Main API Endpoint (api/main.py)

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Optional, List
from datetime import datetime
from models.predict import WaterQualityPredictor
from recommender.rules import WaterQualityRecommender
from database.models import WaterQualityMeasurement, WaterQualityPrediction,
Recommendation
from database.config import SessionLocal, engine
from sqlalchemy.orm import Session

app = FastAPI()

# Create database tables
WaterQualityMeasurement.metadata.create_all(bind=engine)

# Initialize models
predictor = WaterQualityPredictor()
recommender = WaterQualityRecommender()

class WaterQualityRequest(BaseModel):
    location: str
    Lat: float
    Lon: float
    Temperature: float
    DO: float
    pH: float
    Conductivity: float
    BOD: float
    Nitrate: float
    Fecalcaliform: float
    Totalcaliform: float

class WaterQualityResponse(BaseModel):
    location: str
    is_potable: bool
    confidence: float
    recommendations: List[dict]

@app.post("/predict", response_model=WaterQualityResponse)
async def predict_water_quality(request: WaterQualityRequest):
    try:
        # Create database session
        db = SessionLocal()

        # Create measurement record
        measurement = WaterQualityMeasurement(
            location=request.location,
            Lat=request.Lat,
            Lon=request.Lon,
            Temperature=request.Temperature,
            DO=request.DO,
            pH=request.pH,
```

```
    Conductivity=request.Conductivity,
    BOD=request.BOD,
    Nitrate=request.Nitrate,
    Fecalcaliform=request.Fecalcaliform,
    Totalcaliform=request.Totalcaliform
)
db.add(measurement)
db.commit()
db.refresh(measurement)

# Make prediction
prediction = predictor.predict({
    "Lat": request.Lat,
    "Lon": request.Lon,
    "Temperature": request.Temperature,
    "DO": request.DO,
    "pH": request.pH,
    "Conductivity": request.Conductivity,
    "BOD": request.BOD,
    "Nitrate": request.Nitrate,
    "Fecalcaliform": request.Fecalcaliform,
    "Totalcaliform": request.Totalcaliform
})

# Create prediction record
prediction_record = WaterQualityPrediction(
    measurement_id=measurement.id,
    is_potable=prediction["is_potable"],
    confidence=prediction["confidence"],
    model_version="1.0"
)
db.add(prediction_record)

# Generate recommendations
recommendations = recommender.generate_recommendations({
    "pH": request.pH,
    "DO": request.DO,
    "Conductivity": request.Conductivity,
    "BOD": request.BOD,
    "Nitrate": request.Nitrate,
    "Fecalcaliform": request.Fecalcaliform,
    "Totalcaliform": request.Totalcaliform
})

# Create recommendation records
for rec in recommendations:
    recommendation = Recommendation(
        measurement_id=measurement.id,
        parameter=rec["parameter"],
        severity=rec["severity"],
        priority=rec["priority"],
        recommendation=rec["recommendation"]
    )
    db.add(recommendation)
```

```
        db.commit()

        return WaterQualityResponse(
            location=request.location,
            is_potable=prediction["is_potable"],
            confidence=prediction["confidence"],
            recommendations=recommendations
        )

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
    finally:
        db.close()

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Prediction Model (models/predict.py):

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from joblib import dump, load
import os
import matplotlib.pyplot as plt
import seaborn as sns

class WaterQualityPredictor:
    def __init__(self, model_path: str = None):
        self.model = None
        self.scaler = None
        self.label_encoder = None
        self.model_path = model_path or "models/water_quality_model.joblib"
        self.scaler_path = model_path.replace('.joblib', '_scaler.joblib') if
model_path else "models/water_quality_scaler.joblib"
        self.encoder_path = model_path.replace('.joblib', '_encoder.joblib') if
model_path else "models/water_quality_encoder.joblib"
        self.feature_columns = None
        self.target_column = None

    def train(self, data_path: str):
        """Train the model using the provided dataset"""
        try:
            # Load and prepare data
            print(f"Loading data from {data_path}...")
            df = pd.read_excel(data_path)
            print(f"Available columns: {df.columns.tolist()}")
```

