Webserv Development Stages

Non-Parsing Components Implementation Guide

Your Scope

Since parsing (HTTP Request + Config File) is already handled, you'll implement:

- **Server Core Module** - Network layer and event loop
- **Application Module** - Request handling and business logic
- **Utility Module** - Helper functions
- **Integration** - Connect all modules together

Stage 1: Foundation Setup

Goal: Establish project structure and basic utilities

Tasks:

- Create directory structure (`src/`, `include/`, `tests/`)
- Implement Utility Module functions:
  - String helpers (trim, split, case conversion)
  - File operations (exists, readable, size, read)
  - MIME type detection
  - Basic logging system
- Write unit tests for utilities
- Set up Makefile with proper dependencies

Completion Criteria: Utilities compile, tests pass, team can use helper functions

---

Stage 2: Server Core - Socket Management

Goal: Create and manage listening sockets

Tasks:

- Implement `SocketManager` class:
  - `socket()` + `bind()` + `listen()` wrapper
  - Create sockets for all configured ports
  - Set sockets to non-blocking (fcntl)
  - Handle socket errors gracefully
- Store listening socket FDs
- Add basic error handling and logging

Completion Criteria: Server binds to configured ports, doesn't block, handles multiple ports

---

Stage 3: Server Core - Event Loop

Goal: Build the main event loop with poll/select/epoll/kqueue

Tasks:

- Implement `EventLoop` class:
  - Choose multiplexing method (poll recommended for portability)
  - Monitor listening sockets for incoming connections
  - Monitor client sockets for read/write readiness
  - Handle POLLIN, POLLOUT, POLLERR, POLLHUP events
  - Implement timeout mechanism (inactive connections)
- Create main server loop that never blocks

Completion Criteria: Event loop detects new connections and I/O readiness, handles multiple clients simultaneously

---

Stage 4: Server Core - Connection Management

Goal: Track client connections and their state

Tasks:

- Implement `Connection` class:
  - Store FD, address, timestamps
  - Read buffer (for incoming data)
  - Write buffer (for outgoing data)
  - State tracking (READING, PROCESSING, WRITING)
  - Keep-alive flag
- Implement `ConnectionManager`:
  - Accept new connections from listening sockets
  - Store connections in map/vector (FD → Connection)
  - Handle connection closure (timeout or client disconnect)
  - Manage buffers for partial reads/writes

Completion Criteria: Server accepts connections, tracks them, closes them properly, no FD leaks

---

Stage 5: Integration - Request Pipeline

Goal: Connect parsing to your server core

Tasks:

- In event loop, when socket is readable:
  - `recv()` data into Connection's read buffer
  - Call `Request::parse()` (from your teammate)
  - Check parsing state (COMPLETE/INCOMPLETE/ERROR)
- If INCOMPLETE: continue reading
- If ERROR: generate error response
- If COMPLETE: pass Request to Application Module

Completion Criteria: Server receives HTTP requests, parsing works, incomplete requests handled correctly

---

Stage 6: Application - Static File Handler

Goal: Serve files from filesystem

Tasks:

- Implement `StaticFileHandler`:
  - Match request path to filesystem path (using root from config)
  - Check file exists and is readable
  - Read file contents
  - Detect MIME type
  - Generate Response object (200 or 404)
- Handle index files (index.html)
- Handle directory requests

Completion Criteria: Server serves HTML, CSS, JS, images. Returns 404 for missing files

---

Stage 7: Application - Error Handler

Goal: Generate error responses

Tasks:

- Implement `ErrorHandler`:
  - Take error code (400, 404, 500, etc.)
  - Check config for custom error page
  - If custom page exists: serve it
  - If not: generate default HTML error page
  - Set proper status code and headers

Completion Criteria: All error codes return proper responses, custom error pages work

---

Stage 8: Application - Request Router

Goal: Match requests to handlers

Tasks:

- Implement `RequestRouter`:
  - Match request to server block (by Host header + port)
  - Match request to location block (longest prefix match)
  - Check if method is allowed in location
  - Determine handler type (static, CGI, upload, redirect)
  - Call appropriate handler
- Apply location-specific settings (root, autoindex, etc.)

Completion Criteria: Requests routed to correct server/location, method validation works

---

Stage 9: Response Generation

Goal: Convert Response objects to HTTP bytes

Tasks:

- Work with Protocol Module teammate on `Response` struct format
- Implement response formatting:
  - Status line (HTTP/1.1 200 OK)
  - Headers (Content-Type, Content-Length, Connection)
  - Body
- Handle keep-alive vs close
- Buffer response in Connection's write buffer

Completion Criteria: Responses are properly formatted, browsers can parse them

---

Stage 10: Response Transmission

Goal: Send responses back to clients

Tasks:

- In event loop, when socket is writable:
  - `send()` data from Connection's write buffer
  - Handle partial writes (EAGAIN/EWOULDBLOCK)
  - Track bytes sent
  - When complete: reset buffers, handle keep-alive or close
- Implement proper cleanup on connection close

Completion Criteria: Full responses sent, partial writes handled, connections close properly

---

Stage 11: Application - Directory Listing

Goal: Generate HTML directory listings

Tasks:

- Implement autoindex functionality:
  - Check if location has autoindex enabled
  - If directory requested without index file: list contents
  - Generate HTML with file/directory links
  - Include file sizes and modified dates
  - Handle empty directories

