File: __init__.py

File: bot\faq_bot.py

```
import json
from datetime import datetime
from typing import Dict, List
import sys
import os

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from translation.simple_translation_service import SimpleTranslationService

class SimpleFAQBot:
    def __init__(self):
        self.translator = SimpleTranslationService()
        self.conversations = {}

        # Load comprehensive BillMart knowledge base from JSON
        self.knowledge_base = self.load_knowledge_base()
        self.faq_database = self.create_faq_database()

[LINE TOO LONG TO DISPLAY]

    def load_knowledge_base(self):
        """Load BillMart knowledge base from JSON file"""
        try:
            json_path = os.path.join("data", "billmart_complete_knowledge.json")
            with open(json_path, "r", encoding="utf-8") as f:
                return json.load(f)
        except FileNotFoundError:
[LINE TOO LONG TO DISPLAY]
            return self.get_fallback_knowledge()
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return self.get_fallback_knowledge()

    def get_fallback_knowledge(self):
        """Fallback knowledge if JSON file not available"""
        return {
            "company_info": {
                "basic": {
                    "name": "BillMart FinTech Private Limited",
                    "business_model": "Digital marketplace connecting borrowers with financers",
                }
            },
            "general_responses": {
                 "default": "I can help you with BillMart's services. Please ask about our products or company
information."
            },
        }

    def create_faq_database(self):
        """Create searchable FAQ database from knowledge base - NO HARDCODING"""
        faq_db = {}

        try:
            # Company information - Safe access with .get()
            if "company_info" in self.knowledge_base:
                company = self.knowledge_base["company_info"]
                if "basic" in company:
                    basic = company["basic"]
                    faq_db["company information"] = (
                        f"{basic.get('name', 'BillMart')} - {basic.get('business_model', 'Fintech platform')}"
                    )
                    faq_db["billmart"] = basic.get(
                        "description",
                        basic.get("business_model", "Digital lending platform"),
                    )
                    faq_db["about billmart"] = basic.get(
```

```
                    "description",
                    basic.get("business_model", "Digital lending platform"),
                )

            # Contact information
            if "contact" in company:
                contact = company["contact"]
                phone = contact.get("phone", "+91 93269 46663")
                email = contact.get("email", {}).get("support", "care@billmart.com")
                faq_db["contact information"] = f"Call {phone} or email {email}"

        # Products - Safe access
        if "products" in self.knowledge_base:
            products = self.knowledge_base["products"]

            # Business products
            if "business_products" in products:
                bp = products["business_products"]

                if "supply_chain_finance" in bp:
                    scf = bp["supply_chain_finance"]
                    desc = scf.get("description", "Supply Chain Finance")
                    min_amt = scf.get("minimum_amount", "Rs. 50,000")
                    req = scf.get(
                        "requirements", "GST bills not older than 3 months"
                    )
                    faq_db["supply chain finance"] = (
                        f"{desc}. Minimum: {min_amt}, Requirements: {req}"
                    )

            # Individual products
            if "individual_products" in products:
                ip = products["individual_products"]

                if "empcash" in ip:
                    emp = ip["empcash"]
                    desc = emp.get("description", "Employee salary advance")
                    target = emp.get("target", "Salaried employees")
                    limit = emp.get("amount_limit", "Up to 50% of salary")
                    faq_db["empcash"] = f"{desc}. Target: {target}, Limit: {limit}"

                if "gigcash" in ip:
                    gig = ip["gigcash"]
                    desc = gig.get("description", "Gig worker financing")
                    target = gig.get("target", "Gig workers")
                    limit = gig.get("amount_limit", "Up to 50% of earnings")
                    faq_db["gigcash"] = f"{desc}. Target: {target}, Limit: {limit}"

        # Detailed FAQs - Direct mapping from JSON
        if "detailed_faqs" in self.knowledge_base:
            detailed = self.knowledge_base["detailed_faqs"]

            for category, faqs in detailed.items():
                if isinstance(faqs, dict):
                    for faq_key, faq_answer in faqs.items():
                        # Convert underscore keys to searchable phrases
                        search_key = faq_key.replace("_", " ")
                        faq_db[search_key] = faq_answer

        # General responses
        if "general_responses" in self.knowledge_base:
            faq_db.update(self.knowledge_base["general_responses"])

        print(
[LINE TOO LONG TO DISPLAY]
        )
        return faq_db

    except Exception as e:
```

```
[LINE TOO LONG TO DISPLAY]
            return {
                "default": "I can help you with BillMart's services. What would you like to know?"
