A2A Tutorial

This is the first chapter of our A2A tutorial! We'll learn about a powerful concept called "UI Components". Think of building with LEGOs: instead of placing each tiny brick individually to build a car, you use pre-built pieces like wheels, a body, and a windshield. UI components are like those pre-built LEGO pieces for your application's user interface (UI). They make building and maintaining the UI much simpler.

Let's imagine you're building a simple chat application. You'll need a header at the top, a place to display the chat messages, and an area to type and send new messages. Building all this from scratch would be tedious. UI components let us create these elements once and reuse them wherever needed.

Key Concepts:

Our chat application will use three main components:

1. **`header`:** Displays the application title and an icon.
2. **`conversation`:** Shows the chat messages and provides an input area for typing new messages.
3. **`page_scaffold`:** This is a structural component that puts everything together nicely, including a sidebar menu (we'll see what that is soon!).

Let's look at a simple `header` component:

```
import mesop as me

@me.content_component
def header(title: str, icon: str):
    """Header component"""
    with me.box(style=me.Style(display="flex", justify_content="space-between")):
        with me.box(style=me.Style(display="flex", flex_direction="row", gap=5)):
            me.icon(icon=icon)
            me.text(title, type="headline-5", style=me.Style(font_family="Google Sans"))
```

This code defines a function `header` that takes a title (string) and an icon (string) as input. It uses `mesop` (a UI library, don't worry about the details now) to create a header with an icon and the title. The `@me.content_component` decorator makes it a reusable UI component.

Now let's see how we use this `header` in our chat application:

```
header(title="A2A Chat", icon="message")
```

This single line adds a header to our application, displaying "A2A Chat" with a message icon. Simple, right?

A Deeper Look: The `conversation` Component

The `conversation` component is more complex, handling the chat messages and input area. It also relies on features we'll learn in later chapters, such as state management (Chapter 10: State Management) and interacting with A2A agents (Chapter 4: Agent). Let's look at a simplified version:

```
import mesop as me

@me.component
def conversation():
    """Simplified conversation component."""
    with me.box(style=me.Style(display="flex", flex_direction="column")):
        # Display chat messages here (details in later chapters)
        me.text("Chat messages will appear here...")
        me.input(label="Type your message...") # Input area for typing messages
```

This code shows a basic structure. The actual message display and sending logic will be covered in later chapters.

Putting It All Together: `page_scaffold`

The `page_scaffold` component is a container that arranges all the other components. It includes the sidebar for navigation:

```
import mesop as me

@me.content_component
def page_scaffold():
    """page scaffold component (simplified)"""
    # Simplified sidebar - details in later chapters
    me.text("Sidebar will go here...")
    with me.box(style=me.Style(flex="1")): # Main content area
        me.slot() # Placeholder for other components

#Example use:
page_scaffold()
header(title="A2A Chat", icon="message")
conversation()
```

`me.slot()` is a placeholder where other components (like `header` and `conversation`) will be placed. The `page_scaffold` makes sure that our sidebar and the main chat section are nicely positioned within the application page.

Internal Implementation (Simplified)

Let's see a simplified sequence diagram showing how these components interact:

```
sequenceDiagram
    participant User
    participant App
    participant Header
    participant Conversation
    participant PageScaffold

    User->>App: Loads the App
    App->>PageScaffold: Render Page
    PageScaffold->>Header: Render Header
    PageScaffold->>Conversation: Render Conversation
    Header-->>PageScaffold: Header rendered
    Conversation-->>PageScaffold: Conversation rendered
    PageScaffold-->>App: Page rendered
    App->>User: Displays UI
```

The code for these components is spread across several files: `header.py`, `conversation.py`, and `page_scaffold.py`. The `mesop` library handles the rendering and interaction between these components.

Conclusion

This chapter introduced the fundamental concept of UI components. We built a simple chat application using reusable components, `header`, `conversation`, and `page_scaffold`. In the next chapter, we'll delve into the A2A Client (Chapter 2: A2A Client) and how it interacts with our UI.

Building on our understanding of UI Components, let's learn how to connect our user interface to the powerful world of A2A agents. Imagine you're building a task management application. You need a way for your application (the "manager") to send tasks to workers (the "agents") and receive their results. The A2A Client is that bridge – it handles all the communication between your app and the agents.

Let's say our simple task is to translate text from English to Spanish. Our application needs to send the English text to a translation agent, wait for the result (the Spanish text), and display it to the user. This is where the A2A Client comes in.

Key Concepts

The A2A Client simplifies this process by providing a clean interface. It handles the complexities of sending requests to agents, receiving responses, and managing potential errors. Here are some key aspects:

1. **Communication:** The A2A Client communicates with A2A agents using a standardized protocol (Chapter 5: A2A Protocol). This protocol

ensures consistent communication regardless of the agent's specific implementation.