```
            # Look for potability column with case-insensitive match
            self.target_column = None
            for col in df.columns:
                if col.lower() == 'potability':
                    self.target_column = col
                    break

            if self.target_column is None:
                raise ValueError("Could not find 'Potability' column in
dataset")

            # Drop non-numeric columns that aren't relevant for prediction
            columns_to_drop = ['Stationcode', 'Locations', 'Capitalcity',
'State']
            df = df.drop(columns=[col for col in columns_to_drop if col in
df.columns])

            # Convert remaining columns to numeric, replacing non-numeric values
with NaN
            for col in df.columns:
                if col != self.target_column:
                    df[col] = pd.to_numeric(df[col], errors='coerce')

            # Drop rows with NaN values
            df = df.dropna()

            # Prepare features and target
            X = df.drop([self.target_column], axis=1)
            y = df[self.target_column]

            # Encode target variable
            self.label_encoder = LabelEncoder()
            y_encoded = self.label_encoder.fit_transform(y)

            # Store feature columns for later use in prediction
            self.feature_columns = X.columns.tolist()

            print(f"\nFeatures after preprocessing: {self.feature_columns}")
            print(f"Target column: {self.target_column}")
            print(f"Number of samples after preprocessing: {len(df)}")

            # Analyze class distribution
            print("\nClass distribution:")
            print(y.value_counts(normalize=True))

            # Visualize feature distributions
            self._plot_feature_distributions(X, y)

            # Split data
            X_train, X_test, y_train, y_test = train_test_split(X, y_encoded,
test_size=0.2, random_state=42, stratify=y_encoded)
            print(f"\nTraining set size: {len(X_train)}")
            print(f"Test set size: {len(X_test)}")
```

```
            # Scale features
            self.scaler = StandardScaler()
            X_train_scaled = self.scaler.fit_transform(X_train)
            X_test_scaled = self.scaler.transform(X_test)

            # Calculate class weights
            class_weights = dict(zip(np.unique(y_encoded), len(y_encoded) / (2 *
np.bincount(y_encoded))))
            print(f"\nClass weights: {class_weights}")

            # Define parameter grid for grid search
            param_grid = {
                'n_estimators': [50, 100, 200],
                'max_depth': [5, 10, 15],
                'min_samples_split': [2, 5, 10],
                'min_samples_leaf': [1, 2, 4],
                'class_weight': [None, 'balanced', class_weights]
            }

            # Initialize base model
            base_model = RandomForestClassifier(random_state=42)

            # Perform grid search
            print("\nPerforming grid search...")
            grid_search = GridSearchCV(
                estimator=base_model,
                param_grid=param_grid,
                cv=5,
                scoring='roc_auc',
                n_jobs=-1,
                verbose=1
            )
            grid_search.fit(X_train_scaled, y_train)

            # Get best model
            self.model = grid_search.best_estimator_
            print(f"\nBest parameters: {grid_search.best_params_}")

            # Perform cross-validation with best model
            cv_scores = cross_val_score(self.model, X_train_scaled, y_train,
cv=5, scoring='roc_auc')
            print(f"\nCross-validation ROC-AUC scores: {cv_scores}")
            print(f"Mean CV ROC-AUC: {cv_scores.mean():.2%} (+/-
{cv_scores.std() * 2:.2%})")

            # Evaluate on test set
            y_pred = self.model.predict(X_test_scaled)
            y_prob = self.model.predict_proba(X_test_scaled)[:, 1]

            print("\nClassification Report:")
            print(classification_report(y_test, y_pred,
target_names=self.label_encoder.classes_))

            print(f"\nROC-AUC Score: {roc_auc_score(y_test, y_prob):.2%}")
```

```
            # Plot confusion matrix
            self._plot_confusion_matrix(y_test, y_pred)

