Maggie AI Assistant: State and Event Bus Systems

Table of Contents

General Overview

The Maggie AI Assistant is built on a solid foundation of two interconnected core systems: the State Management system and the Event Bus system. These systems work together to create a flexible, responsive, and maintainable application architecture.

Key Concepts

- **Finite State Machine**: The application follows a finite state machine (FSM) pattern, where the system can be in exactly one of a predefined set of states at any given time.
- **Event-Driven Architecture**: Components communicate through events rather than direct method calls, promoting loose coupling and modularity.
- **State-Aware Components**: Application components can react to state changes, enabling coordinated behavior across the system.

- **Prioritized Event Handling**: Events are processed according to priority levels, ensuring critical events are handled promptly.

Benefits of the Architecture

- **Predictable Application Behavior**: The state system ensures that the application behaves consistently based on its current state.
- **Decoupled Components**: Components interact through events, reducing direct dependencies.
- **Extensibility**: New components can be added without modifying existing code.
- **Traceability**: State transitions and events can be logged and monitored for debugging and analysis.
- **Resource Optimization**: Resources can be allocated and freed based on application state.

State System

The state system is implemented as a finite state machine that manages the application's lifecycle and coordinates resource allocation and component behavior.

State Enumeration

The `State` enum defines the possible states the application can be in:

```
class State(Enum):
    INIT = auto()        # Initial state during startup
    STARTUP = auto()     # Application is starting up
    IDLE = auto()        # Application is idle, waiting for activation
    LOADING = auto()     # Application is loading resources
    READY = auto()       # Application is ready for input
    ACTIVE = auto()      # Application is actively processing
    BUSY = auto()        # Application is busy with intensive processing
    CLEANUP = auto()     # Application is cleaning up resources
    SHUTDOWN = auto()    # Application is shutting down
```

Each state also has visual styling properties for UI representation:

```
@property
def bg_color(self) -> str:
    colors = {
        State.INIT: '#E0E0E0',
        State.STARTUP: '#B3E5FC',
        State.IDLE: '#C8E6C9',
        State.LOADING: '#FFE0B2',
        State.READY: '#A5D6A7',
        State.ACTIVE: '#FFCC80',
        State.BUSY: '#FFAB91',
        State.CLEANUP: '#E1BEE7',
        State.SHUTDOWN: '#EF9A9A'
    }
    return colors.get(self, '#FFFFFF')
```

State Transitions

State transitions are represented by the `StateTransition` class, which captures:

- The originating state (`from_state`)
- The destination state (`to_state`)
- The trigger that caused the transition
- The timestamp when the transition occurred
- Optional metadata for additional context

```
@dataclass
class StateTransition:
    from_state: State
    to_state: State
    trigger: str
    timestamp: float
    metadata: Dict[str, Any] = field(default_factory=dict)
```

Transitions also include properties for UI animation when visualizing state changes:

```
@property
def animation_type(self) -> str:
    if self.to_state == State.SHUTDOWN:
        return 'fade'
    elif self.to_state == State.BUSY:
        return 'bounce'
    else:
        return 'slide'
```

State Manager

The `StateManager` class is the core of the state system, responsible for:

1. Maintaining the current state
2. Validating and executing state transitions
3. Notifying components about state changes
4. Maintaining a history of state transitions
5. Executing handler functions during transitions

```
class StateManager(IStateProvider):
    def __init__(self, initial_state: State = State.INIT, event_bus: Any = None):
        self.current_state = initial_state
        self.event_bus = event_bus
        # ... additional initialization ...
```

The `StateManager` enforces rules about valid transitions:

```
self.valid_transitions = {
    State.INIT: [State.STARTUP, State.IDLE, State.SHUTDOWN],
    State.STARTUP: [State.IDLE, State.READY, State.CLEANUP, State.SHUTDOWN],
    State.IDLE: [State.STARTUP, State.READY, State.CLEANUP, State.SHUTDOWN],
    # ... other valid transitions ...
}
```

When a state transition is requested, the manager validates it before executing:

```
def transition_to(self, new_state: State, trigger: str,
                 metadata: Dict[str, Any] = None) -> bool:
    with self._lock:
        if new_state == self.current_state:
            self.logger.debug(f"Already in state {new_state.name}, ignoring transition")
            return True

        if not self.is_valid_transition(self.current_state, new_state):
            error_message = f"Invalid transition from {self.current_state.name} to {new_state.nam
            self.logger.warning(error_message)
            # ... error handling ...
            return False

        old_state = self.current_state
        self.current_state = new_state
        # ... execute handlers and notify ...
```

