# ToolStorage

The `ToolStorage` module provides a structured and efficient way to manage and utilize various tool functions. It is designed to store tool functions, manage settings, and ensure smooth registration and retrieval of tools. This module is particularly useful in applications that require dynamic management of a collection of functions, such as plugin systems, modular software, or any application where functions need to be registered and called dynamically.

## Class: ToolStorage

The `ToolStorage` class is the core component of the module. It provides functionalities to add, retrieve, and list tool functions as well as manage settings.

### Attributes

Attribute	Type	Description
`verbose`	`bool`	A flag to enable verbose logging.
`tools`	`List[Callable]`	A list of tool functions.
`_tools`	`Dict[str, Callable]`	A dictionary that stores the tools, where the key is the tool name and the value is the tool function.
`_settings`	`Dict[str, Any]`	A dictionary that stores the settings, where the key is the setting name and the value is the setting value.

### Methods

`__init__`

Initializes the `ToolStorage` instance.

Parameter	Type	Default	Description
`verbose`	`bool`	`None`	A flag to enable verbose logging.
`tools`	`List[Callable]`	`None`	A list of tool functions to initialize the storage with.
`*args`	`tuple`	`None`	Additional positional arguments.
`**kwargs`	`dict`	`None`	Additional keyword arguments.

#### `add_tool`

Adds a tool to the storage.

Parameter	Type	Description
`func`	`Callable`	The tool function to be added.

Raises:

- `ValueError`: If a tool with the same name already exists.

#### `get_tool`

Retrieves a tool by its name.

Parameter	Type	Description
`name`	`str`	The name of the tool to retrieve.

Returns:

- `Callable`: The tool function.

Raises:

- `ValueError`: If no tool with the given name is found.

#### `set_setting`

Sets a setting in the storage.

Parameter	Type	Description
`key`	`str`	The key for the setting.
`value`	`Any`	The value for the setting.

#### `get_setting`

Gets a setting from the storage.

Parameter	Type	Description
`key`	`str`	The key for the setting.

Returns:

- `Any`: The value of the setting.

Raises:

- `KeyError`: If the setting is not found.

#### `list_tools`

Lists all registered tools.

Returns:

- `List[str]`: A list of tool names.

## Decorator: tool_registry

The `tool_registry` decorator registers a function as a tool in the storage.

Parameter	Type	Description
`storage`	`ToolStorage`	The storage instance to register the tool in.

Returns:

- `Callable`: The decorator function.

## Usage Examples

### Full Example

```python
from swarms import ToolStorage, tool_registry

storage = ToolStorage()

# Example usage
@tool_registry(storage)
def example_tool(x: int, y: int) -> int:
    """
    Example tool function that adds two numbers.

    Args:
        x (int): The first number.
        y (int): The second number.

    Returns:
        int: The sum of the two numbers.
    """
    return x + y
```

```python
# Query all the tools and get the example tool
print(storage.list_tools())  # Should print ['example_tool']
# print(storage.get_tool('example_tool'))  # Should print <function example_tool at 0x...>

# Find the tool by names and call it
print(storage.get_tool("example_tool"))  # Should print 5


# Test the storage and querying
if __name__ == "__main__":
    print(storage.list_tools())  # Should print ['example_tool']
    print(storage.get_tool("example_tool"))  # Should print 5
    storage.set_setting("example_setting", 42)
    print(storage.get_setting("example_setting"))  # Should print 42

```

### Basic Usage

#### Example 1: Initializing ToolStorage and Adding a Tool

```python
from swarms.tools.tool_registry import ToolStorage, tool_registry
```

```
# Initialize ToolStorage
storage = ToolStorage()

# Define a tool function
@tool_registry(storage)
def add_numbers(x: int, y: int) -> int:
    return x + y

# List tools
print(storage.list_tools())  # Output: ['add_numbers']

# Retrieve and use the tool
add_tool = storage.get_tool('add_numbers')
print(add_tool(5, 3))  # Output: 8
```

### Advanced Usage

#### Example 2: Managing Settings

```python
# Set a setting
storage.set_setting('max_retries', 5)

# Get a setting
```

max_retries = storage.get_setting('max_retries')
print(max_retries) # Output: 5

### Error Handling

#### Example 3: Handling Errors in Tool Retrieval

```python
try:
    non_existent_tool = storage.get_tool('non_existent')
except ValueError as e:
    print(e)  # Output: No tool found with name: non_existent
```

#### Example 4: Handling Duplicate Tool Addition

```python
try:
    @tool_registry(storage)
    def add_numbers(x: int, y: int) -> int:
        return x + y
except ValueError as e:
    print(e)  # Output: Tool with name add_numbers already exists.
```

## Conclusion

The `ToolStorage` module provides a robust solution for managing tool functions and settings. Its design allows for easy registration, retrieval, and management of tools, making it a valuable asset in various applications requiring dynamic function handling. The inclusion of detailed logging ensures that the operations are transparent and any issues can be quickly identified and resolved.