            # Analyze feature importance
            self._plot_feature_importance(X)

            # Save model, scaler, and encoder
            os.makedirs(os.path.dirname(self.model_path), exist_ok=True)
            dump(self.model, self.model_path)
            dump(self.scaler, self.scaler_path)
            dump(self.label_encoder, self.encoder_path)

            return cv_scores.mean()

        except Exception as e:
            print(f"Error during training: {str(e)}")
            return None

    def _plot_feature_distributions(self, X, y):
        """Plot distributions of features for each class"""
        plt.figure(figsize=(15, 10))
        for i, feature in enumerate(X.columns):
            plt.subplot(3, 4, i+1)
            sns.boxplot(x=y, y=X[feature])
            plt.title(feature)
        plt.tight_layout()
        plt.savefig('feature_distributions.png')
        plt.close()

    def _plot_confusion_matrix(self, y_true, y_pred):
        """Plot confusion matrix"""
        cm = confusion_matrix(y_true, y_pred)
        plt.figure(figsize=(8, 6))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                   xticklabels=self.label_encoder.classes_,
                   yticklabels=self.label_encoder.classes_)
        plt.title('Confusion Matrix')
        plt.xlabel('Predicted')
        plt.ylabel('True')
        plt.savefig('confusion_matrix.png')
        plt.close()

    def _plot_feature_importance(self, X):
        """Plot feature importance"""
        importance = pd.Series(self.model.feature_importances_, index=X.columns)
        importance = importance.sort_values(ascending=False)
        plt.figure(figsize=(10, 6))
        sns.barplot(x=importance.values, y=importance.index)
        plt.title('Feature Importance')
        plt.tight_layout()
        plt.savefig('feature_importance.png')
        plt.close()
```

```
    def load_model(self):
        """Load the trained model, scaler, and encoder"""
        if all(os.path.exists(path) for path in [self.model_path,
self.scaler_path, self.encoder_path]):
            try:
                self.model = load(self.model_path)
                self.scaler = load(self.scaler_path)
                self.label_encoder = load(self.encoder_path)
                return True
            except Exception as e:
                print(f"Error loading model: {e}")
                return False
        return False

    def predict(self, input_data: dict) -> tuple[bool, float]:
        """Make prediction for new input data"""
        try:
            if self.model is None and not self.load_model():
                raise ValueError("Model not trained or loaded")

            if self.feature_columns is None:
                raise ValueError("Model not properly trained - feature columns
not available")

            # Convert input to DataFrame with correct column order
            input_df = pd.DataFrame([input_data], columns=self.feature_columns)

            # Scale features
            input_scaled = self.scaler.transform(input_df)

            # Make prediction
            prediction = self.model.predict(input_scaled)[0]
            probability = self.model.predict_proba(input_scaled)[0][1]

            # Convert prediction back to original label
            prediction_label =
self.label_encoder.inverse_transform([prediction])[0]

            return bool(prediction_label == 'yes'), float(probability)
        except Exception as e:
            print(f"Error during prediction: {e}")
            raise
```

Recommendation System (recommender/rules.py):

```
from typing import Dict, List
from .guidelines import GUIDELINES

class WaterQualityRecommender:
    def __init__(self):
        self.guidelines = GUIDELINES

    def _get_severity_level(self, value: float, param: str, direction: str) ->
str:
```