State-Aware Components

The `StateAwareComponent` class serves as a base class for components that need to react to state changes:

```
class StateAwareComponent:
    def __init__(self, state_manager: StateManager):
        self.state_manager = state_manager
        self.logger = logging.getLogger(self.__class__.__name__)
        self._registered_handlers = []
        self._register_state_handlers()
```

Components can register handlers for specific states:

```
def _register_state_handlers(self) -> None:
    for state in State:
        method_name = f"on_enter_{state.name.lower()}"
        if hasattr(self, method_name) and callable(getattr(self, method_name)):
            handler = getattr(self, method_name)
            self.state_manager.register_state_handler(state, handler, True)
            self._registered_handlers.append((state, handler, True))

        method_name = f"on_exit_{state.name.lower()}"
        if hasattr(self, method_name) and callable(getattr(self, method_name)):
            handler = getattr(self, method_name)
            self.state_manager.register_state_handler(state, handler, False)
            self._registered_handlers.append((state, handler, False))
```

This enables components to define methods like `on_enter_active()` or `on_exit_busy()` that will be automatically called when corresponding state transitions occur.

Event Bus System

The Event Bus system provides a publish-subscribe mechanism for decoupled communication between components. It manages event distribution, prioritization, and processing.

Event Publishing and Subscription

The `EventBus` class implements the core event handling functionality:

```
class EventBus(IEventPublisher):
    def __init__(self):
        self.subscribers: Dict[str, List[Tuple[int, Callable]]] = {}
        self.queue = queue.PriorityQueue()
        self.running = False
        self._worker_thread = None
        self._lock = threading.RLock()
        # ... additional initialization ...
```

Components can subscribe to specific event types:

```
def subscribe(self, event_type: str, callback: Callable,
              priority: int = EventPriority.NORMAL) -> None:
    with self._lock:
        if event_type not in self.subscribers:
            self.subscribers[event_type] = []
        self.subscribers[event_type].append((priority, callback))
        self.subscribers[event_type].sort(key=lambda x: x[0])
        self.logger.debug(f"Subscription added for event type: {event_type}")
```

And publish events:

```
def publish(self, event_type: str, data: Any = None, **kwargs) -> None:
    if isinstance(data, dict) and self._correlation_id:
        data = data.copy()
        if 'correlation_id' not in data:
            data['correlation_id'] = self._correlation_id
    priority = kwargs.get('priority', EventPriority.NORMAL)
    self.queue.put((priority, (event_type, data)))
    self.logger.debug(f"Event published: {event_type}")
```

Event Priorities

Events can be processed with different priorities, defined in the `EventPriority` class:

```
class EventPriority:
    HIGH = 0         # Critical events that need immediate attention
    NORMAL = 10      # Standard events
    LOW = 20         # Low-priority events that can be delayed
    BACKGROUND = 30  # Background events with lowest priority
```

Event Processing

The `EventBus` processes events in a dedicated thread, respecting priorities:

```
def _process_events(self) -> None:
    while self.running:
        try:
            events_batch = []
            try:
                priority, event = self.queue.get(timeout=.05)
                if event is None:
                    break
                events_batch.append((priority, event))
                self.queue.task_done()
            except queue.Empty:
                time.sleep(.001)
                continue

            # Process more events in batch if available
            batch_size = 10
            while len(events_batch) < batch_size:
                try:
                    priority, event = self.queue.get(block=False)
                    if event is None:
                        break
                    events_batch.append((priority, event))
                    self.queue.task_done()
                except queue.Empty:
                    break

            # Dispatch events
            for (priority, event) in events_batch:
                if event is None:
                    continue
                event_type, data = event
                self._dispatch_event(event_type, data)
        except Exception as e:
            # ... error handling ...
```

Correlation IDs and Context

The event system supports correlation IDs to track related events:

```
def set_correlation_id(self, correlation_id: Optional[str]) -> None:
    self._correlation_id = correlation_id

def get_correlation_id(self) -> Optional[str]:
    return self._correlation_id
```

This allows tracing events that are part of the same logical operation or user interaction.

Implementation Guide

This section provides practical guidance for implementing and working with the state and event systems.