2. **Task Submission:** You use the client to submit tasks to agents. These tasks contain instructions and any necessary data.
3. **Result Retrieval:** The client retrieves the results of the tasks from the agents.
4. **Error Handling:** The client handles potential errors during communication or task execution. This makes your application more robust.

Using the A2A Client

Let's see how we can use the A2A Client to send a translation task and receive the result. We'll focus on a simplified example for clarity.

Here's a minimal example using the JavaScript client:

```
import { A2AClient } from "./client"; // Simplified import

const client = new A2AClient("http://localhost:8080"); // Replace with your server URL

async function translate(text) {
  const result = await client.sendTask({
    id: "translation-task-1",
    message: { text: text }, //The task input
  });
  console.log(result.output.text); // Access the translated text from the result
}

translate("Hello, world!"); //Call the function to translate
```

This code creates an `A2AClient` instance, connects to the A2A server, and sends a task to translate "Hello, world!". The `sendTask` method handles the communication with the agent. The result contains the Spanish translation, which we then print to the console.

This code snippet shows how straightforward it is to interact with an agent. The details of what happens internally (such as the low-level communication and error handling) are all handled by the client. The `sendTask` method makes it appear simple, hiding the complexity.

Internal Implementation (Simplified)

Let's look at a simplified sequence diagram to visualize the interaction:

```
sequenceDiagram
    participant User
    participant App
```

```
    participant A2A Client
    participant A2A Server
    participant Agent

    User->>App: Requests translation
    App->>A2A Client: sendTask("Hello, world!")
    A2A Client->>A2A Server: Translation request
    A2A Server->>Agent: Forward request
    Agent->>A2A Server: "Hola, mundo!"
    A2A Server->>A2A Client: Translation response
    A2A Client->>App: "Hola, mundo!"
    App->>User: Displays translation
```

The `A2AClient` uses HTTP requests (using the `fetch` API in JavaScript or `httpx` in Python) to communicate with the A2A Server. The server then forwards the request to the appropriate Agent. The agent processes the task and sends the result back through the server to the client. The client receives and interprets the response.

Here's a small snippet from the JavaScript client showing how a request is made:

```
private async _makeHttpRequest<Req extends A2ARequest>(method, params) {
  const response = await this.fetchImpl(this.baseUrl, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ jsonrpc: "2.0", method, params }),
  });
  return response;
}
```

This code shows a simplified HTTP POST request to the A2A server. The details of the JSON-RPC protocol are hidden here, abstracted for simplicity in the `sendTask` method. The full implementation handles JSON-RPC formatting, error responses, and streaming capabilities which are discussed in later chapters. The Python client uses similar principles.

Conclusion

This chapter introduced the A2A Client, a crucial component for interacting with agents in the A2A framework. We saw how easily we can send tasks and receive results, without needing to deal with the underlying communication details. In the next chapter, we'll explore the concept of a Task in more detail.

Building on our understanding of the A2A Client, let's explore the core concept of a "Task" in the A2A framework. Think of a task as a single unit of work you want an Agent to perform. It's like giving a to-do item to a helpful assistant. You provide instructions (a message), and the assistant sends back the results (artifacts). The A2A protocol manages the entire lifecycle of this task, ensuring

reliable completion.

Let's use a simple example: imagine you need to translate a sentence from English to Spanish. You'd send this sentence as a task to a translation agent. The agent performs the translation, and returns the Spanish version as the task's result.

Key Concepts

A task in A2A involves several key elements:

1. **Task ID:** Each task is uniquely identified by an ID. This ID is used to track and manage the task throughout its lifecycle. Think of it as a label for your to-do item.
2. **Message:** This is the instruction or data you send to the agent. In our translation example, the message would be the English sentence you want translated.
3. **Artifacts:** These are the results the agent produces. In our example, the artifact would be the translated Spanish sentence. Artifacts can also be files or complex data structures.
4. **Task State:** This indicates the current stage of the task: submitted, working, completed, failed, canceled, etc. It's like the status of your to-do item (e.g., "In progress," "Completed").
5. **Session ID (Optional):** Tasks can optionally belong to a session. A session groups related tasks. This is helpful for conversational tasks where multiple tasks build on each other.

Using Tasks

Let's see how to submit a translation task using the A2A client. Here's a simplified example using the JavaScript client:

```
import { A2AClient } from "./client";

const client = new A2AClient("http://localhost:8080"); // Replace with your server URL

async function translate(text) {
  const taskId = "translation-task-1";
  const result = await client.sendTask({
    id: taskId,
    message: { parts: [{ type: "text", text }] }, //Simple message with text
  });
  console.log(result.artifacts[0].parts[0].text); // Access the translated text
}
```

```
translate("Hello, world!");
```

This code creates a task with an ID, a message containing the English sentence, and sends it to the agent via the `sendTask` method of the A2A client. The `result` contains the translated text from the returned artifact.

This shows how simple it is to send and receive tasks using the A2A client. The complexities of communication and error handling are handled internally by the client.

