Tutorial: Agent Documentor

This project is an **AI-powered documentation generator** , designed to automatically create beginner-friendly tutorials for your codebase. It leverages the power of Large Language Models (LLMs) to understand your project's structure and generate comprehensive documentation.

At its core, the system works by first **identifying key components** or abstractions within your project. It then analyzes the **relationships** between these components to understand how they interact. Based on this analysis, it creates a **structured learning path**, ordering the components logically for a tutorial. Finally, it generates **individual Markdown files** for each component, complete with explanations, code snippets, and diagrams, creating a full tutorial in a `docs` directory.

```
flowchart TD
    A0["Pydantic Models"]
    A1["File Extractor"]
    A2["Agent Nodes Logic"]
    A3["Educator Agent Workflow"]
    A3 -- "builds and compiles" --> A2
    A2 -- "uses" --> A0
    A2 -- "uses" --> A1
    A3 -- "orchestrates" --> A2
    A1 -- "extracts files for" --> A3
    A2 -- "analyzes relationships for" --> A3
    A2 -- "orders components for" --> A3
    A2 -- "plans pages for" --> A3
    A2 -- "processes pages for" --> A3
    A2 -- "writes documentation for" --> A3
```

Chapters

Chapter 1: Educator Agent Workflow

Welcome to the Agent Documentor project! This is the very first chapter, where we'll dive into the core of how our agent works.

Imagine you have a bunch of code files, and you want to automatically generate documentation for them. This sounds like a big task, right? Our **Educator Agent Workflow** is designed to handle exactly this! It's like a super-smart assistant that knows exactly what steps to take to turn your code into helpful documentation.

What is the Educator Agent Workflow?

Think of the Educator Agent Workflow as a recipe or a step-by-step guide for our documentation agent. It tells the agent:

1. **What to do first:** Like gathering all the ingredients for a recipe.
2. **What to do next:** Like chopping vegetables.
3. **How to decide what to do:** Like deciding if you need more salt or pepper.
4. **When to stop:** Like knowing when the dish is perfectly cooked!

This workflow is built using a powerful tool called **LangGraph**. LangGraph helps us create sequences of actions, and even make decisions along the way, which is perfect for our documentation task.

Our Use Case: Documenting Your Code

Let's say you have a Python project with several files. You want to create a nice documentation website for it. The Educator Agent Workflow will orchestrate the process of:

- Understanding your code files.
- Figuring out how different parts of your code relate to each other.
- Planning out the documentation pages.
- Writing the actual documentation content.
- Organizing everything into a neat directory structure.

The Steps Involved: A High-Level View

Our Educator Agent Workflow follows a specific path to achieve this. Here's a simplified look at the journey:

1. **Component Segregator:** First, we need to understand what each piece of your code does. This step separates your code into logical "components."
2. **Component Relationship Analyser:** Next, we figure out how these components talk to each other. Does one component use another?
3. **Component Ordering:** Based on their relationships, we decide the best order to document them. Think of it like arranging chapters in a book.
4. **Components Pages Planner:** Now, we plan out which documentation pages we need and what information should go on each page.
5. **Component Page Processor:** This is where the actual documentation content is generated for each planned page.

6. **Pages to Documentation Directory:** Finally, all the generated documentation pages are organized into a clear directory structure, ready for you to use!

How it Works Under the Hood: A Peek Inside 🕵️‍♀️

To understand how this workflow is built, let's look at the code that defines it.

```
# educator_agent.py

from langgraph.graph import StateGraph, START, END

from state import State
from agent_nodes import (
    component_segregator,
    component_relationship_analyser,
    component_ordering,
    components_pages_planner,
    component_page_processor,
    pages_to_documentation_directory,
    route_based_on_pages_remaining
)
```

This code imports all the necessary pieces to build our workflow. We're using `StateGraph` from `langgraph` to define the steps and `START` and `END` to mark the beginning and end of our process. We also import various "agent nodes" which are the actual functions that perform each step.

Building the Workflow 🏗️

```
def build_educator_agent():
    """Build and return the compiled educator agent workflow."""

    educator_agent_builder = StateGraph(State)

    # Add each step as a node
    educator_agent_builder.add_node("component_segregator", component_segregator)
    educator_agent_builder.add_node("component_relationship_analyser", component_relationship_analyser)
    educator_agent_builder.add_node("component_ordering", component_ordering)
    educator_agent_builder.add_node("components_pages_planner", components_pages_planner)
    educator_agent_builder.add_node("component_page_processor", component_page_processor)
    educator_agent_builder.add_node("pages_to_documentation_directory", pages_to_documentation_directory)
```

Here, we create a `StateGraph` and add each of our steps (like `component_segregator`) as a "node" in the graph. Think of nodes as individual tasks.