            }

    def get_conversation_context(self, user_id: str, limit: int = 3) -> str:
        """Get recent conversation context for better responses"""
        if user_id not in self.conversations:
            return ""
        recent_messages = self.conversations[user_id][-limit * 2 :]
        context_parts = []
        for msg in recent_messages:
            if msg["type"] == "user":
                user_message = msg.get("original", msg.get("message", "Unknown"))
                context_parts.append(f"User previously asked: {user_message}")
            else:
                bot_message = msg.get(
                    "translated", msg.get("english", msg.get("message", "Unknown"))
                )
                context_parts.append(f"Bot responded: {bot_message[:100]}...")
        return " | ".join(context_parts)

    def find_faq_response(self, question: str, context: str = "") -> str:
        """Enhanced FAQ response with better intent recognition"""
        question_lower = question.lower()

        # Check for special commands first
        if any(
            word in question_lower for word in ["end chat", "goodbye", "bye", "finish"]
        ):
            return "END_CHAT_TRIGGER"

        if any(
            word in question_lower
            for word in ["demo", "connect", "talk to someone", "human", "executive"]
        ):
            return "REQUEST_DEMO_TRIGGER"

        # Enhanced intent recognition with scoring
        intent_scores = {}

        # Define intent keyword groups with priorities
        intent_keywords = {
            "supply_chain_finance": {
                "keywords": [
                    "supply chain finance",
                    "scf",
                    "b-scf",
                    "invoice financing",
                    "bill discounting",
                    "vendor financing",
                ],
                "response_key": "supply chain finance",
            },
            "insurance_claim_finance": {
                "keywords": [
                    "insurance claim finance",
                    "icf",
                    "hospital financing",
                    "medical financing",
                    "claim delay",
                ],
                "response_key": "insurance claim finance",
            },
            "empcash": {
                "keywords": [
                    "empcash",
                    "emp cash",
                    "employee cash",
```

```
            "salary advance",
            "employee advance",
        ],
        "response_key": "empcash",
    },
    "gigcash": {
        "keywords": [
            "gigcash",
            "gig cash",
            "freelancer financing",
            "gig worker",
            "uber",
            "zomato",
            "swiggy",
        ],
        "response_key": "gigcash",
    },
    "imark": {
        "keywords": [
            "imark",
            "ai rating",
            "msme rating",
            "credit rating",
            "business rating",
        ],
        "response_key": "imark",
    },
    "company_info": {
        "keywords": ["billmart", "company", "about", "who are you"],
        "response_key": "company information",
    },
    "services": {
        "keywords": [
            "services",
            "products",
            "offerings",
            "what do you offer",
            "solutions",
        ],
        "response_key": "services offered",
    },
    "contact": {
        "keywords": ["contact", "phone", "email", "address", "office"],
        "response_key": "contact information",
    },
}

# Score each intent based on keyword matches
for intent_name, intent_data in intent_keywords.items():
    score = 0
    for keyword in intent_data["keywords"]:
        if keyword in question_lower:
            # Longer matches get higher scores
            score += len(keyword) * 2
            # Exact matches get bonus points
            if question_lower.strip() == keyword:
                score += 10

    if score > 0:
        intent_scores[intent_name] = {
            "score": score,
            "response_key": intent_data["response_key"],
        }

# Return highest scoring intent
if intent_scores:
    best_intent = max(
        intent_scores, key=lambda x: intent_scores[x]["score"]
    )
```

```
            response_key = intent_scores[best_intent]["response_key"]

            # Get response from your JSON knowledge base
            if response_key in self.faq_database:
                return self.faq_database[response_key]

        # Fallback to original keyword matching for other intents
        for keyword, answer in self.faq_database.items():
            if keyword in question_lower:
                return answer

        # Context-aware responses
        if context:
            if "billmart" in context.lower() and any(
                word in question_lower
                for word in ["document", "paper", "need", "require"]
            ):
                reg_info = self.faq_database.get(
                    "how to register",
                    "KYC, GSTIN, CIN, Bank Details, MSME Registration and required documents",
                )
                return f"For BillMart registration: {reg_info}"

            if "empcash" in context.lower() and any(
                word in question_lower for word in ["how", "process", "work"]
            ):
[LINE TOO LONG TO DISPLAY]

        # Default response
        return self.faq_database.get(
            "default",
              "I can help you with BillMart's products and services including Supply Chain Finance, EmpCash,
GigCash, company information, and registration process. What would you like to know?",
        )

    def handle_end_chat(self, user_id: str) -> str:
        """Clear conversation when user ends the chat"""
        if user_id in self.conversations:
            del self.conversations[user_id]
[LINE TOO LONG TO DISPLAY]
        return (
            "Thank you for chatting with BillMart! Feel free to reach out anytime at "
            "+91 93269 46663 or care@billmart.com. Have a great day!"
        )

    def handle_request_demo(self, user_id: str, user_message: str = "") -> str:
        """Handle demo request and notify team"""
        demo_request = {
            "user_id": user_id,
            "timestamp": datetime.now().isoformat(),
            "user_message": user_message,
            "conversation_history": self.conversations.get(user_id, []),
        }

        self.save_demo_request(demo_request)

        return (
            "I'll connect you with our BillMart team for a personalized demo! "
            "Please call +91 93269 46663 or email care@billmart.com. "
            "Our team has been notified and will reach out to you shortly."