Internal Implementation (Simplified)

Here's a simplified sequence diagram showing what happens when a task is submitted:

```
sequenceDiagram
    participant User
    participant App
    participant A2A Client
    participant A2A Server
    participant Agent

    User->>App: Submits translation task
    App->>A2A Client: sendTask(...)
    A2A Client->>A2A Server: Task request (with message)
    A2A Server->>Agent: Forwards the task
    Agent->>A2A Server: Task result (with artifact)
    A2A Server->>A2A Client: Task response (with artifact)
    A2A Client->>App: Returns the result
    App->>User: Displays translation
```

The A2A client packages the task request and sends it to the A2A server using HTTP requests. The server then forwards the task to the appropriate agent. Once the agent completes the task, it sends the result back to the server, which then sends it to the client. The client interprets the response and provides the results to the application.

A simplified snippet from the Python A2A client showing task submission:

```python
async def send_task(self, params: TaskSendParams) -> Task:
    request = SendTaskRequest(params=params) #Create request
    response = await self.send_request(request) #Send request
    if response.error:
        raise A2AClientError(f"Error sending task: {response.error}")
    return response.result  #Return the task
```

This is a highly simplified view. The `send_request` method handles the actual HTTP communication and JSON-RPC messaging.

Conclusion

This chapter introduced the fundamental concept of a "Task" in A2A. We learned how to create and submit tasks, and understand their lifecycle. In the next chapter, we'll explore Agents in more detail, learning how they process these tasks.

Building upon our understanding of the A2A Client and Tasks, let's explore the core concept of an "Agent" within the A2A framework. Think of an agent as an independent, specialized AI application – a skilled worker capable of performing specific tasks. Imagine you need to translate text, write code, or summarize a document. Each of these requires a different skillset; this is where agents come in. Each agent excels at a particular job and communicates using the A2A protocol (Chapter 5: A2A Protocol).

Let's use a simple example: Suppose you want to translate an English sentence into Spanish. You would send this sentence as a task to a *translation agent*. This agent, specialized in translation, performs the task and returns the translated Spanish sentence as the result.

Key Concepts

An agent is characterized by several key aspects:

1. **Specialization:** Agents are designed to perform specific tasks. A translation agent is different from a code-generation agent or a summarization agent. Each has its own unique capabilities and expertise.
2. **A2A Protocol Communication:** Agents communicate with the A2A server and client using the A2A protocol. This ensures consistent and reliable communication.
3. **Task Processing:** Agents receive tasks from the A2A server and process them based on their specialization.
4. **Artifact Generation:** After processing a task, the agent generates *artifacts* which are the results of the task. This could be translated text, generated code, a summary, or any other relevant output.
5. **Framework Agnostic:** Agents can be built using various frameworks such as Google ADK, LangChain, or CrewAI, providing flexibility and options depending on your needs and preferences.

Using an Agent

Let's create a simplified example. The following Python code shows how to send a task to a translation agent and receive the translated text. We'll focus on a high-level interaction using a hypothetical `A2AClient` and `TranslationAgent`.

```python
from a2a_client import A2AClient #Simplified import

client = A2AClient("http://localhost:8080") #Connect to server.
agent = client.get_agent("translation_agent") #Get the Translation Agent

task = agent.submit_task({"text": "Hello, world!"}) # Submit task with input text.

result = task.get_result() # Wait for and get the result.
print(result['translated_text']) # Access and print the translated text.
```

This code first establishes a connection with the A2A server, fetches the translation agent, submits a translation task with the English sentence "Hello, world!", waits for the result, and finally prints the translated Spanish sentence. The specifics of task management and communication are hidden within the `A2AClient` and `TranslationAgent` abstractions.

Internal Implementation (Simplified)

The following sequence diagram illustrates the interaction between the components when a task is sent to an agent:

```
sequenceDiagram
    participant User
    participant App
    participant A2A Client
    participant A2A Server
    participant Translation Agent

    User->>App: Requests translation
    App->>A2A Client: Submit translation task
    A2A Client->>A2A Server: Task request
    A2A Server->>Translation Agent: Forward task
    activate Translation Agent
    Translation Agent->>Translation Agent: Process translation
    deactivate Translation Agent
    Translation Agent->>A2A Server: Translation result
    A2A Server->>A2A Client: Task result
    A2A Client->>App: Display translation
    App->>User: Display "Hola, mundo!"
```

The A2A Client sends the task to the A2A Server, which forwards it to the appropriate agent. The agent processes the task, generates the translated text (the artifact), and returns it to the server. The server, in turn, sends the result back to the client, and then to the application for display to the user.

A simplified example from a hypothetical agent implementation shows how a task might be processed:

```python
class TranslationAgent:
    def process_task(self, task_data):
        text_to_translate = task_data["text"]
        # ...complex translation logic using a translation library...
        translated_text = self.translate(text_to_translate) # Placeholder
        return {"translated_text": translated_text}

    def translate(self, text):
        #simplified placeholder
        if text == "Hello, world!":
            return "Hola, mundo!"
        else:
            return "Translation not implemented for this input."
```

This code snippet shows the core logic within a translation agent. It receives the task data, extracts the text to translate, performs the translation (using a placeholder here for simplicity), and returns the translated text within a dictionary. This dictionary is then used to construct the artifact. The implementation details of the actual translation are omitted for brevity.

