🔄 Request Parsing Flow: From Raw Bytes to Validated Request

🟢 Step 1: Raw Data Accumulated in ServerManager

- In ServerManager::handleClientRead():
  - Incoming data is read into _clientBuffers[clientFD].
  - If \r\n\r\n is found, headers are considered complete.
  - A temporary Request object is created to extract headers and determine if the body is complete.
  - Once the full request (headers + body) is received → the full string is passed to Request::parse().

🧠 Step 2: Request::parse(const std::string& raw)

- Splits the raw string at \r\n\r\n:
  - Header portion → parsed using:
    - parseRequestLine(): extracts method, target, version.
    - parseHeaders(): stores headers into _headers map.
  - Body portion → saved directly in _body.
- **Note**: This step does *not* validate body size or format yet — it just stores and organizes.

🔍 Step 3: ServerManager::processClientRequest()

- Once the full request is parsed and stored in _clientRequests[clientFD]:
  - The method validateBody() is called before further processing.

🧪 Step 4: Request::validateBody(std::size_t maxBodySize)

This is where content validation happens based on headers:

🔸 Case 1: Transfer-Encoding: chunked

- decodeChunkedBody() is called.
  - Parses hex chunk sizes.

  - Reads and joins each chunk.
  - Ensures the structure conforms to the chunked transfer spec.
  - If decoding fails or chunks are malformed → validation fails.

🔸 Case 2: Content-Length + Content-Type

- If Content-Length is present:
  - Ensures the actual body size matches the declared length.
  - Truncates excess if any.
- If Content-Type is multipart/form-data:
  - Iterates through _uploadedFiles and checks if any file exceeds maxBodySize.

If everything is valid → continue to dispatch. Otherwise → respond with 413 Payload Too Large or 400 Bad Request.

🧩 Step 5: parseBody() is called *inside* parse() but interprets only after headers are available

- **Content-Type Driven Logic**:
  - application/x-www-form-urlencoded: key-value pairs are extracted into _formData.
  - multipart/form-data: boundary is extracted and parseMultipartFormData() is called to split form fields vs file uploads.

💡 Summary of Flow Dependencies

text

CopyEdit

```
handleClientRead()
 └── Request::parse(raw)
     ├── parseRequestLine()
     ├── parseHeaders()
     └── parseBody()          ← interprets based on Content-Type
```

```
        ├── form-urlencoded      → _formData
        └── multipart/form-data  → _formData + _uploadedFiles
```

```
processClientRequest()
 └── Request::validateBody(maxSize)
     ├── Transfer-Encoding: chunked → decodeChunkedBody()
     └── Content-Length          → match body size or truncate
         └── multipart/form-data → check uploaded file sizes
```

Let me know if you'd like a diagram or if we should move on to documenting Response next.

Ask ChatGPT

You said:

so how does parsebody work with validatebody? e.g., can there be chunked bodies with requests that have uploaded files and form data

ChatGPT said:

Great question — the relationship between parseBody() and validateBody() is important to understand, especially in edge cases like Transfer-Encoding: chunked with multipart/form-data. Let's walk through how they interact and what cases are allowed in HTTP:

🔁 How parseBody() and validateBody() Work Together

✔ Step 1: Request::parse(raw)

- **Stores the body** in _body.
- **Immediately calls** parseBody() to populate:
  - _formData (from application/x-www-form-urlencoded)
  - _uploadedFiles (from multipart/form-data)

✔ Step 2: Request::validateBody(maxBodySize)

- Called later by ServerManager::processClientRequest().
- Validates that the content matches what's declared in headers:

    - If **chunked**, it tries to **decode** the body with decodeChunkedBody() — replacing _body with the decoded result.
    - If **multipart**, it checks that each uploaded file in _uploadedFiles does not exceed the max size.