Connecting the Steps 🔗

```
    # Define the flow between nodes
    educator_agent_builder.add_edge(START, "component_segregator")
    educator_agent_builder.add_edge("component_segregator", "component_relationship_analyser")
    educator_agent_builder.add_edge("component_relationship_analyser", "component_ordering")
    educator_agent_builder.add_edge("component_ordering", "components_pages_planner")
```

This part defines the sequence. We're saying: "Start, then go to `component_segregator`, then to `component_relationship_analyser`, and so on."

Making Decisions: Conditional Edges 🤔

Sometimes, we need to make decisions. For example, after planning pages, we might need to process more pages or we might be done.

```
    educator_agent_builder.add_conditional_edges(
        "components_pages_planner",
        route_based_on_pages_remaining, # This function decides where to go next
        {
            "pages_to_process": "component_page_processor", # If pages are left, go here
            "no_page_to_process": "pages_to_documentation_directory" # If no pages left, go here
        }
    )
    educator_agent_builder.add_conditional_edges(
        "component_page_processor",
        route_based_on_pages_remaining,
        {
            "pages_to_process": "component_page_processor", # If more pages to process, loop back
            "no_page_to_process": "pages_to_documentation_directory" # If done, move on
        }
    )
```

`add_conditional_edges` is super cool! It lets us use a function (like `route_based_on_pages_remaining`) to decide which node to go to next. This allows our agent to be flexible.

Finishing Up

```
    educator_agent_builder.add_edge("pages_to_documentation_directory", END)

    return educator_agent_builder.compile()

# Create the compiled agent
educator_agent = build_educator_agent()
```

Finally, we connect the last step to `END` and then `compile()` the whole workflow into a runnable agent.

A Simple Walkthrough

Let's visualize the flow with a simple example. Imagine we have 3 components to document.

```
sequenceDiagram
    participant Agent as Educator Agent
    participant Segregator as Component Segregator
    participant Planner as Components Pages Planner
    participant Processor as Component Page Processor
    participant Directory as Pages to Documentation Directory

    Agent->>Segregator: Start
    Segregator-->>Planner: Components identified
    Planner->>Processor: Plan page 1
    Processor-->>Planner: Page 1 done, more pages? (Yes)
    Planner->>Processor: Plan page 2
    Processor-->>Planner: Page 2 done, more pages? (Yes)
    Planner->>Processor: Plan page 3
    Processor-->>Planner: Page 3 done, more pages? (No)
    Planner->>Directory: All pages planned, move to directory
    Directory-->>Agent: Documentation complete!
```

This diagram shows how the agent moves from one step to another, and how it might loop back to process more pages until everything is done.

What's Next? ➡

We've just scratched the surface of the Educator Agent Workflow! We've seen how it orchestrates the entire documentation process. In the next chapter, we'll dive deeper into the logic of each individual Agent Nodes Logic. Get ready to explore what makes each step tick!

Chapter 2: Agent Nodes Logic

Welcome back! In the last chapter, we got a high-level overview of the Educator Agent Workflow. We saw how it acts like a recipe, guiding our documentation agent through a series of steps.

Now, it's time to zoom in and understand the "ingredients" and "cooking techniques" that make each step in that workflow happen. These individual steps are what we call **Agent Nodes Logic**.

What are Agent Nodes Logic?

Think of the Educator Agent Workflow as a play, and each "node" is an actor performing a specific scene. The **Agent Nodes Logic** is the script for each of those actors!

These nodes are essentially Python functions that contain the core intelligence for each part of our documentation process. They are responsible for:

- **Identifying Components:** Figuring out the main building blocks of your code.
- **Analyzing Relationships:** Understanding how these building blocks interact with each other.
- **Ordering Content:** Deciding the best sequence to present information.
- **Generating Pages:** Writing the actual documentation content.

Crucially, these nodes are where we interact with powerful Large Language Models (LLMs) to perform these complex tasks.