        )

    def save_demo_request(self, demo_request: dict):
        """Save demo request for team follow-up"""
        try:
            demo_file = "data/demo_requests.json"

            try:
                with open(demo_file, "r", encoding="utf-8") as f:
```

```
                requests = json.load(f)
            except FileNotFoundError:
                requests = []

            requests.append(demo_request)

            os.makedirs(os.path.dirname(demo_file), exist_ok=True)
            with open(demo_file, "w", encoding="utf-8") as f:
                json.dump(requests, f, indent=2, ensure_ascii=False)

[LINE TOO LONG TO DISPLAY]

        except Exception as e:
[LINE TOO LONG TO DISPLAY]

    def handle_message(self, user_id: str, message: str) -> str:
        """Process user message with translation and conversation memory"""
        try:
            if user_id not in self.conversations:
                self.conversations[user_id] = []

            detected_lang = self.translator.detect_language(message)

            if detected_lang != "english":
                english_message = self.translator.translate_text(
                    message, detected_lang, "english"
                )
            else:
                english_message = message

            self.conversations[user_id].append(
                {
                    "type": "user",
                    "original": message,
                    "english": english_message,
                    "language": detected_lang,
                    "timestamp": datetime.now().isoformat(),
                }
            )

            context = self.get_conversation_context(user_id)
            english_response = self.find_faq_response(english_message, context)

            if english_response == "END_CHAT_TRIGGER":
                return self.handle_end_chat(user_id)
            elif english_response == "REQUEST_DEMO_TRIGGER":
                return self.handle_request_demo(user_id, message)

            if detected_lang != "english":
                final_response = self.translator.translate_text(
                    english_response, "english", detected_lang
                )
            else:
                final_response = english_response

            self.conversations[user_id].append(
                {
                    "type": "bot",
                    "english": english_response,
                    "translated": final_response,
                    "language": detected_lang,
                    "timestamp": datetime.now().isoformat(),
                }
            )

            if len(self.conversations[user_id]) > 20:
                self.conversations[user_id] = self.conversations[user_id][-20:]

            return final_response
```

```
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return "I'm having trouble processing your request. Please try again."


# ====================================================
# COMPREHENSIVE TEST CODE (AT MODULE LEVEL - NOT INSIDE CLASS)
# ====================================================

if __name__ == "__main__":
[LINE TOO LONG TO DISPLAY]
    print("=" * 70)

    # Initialize bot
    try:
        bot = SimpleFAQBot()
[LINE TOO LONG TO DISPLAY]
    except Exception as e:
[LINE TOO LONG TO DISPLAY]
        exit(1)

    # Test 1: Knowledge Base Loading
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
    print(f"   FAQ entries created: {len(bot.faq_database)}")

    # Show some FAQ entries
    print(f"   Sample FAQ keys: {list(bot.faq_database.keys())[:5]}")

    if "company_info" in bot.knowledge_base:
[LINE TOO LONG TO DISPLAY]
    if "products" in bot.knowledge_base:
[LINE TOO LONG TO DISPLAY]
    if "detailed_faqs" in bot.knowledge_base:
[LINE TOO LONG TO DISPLAY]

    # Test 2: FAQ Response Tests (Multiple questions)
[LINE TOO LONG TO DISPLAY]
    print("-" * 50)

    test_questions = [
        "What is BillMart?",
        "Tell me about EmpCash",
        "What is supply chain finance?",
        "How do I contact you?",
        "What are your charges?",
        "Tell me about registration",
        "What is GigCash?",
        "How much can I borrow?",
    ]

    test_user = "test_user_123"

    for i, question in enumerate(test_questions, 1):
[LINE TOO LONG TO DISPLAY]
        try:
            response = bot.handle_message(test_user, question)
            print(
[LINE TOO LONG TO DISPLAY]
            )
        except Exception as e:
[LINE TOO LONG TO DISPLAY]

    # Test 3: Special Commands
[LINE TOO LONG TO DISPLAY]
    print("-" * 30)

[LINE TOO LONG TO DISPLAY]
```

```
    demo_response = bot.handle_message(test_user, "I want a demo")
[LINE TOO LONG TO DISPLAY]

[LINE TOO LONG TO DISPLAY]
    end_response = bot.handle_message(test_user, "End chat")
[LINE TOO LONG TO DISPLAY]

    # Test 4: Multilingual Test (if translation works)
[LINE TOO LONG TO DISPLAY]
    print("-" * 25)

    try:
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
        hindi_response = bot.handle_message("hindi_user", hindi_question)
[LINE TOO LONG TO DISPLAY]
    except Exception as e:
[LINE TOO LONG TO DISPLAY]

    # Test 5: Context Test
[LINE TOO LONG TO DISPLAY]
    print("-" * 25)

    context_user = "context_test_user"

[LINE TOO LONG TO DISPLAY]
    response1 = bot.handle_message(context_user, "What is BillMart?")