❗ Issue: Ordering

- If the body is chunked, then the original call to parseBody() (before decoding) might operate on undecoded/incomplete content.
- Only *after* chunked decoding is done does the body become usable.
- **However**, parseBody() is not called again after decodeChunkedBody() — this is a limitation in your current design.

📦 Can Chunked Encoding Be Used With multipart/form-data?

✅ Yes — it's allowed by the HTTP spec.

According to RFC 7230 §3.3.1, Transfer-Encoding: chunked can be used for **any** content type, including multipart/form-data.

But... your current implementation assumes:

- parseBody() is called **before** the body is decoded.
- Therefore, when the request uses both Transfer-Encoding: chunked **and** multipart/form-data, parsing of the form and uploaded files will fail (or produce incorrect results).

✅ Recommended Fix

To support the full HTTP spec (chunked + form data):

Option A: Call parseBody() after decoding

- Inside validateBody(), after decodeChunkedBody() succeeds:

cpp

CopyEdit

```
if (transferEncoding == "chunked") {
  if (!decodeChunkedBody())
    return false;
    parseBody(); // <-- call again here
}
```

Option B: Delay parseBody() entirely

- Remove the call from parse() and let it be called explicitly only after body validation is done.
    - More correct, but requires more coordination in code.

💡 Summary

Behavior	**Current Implementation**	**Spec-Compliant?**	**Suggested Fix**
Content-Length + multipart	✅ Works	✅ Yes	No change needed
Chunked + plain body	✅ Works	✅ Yes	No change needed
Chunked + multipart/form-data	❌ Broken	✅ Yes	✅ Call parseBody() after decoding

Let me know if you'd like help implementing this patch or want to proceed to documenting the response handling phase.

You said:

what else is missing in my current implementation

ChatGPT said:

Based on what you've shared so far, your Request parsing implementation is **quite thorough** — it already covers:

- Request line and headers
- Basic validation for methods, targets, versions
- Chunked transfer decoding
- Content-Length body validation
- Handling of both application/x-www-form-urlencoded and multipart/form-data
- File upload and form data extraction
- Error handling (e.g., malformed headers, oversized body)

However, here are some **missing or incomplete features** (some are noted in your TODOs) that might matter depending on your project's requirements or RFC compliance goals:

✅ High-Impact / Should Fix

1. 🔁 parseBody() called too early for chunked bodies

- As discussed: it's run before decodeChunkedBody(), so it parses incorrect or incomplete content in chunked + multipart cases.
- **Fix**: Call parseBody() after successful decodeChunkedBody() in validateBody().

---

2. 🧪 Multipart Parsing: Headers in Each Part

- Your parseMultipartFormData() only looks for Content-Disposition.
- But each part may also include headers like Content-Type (e.g., for uploaded images).
- **Improvement**: Parse and store per-part Content-Type to support richer file handling.

---

3. 🔒 Path Traversal Protection in filename

- Uploaded filenames should be sanitized.
- E.g., requests might include filename="../../etc/passwd".
- **Fix**: Strip directory components (basename(filename) style).

---

4. 🌐 Expect Header (HTTP/1.1 feature)

- Clients may send:
  Expect: 100-continue
  Your server should respond with HTTP/1.1 100 Continue before reading the body.
- **Current behavior**: The server likely just reads immediately — could cause errors with compliant clients.
- **Fix**: Detect Expect header and respond accordingly.

🧠 Medium-Impact / RFC Compliance Edge Cases

5. 📦 Support for Transfer-Encoding: chunked + trailers

- After the last chunk (0\r\n\r\n), there may be **trailing headers** before the end of the message.
- **Current behavior**: Ignores them or fails on unexpected data.
- **Fix (optional)**: Parse and merge trailing headers.

---

6. 🧵 Support for Connection: keep-alive

- You're not yet tracking persistent connections — every request seems to be followed by cleanupClient().
- **Fix (future)**: Honor Connection: keep-alive header and allow multiple requests on a single connection.

