**Autonomous Agents Implementation Guide**

**1. Personalization Agent (For Holiday Card & Gift App)**

// personalization-agent.js

export class PersonalizationAgent {

constructor(userData, preferenceHistory) {

this.userData = userData;

this.preferenceHistory = preferenceHistory;

this.recommendations = [];

}

async analyzePreferences() {

// Process user data and preference history

// Extract patterns and preferences

return {

colorPreferences: this.extractColorPreferences(),

stylePreferences: this.extractStylePreferences(),

occasionPreferences: this.extractOccasionPreferences()

};

}

async generateRecommendations() {

const preferences = await this.analyzePreferences();

// Use Claude API to generate personalized recommendations

const claudeResponse = await fetch('/api/claude/recommendations', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ preferences })

});

this.recommendations = await claudeResponse.json();

return this.recommendations;

}

extractColorPreferences() {

// Analyze past selections for color patterns

// Return array of preferred colors with confidence scores

}

extractStylePreferences() {

// Analyze style choices from history

}

extractOccasionPreferences() {

// Determine preference patterns for different occasions

}

}

// Usage

const agent = new PersonalizationAgent(currentUser, userHistory);

const recommendations = await agent.generateRecommendations();

**2. Product Sourcing Agent (For All Apps)**

// product-sourcing-agent.js

export class ProductSourcingAgent {

constructor(productCategory, budget, requirements) {

this.category = productCategory;

this.budget = budget;

this.requirements = requirements;

this.sources = [

{ name: 'API1', endpoint: '/api/products/source1' },

{ name: 'API2', endpoint: '/api/products/source2' },

// Add more product sources/APIs

];

}

async findProducts() {

const results = await Promise.all(

this.sources.map(source => this.querySource(source))

);

return this.rankAndFilterResults(results.flat());

}

async querySource(source) {

const response = await fetch(source.endpoint, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

category: this.category,

budget: this.budget,

requirements: this.requirements

})

});

const products = await response.json();

return products.map(p => ({ ...p, source: source.name }));

}

rankAndFilterResults(products) {

// Filter out products that don't meet requirements

const filtered = products.filter(p => this.meetsRequirements(p));

// Rank by relevance, rating, price, etc.

return filtered.sort((a, b) => this.calculateRelevanceScore(b) - this.calculateRelevanceScore(a));

}

meetsRequirements(product) {

// Check if product meets all requirements

return this.requirements.every(req => this.checkRequirement(product, req));

}

checkRequirement(product, requirement) {

// Check individual requirement

}

calculateRelevanceScore(product) {

// Calculate a relevance score based on how well the product

// matches the requirements, budget, etc.

}

}

// Usage

const agent = new ProductSourcingAgent('tarot-decks', { min: 20, max: 50 }, [

{ type: 'style', value: 'traditional' },

{ type: 'material', value: 'sustainable' }

]);

const products = await agent.findProducts();

**3. Research Agent (For Festival App)**

// research-agent.js

export class ResearchAgent {

constructor(topic, depth = 'medium') {

this.topic = topic;

this.depth = depth; // 'light', 'medium', 'deep'

this.findings = {};

}

async conductResearch() {

// Set research parameters based on depth

const queryCount = this.depth === 'light' ? 3 : this.depth === 'medium' ? 5 : 10;

// Generate related queries

const queries = await this.generateRelatedQueries();

// Select top queries based on depth

const selectedQueries = queries.slice(0, queryCount);

// Search for each query

const searchResults = await Promise.all(

selectedQueries.map(query => this.searchQuery(query))

);

// Process and synthesize results

this.findings = this.synthesizeFindings(searchResults);

return this.findings;

}

async generateRelatedQueries() {

// Use Claude to generate related search queries

const response = await fetch('/api/claude/related-queries', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ topic: this.topic })

});

return await response.json();

}

async searchQuery(query) {

// Use brave search API or similar

const response = await fetch('/api/search', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ query })

});

return await response.json();

}

synthesizeFindings(searchResults) {

// Process and combine search results

// Extract key insights, common themes, etc.

const allResults = searchResults.flat();

return {

topInsights: this.extractTopInsights(allResults),

trends: this.identifyTrends(allResults),

sources: this.extractSources(allResults),

summary: this.generateSummary(allResults)

};

}

extractTopInsights(results) {

// Extract the most important insights from results

}

identifyTrends(results) {

// Identify common trends and patterns

}

extractSources(results) {

// Extract and format source information

}

generateSummary(results) {

// Generate an executive summary of findings

}

}

// Usage

const agent = new ResearchAgent('EDM festival trends 2025', 'deep');

const research = await agent.conductResearch();

**4. Agent Manager (Orchestration Layer)**

// agent-manager.js

export class AgentManager {

constructor() {

this.agents = {};

this.tasks = [];

}

registerAgent(name, agentClass, defaultConfig = {}) {

this.agents[name] = { agentClass, defaultConfig };

}

async createTask(agentName, config = {}) {

if (!this.agents[agentName]) {

throw new Error(`Agent ${agentName} not registered`);

}

const { agentClass, defaultConfig } = this.agents[agentName];

const mergedConfig = { ...defaultConfig, ...config };

const taskId = `task-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;

const agent = new agentClass(mergedConfig);

this.tasks.push({

id: taskId,

agent,

status: 'created',

result: null,

createdAt: new Date(),

updatedAt: new Date()

});

return taskId;

}

async executeTask(taskId) {

const taskIndex = this.tasks.findIndex(t => t.id === taskId);

if (taskIndex === -1) {

throw new Error(`Task ${taskId} not found`);