The Core Functions

Let's look at some of the key functions (nodes) that make up our Agent Nodes Logic:

1. `component_segregator`

This is the first step in our workflow. Its job is to read through your code files and identify the most important "components" or abstractions.

```
# agent_nodes.py

def component_segregator(state: State):
    file_context, file_listing = "", ""
    for i, file in enumerate(state["files"]):
        file_context += f"--- File Index {i}: path: {file['path']} ---\n{file['content']}\n\n"
        file_listing += f"- {i} # {file['path']}\n"

    llm_context = f"""
For the project {state["project_name"]}:

Codebase files context:
{file_context}

Analyse the codebase.
Identify the top 4-{state["max_components"]} core most important abstractions or components to help the new user understand the code

For each component, provide:
- name: concise name for the component
- description: beginner friendly description of the component
- files: list of relevent files indices of the component

List of file indices and paths in the codebase context:
{file_listing}

Return the components in a structured format.
    """

    component_segregation_llm = ChatOpenAI(
            model="gemini-2.5-flash-lite-preview-06-17",
            temperature=0,
            api_key=os.getenv("GOOGLE_API_KEY"),
            base_url="https://generativelanguage.googleapis.com/v1beta/openai/"
        )

    component_segregation_llm_with_structured_output = component_segregation_llm.with_structured_output(Components)

    components = component_segregation_llm_with_structured_output.invoke(llm_context)

    print(f"created {len(components.components)} components")

    return {"components": components.components}
```

What's happening here?

- It gathers all the code content and file paths.
- It crafts a detailed prompt for the LLM, asking it to identify key components, their descriptions, and the files they relate to.
- It uses `ChatOpenAI` (which is our interface to the LLM) and specifically asks for the output in a structured format defined by `Components` (which we'll cover in a later chapter!).
- Finally, it returns the identified components to be used in the next step.

2. `component_relationship_analyser`

Once we know what the components are, we need to understand how they connect. This node figures out those relationships.

```
# agent_nodes.py

def component_relationship_analyser(state: State):
    component_context = "Identified Components or Abstractions:\n"
    component_listing = ""
    relevant_file_indices = set()
    for i, component in enumerate(state["components"]):
        component_context += f"- Index {i}: {component.name} (Relevant file indices: [{', '.join(map(str, component.files))}])\n
        component_listing += f"- {i} # {component.name}\n"
        relevant_file_indices.update(component.files)
    component_context += f"Relevant File Snippets referenced by index and path:\n"
    for i in sorted(list(relevant_file_indices)):
        if i < 0 or i >= len(state["files"]): continue
        component_context += f"\n\n- {i} # {state['files'][i]['path']}\n{state['files'][i]['content']}\n"

    llm_context = f"""
Based on the following abstractions or components and the relevant file snippets from the project {state["project_name"]}:

List of Component or Abstraction Indices and Names:
{component_listing}

Context of Components or Abstractions:
{component_context}

Provide:
1. A high-level `overview` of the project's main purpose and functionality in a few beginner-friendly paragraphs. Use markdown forma
2. A list (`relationships`) describing the key interactions between these abstractions. For each relationship, specify:
- `from_component`: Index of the source component or abstraction (e.g., `0 # ComponentName1`)
- `to_component`: Index of the target component or abstraction (e.g., `1 # ComponentName2`)
- `label`: A brief label for the interaction **in just a few words** (e.g., "Manages", "Inherits", "Uses").
Ideally the relationship should be backed by one component or abstraction calling or passing parameters to another.
Simplify the relationship and exclude those non-important ones.

IMPORTANT: Make sure EVERY component or abstraction is involved in at least ONE relationship (either as source or target). Each comp
    """
    component_relationship_analyser_llm = ChatOpenAI(
            model="gemini-2.5-flash-lite-preview-06-17",
            temperature=0,
            api_key=os.getenv("GOOGLE_API_KEY"),
            base_url="https://generativelanguage.googleapis.com/v1beta/openai/"
        )
    component_relationship_analyser_llm_with_structured_output = component_relationship_analyser_llm.with_structured_output(Relation

    relationship_analysis = component_relationship_analyser_llm_with_structured_output.invoke(llm_context)

    print(relationship_analysis)

    return {"project_overview": relationship_analysis.overview, "component_relationships": relationship_analysis.relationships}
```

What's happening here?

- It takes the identified components and their associated files.
- It prepares a prompt for the LLM that includes the component details and the actual code snippets from the relevant files.
- The LLM is asked to provide a project overview and a list of relationships between components, including a label for each interaction.
- Again, the output is structured using `RelationshipAnalysisOutput`.

