Web UI Design Document - Mythline

Overview

This document outlines the design and implementation plan for a web-based user interface for the Mythline storytelling system. The web UI will provide browser-based access to the two primary CLI workflows: Research Story and Create Story.

Design Decisions

Technology Stack

Backend:

- Framework: FastAPI
- Rationale: Native async support, WebSocket capabilities, automatic API documentation, perfect match for existing async agents

Frontend:

- Framework: React
- Rationale: Component-based architecture, rich ecosystem, excellent for real-time updates

Authentication:

- Approach: No authentication (single-user deployment)
- Rationale: Simplifies initial implementation, suitable for local/personal use

Storage:

- Approach: Filesystem-based (continue existing pattern)
- Rationale: No new dependencies, leverages existing context/memory system

CLI Analysis Summary

Research Story CLI (`src/ui/cli/research_story.py`)

Characteristics:

- Interactive chat loop with the StoryResearcher agent
- Synchronous execution with `input()` prompts
- Session-based with context memory persistence
- Uses MCP servers: web-search, web-crawler, filesystem
- Delegates to sub-agents: NarratorAgent, DialogCreatorAgent, UserPreferenceAgent
- Output: Unstructured text responses
- Session management: New/resume/specific via CLI args

User Flow:

```
```

```
Start → Choose session → Chat loop → Ask questions → Get responses → Exit
```

Create Story CLI (`src/ui/cli/create_story.py`)

Characteristics:

- Non-interactive batch process with StoryCreatorAgent
- Asynchronous execution using Pydantic Graph workflow
- Session ID derived from subject name
- Requires pre-existing research file: `output/{subject}/research.md`
- Output: Structured JSON file: `output/{subject}/story.json`
- Progress displayed via colored console logging

Graph Workflow:

1. GetStoryResearch → Read research.md
2. CreateTODO → Generate story plan via StoryPlannerAgent
3. Loop: GetNextTODO → CreateStorySegment → ReviewOutput → WriteToFile
4. End when all TODOs complete

User Flow:

```
Start → Provide subject + player name → Validate research exists → Execute graph →
Watch progress → Complete
```

Architecture Design

Directory Structure

```
src/ui/web/
├── backend/
│   ├── __init__.py
│   ├── main.py                    # FastAPI application entry point
│   ├── routers/
│   │   ├── __init__.py
│   │   ├── research.py            # Research Story API endpoints
│   │   ├── story.py               # Create Story API endpoints
│   │   └── files.py               # File management endpoints
│   ├── services/
│   │   ├── __init__.py
│   │   ├── research_service.py    # Business logic for research
│   │   └── story_service.py       # Business logic for story creation
│   ├── models/
│   │   ├── __init__.py
│   │   └── api_models.py          # Pydantic request/response models
│   └── websocket/
│       ├── __init__.py
```

```
│       ├── research_ws.py         # Research WebSocket handler
│       └── story_ws.py            # Story progress WebSocket handler
└── frontend/
    ├── public/
    ├── src/
    │   ├── App.jsx                # Main React component
    │   ├── main.jsx               # React entry point
    │   ├── components/
    │   │   ├── ResearchChat/
    │   │   │   ├── ResearchChat.jsx       # Main research interface
    │   │   │   ├── MessageList.jsx        # Chat message display
    │   │   │   ├── MessageInput.jsx       # Message input box
    │   │   │   └── SessionSelector.jsx    # Session dropdown
    │   │   ├── StoryCreator/
    │   │   │   ├── StoryCreator.jsx       # Main story creator interface
    │   │   │   ├── JobForm.jsx            # Subject/player input form
    │   │   │   ├── ProgressMonitor.jsx    # Real-time progress display
    │   │   │   └── StoryViewer.jsx        # Story JSON viewer
    │   │   ├── FileExplorer/
    │   │   │   └── FileExplorer.jsx       # Browse output files
    │   │   └── common/
    │   │       ├── Layout.jsx             # App layout wrapper
    │   │       └── Navigation.jsx         # Navigation bar
    │   ├── services/
    │   │   ├── api.js                     # HTTP client (axios/fetch)
    │   │   └── websocket.js               # WebSocket client wrapper
    │   ├── hooks/
    │   │   ├── useResearchChat.js         # Research chat state hook
    │   │   └── useStoryJob.js             # Story job state hook
    │   └── styles/
    │       └── index.css                  # Global styles
    ├── package.json
    ├── vite.config.js
    └── index.html
```