}

this.tasks[taskIndex].status = 'running';

this.tasks[taskIndex].updatedAt = new Date();

try {

// Assuming each agent has a main method that returns a promise

const result = await this.tasks[taskIndex].agent.execute();

this.tasks[taskIndex].status = 'completed';

this.tasks[taskIndex].result = result;

} catch (error) {

this.tasks[taskIndex].status = 'failed';

this.tasks[taskIndex].error = error.message;

}

this.tasks[taskIndex].updatedAt = new Date();

return this.tasks[taskIndex];

}

getTaskStatus(taskId) {

const task = this.tasks.find(t => t.id === taskId);

if (!task) {

throw new Error(`Task ${taskId} not found`);

}

return {

id: task.id,

status: task.status,

createdAt: task.createdAt,

updatedAt: task.updatedAt,

result: task.status === 'completed' ? task.result : null,

error: task.status === 'failed' ? task.error : null

};

}

}

// Usage

const manager = new AgentManager();

// Register agents

manager.registerAgent('personalization', PersonalizationAgent);

manager.registerAgent('productSourcing', ProductSourcingAgent);

manager.registerAgent('research', ResearchAgent);

// Create and execute tasks

async function runPersonalizationWorkflow(userId) {

const userData = await fetchUserData(userId);

const taskId = await manager.createTask('personalization', {

userData,

preferenceHistory: await fetchUserHistory(userId)

});

await manager.executeTask(taskId);

const result = manager.getTaskStatus(taskId);

if (result.status === 'completed') {

// Use the recommendations

return result.result;

} else {

// Handle error

console.error(result.error);

return null;

}

}

**5. Integration with Next.js**

// pages/api/agent/[agentType].js

import { AgentManager } from '../../../lib/agents/agent-manager';

import { PersonalizationAgent } from '../../../lib/agents/personalization-agent';

import { ProductSourcingAgent } from '../../../lib/agents/product-sourcing-agent';

import { ResearchAgent } from '../../../lib/agents/research-agent';

// Initialize agent manager

const manager = new AgentManager();

manager.registerAgent('personalization', PersonalizationAgent);

manager.registerAgent('productSourcing', ProductSourcingAgent);

manager.registerAgent('research', ResearchAgent);

export default async function handler(req, res) {

const { agentType } = req.query;

if (req.method === 'POST') {

try {

// Create a new task

const taskId = await manager.createTask(agentType, req.body);

// Execute immediately or queue for background processing

if (req.body.executeImmediately) {

await manager.executeTask(taskId);

} else {

// In a real app, you might use a job queue here

process.nextTick(() => manager.executeTask(taskId));

}

res.status(200).json({ taskId });

} catch (error) {

res.status(400).json({ error: error.message });

}

} else if (req.method === 'GET') {

// Get task status

const { taskId } = req.query;

try {

const status = manager.getTaskStatus(taskId);

res.status(200).json(status);

} catch (error) {

res.status(404).json({ error: error.message });

}

} else {

res.status(405).json({ error: 'Method not allowed' });

}

}

**Front-end Integration Example**

// components/TarotReadingForm.jsx

import { useState } from 'react';

import { useAgent } from '../hooks/useAgent';

export default function TarotReadingForm() {

const [question, setQuestion] = useState('');

const [readingType, setReadingType] = useState('three-card');

const { createTask, executeTask, taskStatus, result, error } = useAgent('tarotReading');

const handleSubmit = async (e) => {

e.preventDefault();

const taskId = await createTask({

question,

readingType,

timestamp: new Date().toISOString()

});

await executeTask(taskId);

};

return (

<div className="max-w-md mx-auto bg-violet-50 p-6 rounded-lg shadow-lg">

<h2 className="text-2xl font-serif text-violet-900 mb-4">Tarot Reading</h2>

<form onSubmit={handleSubmit} className="space-y-4">

<div>

<label className="block text-sm font-medium text-violet-700 mb-1">

Your Question

</label>

<textarea

value={question}

onChange={(e) => setQuestion(e.target.value)}

className="w-full px-3 py-2 border border-violet-300 rounded-md focus:ring-2 focus:ring-violet-500"

rows={3}

placeholder="What would you like to ask the cards?"

required

/>

</div>

<div>

<label className="block text-sm font-medium text-violet-700 mb-1">

Reading Type

</label>

<select

value={readingType}

onChange={(e) => setReadingType(e.target.value)}

className="w-full px-3 py-2 border border-violet-300 rounded-md focus:ring-2 focus:ring-violet-500"

>

<option value="single-card">Single Card</option>

<option value="three-card">Three Card Spread</option>

<option value="celtic-cross">Celtic Cross</option>

</select>

</div>

<button

type="submit"

className="w-full bg-violet-600 text-white py-2 px-4 rounded-md hover:bg-violet-700 transition-colors"

disabled={taskStatus === 'running'}

>

{taskStatus === 'running' ? 'Reading cards...' : 'Begin Reading'}

</button>

</form>

{error && (

<div className="mt-4 p-3 bg-red-100 text-red-700 rounded-md">

{error}

</div>

)}

{result && (

<div className="mt-6 space-y-4">

<h3 className="text-xl font-serif text-violet-900">Your Reading</h3>

<div className="grid grid-cols-3 gap-4">

{result.cards.map((card, index) => (

<div key={index} className="bg-white p-3 rounded-md shadow text-center">

<div className="text-violet-800 font-medium">{card.name}</div>

<div className="text-sm text-gray-600">{card.position}</div>

</div>

))}

</div>

<div className="bg-white p-4 rounded-md shadow">

<h4 className="font-medium text-violet-800 mb-2">Interpretation</h4>

<p className="text-gray-700">{result.interpretation}</p>

</div>

</div>

)}

</div>

);

}