Working with States

To work with the state system, you typically need to:

1. Access the `StateManager` instance
2. Check or monitor the current state
3. Request state transitions when appropriate

Accessing the StateManager

The `StateManager` is typically available through the `ServiceLocator`:

```
from maggie.service.locator import ServiceLocator

state_manager = ServiceLocator.get('state_manager')
```

Checking the Current State

```
current_state = state_manager.get_current_state()
if current_state == State.IDLE:
    # Do something specific to IDLE state
```

Requesting State Transitions

```
# Request transition to READY state triggered by "user_input"
success = state_manager.transition_to(State.READY, "user_input")
if not success:
    # Handle transition failure
```

Handling State Transitions

There are two main approaches to handling state transitions:

1. **Register handlers with the StateManager**:

```
def handle_active_entry(transition):
    print(f"Entered ACTIVE state from {transition.from_state.name}")

state_manager.register_state_handler(State.ACTIVE, handle_active_entry, True)
```

2. **Extend StateAwareComponent**:

```
class MyComponent(StateAwareComponent):
    def __init__(self, state_manager):
        super().__init__(state_manager)

    def on_enter_active(self, transition):
        print(f"Entered ACTIVE state from {transition.from_state.name}")
```

```
    def on_exit_busy(self, transition):
        print(f"Exiting BUSY state to {transition.to_state.name}")
```

Publishing and Subscribing to Events

Working with the event system involves:

1. Accessing the `EventBus` instance
2. Subscribing to events of interest
3. Publishing events when relevant

Accessing the EventBus

```
from maggie.service.locator import ServiceLocator

event_bus = ServiceLocator.get('event_bus')
```

Subscribing to Events

```
def handle_wake_word(data=None):
    print("Wake word detected!")

event_bus.subscribe('wake_word_detected', handle_wake_word, EventPriority.HIGH)
```

Publishing Events

```
# Publish an event with data
event_bus.publish('command_detected', {'command': 'hello', 'confidence': 0.95})

# Publish an event without data
event_bus.publish('wake_word_detected')

# Publish with priority
event_bus.publish('error_logged', error_data, priority=EventPriority.HIGH)
```

Creating State-Aware Event-Driven Components

Many components will need to be both state-aware and event-driven. The framework provides base classes to streamline this:

```
class MyComponent(StateAwareComponent, EventListener):
    def __init__(self, state_manager, event_bus):
        StateAwareComponent.__init__(self, state_manager)
        EventListener.__init__(self, event_bus)
        self._register_event_handlers()

    def _register_event_handlers(self):
        event_handlers = [
            ('wake_word_detected', self._handle_wake_word, EventPriority.HIGH),
```

```python
            ('command_detected', self._handle_command, EventPriority.NORMAL)
        ]

        for (event_type, handler, priority) in event_handlers:
            self.listen(event_type, handler, priority=priority)

    def _handle_wake_word(self, data=None):
        current_state = self.state_manager.get_current_state()
        if current_state == State.IDLE:
            self.state_manager.transition_to(State.READY, 'wake_word_detected')

    def _handle_command(self, command):
        if self.state_manager.get_current_state() == State.READY:
            self.state_manager.transition_to(State.ACTIVE, 'command_detected')
            # Process command...

    def on_enter_active(self, transition):
        # Handle entering active state
        pass
```

Code Examples

The following examples illustrate common patterns and scenarios for working with the state and event systems.

Example 1: Basic State Transitions

This example shows a simple component that manages state transitions based on user input:

```python
from maggie.core.state import State, StateAwareComponent
from maggie.utils.logging import ComponentLogger

class InputHandler(StateAwareComponent):
    def __init__(self, state_manager):
        super().__init__(state_manager)
        self.logger = ComponentLogger('InputHandler')

    def process_user_input(self, text):
        """Process user input and update application state accordingly."""
        current_state = self.state_manager.get_current_state()

        if current_state == State.IDLE:
            # If idle, wake up on any input
            self.logger.info("Waking up from IDLE state")
            self.state_manager.transition_to(State.READY, "user_input")
            return True

        elif current_state == State.READY:
            # If ready, start processing
            self.logger.info("Processing user input")
            self.state_manager.transition_to(State.ACTIVE, "command_processing")

            # Simulate operation that takes time
            import time
```

```python
            time.sleep(1)  # Simulate processing

            # Return to ready state
            self.state_manager.transition_to(State.READY, "processing_complete")
            return True

        elif current_state == State.ACTIVE or current_state == State.BUSY:
            self.logger.warning("Already processing, cannot handle new input")
            return False

        return False

    def on_enter_ready(self, transition):
        self.logger.info("System is now ready for input")

    def on_enter_active(self, transition):
        self.logger.info("System is actively processing")

    def on_exit_active(self, transition):
        self.logger.info("Finished processing")
```