Conclusion

This chapter introduced the crucial concept of an "Agent" in the A2A framework. We learned how agents specialize in specific tasks, communicate using the A2A protocol, and process tasks to generate artifacts. In the next chapter, we'll delve into the specifics of the A2A protocol itself: Chapter 5: A2A Protocol.

Building on our understanding of Agents, let's explore the A2A protocol – the communication standard that allows different AI agents to talk to each other. Imagine you have a team of specialized agents: one for translation, one for summarization, and one for question answering. The A2A protocol is the common language that lets them work together seamlessly, even if they're built using different technologies or from different vendors.

Let's say you want to translate a document, summarize it, and then answer a question about the summary. You'd need to coordinate three separate agents. This is where the A2A protocol comes in. It handles the communication and data exchange between these agents, making it possible to chain their operations together.

Key Concepts

The A2A protocol is based on JSON-RPC 2.0, a standard way for applications to communicate over a network. Here are the essential pieces:

1. **JSON-RPC 2.0:** This is the underlying communication framework. It defines the structure of requests and responses, making sure everyone

speaks the same "language." Think of it as the grammar and vocabulary of our inter-agent conversations.

2. **`tasks/send`:** This is the main method for submitting a task to an agent. You send a request with a message containing instructions and data, and the agent sends back the results. It's like giving an instruction to a worker and getting the finished product.

3. **Tasks:** Each individual job for an agent is a "task". It includes an ID, the message, and the resulting artifacts. It's like assigning a specific to-do item on a list to your team of agents.

4. **Artifacts:** These are the results of a task – the output generated by an agent. It could be translated text, a summary, or an answer to a question. Think of these as the completed "to-do" items.

5. **Messages:** These are the instructions and data sent to and from agents. A message can contain multiple "parts" like text, files, or structured data (JSON). Messages are the core data units being exchanged.

6. **Task States:** This tracks the status of a task: submitted, working, completed, failed, etc. It's like seeing the progress of a to-do item (e.g., "In progress", "Done").

Using the A2A Protocol

Let's see a simple example using the Python client to send a translation task:

```
from a2a_client import A2AClient #Simplified import

client = A2AClient("http://localhost:8080") #Simplified connection
task_id = "my_translation_task"
message = {"text": "Hello, world!"}
response = client.send_task(task_id, message) #send the task

print(response.result.artifacts[0].parts[0].text) #Access the result
```

This code creates an A2AClient, sends a task (with an ID and a message containing text), and prints the translated text from the returned artifact. The details of the underlying JSON-RPC communication are hidden within the client.

This example shows how to use the `tasks/send` method. The client handles the JSON-RPC packaging and communication. The response contains the task's result including the translated text.

Internal Implementation (Simplified)

Here's a simplified sequence diagram showing how the A2A protocol works:

```
sequenceDiagram
```

```
participant App
participant A2A Client
participant A2A Server
participant Translation Agent

App->>A2A Client: send_task("my_translation_task", {"text": "Hello, world!"})
A2A Client->>A2A Server: JSON-RPC request (tasks/send)
A2A Server->>Translation Agent: Task request
activate Translation Agent
Translation Agent->>A2A Server: JSON-RPC response (task result)
deactivate Translation Agent
A2A Server->>A2A Client: JSON-RPC response
A2A Client->>App: Task result ("Hola, mundo!")
```

The App calls the `send_task` method on the A2A Client. The client packages the request using JSON-RPC 2.0, sends it to the A2A Server, which then forwards it to the appropriate agent. The agent processes the task, generates an artifact, and sends back a response via the server. The client then delivers the result to the application.

Here's a simplified Python snippet showing how the client builds the JSON-RPC request (from the file `a2a_client.py`):

```python
async def send_request(self, request: A2ARequest): #Simplified method
  # ... (HTTP request setup) ...
  async with self.session.post(self.server_url, json=request.model_dump_json()) as response:
    # ... (Handle HTTP response) ...
```

This code shows how the client sends the HTTP request to the A2A server. The request is formatted as a JSON-RPC request object. The full implementation also handles error cases and other details.

Conclusion

This chapter introduced the A2A protocol, the backbone of communication between agents. We learned how it simplifies the interaction between agents using JSON-RPC and the `tasks/send` method. In the next chapter, we'll explore the A2A Server that facilitates this communication.

Following our exploration of the A2A Protocol, let's understand the crucial role of the A2A Server. Imagine you have multiple Agents, each specialized in a different task (like translation, summarization, or code generation). You need a central hub to manage communication between your A2A Client and these agents. The A2A Server acts as this central hub, routing requests, handling responses, and managing the entire task lifecycle.