[LINE TOO LONG TO DISPLAY]

[LINE TOO LONG TO DISPLAY]
    response2 = bot.handle_message(
        context_user, "What documents do I need for that?"
    )
[LINE TOO LONG TO DISPLAY]

    # Test 6: System Statistics
[LINE TOO LONG TO DISPLAY]
    print("-" * 20)
    print(f"   Active conversations: {len(bot.conversations)}")
    print(f"   FAQ database size: {len(bot.faq_database)} entries")

    # Check demo requests
    if os.path.exists("data/demo_requests.json"):
        try:
            with open("data/demo_requests.json", "r") as f:
                demo_requests = json.load(f)
            print(f"   Demo requests logged: {len(demo_requests)}")
        except:
            print(f"   Demo requests: File exists but couldn't read")
    else:
        print(f"   Demo requests: No file created yet")

[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
    print("=" * 70)
```

File: bot\multilingual_rasa_wrapper.py

```
# src/bot/multilingual_rasa_wrapper.py
import asyncio
import os
import sys
from typing import Dict, Optional, List
from datetime import datetime
import logging

# Configure logging
logging.basicConfig(
    format='%(asctime)s [%(levelname)s] %(message)s',
    level=logging.INFO
)
logger = logging.getLogger(__name__)

# Rasa imports
from rasa.core.agent import Agent
from rasa.shared.core.trackers import DialogueStateTracker
from rasa.shared.core.domain import Domain

# Translation service
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from translation.simple_translation_service import SimpleTranslationService

class MultilingualRasaWrapper:
    """Production-grade multilingual wrapper with critical fixes"""

    def __init__(self, model_path: Optional[str] = None):
[LINE TOO LONG TO DISPLAY]

        # Initialize translation service with fallback
        self.translator = self._initialize_translator()

        # Load Rasa agent with enhanced error handling
        self.rasa_agent = self._load_rasa_agent(model_path)

        # Conversation tracking with slots
        self.conversations = {}

    def _initialize_translator(self):
        """Initialize translation service with fallback"""
        try:
            translator = SimpleTranslationService()
[LINE TOO LONG TO DISPLAY]
            return translator
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return None

    def _load_rasa_agent(self, model_path: Optional[str]):
        """Load Rasa agent with production-grade checks"""
        try:
            model_path = model_path or self._find_latest_model()
            agent = Agent.load(model_path)

            # Validate critical components
            if not agent.domain:
                raise ValueError("Loaded agent has no domain")
            if not agent.processor:
                raise ValueError("Agent processor not initialized")

[LINE TOO LONG TO DISPLAY]
            return agent
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            raise
```

```
async def process_message_async(self, user_id: str, message: str) -> Dict:
    """Production-grade message processing pipeline"""
    try:
        # Language detection with fallback
        detected_lang = self._detect_language_with_fallback(message)

        # Translation to English
        english_input = self._translate_input(message, detected_lang)

        # Get Rasa response with context
        tracker = await self._get_tracker(user_id)
        rasa_response = await self.rasa_agent.handle_text(
            english_input, sender_id=user_id, tracker=tracker
        )

        # Extract and translate response
        response_text = self._extract_response_text(rasa_response)
        translated_response = self._translate_response(
            response_text, detected_lang
        )

        # Update conversation context
        self._update_conversation_context(
            user_id, message, translated_response, detected_lang, tracker
        )

        return self._format_final_response(
            translated_response, detected_lang, rasa_response
        )

    except Exception as e:
        logger.error(f"Processing error: {e}", exc_info=True)
        return self._error_response()

def _detect_language_with_fallback(self, text: str) -> str:
    """Robust language detection with fallback to English"""
    if not self.translator:
        return 'en'

    try:
        return self.translator.detect_language(text) or 'en'
    except:
        return 'en'

def _translate_input(self, text: str, source_lang: str) -> str:
    """Translate user input to English"""
    if source_lang == 'en' or not self.translator:
        return text

    try:
        return self.translator.translate_text(text, source_lang, 'en')
    except:
        return text  # Fallback to original text

def _translate_response(self, text: str, target_lang: str) -> str:
    """Translate bot response to user's language"""
    if target_lang == 'en' or not self.translator:
        return text

    try:
        return self.translator.translate_text(text, 'en', target_lang)
    except:
        return text  # Fallback to English

async def _get_tracker(self, user_id: str) -> DialogueStateTracker:
    """Get conversation tracker with slot management"""
    tracker = await self.rasa_agent.tracker_store.get_or_create_tracker(user_id)
```

```
        # Initialize slots if missing
        if 'user_language' not in tracker.slots:
            tracker.slots['user_language'] = None
        if 'conversation_context' not in tracker.slots:
            tracker.slots['conversation_context'] = None

        return tracker

    def _extract_response_text(self, rasa_response: List) -> str:
        """Robust response extraction with markdown support"""
        if not rasa_response:
            return "I'm having trouble understanding. Could you please rephrase?"