Completion Criteria: Directory listings work when enabled, match NGINX format

---

Stage 12: Application - Redirect Handler

Goal: Handle HTTP redirects

Tasks:

- Implement `RedirectHandler`:
  - Check if location has redirect configured
  - Generate 301/302 response
  - Set Location header
  - Return minimal HTML body

Completion Criteria: Redirects work, browsers follow them

Stage 13: Application - Upload Handler

Goal: Handle file uploads

Tasks:

- Implement `UploadHandler`:
  - Check if uploads enabled in location
  - Parse multipart/form-data (if POST with file)
  - Validate upload directory exists and is writable
  - Write uploaded file to disk
  - Respect max body size limits
  - Return 201 Created or error

Completion Criteria: Files can be uploaded via POST, saved to configured directory

---

Stage 14: Application - CGI Handler

Goal: Execute CGI scripts

Tasks:

- Implement `CGIHandler`:
  - Detect CGI by file extension
  - Set up environment variables (REQUEST_METHOD, QUERY_STRING, etc.)
  - Create pipes for stdin/stdout

- `fork()` process
- Child: `execve()` CGI script
- Parent: write request body to stdin, read output from stdout
- Parse CGI output (headers + body)
- Handle CGI errors and timeouts
- Reap child process (`waitpid`)

Completion Criteria: CGI scripts execute, output returned to client, no zombie processes

---

Stage 15: Stress Testing & Refinement

Goal: Ensure stability under load

Tasks:

- Test with multiple simultaneous clients
- Test with slow clients (partial reads/writes)
- Test with malformed requests
- Test with large files and uploads
- Check for memory leaks (valgrind)
- Check for FD leaks (lsof)
- Profile performance bottlenecks
- Fix bugs and edge cases

Completion Criteria: Server handles 100+ simultaneous connections, no leaks, stable under stress

---

Stage 16: NGINX Comparison

Goal: Match NGINX behavior

Tasks:

- Set up NGINX with identical config
- Compare outputs for:

    - Static file serving
    - Error responses
    - Directory listings
    - Redirects
    - CGI execution
- Fix discrepancies
- Document differences (if any)

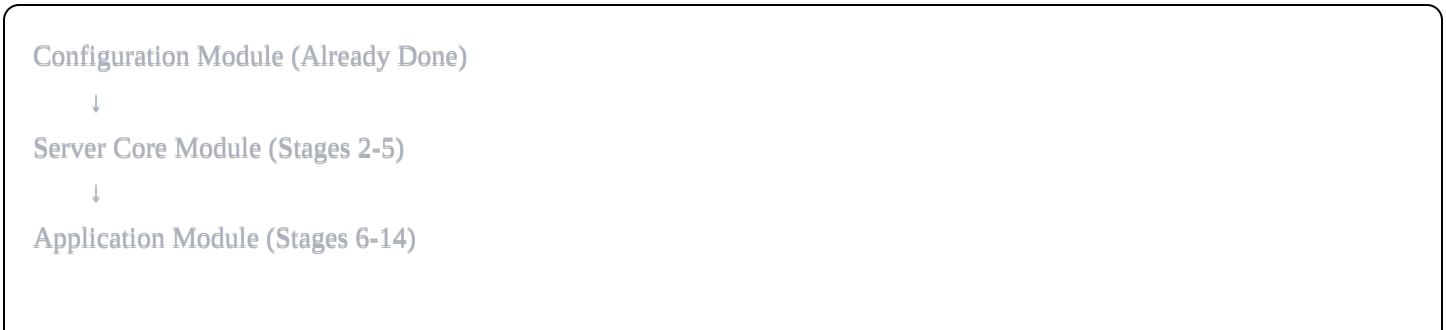
Completion Criteria: Your server behaves like NGINX for all test cases

---

Critical Learning Points Per Stage

Stages	Focus Area
2-4	Understanding non-blocking I/O, poll(), socket lifecycle
5	Integrating separate modules, handling incomplete data
6-8	HTTP semantics, routing logic, filesystem operations
9-10	Response formatting, partial writes, connection state
11-13	HTTP features (directory listing, redirects, uploads)
14	Process management, pipes, environment variables, CGI protocol
15-16	Debugging, testing, production readiness

Module Dependencies



↓

Testing & Refinement (Stages 15-16)

Utility Module supports all stages throughout development.

Recommended Development Order

1. **Foundation First** (Stage 1) - Everyone needs utilities
2. **Core Infrastructure** (Stages 2-4) - Build the engine
3. **Basic Integration** (Stage 5) - Connect parsing to core
4. **Simple Handler** (Stage 6) - Prove the pipeline works
5. **Error Handling** (Stage 7) - Handle failures gracefully
6. **Routing Logic** (Stage 8) - Enable configuration flexibility
7. **Response Pipeline** (Stages 9-10) - Complete the request cycle
8. **Extended Features** (Stages 11-14) - Add remaining functionality
9. **Production Ready** (Stages 15-16) - Polish and validate

Team Collaboration Notes

- **Clear Interfaces:** Define Request/Response structs with parsing teammate
- **Incremental Testing:** Test each stage before moving forward
- **Code Reviews:** Review each other's work at stage boundaries
- **Daily Syncs:** Share progress, blockers, and integration points
- **Git Workflow:** Feature branches → PR → Review → Merge to dev → Weekly integration to main

---

Document Version: 1.0
Last Updated: December 2025