```
        """Determine severity level based on value and parameter"""
        if param not in self.guidelines:
            return "unknown"

        param_guidelines = self.guidelines[param]
        severity_levels = param_guidelines["severity_levels"]

        if direction == "low":
            for level, threshold in severity_levels.items():
                if value <= threshold:
                    return level
        else:  # high
            for level, threshold in severity_levels.items():
                if value >= threshold:
                    return level

        return "normal"

    def _get_parameter_description(self, param: str) -> str:
        """Get parameter description and health implications"""
        descriptions = {
            "ph": "pH measures water's acidity or alkalinity. Extreme values can
affect water treatment efficiency and pipe corrosion.",
            "dissolved_oxygen": "Dissolved oxygen is crucial for aquatic life.
Low levels can cause fish kills and anaerobic conditions.",
            "conductivity": "Conductivity indicates water's ability to conduct
electricity, reflecting dissolved solids content.",
            "bod": "Biochemical Oxygen Demand measures organic matter content.
High BOD indicates poor water quality.",
            "nitrate": "Nitrate levels above 10 mg/L can cause methemoglobinemia
(blue baby syndrome) in infants.",
            "fecal_coliform": "Fecal coliform indicates potential presence of
disease-causing organisms from human/animal waste.",
            "total_coliform": "Total coliform indicates overall microbial water
quality and potential contamination."
        }
        return descriptions.get(param, "No description available")

    def _get_health_implications(self, param: str, severity: str) -> List[str]:
        """Get health implications based on parameter and severity"""
        implications = {
            "ph": {
                "mild": ["Slight irritation to eyes and skin", "Reduced
effectiveness of disinfection"],
                "moderate": ["Increased corrosion of pipes", "Reduced
effectiveness of water treatment"],
                "severe": ["Significant corrosion of infrastructure", "Potential
health risks from heavy metal leaching"],
                "critical": ["Immediate health risks", "Severe infrastructure
damage"]
            },
            "dissolved_oxygen": {
                "mild": ["Stress on aquatic life", "Reduced water quality"],
                "moderate": ["Fish kills possible", "Anaerobic conditions
```

```
developing"],
                "severe": ["Mass fish kills", "Severe ecosystem damage"],
                "critical": ["Complete ecosystem collapse", "Production of toxic
gases"]
            },
            "conductivity": {
                "mild": ["Slight taste changes", "Minor scaling in pipes"],
                "moderate": ["Increased scaling", "Reduced effectiveness of
treatment"],
                "severe": ["Severe scaling", "Potential health risks from high
mineral content"],
                "critical": ["Immediate health risks", "Infrastructure damage"]
            },
            "bod": {
                "mild": ["Slight odor issues", "Minor water quality
degradation"],
                "moderate": ["Significant odor problems", "Reduced oxygen
levels"],
                "severe": ["Severe water quality issues", "Potential health
risks"],
                "critical": ["Immediate health risks", "Complete water quality
failure"]
            },
            "nitrate": {
                "mild": ["Slight risk to sensitive populations", "Minor water
quality issues"],
                "moderate": ["Risk to infants and pregnant women", "Potential
health impacts"],
                "severe": ["Significant health risks", "Potential for
methemoglobinemia"],
                "critical": ["Immediate health risks", "Life-threatening
conditions possible"]
            },
            "fecal_coliform": {
                "mild": ["Low risk of waterborne illness", "Minor
contamination"],
                "moderate": ["Moderate risk of illness", "Significant
contamination"],
                "severe": ["High risk of illness", "Severe contamination"],
                "critical": ["Immediate health risks", "Outbreak potential"]
            },
            "total_coliform": {
                "mild": ["Low risk of contamination", "Minor water quality
issues"],
                "moderate": ["Moderate risk of contamination", "Significant
water quality issues"],
                "severe": ["High risk of contamination", "Severe water quality
issues"],
                "critical": ["Immediate health risks", "Outbreak potential"]
            }
        }
        return implications.get(param, {}).get(severity, ["Unknown health
implications"])
```

```
    def generate_recommendations(self, input_values: Dict[str, float]) ->
Dict[str, List[Dict]]:
        """Generate comprehensive recommendations based on input values and WHO
guidelines"""
        recommendations = {
            "immediate": [],
            "short_term": [],
            "long_term": [],
            "preventive": []
        }