        # Handle multiple response types
        texts = []
        for resp in rasa_response:
            if isinstance(resp, dict):
                text = resp.get('text', '')
                if 'buttons' in resp:
                    text += self._format_buttons(resp['buttons'])
                texts.append(text)
            elif hasattr(resp, 'text'):
                texts.append(resp.text)

        return '\n'.join(texts).strip()

    def _format_buttons(self, buttons: List) -> str:
        """Format buttons for non-rich clients"""
        return '\n'.join([f"- {btn['title']}" for btn in buttons])

    def _update_conversation_context(self, user_id: str, user_msg: str, bot_resp: str,
                                     language: str, tracker: DialogueStateTracker):
        """Maintain conversation context in slots"""
        # Update language slot
        tracker.slots['user_language'] = language

        # Maintain last 3 messages as context
        context = tracker.slots['conversation_context'] or []
        context.append({
            'user': user_msg,
            'bot': bot_resp,
            'timestamp': datetime.now().isoformat()
        })
        tracker.slots['conversation_context'] = context[-3:]  # Keep last 3 exchanges

        # Persist updated tracker
        self.rasa_agent.tracker_store.save(tracker)

    def _format_final_response(self, text: str, lang: str, rasa_response: List) -> Dict:
        """Structure final response with debug info"""
        return {
            'text': text,
            'language': lang,
            'intent': self._get_top_intent(rasa_response),
            'confidence': self._get_confidence(rasa_response),
            'success': True,
            'timestamp': datetime.now().isoformat()
        }

    def _get_top_intent(self, rasa_response: List) -> Optional[str]:
        """Extract top intent from response"""
        for resp in rasa_response:
            if isinstance(resp, dict):
                intent = resp.get('intent', {}).get('name')
                if intent:
                    return intent
        return None

    def _get_confidence(self, rasa_response: List) -> float:
```

```
    """Extract highest confidence score"""
    confidences = [
        r.get('intent', {}).get('confidence', 0)
        for r in rasa_response
        if isinstance(r, dict)
    ]
    return max(confidences) if confidences else 0.0

def _error_response(self) -> Dict:
    """Standard error response"""
    return {
        'text': "I'm experiencing technical difficulties. Please try again later.",
        'language': 'en',
        'intent': None,
        'confidence': 0.0,
        'success': False,
        'timestamp': datetime.now().isoformat()
    }

# Public interface
async def process_message(self, user_id: str, message: str) -> Dict:
    """Public async processing method"""
    return await self.process_message_async(user_id, message)
```

File: bot\simple_multilingual_wrapper.py

```
#!/usr/bin/env python3
"""
PRODUCTION MULTILINGUAL RASA WRAPPER
Complete implementation with AI4Bharat translation and proper main method
"""
import asyncio
import argparse
import sys
import os
from typing import Dict, List, Optional
import logging

# Add project root to path
sys.path.append(os.path.dirname(os.path.dirname(os.path.dirname(__file__))))

try:
    from rasa.core.agent import Agent
    import uvicorn
    from fastapi import FastAPI, HTTPException
    from fastapi.middleware.cors import CORSMiddleware
    from pydantic import BaseModel

    # Translation services - AI4Bharat + langdetect
    from langdetect import detect
    from ai4bharat.transliteration import XlitEngine

    # AI4Bharat imports
    import torch
    from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

except ImportError as e:
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
    sys.exit(1)

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s [%(levelname)s] %(name)s: %(message)s')
logger = logging.getLogger(__name__)

class MessageRequest(BaseModel):
    sender: str
    message: str
    metadata: dict = {}

class AI4BharatTranslator:
    """AI4Bharat-based translation service for Indian languages"""

    def __init__(self):
        """Initialize AI4Bharat translation models"""
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
[LINE TOO LONG TO DISPLAY]

        # Initialize models
        self.model_name = "ai4bharat/indictrans2-indic-en-1B"
        self.reverse_model_name = "ai4bharat/indictrans2-en-indic-1B"

        try:
            # Load tokenizers and models
[LINE TOO LONG TO DISPLAY]

            # Indic to English
            self.tokenizer_indic_en = AutoTokenizer.from_pretrained(self.model_name, trust_remote_code=True)
                            self.model_indic_en   =   AutoModelForSeq2SeqLM.from_pretrained(self.model_name,
trust_remote_code=True)
            self.model_indic_en.to(self.device)
```

```
            # English to Indic
                                self.tokenizer_en_indic   =   AutoTokenizer.from_pretrained(self.reverse_model_name,
trust_remote_code=True)
                            self.model_en_indic   =   AutoModelForSeq2SeqLM.from_pretrained(self.reverse_model_name,
trust_remote_code=True)
            self.model_en_indic.to(self.device)

[LINE TOO LONG TO DISPLAY]

        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            self.model_indic_en = None
            self.model_en_indic = None

    def detect_language(self, text: str) -> str:
        """Detect language using langdetect"""
        try:
            detected = detect(text.strip())
[LINE TOO LONG TO DISPLAY]
            return detected
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return 'en'

    def translate_indic_to_english(self, text: str, source_lang: str) -> str:
        """Translate from Indian language to English using AI4Bharat"""
        if not self.model_indic_en or source_lang == 'en':