Backend API Specification

Research Story API

Create Session

```
POST /api/research/sessions
Request: { "session_id": "optional_custom_id" }
Response: { "session_id": "20250110_143022", "message_count": 0 }
```

List Sessions

```
GET /api/research/sessions
Response: {
  "sessions": [
    { "session_id": "20250110_143022", "message_count": 5, "last_updated": "2025-
01-10T14:35:00Z" }
  ]
}
```

Get Session Messages

```
GET /api/research/sessions/{session_id}
Response: {
  "session_id": "20250110_143022",
  "messages": [
    { "role": "user", "content": "Research shadowglen" },
    { "role": "assistant", "content": "..." }
  ]
}
```

Send Message (Streaming)

```
POST /api/research/sessions/{session_id}/message
Request: { "content": "Tell me about shadowglen" }
Response: SSE stream
  data: {"type": "token", "content": "Shadowglen"}
  data: {"type": "token", "content": " is"}
  data: {"type": "done"}
```

Delete Session

```
DELETE /api/research/sessions/{session_id}
Response: { "success": true }
```

WebSocket Connection

```
WS /ws/research/{session_id}
Send: { "type": "message", "content": "Research shadowglen" }
Receive: { "type": "token", "content": "..." }
Receive: { "type": "tool_call", "tool": "create_narration", "args": {...} }
Receive: { "type": "done" }
```

Create Story API

Submit Job

```
POST /api/story/jobs
Request: { "subject": "shadowglen", "player": "Sarephine" }
Response: { "job_id": "shadowglen", "status": "running" }
```

List Jobs

```
GET /api/story/jobs
Response: {
  "jobs": [
    {
      "job_id": "shadowglen",
      "status": "running",
      "progress": { "current": 5, "total": 12 },
      "started_at": "2025-01-10T14:40:00Z"
    }
  ]
}
```

Get Job Status

```
GET /api/story/jobs/{subject}
Response: {
  "job_id": "shadowglen",
  "status": "running",
  "progress": { "current": 5, "total": 12, "current_task": "Creating dialogue for
quest 2" },
  "logs": [
    { "level": "info", "message": "Processing todo 5/12", "timestamp": "..." }
  ]
}
```

Cancel Job

```
DELETE /api/story/jobs/{subject}
Response: { "success": true, "message": "Job cancelled" }
```

WebSocket Connection

```
WS /ws/story/{subject}
Receive: { "type": "progress", "current": 1, "total": 12, "task": "Reading
research" }
Receive: { "type": "log", "level": "info", "message": "Research loaded
successfully" }
Receive: { "type": "review", "score": 0.92, "retry": false }
Receive: { "type": "complete", "story_path": "output/shadowglen/story.json" }
Receive: { "type": "error", "message": "...", "traceback": "..." }
```

File Management API

List Research Files

```
GET /api/files/research
Response: {
  "files": [
    { "name": "shadowglen", "path": "output/shadowglen/research.md", "size":
15230, "modified": "..." }
  ]
}
```

List Story Files

```
GET /api/files/stories
Response: {
  "files": [
    { "name": "shadowglen", "path": "output/shadowglen/story.json", "size": 45120,
"modified": "..." }
  ]
}
```

Download File

```
GET /api/files/{path}
Response: File download (application/octet-stream)
```

Frontend Component Design

Research Chat Interface

Components:

- `ResearchChat.jsx` - Main container

    - `SessionSelector` - Dropdown to choose/create sessions
    - `MessageList` - Scrollable message thread
    - `MessageInput` - Text input with send button

Features:

- Chat bubble UI (user messages right-aligned, assistant left-aligned)
- Real-time message streaming via WebSocket
- Tool call indicators (colored badges: "Calling NarratorAgent...", "Searching web...")
- Loading spinner during agent thinking
- Session management (new/resume/delete)
- Export conversation button

State Management:

```
const [sessionId, setSessionId] = useState(null);
const [messages, setMessages] = useState([]);
const [isLoading, setIsLoading] = useState(false);
const [wsConnection, setWsConnection] = useState(null);
```

