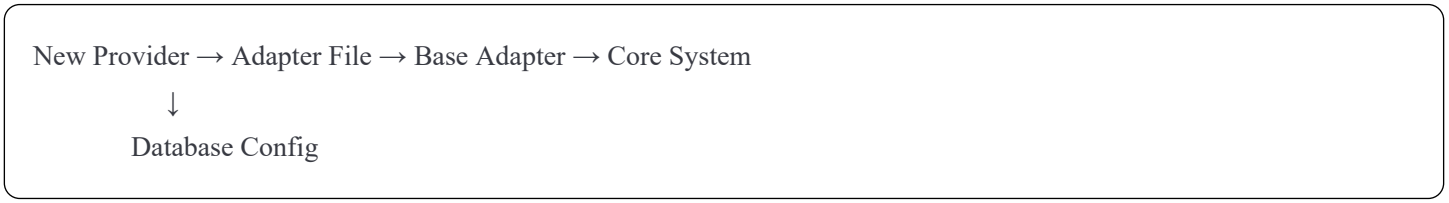
How to Add a New AI Provider

The system now supports **dynamic AI provider addition** with a plugin architecture. You can add new providers without modifying core code!

Architecture Overview



Key Components:

1. **BaseAdapter** - Abstract class defining the interface all providers must implement
2. **Provider Adapters** - Individual files for each AI provider (openai.js, anthropic.js, etc.)
3. **aiProviders Service** - Dynamically loads adapters based on database configuration
4. **Database** - Stores provider configuration including which adapter to use

Step-by-Step: Adding a New Provider

Let's add **Mistral AI** as an example.

Step 1: Create the Adapter File

Create `backend/src/services/adapters/mistral.js`:

```javascript
const BaseAdapter = require('./BaseAdapter');

class MistralAdapter extends BaseAdapter {
  // Transform messages to Mistral format
  transformMessages(messages) {
    return messages.map(m => ({
      role: m.role,
      content: m.content
    }));
  }

  // Build the API request payload
  buildRequestPayload(messages) {
    return {
      model: this.modelName,
      messages: this.transformMessages(messages),
      temperature: this.config.temperature || 0.7,
      max_tokens: this.config.max_tokens || 1000
    };
  }

  // Set HTTP headers for authentication
  getHeaders(apiKey) {
    return {
      'Authorization': `Bearer ${apiKey}`,
      'Content-Type': 'application/json'
    };
  }

  // Extract the AI response text
  extractResponse(responseData) {
    return responseData.choices[0].message.content;
  }

  // Extract token usage (optional)
  extractTokenCount(responseData) {
    return responseData.usage?.total_tokens || 0;
  }
}

module.exports = MistralAdapter;
```

Step 2: Add Provider to Database

```sql
INSERT INTO ai_providers (
  name,
  display_name,
  api_endpoint,
  model_name,
  adapter_module,
  config,
  is_active
) VALUES (
  'mistral',                          -- unique identifier
  'Mistral AI',                       -- display name
  'https://api.mistral.ai/v1/chat/completions', -- API endpoint
  'mistral-medium',                   -- default model
  'mistral',                          -- adapter file name (without .js)
  '{"temperature": 0.7, "max_tokens": 1000}',   -- JSON configuration
  true                                -- is active
);
```

Or use the API:

```bash
curl -X POST http://localhost/api/providers/admin/create \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "mistral",
    "display_name": "Mistral AI",
    "api_endpoint": "https://api.mistral.ai/v1/chat/completions",
    "model_name": "mistral-medium",
    "adapter_module": "mistral",
    "config": {
      "temperature": 0.7,
      "max_tokens": 1000
    }
  }'
```

Step 3: Add API Keys

```bash
# Get the provider ID (let's say it's 5)
curl http://localhost/api/providers

# Add API key
curl -X POST http://localhost/api/providers/keys \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "provider_id": 5,
    "key_value": "your-mistral-api-key"
  }'
```

Step 4: Test It!

```bash
# Create a chat with Mistral
curl -X POST http://localhost/api/chats/project/1 \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Testing Mistral",
    "provider_id": 5
  }'

# Send a message
curl -X POST http://localhost/api/chats/CHAT_ID/messages \
  -H "Authorization: Bearer YOUR_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "content": "Hello Mistral!"
  }'
```

BaseAdapter Methods Reference

Every adapter **must implement** these methods:

Required Methods

```javascript
transformMessages(messages)
// Input: [{ role: 'user', content: '...' }, ...]
// Output: Provider-specific message format
// Purpose: Convert our standard format to provider's format

buildRequestPayload(messages)
// Input: Array of messages
// Output: Complete request body object
// Purpose: Build the full API request payload

getHeaders(apiKey)
// Input: API key string
// Output: Object with HTTP headers
// Purpose: Set authentication and content-type headers

extractResponse(responseData)
// Input: Full API response data
// Output: String (the AI's response text)
// Purpose: Extract the actual message content
```

Optional Methods

```javascript
extractTokenCount(responseData)
// Input: Full API response data
// Output: Number (token count)
// Default: Returns 0
// Purpose: Extract token usage for analytics

getEndpoint(apiKey)
// Input: API key string
// Output: String (full URL)
// Default: Uses this.endpoint from database
// Purpose: Customize endpoint URL (e.g., add API key as query param)

parseRateLimitError(error)
// Input: Axios error object
// Output: { resetAt: Date } or null
// Default: BaseAdapter provides standard implementation
// Purpose: Extract rate limit reset time from error
```