            return text

        try:
            # Prepare input with language code
            input_text = f"{source_lang}: {text}"

            # Tokenize
              inputs = self.tokenizer_indic_en(input_text, return_tensors="pt", padding=True, truncation=True,
max_length=512)
            inputs = {k: v.to(self.device) for k, v in inputs.items()}

            # Generate translation
            with torch.no_grad():
                generated_tokens = self.model_indic_en.generate(
                    **inputs,
                    max_length=512,
                    num_beams=5,
                    num_return_sequences=1,
                    temperature=1.0,
                    do_sample=False
                )

            # Decode result
            translated = self.tokenizer_indic_en.decode(generated_tokens[0], skip_special_tokens=True)

[LINE TOO LONG TO DISPLAY]
            return translated.strip()

        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return text

    def translate_english_to_indic(self, text: str, target_lang: str) -> str:
        """Translate from English to Indian language using AI4Bharat"""
        if not self.model_en_indic or target_lang == 'en':
            return text

        try:
            # Prepare input with target language code
            input_text = f"en: {text}"

            # Tokenize
```

```
              inputs = self.tokenizer_en_indic(input_text, return_tensors="pt", padding=True, truncation=True,
max_length=512)
            inputs = {k: v.to(self.device) for k, v in inputs.items()}

            # Generate translation
            with torch.no_grad():
                generated_tokens = self.model_en_indic.generate(
                    **inputs,
                    max_length=512,
                    num_beams=5,
                    num_return_sequences=1,
                    temperature=1.0,
                    do_sample=False,
                    forced_bos_token_id=self.tokenizer_en_indic.lang_code_to_id[target_lang]
                )

            # Decode result
            translated = self.tokenizer_en_indic.decode(generated_tokens[0], skip_special_tokens=True)

[LINE TOO LONG TO DISPLAY]
            return translated.strip()

        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return text

class MultilingualRasaWrapper:
    """Production-grade multilingual wrapper with AI4Bharat translation"""

    def __init__(self, model_path: str = None):
        """Initialize with AI4Bharat translation services"""
        self.model_path = model_path or self._find_latest_model()
        self.rasa_agent = None

        # Initialize AI4Bharat translator
        self.translator = AI4BharatTranslator()

        # Supported language mappings
        self.supported_languages = {
            'hi': 'hi',  # Hindi
            'gu': 'gu',  # Gujarati
            'ta': 'ta',  # Tamil
            'te': 'te',  # Telugu
            'bn': 'bn',  # Bengali
            'mr': 'mr',  # Marathi
            'en': 'en'   # English
        }

        self.load_agent()

    def _find_latest_model(self):
        """Find latest model file"""
        models_dir = "models"
        if not os.path.exists(models_dir):
            raise FileNotFoundError("Models directory not found")

        model_files = [f for f in os.listdir(models_dir) if f.endswith('.tar.gz')]
        if not model_files:
            raise FileNotFoundError("No model files found")

        latest_model = max(model_files, key=lambda f: os.path.getctime(os.path.join(models_dir, f)))
        return os.path.join(models_dir, latest_model)

    def load_agent(self):
        """Load Rasa agent"""
        try:
[LINE TOO LONG TO DISPLAY]
            self.rasa_agent = Agent.load(self.model_path)
[LINE TOO LONG TO DISPLAY]
```

```
        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            raise

    async def process_message(self, sender_id: str, message: str) -> Dict:
        """Complete multilingual processing pipeline with AI4Bharat"""
        try:
            # Step 1: Detect language
            detected_lang = self.translator.detect_language(message)
[LINE TOO LONG TO DISPLAY]

            # Step 2: Translate to English for Rasa processing
            english_input = message
            if detected_lang != 'en' and detected_lang in self.supported_languages:
                english_input = self.translator.translate_indic_to_english(message, detected_lang)
[LINE TOO LONG TO DISPLAY]

            # Step 3: Get Rasa response
            rasa_responses = await self.rasa_agent.handle_text(english_input, sender_id=sender_id)

            # Step 4: Extract response text
            response_text = self._extract_response_text(rasa_responses)
[LINE TOO LONG TO DISPLAY]

            # Step 5: Translate back to user's language
            final_response = response_text
            if detected_lang != 'en' and detected_lang in self.supported_languages:
                final_response = self.translator.translate_english_to_indic(response_text, detected_lang)
[LINE TOO LONG TO DISPLAY]

            return {
                "recipient_id": sender_id,
                "text": final_response,
                "language": detected_lang,
                "original_message": message,
                "english_processed": english_input,
                "success": True
            }

        except Exception as e:
[LINE TOO LONG TO DISPLAY]
            return {
                "recipient_id": sender_id,
                "text": "Sorry, I'm having technical difficulties. Please try again.",
                "language": "en",
                "success": False,
                "error": str(e)
            }

    def _extract_response_text(self, responses: List) -> str:
        """Extract text from Rasa responses"""
        if not responses:
            return "I'm not sure how to respond to that."