Let's say you want to translate a sentence, then summarize the translation. You'd need to send the sentence to a translation agent, get the result, then send

that result to a summarization agent. The A2A Server handles this communication flow, making it much simpler.

Key Concepts

The A2A Server acts as a bridge, handling several important functions:

1. **Request Routing:** It receives requests from the A2A client and forwards them to the correct agent based on the task's instructions. This is like a receptionist directing calls to the right department.
2. **Response Handling:** After the agent completes the task, the server receives the results (artifacts) and sends them back to the client. It's like the receptionist relaying messages back to the caller.
3. **Task Management:** The server keeps track of the tasks' status (submitted, working, completed, failed, etc.). This is like a manager monitoring project progress.
4. **Streaming Support:** For long-running tasks, the server enables streaming results, allowing the client to receive updates as the agent works. This is like getting progress reports during a lengthy project.
5. **Error Management:** The server handles errors that occur during task processing or communication, providing helpful error messages to the client. It's like a problem-solver identifying and reporting issues.

Using the A2A Server

You don't directly interact with the A2A Server; the A2A Client handles the communication. However, understanding the server's role is crucial. The client sends requests to a specific URL (e.g., `http://localhost:8080`), and the server at that address handles the routing and response.

Here's a simplified example of how a client might interact with the server (using the JavaScript client):

```
import { A2AClient } from "./client"; //Simplified import

const client = new A2AClient("http://localhost:8080"); // Server URL

async function translateAndSummarize(text){
  // ... (Client handles task submission to the server) ...
}
translateAndSummarize("This is a long sentence to translate and then summarize.");
```

This code creates a client connected to the A2A server running at `http://localhost:8080`. The `translateAndSummarize` function (details omitted for brevity) uses the client to interact with the server which then routes to appropriate agents. The server is responsible for forwarding requests

and delivering responses back to the client. The client doesn't directly communicate with individual agents.

Internal Implementation (Simplified)

Let's visualize the interaction using a sequence diagram:

```
sequenceDiagram
    participant Client
    participant A2A Server
    participant Translation Agent
    participant Summarization Agent

    Client->>A2A Server: Translation request
    A2A Server->>Translation Agent: Forward request
    activate Translation Agent
    Translation Agent-->>A2A Server: Translated text
    deactivate Translation Agent
    A2A Server->>Client: Translated text
    Client->>A2A Server: Summarization request (using translated text)
    A2A Server->>Summarization Agent: Forward request
    activate Summarization Agent
    Summarization Agent-->>A2A Server: Summary
    deactivate Summarization Agent
    A2A Server->>Client: Summary
```

The client sends a request to the A2A Server. The server forwards it to the Translation Agent, receives the result, and sends it back to the client. The client then sends a second request for summarization, and the process repeats. The server manages the entire flow.

A simplified Python snippet from the server (from `server.py`):

```python
async def _process_request(self, request: Request):
    try:
        body = await request.json()
        # ... (Server handles request routing and response based on the request method) ...
    except Exception as e:
        # ... (Error handling) ...
```

This code shows a simplified request handling function. The server receives the request, parses the JSON, determines the request type, routes the request to the appropriate agent (or handler), and sends the response back. Error handling and task management are significantly simplified here. The full implementation is much more complex, involving task persistence, streaming, and extensive error management.

Conclusion

This chapter provided a high-level overview of the A2A Server and its crucial role in facilitating communication between clients and agents. It's the central hub that manages tasks, routes requests, and handles responses in the A2A framework. In the next chapter, we'll discuss Task Handlers, which are the core logic behind agent task processing.

Building on our understanding of Agents and how they perform tasks, let's explore the "Task Handler"—the core logic that dictates how an agent processes a specific request. Think of it as the detailed instruction manual for a single agent task. It defines exactly how the agent works and how it produces results.

Let's imagine a simple scenario: you want an agent to translate an English sentence into Spanish. The task handler would contain the specific instructions for this translation—it would define how the agent receives the English sentence, uses its translation capabilities (maybe leveraging an external library), and finally generates the Spanish translation as its output.

Key Concepts

The task handler is implemented as an asynchronous generator, meaning it can yield intermediate results as it works. This is especially important for long-running tasks, enabling the A2A Server to provide real-time updates to the A2A Client.

1. **Asynchronous Generator:** The task handler is an `async generator`. This means it can pause its execution, yield a result, and then resume later. This allows for gradual updates as the task progresses.
2. **Yielding Updates:** The generator *yields* updates about the task's progress. These updates can be either partial task statuses or complete artifacts. Think of it like receiving progress reports during a complex project.
3. **Task Context:** The handler receives a `TaskContext` object which provides all the necessary information about the task: the task itself, the user's message, the task's history, and a function to check for cancellation requests. It's like having all the details and tools needed for the job.