Real-World Examples

Example 1: OpenAI-Compatible Provider (Groq, Together.ai)

Most providers following OpenAI's API format need minimal code:

```javascript
const BaseAdapter = require('./BaseAdapter');

class GroqAdapter extends BaseAdapter {
  transformMessages(messages) {
    return messages.map(m => ({ role: m.role, content: m.content }));
  }

  buildRequestPayload(messages) {
    return {
      model: this.modelName,
      messages: this.transformMessages(messages)
    };
  }

  getHeaders(apiKey) {
    return {
      'Authorization': `Bearer ${apiKey}`,
      'Content-Type': 'application/json'
    };
  }

  extractResponse(responseData) {
    return responseData.choices[0].message.content;
  }

  extractTokenCount(responseData) {
    return responseData.usage?.total_tokens || 0;
  }
}

module.exports = GroqAdapter;
```

Example 2: Provider with API Key in URL (Like Gemini)

javascript

```javascript
class CustomAdapter extends BaseAdapter {
  // ... other methods ...

  getEndpoint(apiKey) {
    // Add API key to URL instead of header
    return `${this.endpoint}?apiKey=${apiKey}&format=json`;
  }

  getHeaders(apiKey) {
    // No auth header needed
    return {
      'Content-Type': 'application/json'
    };
  }
}
```

Example 3: Provider with Custom Message Format

javascript

```javascript
class CustomAdapter extends BaseAdapter {
  transformMessages(messages) {
    // Custom format with nested structure
    return {
      conversation: messages.map((m, idx) => ({
        id: idx,
        speaker: m.role === 'user' ? 'human' : 'ai',
        text: m.content,
        timestamp: new Date().toISOString()
      }))
    };
  }

  buildRequestPayload(messages) {
    return {
      model_id: this.modelName,
      input: this.transformMessages(messages),
      parameters: this.config
    };
  }

  extractResponse(responseData) {
    // Navigate nested response structure
    return responseData.output.generated_text.content;
  }
}
```

Configuration Options

The `config` JSONB field in `ai_providers` table allows provider-specific settings:

```javascript
{
  "temperature": 0.7,          // Control randomness
  "max_tokens": 1000,          // Maximum response length
  "top_p": 0.9,                // Nucleus sampling
  "frequency_penalty": 0,      // Reduce repetition
  "presence_penalty": 0,       // Encourage new topics
  "custom_param": "value"      // Any provider-specific parameter
}
```

Access in adapter via `this.config`:

```javascript
buildRequestPayload(messages) {
  return {
    model: this.modelName,
    messages: this.transformMessages(messages),
    temperature: this.config.temperature || 0.7,
    my_custom_setting: this.config.custom_param
  };
}
```

Testing Your Adapter

1. Test Locally First

```javascript
// Create a test file: backend/test-adapter.js
const MistralAdapter = require('./src/services/adapters/mistral');

const mockProvider = {
  name: 'mistral',
  api_endpoint: 'https://api.mistral.ai/v1/chat/completions',
  model_name: 'mistral-medium',
  config: { temperature: 0.7 }
};

const adapter = new MistralAdapter(mockProvider);

const messages = [
  { role: 'user', content: 'Hello!' }
];

console.log('Transformed:', adapter.transformMessages(messages));
console.log('Payload:', adapter.buildRequestPayload(messages));
console.log('Headers:', adapter.getHeaders('test-key'));
```

Run: `node backend/test-adapter.js`

2. Test with Real API

Use the provider management endpoints to add it to the system and test end-to-end.

Troubleshooting

Adapter Not Found

```
Error: Adapter module 'mistral' not found
```

Solution: Ensure the file exists at `backend/src/services/adapters/mistral.js`

Method Not Implemented

```
Error: transformMessages() must be implemented by adapter
```

Solution: Make sure all required methods are implemented in your adapter

Rate Limit Handling

If the provider has custom rate limit headers, override `parseRateLimitError()`:

```javascript
parseRateLimitError(error) {
  if (error.response?.status !== 429) return null;

  const resetTime = error.response.headers['x-custom-reset-header'];
  const resetAt = new Date(resetTime * 1000);

  return { resetAt };
}
```

Benefits of This Architecture

✅ **No Core Code Changes** - Add providers without touching main codebase ✅ **Consistent Interface** - All providers work the same way from the app's perspective
✅ **Easy Maintenance** - Each provider's logic is isolated ✅ **Type Safety** - BaseAdapter enforces required methods ✅ **Flexible Configuration** - Store provider-specific settings in database ✅ **Hot Reload Ready** - Can add providers without restarting (in development)

Advanced: Dynamic Model Selection

You can even support multiple models per provider:

```sql
-- Add multiple entries for same provider with different models
INSERT INTO ai_providers (name, display_name, model_name, adapter_module) VALUES
  ('gpt-4', 'GPT-4', 'gpt-4', 'openai'),
  ('gpt-3.5', 'GPT-3.5 Turbo', 'gpt-3.5-turbo', 'openai');
```

Users can then choose between GPT-4 and GPT-3.5 as separate options!