---

7. 🔄 Handling of CRLF vs LF

- Strict HTTP parsing expects \r\n, but some clients (esp. curl in loose mode) may send just \n.
- **Fix (optional)**: Tolerate \n only if you want robustness (but less strict parsing).

---

8. ❓ Unknown/Unsupported Headers

- Your parser accepts all headers without validation (not wrong, just liberal).
- Some servers normalize or reject malformed/unknown headers.

---

9. ⚠️ Header Size Limits

- RFC recommends imposing limits on:
  - Total header size (e.g., 8KB)
  - Number of headers
- **Fix (optional)**: Add header parsing limits to prevent abuse.

✅ Overall: You're Doing Great

Your parser already handles far more than basic HTTP: chunked transfer, form parsing, and multipart uploads. With the fixes below, it would be production-grade in behavior.

🎯 Recommended Next Steps

Fix	Why It Matters
Call parseBody() after decoding chunked body	Fix bug for chunked + multipart
Add path sanitization for uploaded filenames	Prevent directory traversal
Handle Expect: 100-continue	Improve standards compliance
(Later) Keep-alive connection handling	Required for full HTTP/1.1 support

Let me know if you'd like code suggestions or want to proceed with the next documentation stage.

You said:

are all these steps required for mandatory?

ChatGPT said:

Good question. Here's a breakdown of which improvements are **mandatory**, **recommended**, or **optional**, depending on your goals — especially if you're building this for the **Webserv project at 42**, which often balances correctness with scope control.

✅ Must-Have (Mandatory for Webserv subject)

These are required to meet the **core HTTP spec requirements** and to handle the most common test cases:

Feature	Why it's Required
✅ parseBody() called after decodeChunkedBody()	Otherwise, chunked + multipart/form-data will fail — a correctness bug.
✅ Content-Length body validation	Required for POST uploads and to prevent abuse.
✅ Chunked transfer decoding	Required by subject (must support Transfer-Encoding: chunked).

Feature	Why it's Required
✅ Proper parsing of multipart/form-data	Needed to support file uploads.
✅ Correct request line and header parsing	Required to parse and route HTTP requests at all.

🟡 Should-Have (Strongly Recommended)

These aren't explicitly required by the subject but **will be tested** if you use tools like curl, Postman, or browsers — and might be part of **bonus tests or peer feedback**.

Feature	Why it's Important
🔒 Sanitizing filename in uploads	Prevents path traversal vulnerabilities.
🧪 Better multipart parsing (support Content-Type per part)	Improves correctness for real-world clients (e.g., image uploads).
🔄 Connection: keep-alive support	Required for HTTP/1.1 compliance, often tested implicitly.
🌐 Handling Expect: 100-continue	curl --expect100 may cause hangs without this.

If you're going for **bonus** or want your server to be robust across real-world use, these matter.

⚪ Nice-to-Have (Optional/Advanced)

These improve your server's robustness or RFC compliance but **are not required** for a passing Webserv project:

Feature	Why it's Optional
🧵 Parsing trailing headers after chunked body	Rarely used in real-world traffic.
🔄 LF-only line ending tolerance	Not strictly compliant with the RFC.

Feature	Why it's Optional
📦 Header size limits	Useful for hardening but not part of Webserv scope.
❓ Unknown headers validation	Liberal handling is fine for now.

🔚 TL;DR Summary

Priority	Feature
✅ Required	Correct request parsing, chunked decoding, form-data parsing, validation logic
🟡 Recommended	Sanitizing filenames, calling parseBody after decoding, keep-alive, Expect header
○ Optional	Trailer headers, header limits, LF-only, unknown header filtering

If you're working on **finishing mandatory features first**, focus on:

- Fixing the parseBody() timing bug
- Validating and parsing Content-Length and chunked bodies correctly
- Correctly parsing multipart form data

Let me know if you want a checklist or help implementing one of the fixes!