Story Creator Interface

Components:

- `StoryCreator.jsx` - Main container
    - `JobForm` - Subject and player name inputs
    - `ProgressMonitor` - Real-time progress display
    - `StoryViewer` - Formatted story display

Features:

- Form validation (check research file exists)
- Real-time progress bar (5/12 TODOs complete)
- Live log stream with color coding (info/success/error)
- Current task indicator
- Review score display
- Story preview with collapsible sections
- Download JSON button
- Cancel job button

State Management:

```
const [jobId, setJobId] = useState(null);
const [jobStatus, setJobStatus] = useState('idle');
const [progress, setProgress] = useState({ current: 0, total: 0 });
const [logs, setLogs] = useState([]);
const [story, setStory] = useState(null);
```

File Explorer Interface

Components:

- `FileExplorer.jsx` - File browser

Features:

- List research and story files
- Filter by type (research/stories)
- Download button per file
- View file content in modal
- Delete file button (with confirmation)
- File metadata (size, date)

Navigation

Routes:

- `/` - Home/Landing page
- `/research` - Research Chat interface
- `/create-story` - Story Creator interface
- `/files` - File Explorer

Backend Implementation Details

Research Service (`research_service.py`)

Responsibilities:

- Manage StoryResearcher agent instances (in-memory cache per session)
- Load/save context memory using existing functions
- Stream agent responses token-by-token
- Handle session lifecycle (create/delete)

Key Functions:

```
class ResearchService:
    def __init__(self):
        self.agents = {}  # session_id -> StoryResearcher instance

    def get_or_create_agent(self, session_id: str) -> StoryResearcher:
        # Return cached agent or create new one

    async def send_message(self, session_id: str, message: str) -> AsyncGenerator:
        # Run agent and yield tokens

    def list_sessions(self) -> list[dict]:
        # Scan .mythline/story_researcher/context_memory/

    def get_session_messages(self, session_id: str) -> list[dict]:
```

```
        # Load context memory file

    def delete_session(self, session_id: str):
        # Delete context memory file
```

Story Service (`story_service.py`)

Responsibilities:

- Manage StoryCreatorAgent job execution
- Track job status in `.mythline/jobs/{subject}.json`
- Capture and broadcast progress updates
- Handle job cancellation

Key Functions:

```
class StoryService:
    def __init__(self):
        self.running_jobs = {}  # subject -> asyncio.Task

    async def submit_job(self, subject: str, player: str) -> str:
        # Validate research file exists
        # Create job tracking file
        # Start background task
        # Return job_id

    async def run_story_creation(self, subject: str, player: str,
progress_callback):
        # Instantiate StoryCreatorAgent
        # Run with progress interception
        # Broadcast updates via callback

    def get_job_status(self, subject: str) -> dict:
        # Read job tracking file

    def cancel_job(self, subject: str):
        # Cancel asyncio task
```

Job Tracking File Format (`.mythline/jobs/{subject}.json`):

```
{
  "job_id": "shadowglen",
  "subject": "shadowglen",
  "player": "Sarephine",
  "status": "running",
  "progress": {
    "current": 5,
    "total": 12,
    "current_task": "Creating dialogue for quest acceptance"
```

```
  },
  "started_at": "2025-01-10T14:40:00Z",
  "completed_at": null,
  "logs": [
    {"level": "info", "message": "Research loaded", "timestamp": "..."}
  ]
}
```

WebSocket Handlers

Research WebSocket (`research_ws.py`):

```python
@router.websocket("/ws/research/{session_id}")
async def research_websocket(websocket: WebSocket, session_id: str):
    await websocket.accept()
    try:
        while True:
            data = await websocket.receive_json()
            message = data["content"]

            async for token in research_service.send_message(session_id, message):
                await websocket.send_json({"type": "token", "content": token})

            await websocket.send_json({"type": "done"})
    except WebSocketDisconnect:
        pass
```