Using a Task Handler

Let's illustrate with a simplified example of a translation task handler in JavaScript. This example won't include an actual translation library for simplicity.

```
// Simplified Task Handler (JavaScript)
async function* translationHandler(context) {
  // Access the English sentence from the context
```

```
  const englishSentence = context.userMessage.parts[0].text;
  console.log("Received English sentence:", englishSentence);

  // Simulate translation work... (replace with actual translation code)
  await new Promise(resolve => setTimeout(resolve, 1000)); // Simulate delay

  // Simulate generating the translated sentence.
  const spanishSentence = "Hola, mundo!";

  // Yield a partial update (optional)
  yield { state: "working", message: { parts: [{ type: "text", text: "Translating..." }] } };

  // Yield the translated sentence as an artifact
  yield { parts: [{ type: "text", text: spanishSentence }], index:0, append:false };

  console.log("Translation complete.");
  //Return the final task (for non streaming modes)
  return context.task;
}
```

This code shows a simple task handler function that receives context information, simulates translation work (using `setTimeout` as a placeholder), yields a "working" state update, and then yields the translated sentence. This is an async generator, indicated by the `async function*` syntax. The function can be started using something like `for await(const update of translationHandler(context)) {...}`. Each iteration will provide a new update.

Internal Implementation (Simplified)

The A2A server uses the task handler to process incoming tasks. Let's illustrate with a sequence diagram.

```
sequenceDiagram
    participant Client
    participant A2A Server
    participant Task Handler
    participant Translation Agent

    Client->>A2A Server: Translation request
    A2A Server->>Task Handler: Execute handler with context
    activate Task Handler
    Task Handler->>Translation Agent: Translate
    activate Translation Agent
    Translation Agent-->>Task Handler: Translated text
    deactivate Translation Agent
    Task Handler->>A2A Server: Yield updates and final result
```

```
    deactivate Task Handler
    A2A Server->>Client: Stream updates and final result
```

The client sends a request to the A2A server. The server then calls the appropriate task handler with a `TaskContext` object providing all the needed information. The handler uses this context to interact with the agent (or other external resources). The handler uses `yield` to send updates back to the server, which then streams these updates to the client. Once complete, the handler returns.

A simplified snippet from the A2A Server code (from `server.py` - highly simplified):

```python
async def process_task(self, task, handler):
    context = TaskContext(task=task, ...) #Create the context
    async for update in handler(context):
        await self.update_task(task, update) #Update the task in store
```

This shows a simplified task processing loop on the server. It creates a `TaskContext`, calls the handler, and iterates through yielded updates, updating the task's status and streaming updates to the client. The actual implementation is considerably more complex, handling errors and various other edge cases.

Conclusion

This chapter explained the crucial role of the Task Handler as the core logic for agent task processing. We saw how it's implemented as an asynchronous generator that yields updates, enabling streaming capabilities and providing real-time feedback. In the next chapter, we will delve into the concept of Messages and how they structure communication within the A2A framework.

Building on our understanding of Tasks, let's explore the "Message"—the fundamental unit of communication in the A2A framework. Think of a message as a container for instructions and data exchanged between a Client and an Agent. It's how you tell an agent what to do and how you receive its results. A message isn't just plain text; it's a structured container that can hold various types of data.

Let's use a simple example: you want to translate the sentence "Hello, world!" into Spanish. You'll send this sentence as part of a message to a translation agent. The agent will process the message, perform the translation, and return the translated Spanish sentence within a response message.

Key Concepts

A message in A2A is composed of several key elements:

1. **Parts:** A message can contain multiple *parts*. Each part represents a distinct piece of information. A part could be plain text, a file, or structured data (like JSON). It's like attaching different files or text sections to an email.
2. **Role:** Each message has a designated *role*, indicating whether it's from the *user* (the client) or the *agent*. This helps track the communication flow. It's like labeling an email as "Sent" or "Received."
3. **Metadata:** Each message can include optional *metadata*, providing additional context or information. This is like adding extra details or labels to your email for better organization. It's completely optional but can be very useful.

Using Messages

Let's look at how to create a message to send a translation task. Here's a simplified example using the JavaScript client:

```
//Create a simple text message part
const textPart = { type: "text", text: "Hello, world!" };

//Create a message
const message = {
  role: "user",
  parts: [textPart],
};

//Send the message as part of a task.  (Details omitted for brevity.)
// ... client.sendTask({id: "myTask", message: message }) ...
```

This code creates a message with a single text part containing "Hello, world!". The `role` is set to "user" because it's the client sending the message. This message is then sent as part of a task using the `client.sendTask` method which was explained in earlier chapters.

This demonstrates how straightforward it is to create and use messages. Each message part is a simple object. This message is packed into a task, but this message is the core data unit of communication.