        # Convert string values to numeric values
        value_mapping = {
            "low": 0.0,
            "high": 1000.0,  # Use appropriate high value based on parameter
            "normal": 50.0   # Use appropriate normal value based on parameter
        }

        for param, value in input_values.items():
            try:
                # Convert string values to numeric
                if isinstance(value, str):
                    value = value_mapping.get(value.lower(), 0.0)

                # Check if parameter exists in guidelines
                if param not in self.guidelines:
                    print(f"Warning: No guidelines found for parameter {param}")
                    continue

                param_guidelines = self.guidelines[param]
                min_val, max_val = param_guidelines["range"]

                # Determine if value is low or high
                if value < min_val:
                    direction = "low"
                    severity = self._get_severity_level(value, param, direction)
                elif value > max_val:
                    direction = "high"
                    severity = self._get_severity_level(value, param, direction)
                else:
                    continue  # Value is within acceptable range

                # Get measures for the current direction and severity
                measures = param_guidelines["measures"][direction]

                # Get parameter description and health implications
                description = self._get_parameter_description(param)
                health_implications = self._get_health_implications(param,
severity)

                # Add recommendations only for priority levels that have
specific measures
                for priority in ["immediate", "short_term", "long_term",
"preventive"]:
```

```
                    if priority in measures and measures[priority] and
len(measures[priority]) > 0:
                        for action in measures[priority]:
                            recommendation = {
                                "parameter": param,
                                "severity": severity,
                                "description": description,
                                "health_implications": health_implications,
                                "action": action,
                                "priority": priority,
                                "current_value": value,
                                "acceptable_range": [min_val, max_val]
                            }
                            recommendations[priority].append(recommendation)

            except Exception as e:
                print(f"Error processing {param}: {str(e)}")
                continue

        return recommendations
```

Database Models (database/models.py):

```
from sqlalchemy import Column, Integer, Float, String, DateTime, ForeignKey,
Boolean
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func
from database.config import Base

class WaterQualityMeasurement(Base):
    __tablename__ = "water_quality_measurements"

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey("users.id"))
    latitude = Column(Float)
    longitude = Column(Float)
    temperature = Column(Float)
    dissolved_oxygen = Column(Float)
    ph = Column(Float)
    conductivity = Column(Float)
    bod = Column(Float)
    nitrate = Column(Float)
    fecal_coliform = Column(Float)
    total_coliform = Column(Float)
    timestamp = Column(DateTime(timezone=True), server_default=func.now())

    predictions = relationship("WaterQualityPrediction",
back_populates="measurement")
    recommendations = relationship("Recommendation",
back_populates="measurement")
    user = relationship("User", back_populates="measurements")

class WaterQualityPrediction(Base):
    __tablename__ = "water_quality_predictions"
```

```
    id = Column(Integer, primary_key=True, index=True)
    measurement_id = Column(Integer,
ForeignKey("water_quality_measurements.id"))
    is_potable = Column(Boolean)
    confidence = Column(Float)
    wqi_value = Column(Float)
    quality_category = Column(String)
    timestamp = Column(DateTime(timezone=True), server_default=func.now())

    measurement = relationship("WaterQualityMeasurement",
back_populates="predictions")

class Recommendation(Base):
    __tablename__ = "recommendations"

    id = Column(Integer, primary_key=True, index=True)
    measurement_id = Column(Integer,
ForeignKey("water_quality_measurements.id"))
    parameter = Column(String)
    severity = Column(String)
    priority = Column(String)
    description = Column(String)
    estimated_cost = Column(Float, nullable=True)
    implementation_timeframe = Column(String, nullable=True)
    timestamp = Column(DateTime(timezone=True), server_default=func.now())

    measurement = relationship("WaterQualityMeasurement",
back_populates="recommendations")

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    # Relationships
    measurements = relationship("WaterQualityMeasurement",
back_populates="user")
```