Example 2: Component Responding to State Changes

This example demonstrates a resource manager that optimizes resource allocation based on the application state:

```python
from maggie.core.state import State, StateTransition, StateAwareComponent
from maggie.utils.logging import ComponentLogger

class ResourceOptimizer(StateAwareComponent):
    def __init__(self, state_manager):
        super().__init__(state_manager)
        self.logger = ComponentLogger('ResourceOptimizer')

    def on_enter_init(self, transition: StateTransition) -> None:
        """Minimal resource allocation during initialization."""
        self.logger.info("Initializing with minimal resources")
        self._set_process_priority('low')
        self._free_gpu_memory()

    def on_enter_idle(self, transition: StateTransition) -> None:
        """Free up resources when idle."""
        self.logger.info("Freeing resources in IDLE state")
        self._set_process_priority('below_normal')
        self._free_gpu_memory()
        self._reduce_memory_usage()

    def on_enter_active(self, transition: StateTransition) -> None:
        """Allocate resources for active processing."""
        self.logger.info("Allocating resources for ACTIVE state")
        self._set_process_priority('above_normal')
        self._optimize_for_active_state()

    def on_enter_busy(self, transition: StateTransition) -> None:
        """Maximize resources for intensive processing."""
        self.logger.info("Maximizing resources for BUSY state")
        self._set_process_priority('high')
```

```python
            self._optimize_for_busy_state()

    def on_exit_busy(self, transition: StateTransition) -> None:
        """Clean up after intensive processing."""
        if transition.to_state == State.READY:
            self.logger.info("Cleaning up after BUSY state")
            self._free_gpu_memory()

    def _set_process_priority(self, priority: str) -> None:
        """Set the process priority based on state."""
        try:
            import psutil
            process = psutil.Process()

            priority_map = {
                'low': psutil.IDLE_PRIORITY_CLASS,
                'below_normal': psutil.BELOW_NORMAL_PRIORITY_CLASS,
                'normal': psutil.NORMAL_PRIORITY_CLASS,
                'above_normal': psutil.ABOVE_NORMAL_PRIORITY_CLASS,
                'high': psutil.HIGH_PRIORITY_CLASS
            }

            if priority in priority_map:
                process.nice(priority_map[priority])
                self.logger.debug(f"Set process priority to {priority}")
        except ImportError:
            self.logger.warning("psutil not available, cannot set priority")
        except Exception as e:
            self.logger.error(f"Error setting process priority: {e}")

    def _free_gpu_memory(self) -> None:
        """Free GPU memory."""
        try:
            import torch
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
                self.logger.debug("Cleared GPU memory cache")
        except ImportError:
            self.logger.debug("PyTorch not available, skipping GPU memory cleanup")
        except Exception as e:
            self.logger.error(f"Error clearing GPU memory: {e}")

    def _reduce_memory_usage(self) -> None:
        """Reduce memory usage."""
        try:
            import gc
            gc.collect()
            self.logger.debug("Garbage collection performed")
        except Exception as e:
            self.logger.error(f"Error reducing memory usage: {e}")

    def _optimize_for_active_state(self) -> None:
        """Apply optimizations for active state."""
        self.logger.debug("Applied active state optimizations")

    def _optimize_for_busy_state(self) -> None:
        """Apply optimizations for busy state."""
        self.logger.debug("Applied busy state optimizations")
```