Internal Implementation (Simplified)

Let's visualize the message flow using a sequence diagram. We'll focus on the core message handling, omitting details of the A2A protocol and server aspects.

```
sequenceDiagram
    participant Client
    participant Agent
    participant Task
```

```
    participant Message

    Client->>Task: Submits task with message
    Task->>Agent: Sends Message (role: user)
    activate Agent
    Agent->>Message: Processes message parts
    Agent->>Task: Sends response Message (role: agent)
    deactivate Agent
    Task->>Client: Delivers response message
```

The client submits a task, sending a user message to the agent. The agent receives and processes the message. It then generates its response and sends a response message back through the Task to the client.

Here's a simplified part of the JavaScript `client.sendTask` method demonstrating message handling:

```javascript
// ... other code ...
const response = await this._makeHttpRequest("tasks/send", {
    id: params.id,
    message: params.message,
    // ... other parameters ...
});
// ... handle the response ...
```

This snippet shows how the client's `sendTask` method makes an HTTP request to the A2A Server. The `message` is included directly in the request parameters. This message will be sent to the agent for processing. The server then forwards this message to the agent using the A2A protocol, further details of which are omitted for simplicity.

Conclusion

This chapter introduced the concept of a "Message" in A2A. We saw how messages, composed of parts and metadata, form the core units of communication between clients and agents. In the next chapter, we'll explore Artifacts, the results produced by agents and returned in their response messages.

Following our discussion of Messages, let's explore "Artifacts"—the tangible results produced by agents after completing a task. Think of an artifact as the final product, the output of an agent's hard work. It's the answer to your question, the translated text, the generated code, or any other data the agent creates to fulfill a task. These can even be sent in parts (streaming), just like a large file might download piece-by-piece.

Let's imagine you ask a translation agent to convert an English sentence into Spanish. The translated Spanish sentence is the *artifact*—the concrete outcome of the agent's work.

Key Concepts

Artifacts are structured to allow for flexibility and efficient handling of different data types:

1. **Parts:** An artifact can consist of multiple *parts*. Each part can be a different data type, like plain text, an image, or a JSON object. This enables the agent to return complex results beyond a simple text response. Think of it as a package containing different items – each part being an individual item.
2. **Streaming Support:** For long tasks, agents can send artifacts in parts (streaming). The A2A Client receives these parts gradually, allowing for a more responsive user experience. This is like receiving a large file download – not all at once, but piece by piece.
3. **Metadata:** Optional metadata can provide additional context about the artifact. This could include information like the file type, creation timestamp, or any other relevant details. This is like adding labels to a file to help organize and understand its contents.

Using Artifacts

Let's see how to access the artifact returned by a translation agent. Here's a simplified example using the JavaScript client:

```
// ... (previous code to send a translation task) ...

const result = await client.sendTask({id:"mytask", message: { parts: [{ type: "text", text:

const translatedText = result.artifacts[0].parts[0].text;  // Access the translated text
console.log("Translated text:", translatedText); // Output: Hola, mundo! (or similar)
```

This code snippet sends a translation task, and then accesses the translated text from the artifact. The `result` object contains the details of the task, including an array of artifacts. In this simplified case, we assume there's only one artifact, and it contains one part (the translated text).

This shows how to access the artifact after the task is complete. The `result.artifacts` array is where the output of the agent's operation is stored.

Internal Implementation (Simplified)

Here's a simplified view of how artifacts are handled:

```
sequenceDiagram
    participant Client
    participant Server
    participant Agent
    participant Artifact
```

```
    Client->>Server: Task request
    Server->>Agent: Forward request
    activate Agent
    Agent->>Artifact: Create artifact
    Agent->>Server: Send artifact parts (streaming)
    deactivate Agent
    Server->>Client: Stream artifact parts
    Client->>Client: Process artifact parts
```

The client sends a task request to the server. The server forwards it to the agent. The agent processes the task and creates an artifact, sending its parts to the server as a stream. The server then streams these parts to the client, which processes them as they arrive.

Here's a simplified code snippet (Python) showing artifact creation within an agent:

```python
#Simplified Agent code
def process_task(self, task_data):
    # ... (translation logic) ...
    translated_text = "Hola, mundo!" # Placeholder translation
    artifact = Artifact(parts=[Part(type="text", text=translated_text)]) #Create artifact
    return artifact #Return the artifact
```

This is a minimal example showing artifact creation. The agent's logic generates the translated text, and this text is then packaged into an artifact object. The `Artifact` class is used to create and handle the structured artifact object. This function then returns the artifact to the server for delivery to the client.

Conclusion

This chapter covered the concept of "Artifacts" in A2A—the tangible results produced by agents. We learned about parts, streaming, and metadata, essential for handling diverse outputs. In the next chapter, we will look at State Management and how to track the progress of long-running tasks.

Building on our understanding of Agents and how they process tasks, let's tackle the concept of "State Management" in A2A. Imagine you're using a complex multi-agent system for a project – you might have multiple agents working concurrently, each updating the project's progress in different ways. How do you track it all? That's where state management comes in. It provides a central place to monitor and keep track of the overall progress and data involved in the process.