        texts = []
        for response in responses:
            if isinstance(response, dict) and 'text' in response:
                texts.append(response['text'])
            elif hasattr(response, 'text'):
                texts.append(response.text)

        return ' '.join(texts) if texts else "I'm not sure how to respond to that."

# FastAPI app setup
app = FastAPI(title="BillMart Multilingual Assistant with AI4Bharat")

# CORS middleware
app.add_middleware(
    CORSMiddleware,
```

```
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Global wrapper instance
wrapper = None

@app.on_event("startup")
async def startup_event():
    global wrapper
[LINE TOO LONG TO DISPLAY]
    try:
        wrapper = MultilingualRasaWrapper()
[LINE TOO LONG TO DISPLAY]
    except Exception as e:
[LINE TOO LONG TO DISPLAY]
        raise

@app.post("/webhooks/rest/webhook")
async def webhook(request: MessageRequest):
    """Main webhook endpoint with AI4Bharat translation"""
    if not wrapper:
        raise HTTPException(status_code=500, detail="Wrapper not initialized")

    try:
        response = await wrapper.process_message(request.sender, request.message)
        return [response]  # Rasa expects array format
    except Exception as e:
[LINE TOO LONG TO DISPLAY]
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/")
async def health_check():
    """Health check endpoint"""
    return {
        "status": "running",
        "service": "BillMart Multilingual Assistant",
        "translation_service": "AI4Bharat IndicTrans2",
        "agent_loaded": wrapper and wrapper.rasa_agent is not None,
        "supported_languages": ["Hindi", "English", "Gujarati", "Tamil", "Telugu", "Bengali", "Marathi"]
    }

def main():
    """Main entry point - THE MISSING PIECE!"""
    parser = argparse.ArgumentParser(description="BillMart Multilingual Rasa Assistant with AI4Bharat")
    parser.add_argument("--debug", action="store_true", help="Enable debug mode")
    parser.add_argument("--port", type=int, default=5005, help="Port to run on")
    parser.add_argument("--host", default="0.0.0.0", help="Host to bind to")

    args = parser.parse_args()

    if args.debug:
        logging.getLogger().setLevel(logging.DEBUG)

[LINE TOO LONG TO DISPLAY]
    logger.info("=" * 70)
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
    logger.info("=" * 70)

    try:
        uvicorn.run(app, host=args.host, port=args.port, log_level="info")
    except KeyboardInterrupt:
[LINE TOO LONG TO DISPLAY]
    except Exception as e:
```

```
[LINE TOO LONG TO DISPLAY]

if __name__ == "__main__":
    main()  # THE MISSING MAIN METHOD THAT WAS CAUSING THE ISSUE!
```

File: bot__init__.py

File: translation\simple_translation_service.py

```
import torch
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
from langdetect import detect

class SimpleTranslationService:
    def __init__(self):

        self.device = "cuda" if torch.cuda.is_available() else "cpu"

        # Lazy loading: Initialize as None, load when needed
        self.model = None
        self.tokenizer = None

[LINE TOO LONG TO DISPLAY]
        self.language_codes = {
            'hi': 'hin_Deva',      # Hindi - Devanagari script
            'bn': 'ben_Beng',      # Bengali - Bengali script
            'mr': 'mar_Deva',      # Marathi - Devanagari script
            'ta': 'tam_Taml',      # Tamil - Tamil script
            'te': 'tel_Telu',      # Telugu - Telugu script
            'gu': 'guj_Gujr',      # Gujarati - Gujarati script
            'kn': 'kan_Knda',      # Kannada - Kannada script
            'ml': 'mal_Mlym',      # Malayalam - Malayalam script
            'ur': 'urd_Arab',      # Urdu - Arabic script
            'or': 'ory_Orya'       # Odia - Oriya script
        }

    def detect_language(self, text):
        """
        Detect the language of input text and map to IndicTrans2 format.