3. `component_ordering`

With components and their relationships known, we need to decide the best order to explain them. This node handles that.

```
 # agent_nodes.py

def component_ordering(state: State):
    components_list = "\n".join([f"- {i} # {component.name}" for i, component in enumerate(state["components"])])
    relationships_list = "\n".join([f"- from {state['components'][relationship.from_component].name} to {state['components'][relatio

    llm_context = f"""

Given the following project abstractions and their relationships for the project {state["project_name"]}:

Components or Abstractions (Index # Name):

{components_list}

Project Overview:
{state["project_overview"]}

Relationships between Components:

{relationships_list}

If you are going to make a tutorial for {state["project_name"]}, what is the best order to explain these abstractions, from first to
Ideally, first explain those that are most important or foundational, perhaps user-facing concepts or entry points. Then move to mor

Output the ordered list of component indices in a list, should include all components and should be a non-repeating valid list of in

example output:
[3, 1, 2, 0]

Now provide the ordered list of component indices.

"""

    component_ordering_llm = ChatOpenAI(
            model="gemini-2.5-flash-lite-preview-06-17",
            temperature=0,
            api_key=os.getenv("GOOGLE_API_KEY"),
            base_url="https://generativelanguage.googleapis.com/v1beta/openai/"
        )

    component_ordering_llm_with_structured_output = component_ordering_llm.with_structured_output(OrderedComponents)

    ordered_components = component_ordering_llm_with_structured_output.invoke(llm_context)

    print(ordered_components)

    return {"ordered_components": ordered_components.ordered_components}
```

What's happening here?

- It takes the project overview and the relationships we just analyzed.
- It prompts the LLM to suggest an optimal order for explaining the components, prioritizing foundational concepts.
- The LLM's output is a list of component indices, structured by `OrderedComponents`.

4. `components_pages_planner`

This node takes the ordered components and plans out the structure of our documentation, including titles and filenames.

```
# agent_nodes.py

def components_pages_planner(state: State):
    docs_metadata = []
    print(state)
    for i, component_index in enumerate(state["ordered_components"]):
        if 0 > component_index >= len(state["components"]):
            continue
        docs_metadata.append({
            "title": state["components"][component_index].name,
            "number": i + 1,
            "component_index": component_index,
            "file_name": f"{i+1:02d}_{''.join(c if c.isalnum() else '_' for c in state['components'][component_index].name).lower()}
        })
    all_docs_names_listing = "\n".join([f"{doc_metadata['number']}. [{doc_metadata['title']}]({doc_metadata['file_name']})" for doc_
    pages_to_process = []
    for i, component_index in enumerate(state["ordered_components"]):
        if 0 > component_index >= len(state["components"]):
            continue
        component_details = state["components"][component_index]
        relevant_file_indices = component_details.files
        files_map = [(f"{i} # {state['files'][i]['path']}", state["files"][i]["content"]) for i in relevant_file_indices]
        prev_page = None
        if i > 0:
            prev_component_index = state["ordered_components"][i - 1]
            prev_page = docs_metadata[prev_component_index]

        next_page = None
        if i < len(state["ordered_components"]) - 1:
            next_component_index = state["ordered_components"][i + 1]
            next_page = docs_metadata[next_component_index]

        pages_to_process.append({
            "component_num": i + 1,
            "component_index": component_index,
            "all_pages_names_listing": all_docs_names_listing,
            "prev_page": prev_page,
            "next_page": next_page,
            "file_name": docs_metadata[i]["file_name"]
        })

    return {"pages_to_process": pages_to_process, "pages_processed": 0}
```

What's happening here?

- It iterates through the `ordered_components`.
- For each component, it creates metadata like a chapter number, title, and a clean filename.
- It also prepares context about the previous and next pages, which will be useful for the LLM when writing the content.
- The output is a list called `pages_to_process`, which contains all the information needed to generate each documentation page.