Example 3: Event Publishing and Subscription

This example shows a component that publishes events and another that subscribes to them:

```python
from maggie.core.event import EventEmitter, EventListener, EventPriority
from maggie.utils.logging import ComponentLogger

class SpeechDetector(EventEmitter):
    def __init__(self, event_bus):
        super().__init__(event_bus)
        self.logger = ComponentLogger('SpeechDetector')
        self.is_listening = False

    def start_listening(self):
        """Start listening for speech."""
        self.is_listening = True
        self.logger.info("Started listening for speech")
        self.emit('speech_detector_started')

    def stop_listening(self):
        """Stop listening for speech."""
        self.is_listening = False
        self.logger.info("Stopped listening for speech")
        self.emit('speech_detector_stopped')

    def simulate_wake_word_detection(self):
        """Simulate detecting a wake word."""
        if not self.is_listening:
            self.logger.warning("Cannot detect wake word: not listening")
            return False

        self.logger.info("Wake word detected!")
        # Emit event with high priority
        self.emit('wake_word_detected', None, priority=EventPriority.HIGH)
        return True

    def simulate_speech_detection(self, text):
        """Simulate detecting speech."""
        if not self.is_listening:
            self.logger.warning("Cannot detect speech: not listening")
            return False

        self.logger.info(f"Detected speech: {text}")
        # Emit event with normal priority and data
        self.emit('speech_detected', {
            'text': text,
            'confidence': 0.95,
            'timestamp': time.time()
        })
        return True

class SpeechHandler(EventListener):
    def __init__(self, event_bus, speech_processor=None):
        super().__init__(event_bus)
        self.logger = ComponentLogger('SpeechHandler')
        self.speech_processor = speech_processor
        self._register_event_handlers()

    def _register_event_handlers(self):
```

```python
        """Register handlers for speech-related events."""
        event_handlers = [
            ('wake_word_detected', self._handle_wake_word, EventPriority.HIGH),
            ('speech_detected', self._handle_speech, EventPriority.NORMAL),
            ('speech_detector_started', self._handle_detector_started, EventPriority.LOW),
            ('speech_detector_stopped', self._handle_detector_stopped, EventPriority.LOW),
        ]

        for (event_type, handler, priority) in event_handlers:
            self.listen(event_type, handler, priority=priority)

    def _handle_wake_word(self, data=None):
        """Handle wake word detection."""
        self.logger.info("Handling wake word detection")
        # Here we might activate the assistant

    def _handle_speech(self, data):
        """Handle detected speech."""
        text = data.get('text', '')
        confidence = data.get('confidence', 0)
        self.logger.info(f"Processing speech: '{text}' (confidence: {confidence:.2f})")

        if self.speech_processor:
            self.speech_processor.process(text)

    def _handle_detector_started(self, data=None):
        """Handle speech detector start event."""
        self.logger.debug("Speech detector started")

    def _handle_detector_stopped(self, data=None):
        """Handle speech detector stop event."""
        self.logger.debug("Speech detector stopped")
```

