ] Add logging (Winston/Sentry)
- [ ] Add error tracking
- [ ] Add performance monitoring
- [ ] Set up alerts
- [ ] Add health check endpoints

## Infrastructure

- [ ] Use production WSGI server (Gunicorn)
- [ ] Set up reverse proxy (Nginx)
- [ ] Configure environment variables
- [ ] Set up CI/CD pipeline
- [ ] Configure backups

## 📞 Support

If you have questions about the codebase:

1. **Read this guide first** - Most common questions are answered here
2. **Check the code comments** - Critical sections have inline documentation
3. **Review the API** - Use Postman to test endpoints directly
4. **Check the logs** - Backend prints debug info to console
5. **Ask the team** - We're here to help!

---

## 🎓 Learning Resources

Want to understand the code better?

**Flask:**

- Official docs: https://flask.palletsprojects.com/
- SQLAlchemy ORM: https://docs.sqlalchemy.org/

**React:**

- Official docs: https://react.dev/
- React Router: https://reactrouter.com/

**Email Security:**

- OWASP Phishing Guide: https://owasp.org/www-community/attacks/Phishing
- Email Headers Explained: https://mxtoolbox.com/emailheaders.aspx

**NLP & Analysis:**

- Regex Tutorial: https://regexone.com/
- Text Analysis: https://realpython.com/natural-language-processing-spacy-python/

---

**Last Updated:** November 29, 2024

**Version:** 1.0

**Maintainers:** PhishVision Team