5. `component_page_processor`

This is where the magic of content generation happens! This node uses the LLM to write the actual Markdown content for each documentation page.

```
# agent_nodes.py

def component_page_processor(state: State):
    print(state)
    page = state["pages_to_process"][state["pages_processed"]]
    component_details = state["components"][page["component_index"]]

    files_context = "\n".join([f"# {state['files'][i]['path']}\n\n{state['files'][i]['content']}" for i in component_details.files])
    prev_pages_context = "\n--------\n".join(state["pages"])

    llm_context = f"""
Write a very beginner-friendly tutorial chapter (in Markdown format) for the project {state["project_name"]} about the concept: "{co
```

```
Write a very beginner-friendly tutorial chapter (in Markdown format) for the project {state[ project_name ]} about the concept:  {co

Concept Details:
- Name: {component_details.name}
- Description:
{component_details.description}

Complete Tutorial Structure:
{page["all_pages_names_listing"]}

Context from previous chapters:
{prev_pages_context if prev_pages_context else "This is the first chapter."}

Relevant Code Snippets (Code itself remains unchanged):
{files_context if files_context else "No specific code snippets provided for this abstraction."}

Instructions for the chapter:
- Start with a clear heading (e.g., `# Chapter {page['component_num']}: {component_details.name}`). Use the provided concept name.

- If this is not the first chapter, begin with a brief transition from the previous chapter, referencing it with a proper Markdown l

- Begin with a high-level motivation explaining what problem this abstraction solves. Start with a central use case as a concrete ex

- If the abstraction is complex, break it down into key concepts. Explain each concept one-by-one in a very beginner-friendly way.

- Explain how to use this abstraction to solve the use case. Give example inputs and outputs for code snippets (if the output isn't

- Each code block should be BELOW 10 lines! If longer code blocks are needed, break them down into smaller pieces and walk through t

- Describe the internal implementation to help understand what's under the hood. First provide a non-code or code-light walkthrough

- Then dive deeper into code for the internal implementation with references to files. Provide example code blocks, but make them si

- IMPORTANT: When you need to refer to other core abstractions covered in other chapters, ALWAYS use proper Markdown links like this

- Use mermaid diagrams to illustrate complex concepts (```mermaid``` format)..

- Heavily use analogies and examples throughout to help beginners understand.

- End the chapter with a brief conclusion that summarizes what was learned and provides a transition to the next chapter. If there i

- Ensure the tone is welcoming and easy for a newcomer to understand, feel free to use emojis to make it more engaging.

- Output *only* the Markdown content for this chapter.

Now, directly provide a super beginner-friendly Markdown output (DON'T need ```markdown``` tags):
"""

    page_processor_llm = ChatOpenAI(
            model="gemini-2.5-flash-lite-preview-06-17",
            temperature=0,
            api_key=os.getenv("GOOGLE_API_KEY"),
            base_url="https://generativelanguage.googleapis.com/v1beta/openai/"
        )

    page_content = page_processor_llm.invoke(llm_context).content
    page_heading = f"# Chapter {page['component_num']}: {component_details.name}"
    if not page_content.strip().startswith(f"# Chapter {page['component_num']}"):
        lines = page_content.strip().split("\n")
        if lines and lines[0].strip().startswith("#"):
            lines[0] = page_heading
            page_content = "\n".join(lines)
        else:
            page_content = f"{page_heading}\n\n{page_content}"

    return {"pages": [page_content], "pages_processed": state["pages_processed"] + 1}
```

What's happening here?

- It takes the details for the current page to be processed.
- It constructs a very detailed prompt for the LLM, including instructions on tone, structure, code block length, use of diagrams, and how to link to other chapters.
- The LLM generates the Markdown content for the chapter.
- It ensures the output starts with the correct chapter heading and then updates the state with the generated page content and increments the `pages_processed` count.