Story WebSocket (`story_ws.py`):

```python
@router.websocket("/ws/story/{subject}")
async def story_websocket(websocket: WebSocket, subject: str):
    await websocket.accept()

    async def progress_callback(update: dict):
        await websocket.send_json(update)

    try:
        await story_service.run_story_creation(subject, player, progress_callback)
    except Exception as e:
        await websocket.send_json({"type": "error", "message": str(e)})
    finally:
        await websocket.close()
```

Frontend Implementation Details

WebSocket Client Hook (`useResearchChat.js`)

```
export const useResearchChat = (sessionId) => {
  const [messages, setMessages] = useState([]);
  const [isConnected, setIsConnected] = useState(false);
  const [isLoading, setIsLoading] = useState(false);
  const wsRef = useRef(null);

  useEffect(() => {
    if (!sessionId) return;

    const ws = new WebSocket(`ws://localhost:8080/ws/research/${sessionId}`);

    ws.onopen = () => setIsConnected(true);
    ws.onclose = () => setIsConnected(false);

    ws.onmessage = (event) => {
      const data = JSON.parse(event.data);

      if (data.type === 'token') {
        // Append token to last message
        setMessages(prev => {
          const last = prev[prev.length - 1];
          if (last && last.role === 'assistant') {
            return [...prev.slice(0, -1), { ...last, content: last.content +
data.content }];
          }
          return [...prev, { role: 'assistant', content: data.content }];
        });
      } else if (data.type === 'done') {
        setIsLoading(false);
      }
    };

    wsRef.current = ws;
    return () => ws.close();
  }, [sessionId]);

  const sendMessage = (content) => {
    setMessages(prev => [...prev, { role: 'user', content }]);
    setIsLoading(true);
    wsRef.current.send(JSON.stringify({ type: 'message', content }));
  };

  return { messages, sendMessage, isConnected, isLoading };
};
```

Story Job Hook (`useStoryJob.js`)

```
export const useStoryJob = () => {
  const [jobId, setJobId] = useState(null);
  const [status, setStatus] = useState('idle');
```

```
  const [progress, setProgress] = useState({ current: 0, total: 0 });
  const [logs, setLogs] = useState([]);
  const wsRef = useRef(null);

  const submitJob = async (subject, player) => {
    const response = await fetch('/api/story/jobs', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ subject, player })
    });

    const data = await response.json();
    setJobId(data.job_id);
    setStatus('running');

    // Connect WebSocket
    const ws = new WebSocket(`ws://localhost:8080/ws/story/${subject}`);

    ws.onmessage = (event) => {
      const data = JSON.parse(event.data);

      if (data.type === 'progress') {
        setProgress({ current: data.current, total: data.total, task: data.task
});
      } else if (data.type === 'log') {
        setLogs(prev => [...prev, data]);
      } else if (data.type === 'complete') {
        setStatus('completed');
      } else if (data.type === 'error') {
        setStatus('error');
      }
    };

    wsRef.current = ws;
  };

  const cancelJob = async () => {
    await fetch(`/api/story/jobs/${jobId}`, { method: 'DELETE' });
    wsRef.current?.close();
    setStatus('cancelled');
  };

  return { jobId, status, progress, logs, submitJob, cancelJob };
};
```

Development Setup

Dependencies

Backend (`requirements.txt`):

```
```

```
fastapi
uvicorn[standard]
websockets
python-multipart
```

Frontend (`package.json`):

```
{
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.2.0",
    "vite": "^5.0.0"
  }
}
```

Startup Scripts

`start_web_ui.bat`:

```
@echo off
echo Starting Mythline Web UI...

echo Starting MCP servers...
start "MCP Web Search" cmd /k "cd src\mcp_servers\mcp_web_search && python
server.py"
start "MCP Web Crawler" cmd /k "cd src\mcp_servers\mcp_web_crawler && python
server.py"
start "MCP Filesystem" cmd /k "cd src\mcp_servers\mcp_filesystem && python
server.py"

timeout /t 5

echo Starting backend API...
start "Backend API" cmd /k "uvicorn src.ui.web.backend.main:app --reload --port
8080"

timeout /t 3

echo Starting frontend dev server...
start "Frontend Dev" cmd /k "cd src\ui.web\frontend && npm run dev"

echo.
echo Mythline Web UI started!
```

```
echo Backend: http://localhost:8080
echo Frontend: http://localhost:5173
echo API Docs: http://localhost:8080/docs
pause
```