        Args:
            text (str): Input text to analyze

        Returns:
            str: IndicTrans2 language code or 'english' if detection fails
        """
        try:
            # Use langdetect to identify language (returns ISO codes like 'hi', 'bn')
            detected = detect(text)

            # Map to IndicTrans2 format, default to 'english' if not found
            return self.language_codes.get(detected, 'english')
        except Exception:
            # Fallback to English if detection fails (empty text, special chars, etc.)
            return 'english'

    def load_translation_model(self):
        """load AI4Bharat IndicTrans2 translation model- heavy model"""
        if self.model is None:
            print("Loading IndicTrans2 model...")
            print("this may take some time to load 800mb model...")
            model_name = "ai4bharat/indictrans2-en-indic-dist-200M"
            self.tokenizer=AutoTokenizer.from_pretrained(
                model_name,trust_remote_code=True
            )
            self.model = AutoModelForSeq2SeqLM.from_pretrained(
                model_name,trust_remote_code=True,
                torch_dtype=torch.float16).to(self.device)
            print(" AI4Bharat Model loaded successfully!")
        else:
            print("Model already loaded, skipping...")

    def _get_language_code(self, language):
        """convert human readable text into IndicTrans2 code"""
        if language.lower()=='english':
```

```
            return 'eng_Latn'
        language_mapping={
            'hindi': 'hin_Deva',
            'hin_deva': 'hin_Deva',
            'bengali': 'ben_Beng',
            'marathi': 'mar_Deva',
            'tamil': 'tam_Taml',
            'telugu': 'tel_Telu',
            'gujarati': 'guj_Gujr',
            'kannada': 'kan_Knda',
            'malayalam': 'mal_Mlym',
            'urdu': 'urd_Arab',
            'odia': 'ory_Orya'
        }
        return language_mapping.get(language.lower(),'eng_Latn')
    def translate_text(self,text,source_lang, target_lang):

        self.load_translation_model()
        src_code=self._get_language_code(source_lang)
        tgt_code=self._get_language_code(target_lang)

        input_text=f"{src_code} {tgt_code} {text}"
        inputs = self.tokenizer(input_text,return_tensors="pt",padding=True,
                    max_length=512,truncation=True).to(self.device)
        with torch.no_grad():
            outputs=self.model.generate(**inputs,max_length=256,
                        num_beams=3,early_stopping=True,do_sample=False)
            translation=self.tokenizer.decode(outputs[0],skip_special_tokens=True,
                                              clean_up_tokenization_spaces=True)
        return translation.strip()




    def debug_gpu_info(self):
        """Debug method to display GPU availability and system information"""
        print("=== GPU Debug Information ===")
        print(f"PyTorch version: {torch.__version__}")
        print(f"CUDA available: {torch.cuda.is_available()}")

        if torch.cuda.is_available():
            print(f"CUDA version: {torch.version.cuda}")
            print(f"Number of GPUs: {torch.cuda.device_count()}")
            print(f"GPU name: {torch.cuda.get_device_name(0)}")
        else:
            print(" CUDA not available - possible reasons:")
            print("  1. PyTorch CPU-only version installed")
            print("  2. NVIDIA drivers not installed")
            print("  3. GPU not CUDA-compatible")
        print("==========================")

# Test code - only runs when script is executed directly
if __name__ == "__main__":
    print(" Translation Service - Development Testing")
    print("=" * 60)

    # Initialize the service
    service = SimpleTranslationService()

    # Display system information
    service.debug_gpu_info()

    # Test language detection first
    print(f"\n Language Detection Tests:")
    test_cases = [
        "Hello, how are you?",
[LINE TOO LONG TO DISPLAY]
[LINE TOO LONG TO DISPLAY]
    ]
```

```
    for text in test_cases:
        detected = service.detect_language(text)
[LINE TOO LONG TO DISPLAY]

    # Test AI Translation
    print(f"\n AI Translation Tests:")
    print("   Loading AI4Bharat IndicTrans2 model...")

    translation_tests = [
        ("Hello, how are you?", "english", "hindi"),
        ("What is your name?", "english", "tamil"),
        ("I need loan information", "english", "bengali"),
        ("Thank you very much", "english", "gujarati")
    ]

    for text, src_lang, tgt_lang in translation_tests:
        try:
            print(f"\n    Translating: '{text}'")
[LINE TOO LONG TO DISPLAY]

            translation = service.translate_text(text, src_lang, tgt_lang)
            print(f"      Result: '{translation}'")
            print("       Translation successful!")

        except Exception as e:
            print(f"       Translation failed: {e}")

    print(f"\n Final Service Status:")
    print(f"   Device: {service.device}")
    print(f"   Model loaded: {service.model is not None}")
    print(f"   Ready for production!")

    print("\n AI Translation Service Test Complete!")
```

File: translation__init__.py