6. `pages_to_documentation_directory`

The final step! This node takes all the generated Markdown content and organizes it into a proper documentation directory structure.

```
# agent_nodes.py

def pages_to_documentation_directory(state: State):
    output_path = os.path.join("docs", state["project_name"])
    mermaid_lines = ["flowchart TD"]
    components = state["components"]
    ordered_components = state["ordered_components"]
    project_overview = state["project_overview"]
    pages = state["pages"]
    for i, component in enumerate(state["components"]):
        node_id = f"A{i}"
        # Use potentially translated name, sanitize for Mermaid ID and label
        sanitized_name = component.name.replace('"', "")
        node_label = sanitized_name  # Using sanitized name only
        mermaid_lines.append(
            f'    {node_id}["{node_label}"]'
        )  # Node label uses potentially translated name
    # Add edges for relationships using potentially translated labels
    for rel in state["component_relationships"]:
        from_node_id = f"A{rel.from_component}"
        to_node_id = f"A{rel.to_component}"
        # Use potentially translated label, sanitize
        edge_label = (
            rel.label.replace('"', "").replace("\n", " ")
        )  # Basic sanitization
        max_label_len = 30
        if len(edge_label) > max_label_len:
            edge_label = edge_label[: max_label_len - 3] + "..."
        mermaid_lines.append(
            f'    {from_node_id} -- "{edge_label}" --> {to_node_id}'
        )  # Edge label uses potentially translated label
    mermaid_diagram = "\n".join(mermaid_lines)

    index_content = f"# Tutorial: {state['project_name']}\n\n"
    index_content += f"{state['project_overview']}\n\n"  # Use the potentially translated summary directly
    # Keep fixed strings in English
    # index_content += f"**Source Repository:** [{repo_url}]({repo_url})\n\n"

    # Add Mermaid diagram for relationships (diagram itself uses potentially translated names/labels)
    index_content += "```mermaid\n"
    index_content += mermaid_diagram + "\n"
    index_content += "```\n\n"

    # Keep fixed strings in English
    index_content += f"## Chapters\n\n"

    chapter_files = []
    # Generate chapter links based on the determined order, using potentially translated names
    for i, component_index in enumerate(ordered_components):
        # Ensure index is valid and we have content for it
        if 0 <= component_index < len(components) and i < len(pages):
            component_name = components[component_index].name
            # Sanitize potentially translated name for filen
```

```
            # Sanitize potentially translated name for filename
            safe_name = "".join(
                c if c.isalnum() else "_" for c in component_name
            ).lower()
            filename = f"{i+1:02d}_{safe_name}.md"
            index_content += f"{i+1}. [{component_name}]({filename})\n"  # Use potentially translated name in link text

            chapter_content = pages[i]
            if not chapter_content.endswith("\n\n"):
                chapter_content += "\n\n"
            # Keep fixed strings in English
            chapter_content += f"---\n\nGenerated by Kritagya Khandelwal"

            # Store filename and corresponding content
            chapter_files.append({"filename": filename, "content": chapter_content})
        else:
            print(
                f"Warning: Mismatch between chapter order, abstractions, or content at index {i} (abstraction index {component_index
            )

    # Add attribution to index content (using English fixed string)
    index_content += f"\n\n---\n\nGenerated by Kritagya Khandelwal"

    os.makedirs(output_path, exist_ok=True)

    # Write index.md
    index_filepath = os.path.join(output_path, "index.md")
    with open(index_filepath, "w", encoding="utf-8") as f:
        f.write(index_content)
    print(f"  - Wrote {index_filepath}")

    # Write chapter files
    for chapter_info in chapter_files:
        chapter_filepath = os.path.join(output_path, chapter_info["filename"])
        with open(chapter_filepath, "w", encoding="utf-8") as f:
            f.write(chapter_info["content"])
        print(f"  - Wrote {chapter_filepath}")

    return {"output_path": output_path}
```

What's happening here?

- It creates a `docs` directory and a subdirectory for your project.
- It generates an `index.md` file that serves as the table of contents, including the project overview and links to each chapter.
- It also creates each individual chapter file with its generated content.
- It organizes everything neatly, ready for you to view!

Orchestrating the LLM

Notice a pattern? Each of these nodes is designed to:

1. **Prepare Input:** Gather the necessary data from the previous steps.
2. **Craft a Prompt:** Create a specific, detailed instruction for the LLM.
3. **Call the LLM:** Use `ChatOpenAI` to send the prompt and get a response.
4. **Process Output:** Parse the LLM's response and structure it for the next step.

This modular approach, where each node has a single, well-defined responsibility, makes our documentation agent robust and easier to manage.

What's Next? ➡

We've now seen the individual "brains" behind each step of our documentation workflow. In the next chapter, we'll dive into the File Extractor, which is responsible for getting the code files into our agent's hands in the first place! Stay tuned!

Chapter 3: File Extractor

Welcome back! In the last chapter, we explored the Agent Nodes Logic, the individual functions that power our documentation agent. Now, we're going to focus on a crucial first step: getting the code files into our agent's hands. This is where the **File Extractor** comes in!

What is the File Extractor?

Imagine you have a project with many files. Some are important code files, others might be temporary files, configuration files, or large binary files that we don't need for documentation. The File Extractor is like a smart librarian for your code. Its job is to go into a specified folder (your project directory) and carefully select *only* the files that are relevant for documentation.

It's designed to be flexible, allowing you to:

- **Specify a Directory:** Tell it exactly where to look for files.
- **Include/Exclude Patterns:** Define rules for which files to grab (e.g., only Python files) and which to ignore (e.g., `.log` files, `venv` directories).
- **Respect `.gitignore`:** It's smart enough to understand your project's `.gitignore` file, automatically skipping files that you've already told Git to ignore. This is super handy for keeping your documentation focused on your actual code.
- **Size Limits:** You can even set a maximum file size, so it won't try to read massive files that aren't suitable for documentation.

Why Do We Need It?

Our documentation agent needs to read your code to understand it. But it can't just blindly read every single file in your project. That would be inefficient and could lead to errors or irrelevant information. The File Extractor acts as a filter, ensuring that only the necessary code files are passed on to the next stages of our workflow.