CORS Configuration

In `main.py`:

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],  # Frontend dev server
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Production Deployment

Build Process

1. **Frontend build:**

```
cd src/ui/web/frontend
npm run build
```

2. **Serve static files from FastAPI:**

```
from fastapi.staticfiles import StaticFiles

app.mount("/", StaticFiles(directory="src/ui/web/frontend/dist", html=True),
name="frontend")
```

3. **Run backend:**

```
uvicorn src.ui.web.backend.main:app --host 0.0.0.0 --port 8080
```

Deployment Considerations

- MCP servers must be running on configured ports
- `.env` file must be present with API keys

- `output/` directory must be writable
- Consider using `gunicorn` with uvicorn workers for production
- Set appropriate CORS origins for production domain
- Use reverse proxy (nginx) for SSL/TLS termination

Testing Strategy

Backend Tests

1. **Research API:**
    - Create session and verify context memory file created
    - Send message and verify response streaming
    - List sessions and verify all sessions returned
    - Delete session and verify file removed
2. **Story API:**
    - Submit job and verify job tracking file created
    - Monitor progress via WebSocket and verify updates
    - Cancel job and verify task stopped
    - Verify story.json created on completion
3. **File API:**
    - List files and verify correct paths returned
    - Download file and verify content matches filesystem

Frontend Tests

1. **Research Chat:**
    - Create new session and send message
    - Resume existing session and verify messages loaded
    - Delete session and verify removed from list
    - Test WebSocket reconnection on disconnect
2. **Story Creator:**
    - Submit job with valid subject/player
    - Monitor progress and verify real-time updates
    - Cancel running job
    - View completed story
3. **File Explorer:**
    - List files and verify display
    - Download file and verify content
    - Delete file and verify removed

Integration Tests

- Start all MCP servers
- Start backend API
- Run complete Research Story workflow
- Run complete Create Story workflow
- Test concurrent story creation jobs
- Test error handling (missing research file, API errors)

Implementation Phases

Phase 1: Backend Foundation (2-3 hours)

- Set up FastAPI application structure
- Implement Research API endpoints
- Implement Story API endpoints
- Implement File API endpoints
- Create Pydantic models for requests/responses

Phase 2: WebSocket Implementation (1-2 hours)

- Implement Research WebSocket handler
- Implement Story WebSocket handler
- Add progress tracking for story jobs
- Test WebSocket connections

Phase 3: Frontend Setup (1 hour)

- Initialize React + Vite project
- Set up routing
- Create basic layout and navigation
- Configure API client

Phase 4: Research Chat UI (2 hours)

- Build ResearchChat components
- Implement WebSocket client hook
- Add session management UI
- Style chat interface

Phase 5: Story Creator UI (2 hours)

- Build StoryCreator components
- Implement job submission form
- Add progress monitor
- Add story viewer

Phase 6: File Explorer UI (1 hour)

- Build FileExplorer component
- Add download/delete functionality
- Display file metadata

Phase 7: Integration & Polish (1-2 hours)

- Connect all components
- Add error handling
- Improve styling
- Test complete workflows
- Create startup scripts

Phase 8: Documentation & Testing (1 hour)

- Write deployment instructions
- Create user guide
- Test in clean environment

Total Estimated Time: 11-13 hours

Success Criteria

1. Research Chat interface successfully replicates CLI functionality
2. Story Creator interface successfully executes story creation workflow
3. Real-time updates work smoothly via WebSocket
4. Session management works correctly (create/resume/delete)
5. File operations work (list/download/delete)
6. Concurrent story creation jobs execute without interference
7. Error handling provides clear feedback to users
8. UI is responsive and intuitive
9. All existing CLI features are accessible via web UI
10. System is deployable with simple startup script

Future Enhancements (Out of Scope)

- Multi-user support with authentication
- Database for session/job metadata
- Job queue system for resource management
- Story editing interface
- Research file upload
- Export story to multiple formats (PDF, HTML, EPUB)
- Preference management UI
- MCP server health monitoring dashboard
- Job scheduling (run story creation at specific time)
- Email notifications on job completion
- Story comparison/versioning
- Advanced search and filtering
- Analytics dashboard (usage stats, agent performance)