Example 4: Combined State and Event Handling

This example demonstrates the integration of state and event systems in a comprehensive component:

```python
from maggie.core.state import State, StateTransition, StateAwareComponent
from maggie.core.event import EventListener, EventEmitter, EventPriority
from maggie.utils.logging import ComponentLogger, log_operation

class AssistantCore(StateAwareComponent, EventListener, EventEmitter):
    def __init__(self, state_manager, event_bus):
        StateAwareComponent.__init__(self, state_manager)
        EventListener.__init__(self, event_bus)
        EventEmitter.__init__(self, event_bus)

        self.logger = ComponentLogger('AssistantCore')
        self._register_event_handlers()

    def _register_event_handlers(self):
        """Register handlers for events."""
        event_handlers = [
            ('wake_word_detected', self._handle_wake_word, EventPriority.HIGH),
            ('speech_detected', self._handle_speech, EventPriority.NORMAL),
            ('command_processed', self._handle_command_processed, EventPriority.NORMAL)
        ]
```

```python
        for (event_type, handler, priority) in event_handlers:
            self.listen(event_type, handler, priority=priority)

    @log_operation(component='AssistantCore')
    def _handle_wake_word(self, data=None):
        """Handle wake word detection event."""
        current_state = self.state_manager.get_current_state()

        if current_state == State.IDLE:
            self.logger.info("Wake word detected in IDLE state, transitioning to READY")
            self.state_manager.transition_to(State.READY, "wake_word_detected")
            self.emit('assistant_activated')
        elif current_state == State.READY:
            self.logger.info("Wake word acknowledged in READY state")
            self.emit('ready_for_input')

    @log_operation(component='AssistantCore')
    def _handle_speech(self, data):
        """Handle detected speech event."""
        text = data.get('text', '')
        current_state = self.state_manager.get_current_state()

        if current_state == State.READY:
            self.logger.info(f"Processing speech in READY state: {text}")
            self.state_manager.transition_to(State.ACTIVE, "speech_processing")

            # Process the speech as a command
            self._process_command(text)

        elif current_state == State.ACTIVE:
            self.logger.info(f"Additional speech received while ACTIVE: {text}")
            # Queue the speech for later processing or handle interruption

        else:
            self.logger.warning(f"Received speech in unexpected state {current_state.name}: {text

    @log_operation(component='AssistantCore')
    def _process_command(self, text):
        """Process a command from speech input."""
        # Simulate complex processing
        self.logger.info(f"Processing command: {text}")

        # If the processing will be intensive, transition to BUSY
        if len(text) > 20:  # Simple heuristic for complex request
            self.state_manager.transition_to(State.BUSY, "complex_processing")

        # Simulate processing time
        import time
        time.sleep(1)

        # Emit completion event
        self.emit('command_processed', {'command': text, 'success': True})

    def _handle_command_processed(self, data):
        """Handle command processing completion."""
        success = data.get('success', False)
        current_state = self.state_manager.get_current_state()
```

```python
        if success:
            if current_state == State.ACTIVE or current_state == State.BUSY:
                self.logger.info("Command processed successfully, returning to READY state")
                self.state_manager.transition_to(State.READY, "command_complete")
        else:
            self.logger.warning("Command processing failed")
            if current_state == State.BUSY:
                self.state_manager.transition_to(State.READY, "command_failed")

    def on_enter_idle(self, transition: StateTransition):
        """Handle entering IDLE state."""
        self.logger.info("Entered IDLE state, waiting for wake word")

    def on_enter_ready(self, transition: StateTransition):
        """Handle entering READY state."""
        self.logger.info("Entered READY state, waiting for commands")
        self.emit('ready_for_input')

    def on_enter_active(self, transition: StateTransition):
        """Handle entering ACTIVE state."""
        self.logger.info("Entered ACTIVE state, processing input")

    def on_enter_busy(self, transition: StateTransition):
        """Handle entering BUSY state."""
        self.logger.info("Entered BUSY state, performing intensive processing")

    def on_exit_busy(self, transition: StateTransition):
        """Handle exiting BUSY state."""
        self.logger.info(f"Exiting BUSY state to {transition.to_state.name}")
```

Reference

This section provides a reference of the key APIs and best practices for the state and event systems.

State System API

State Enum

The `State` enum defines the possible application states:

- `INIT` - Initial application state during startup
- `STARTUP` - Application is starting up and initializing components
- `IDLE` - Application is idle, waiting for activation
- `LOADING` - Application is loading resources or models
- `READY` - Application is ready to process input
- `ACTIVE` - Application is actively processing
- `BUSY` - Application is performing intensive processing
- `CLEANUP` - Application is cleaning up resources
- `SHUTDOWN` - Application is shutting down

StateTransition Class

The `StateTransition` dataclass represents a transition between states:

- `from_state: State` - The originating state
- `to_state: State` - The destination state
- `trigger: str` - The event or action that triggered the transition
- `timestamp: float` - When the transition occurred
- `metadata: Dict[str, Any]` - Additional context for the transition

StateManager Class

The `StateManager` class manages state transitions:

- `__init__(initial_state: State, event_bus: Any)` - Initialize with starting state
- `transition_to(new_state: State, trigger: str, metadata: Dict[str, Any])` - Request state transition
- `get_current_state() -> State` - Get the current application state
- `is_valid_transition(from_state: State, to_state: State) -> bool` - Check if transition is valid
- `register_state_handler(state: State, handler: Callable, is_entry: bool)` - Register entry/exit handler
- `register_transition_handler(from_state: State, to_state: State, handler: Callable)` - Register transition handler
- `get_transition_history(limit: int = 10) -> List[StateTransition]` - Get recent transitions

StateAwareComponent Class

The `StateAwareComponent` base class for state-aware components:

- `__init__(state_manager: StateManager)` - Initialize with state manager reference
- `on_enter_<state_name>(transition: StateTransition)` - Define state entry handlers
- `on_exit_<state_name>(transition: StateTransition)` - Define state exit handlers
- `get_component_state() -> Dict[str, Any]` - Get component state information

Event Bus API

EventPriority Class

The `EventPriority` class defines constants for event priorities:

- `HIGH = 0` - Highest priority events
- `NORMAL = 10` - Normal priority events
- `LOW = 20` - Low priority events
- `BACKGROUND = 30` - Lowest priority background events

EventBus Class

The `EventBus` class implements the central event system:

- `__init__()` - Initialize the event bus
- `subscribe(event_type: str, callback: Callable, priority: int)` - Subscribe to events
- `unsubscribe(event_type: str, callback: Callable) -> bool` - Remove subscription
- `publish(event_type: str, data: Any = None, **kwargs)` - Publish an event
- `start() -> bool` - Start the event processing thread

- `stop() -> bool` - Stop the event processing thread
- `set_correlation_id(correlation_id: Optional[str])` - Set correlation ID for tracking
- `get_correlation_id() -> Optional[str]` - Get current correlation ID
- `add_event_filter(event_type: str, filter_func: Callable)` - Add event filter
- `remove_event_filter(event_type: str, filter_id: str) -> bool` - Remove event filter

EventEmitter Class

The `EventEmitter` base class for components that publish events:

- `__init__(event_bus: EventBus)` - Initialize with event bus reference
- `emit(event_type: str, data: Any = None, priority: int = EventPriority.NORMAL)` - Emit event
- `set_correlation_id(correlation_id: Optional[str])` - Set correlation ID for this emitter
- `get_correlation_id() -> Optional[str]` - Get emitter's correlation ID
- `cleanup()` - Clean up resources

EventListener Class

The `EventListener` base class for components that subscribe to events:

- `__init__(event_bus: EventBus)` - Initialize with event bus reference
- `listen(event_type: str, callback: Callable, priority: int = EventPriority.NORMAL)` - Subscribe
- `stop_listening(event_type: str = None, callback: Callable = None)` - Unsubscribe
- `add_filter(event_type: str, filter_func: Callable)` - Add event filter
- `remove_filter(event_type: str, filter_id: str) -> bool` - Remove event filter
- `cleanup()` - Clean up and unsubscribe from all events

Common Events

The following predefined events are used within the system:

- `STATE_CHANGED_EVENT = 'state_changed'` - Emitted when application state changes
- `STATE_ENTRY_EVENT = 'state_entry'` - Emitted when entering a state
- `STATE_EXIT_EVENT = 'state_exit'` - Emitted when exiting a state
- `TRANSITION_COMPLETED_EVENT = 'transition_completed'` - Emitted after a successful transition
- `TRANSITION_FAILED_EVENT = 'transition_failed'` - Emitted when a transition fails
- `UI_STATE_UPDATE_EVENT = 'ui_state_update'` - Emitted to update UI with state changes
- `INPUT_ACTIVATION_EVENT = 'input_activation'` - Emitted when input is activated
- `INPUT_DEACTIVATION_EVENT = 'input_deactivation'` - Emitted when input is deactivated

Best Practices

When working with the state and event systems, follow these best practices:

1. **Keep Components Decoupled**
    - Communicate via events rather than direct method calls
    - Components should not assume the existence of other components
2. **Respect State Constraints**
    - Operations should check current state before proceeding

- Use valid transitions defined by the state manager
- Be mindful of resource allocation in different states

3. **Prioritize Events Appropriately**

    - Use `EventPriority.HIGH` sparingly for truly critical events
    - Most events should use `EventPriority.NORMAL`
    - Background tasks should use `EventPriority.LOW` or `EventPriority.BACKGROUND`

4. **Use Correlation IDs for Tracing**

    - Set correlation IDs for related events
    - This helps with debugging and tracking user interactions

5. **Handle Errors Gracefully**

    - Always catch exceptions in event handlers
    - Publish error events when something goes wrong
    - Return to a stable state after errors

6. **Clean Up Resources**

    - Listen for state changes to clean up resources
    - Implement `cleanup()` methods for all components
    - Ensure event listeners are unsubscribed when no longer needed

7. **Test State Transitions**

    - Verify that components transition correctly
    - Test invalid transitions and ensure they're rejected
    - Check that state-specific behaviors are correct

8. **Document Events**

    - Document all events a component publishes
    - Document expected event data structure
    - Include information about when events are published

9. **Use Logging**

    - Log state transitions for debugging
    - Log important events and their data
    - Use appropriate log levels based on importance

10. **Follow Naming Conventions**

```
-   Use past tense for events (e.g., `wake_word_detected`)
-   Use clear state names (e.g., `ACTIVE` vs `RUNNING`)
-   Use descriptive trigger names (e.g., `user_command` vs `trigger`)
```

Following these best practices will help create a robust, maintainable, and decoupled application architecture.