Think of it as preparing your ingredients before cooking. You wouldn't throw the whole vegetable garden into the pot, right? You select the best, freshest ingredients. The File Extractor does the same for your code files.

How Does It Work? ⚙

Let's look at the code that makes the File Extractor work:

```python
# file_extractor.py

import fnmatch
import os
import pathspec

def file_extractor(directory: str, include_patterns: list[str] = [], exclude_patterns: list[str] = [], max_file_size: int = None) ->
    """
    Extracts files from a directory based on specified patterns and size constraints.
    ... (rest of the function)
    """
    extracted_files = []
    if not os.path.isdir(directory):
        raise ValueError(f"Directory does not exist: {directory}")

    # ... (code to handle .gitignore) ...

    all_files = []
    for root, dirs, files in os.walk(directory):
        # ... (code to handle directory exclusions) ...
        for filename in files:
            filepath = os.path.join(root, filename)
            all_files.append(filepath)

    # ... (loop through all_files to check patterns and size) ...
    for filepath in all_files:
        relpath = os.path.relpath(filepath, directory)

        # --- Exclusion check ---
        excluded = False
        # ... (check against .gitignore and exclude_patterns) ...

        included = False
        # ... (check against include_patterns) ...

        if not included or excluded:
            continue # Skip to next file if not included or excluded

        if max_file_size and os.path.getsize(filepath) > max_file_size:
            continue # Skip large files

        # --- File is being processed ---
        try:
            with open(filepath, "r", encoding="utf-8-sig") as f:
                content = f.read()
            extracted_files.append({"path": relpath, "content": content})
        except Exception as e:
            print(f"Warning: Could not read file {filepath}: {e}")

    return extracted_files
```

Let's break down the key parts:

1. **Importing Libraries:**
    - `fnmatch` : This is used for matching filenames against patterns (like `*.py`).
    - `os` : This module provides functions for interacting with the operating system, like walking through directories (`os.walk`) and getting file sizes (`os.path.getsize`).
    - `pathspec` : This powerful library helps us work with `.gitignore` files and other pattern matching.
2. **The `file_extractor` Function:**
    - It takes the `directory` , `include_patterns` , `exclude_patterns` , and `max_file_size` as input.
    - It first checks if the provided `directory` actually exists.
    - **`.gitignore` Handling:** It looks for a `.gitignore` file in the specified directory. If found, it reads the patterns from it using `pathspec` to create a filter.
    - **Walking the Directory:** `os.walk(directory)` is used to go through every file and folder within the `directory` . It's like a recursive search.

- **Filtering Files:** For each file found, it performs several checks:
  - **Exclusion:** It checks if the file's path matches any of the `exclude_patterns` or if it's ignored by the `.gitignore` rules.
  - **Inclusion:** It checks if the file's path matches any of the `include_patterns`. If no `include_patterns` are given, all files are considered for inclusion by default.
  - **Size Limit:** If `max_file_size` is set, it checks if the file's size exceeds this limit.
- **Reading Content:** If a file passes all the checks (i.e., it's included and not excluded, and within the size limit), its content is read using `f.read()`.
- **Storing Results:** The file's relative path and its content are stored in a list called `extracted_files`.

A Simple Example

Let's say you have a project structure like this:

```
my_project/
├── src/
│   ├── main.py
│   └── utils.py
├── tests/
│   ├── test_main.py
│   └── test_utils.py
├── data/
│   └── sample.csv
├── .gitignore
└── README.md
```

And your `.gitignore` file contains:

```
*.csv
*.log
venv/
```

If you call `file_extractor('my_project', include_patterns=['*.py'], exclude_patterns=['tests/'])`, here's what would happen:

- It would look inside `my_project`.
- It would see `*.py` in `include_patterns`, so it wants Python files.
- It sees `tests/` in `exclude_patterns`, so it will skip the `tests` directory.
- It reads `.gitignore` and knows to skip `*.csv` and `*.log` files.

The files that would be extracted are:

- `src/main.py`
- `src/utils.py`

The files that would be skipped are:

- `tests/test_main.py` (because of `exclude_patterns`)
- `tests/test_utils.py` (because of `exclude_patterns`)
- `data/sample.csv` (because of `.gitignore`)
- `README.md` (because it's not a `.py` file and no `include_patterns` were set to catch it)

What's Next? ➡

The File Extractor is our gatekeeper, ensuring our agent works with the right data. Now that we know how files are selected, in the next chapter, we'll dive into the Pydantic Models. These are like blueprints that help us structure the data our agent works with, making everything organized and predictable!