Let's use a concrete example: You're building a chatbot using several agents – one for understanding questions, one for generating answers, and one for sending final responses. State management allows you to track the current state of the conversation, the status of each task handled by each agent, and the results

generated, enabling your application to keep up with the agents' work in real-time.

Key Concepts

State management in A2A involves tracking various aspects of your multi-agent system:

1. **Task States:** This tracks the progress of each task – submitted, in progress, completed, failed, canceled, etc. It's like a to-do list that shows the current status of each item.
2. **Agent States:** This manages the status of each agent. Are they currently working on a task? Are they available? Have they encountered any errors? This is like monitoring each worker on your team and knowing their status.
3. **Conversation States:** For applications with conversational flows, this tracks the overall state of a conversation. It records messages, task progress, and other context, creating a detailed history of the interaction. This is like having a record of the entire conversation for reference and follow-up.
4. **Application State:** This encompasses the state of the entire application, including user settings, the active task, open conversations, and other relevant data. It's the overview of the whole system.

Using State Management

Let's show a simplified example of how you might use state management in a Python application. We'll leverage the `mesop` UI library for managing the state in a user interface. We'll focus on tracking a single task's progress.

First, we'll define a state class using `mesop`:

```
import mesop as me

@me.stateclass
class TaskState:
  task_id: str = ""
  status: str = "pending" # pending, running, success, failure
```

This defines a `TaskState` class with fields for `task_id` and `status`. The `@me.stateclass` decorator makes this class compatible with the `mesop` UI library.

Next, we can update the state as the task progresses:

```
# ... inside the task processing loop ...
task_state.status = "running"  #Update status to running
# ... after successful task completion ...
```

```
task_state.status = "success" # Update status to success
# ... after failure...
task_state.status = "failure" # Update status to failure
```

This code demonstrates how to update the `TaskState`. `mesop` automatically reflects changes in the UI, providing real-time feedback to the user. The user interface will update itself to reflect changes in the status field.

Internal Implementation (Simplified)

Let's visualize the state management flow.

```
sequenceDiagram
    participant User
    participant App
    participant Task Handler
    participant State Manager
    participant UI

    User->>App: Initiates Task
    App->>Task Handler: Process Task
    activate Task Handler
    loop Task Progress
        Task Handler->>State Manager: Update Task State
        State Manager->>UI: Update UI
    end
    Task Handler-->>App: Task Complete
    deactivate Task Handler
    App->>User: Displays Results
```

The user initiates a task, the application sends the task to a handler. The handler periodically updates the state manager, which then updates the UI. This provides real-time feedback to the user.

A simplified snippet from a hypothetical `State Manager` class (from `state_manager.py`):

```
class StateManager:
    def update_task_state(self, task_id, new_status):
        # ... (logic to update the task state in the store) ...
        # ... (notify UI using mesop) ...
```

This is a highly simplified example. The actual implementation would involve mechanisms to persistently store the state and handle concurrency efficiently. This would likely involve a database or in-memory store which is accessible by the application and the UI.

Conclusion

This chapter introduced state management in A2A. We learned how to track the progress of tasks and the status of agents, which is essential for building interactive multi-agent systems. This will enable the creation of more sophisticated and responsive applications. In the next chapter, we'll move on to more advanced topics such as error handling.

The A2A project aims to create *interoperability* between different **AI agents**. It achieves this by defining a common **communication protocol** (A2A Protocol) that enables agents built using various frameworks (like LangChain, Google ADK) to exchange information and collaborate on **tasks**. The system includes a **server** and **client** for managing agent interactions and a user **interface** for task assignment and monitoring.

Source Repository: https://github.com/kumar045/A2A.git

```
flowchart TD
    A0["A2A Protocol"]
    A1["Agent"]
    A2["Task"]
    A3["Task Handler"]
    A4["A2A Server"]
    A5["A2A Client"]
    A6["Message"]
    A7["Artifact"]
    A8["State Management"]
    A9["UI Components"]
    A0 -- "Defines communication" --> A1
    A0 -- "Manages lifecycle" --> A2
    A0 -- "Structures format" --> A6
    A0 -- "Defines structure" --> A7
    A1 -- "Executes logic" --> A3
    A1 -- "Registers with" --> A4
    A2 -- "Assigned by" --> A5
    A2 -- "Processed by" --> A4
    A3 -- "Produces" --> A7
    A4 -- "Uses" --> A0
    A4 -- "Sends/Receives" --> A6
    A4 -- "Updates state" --> A8
    A5 -- "Uses" --> A0
    A5 -- "Submits" --> A2
    A5 -- "Reads from" --> A8
    A5 -- "Uses" --> A9
    A8 -- "Updates" --> A9
    A6 -- "Part of" --> A2
    A7 -- "Result of" --> A2
```

Chapters

1. UI Components
2. A2A Client
3. Task
4. Agent
5. A2A Protocol
6. A2A Server
7. Task Handler
8. Message
9. Artifact
10. State Management