Chapter 4: Pydantic Models

Welcome back! We've explored how our documentation agent works, the logic behind its steps, and how it finds the right files. Now, let's talk about how we keep all the information organized and validated. This is where **Pydantic Models** come into play!

What are Pydantic Models?

Think of Pydantic Models as **blueprints** or **templates** for the data our agent uses. In any software project, especially one that deals with a lot of information, it's crucial to have a clear structure for that data. Pydantic Models help us define exactly what kind of data we expect, what its fields are, and what type of data each field should hold (like text, numbers, or lists).

Why is this important?

1. **Data Validation:** Pydantic automatically checks if the data we receive matches our defined structure. If you expect a number but get text, Pydantic will raise an error, preventing bugs and ensuring data integrity.
2. **Data Serialization/Deserialization:** Pydantic makes it easy to convert data between different formats, like Python dictionaries and JSON. This is super useful when communicating with APIs or saving data.
3. **Readability and Maintainability:** By defining our data structures clearly, our code becomes much easier to understand and manage. Anyone looking at the

Pydantic models can immediately grasp what kind of data is being handled.

In our Agent Documentor project, we use Pydantic Models to define the structure for:

- **Components:** What information we need about each code component (its name, description, and related files).
- **Relationships:** How different components interact with each other.
- **Analysis Outputs:** The structured results from our LLM analysis, like project overviews and ordered component lists.

Our Pydantic Models

Let's look at the Pydantic Models we've defined in `models.py`:

```
# models.py

from pydantic import BaseModel, Field


class Component(BaseModel):
    name: str = Field(description="concise name for the component")
    description: str = Field(description="beginner friendly description of the component")
    files: list[int] = Field(description="list of relevent files indices of the component")
```

This `Component` model is a blueprint for a single component identified in our code. It tells us that each component must have:

- `name`: A string representing the component's name.
- `description`: A string describing the component.
- `files`: A list of integers, where each integer is an index pointing to a file identified by our File Extractor.

The `Field` function is used here to add descriptions to each attribute, which helps document the model itself and can be used by tools.

```
class Components(BaseModel):
    components: list[Component] = Field(description="list of components")
```

The `Components` model is a container for multiple `Component` models. When our `component_segregator` node asks the LLM to identify components, it expects the output to be structured like this `Components` model, containing a list of individual `Component` objects.

```
class Relationship(BaseModel):
    from_component: int = Field(description="index of the source component or abstraction")
    to_component: int = Field(description="index of the target component or abstraction")
    label: str = Field(description="brief label for the interaction in just a few words")
```

This `Relationship` model defines how one component connects to another. It includes:

- `from_component`: The index of the component that initiates the relationship.
- `to_component`: The index of the component that is the target of the relationship.
- `label`: A short description of the relationship (e.g., "uses", "manages").

```
class RelationshipAnalysisOutput(BaseModel):
    overview: str = Field(description="A high-level `overview` of the project's main purpose and functionality in a few beginner-fri
    relationships: list[Relationship] = Field(description="list of `relationships` between the components")
```

This model is used for the output of the `component_relationship_analyser` node. It bundles together the overall project `overview` (a string) and a list of `relationships` (using our `Relationship` model).

```
class OrderedComponents(BaseModel):
    ordered_components: list[int] = Field(description="ordered list of component indices in a list, should include all components an
```

Finally, the `OrderedComponents` model is used to structure the output from the `component_ordering` node. It simply expects a list of integers, representing the indices of the components in the desired order for documentation.

How Pydantic is Used in the Agent

In the Agent Nodes Logic chapter, you saw how we used these models. For example, in `component_segregator`:

```
    component_segregation_llm_with_structured_output = component_segregation_llm.with_structured_output(Components)
    components = component_segregation_llm_with_structured_output.invoke(llm_context)
```

Here, we tell the LLM to output its response in the structure defined by our `Components` Pydantic model. Pydantic then takes the LLM's raw text output, parses it, validates it against the `Components` model, and if everything matches, it returns a Python object of type `Components`. If there's a mismatch (e.g., the LLM didn't provide a list of files for a component), Pydantic would raise an error, letting us know something went wrong.

This structured approach makes our agent much more reliable and easier to debug.

What's Next? ➡

We've now covered the essential building blocks of our Agent Documentor: the overall workflow, the logic within each node, how it finds files, and how it structures its data using Pydantic Models.

In our next chapter, we'll look at the File Extractor again, but this time we'll focus on how it's integrated into the agent's workflow to actually fetch the files needed for documentation. Get ready to see how the pieces fit together!

Generated by Kritagya Khandelwal