Barebones Web Framework: Design Document

Overview

This project designs and builds a minimal, educational web framework from scratch. The core architectural challenge is creating a flexible, extensible pipeline that processes HTTP requests through layers of middleware, matches routes with parameters, and renders dynamic content, all while providing a clean developer API.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** All (Foundational)

A web framework is the foundational layer upon which modern web applications are built. It provides the structural patterns, abstractions, and utilities that transform the raw, low-level HTTP protocol into a productive development environment. Building such a framework from scratch reveals the intricate dance between HTTP's stateless request-response model and the developer's need for stateful, structured, and reusable code.

At its core, every web framework must solve the same fundamental problem: how to efficiently and flexibly connect incoming HTTP requests to the appropriate business logic, transform data along the way, and produce meaningful HTTP responses. The complexity arises from the myriad of concerns—routing, parameter parsing, security, templating, error handling—that must be orchestrated in a cohesive, predictable manner.

This document explores the architecture of **Barebones**, a minimal yet complete web framework designed for educational purposes. By understanding the foundational patterns—the **middleware pipeline**, **router trie**, and **template compilation**—developers gain the insight needed to work effectively with any web framework and appreciate the trade-offs between simplicity and feature richness.

The Factory Pipeline Analogy

Imagine an **automated factory assembly line** where raw materials (HTTP requests) enter at one end and finished products (HTTP responses) exit at the other. Each request is a "workpiece" that moves along a conveyor belt, stopping at various **processing stations** (middleware) where it is inspected, modified, or enhanced before reaching its final assembly station (the route handler).

Station	Purpose	Real-World Equivalent
Quality Control	Validates incoming materials	Request validation middleware
Stamp Machine	Marks the workpiece with metadata	Logging middleware
Painting Booth	Applies a uniform coating	Authentication middleware
Assembly Robot	Attaches specific components based on blueprint	Route handler
Packaging	Wraps the final product for shipment	Response formatting middleware

This mental model captures several critical framework concepts:

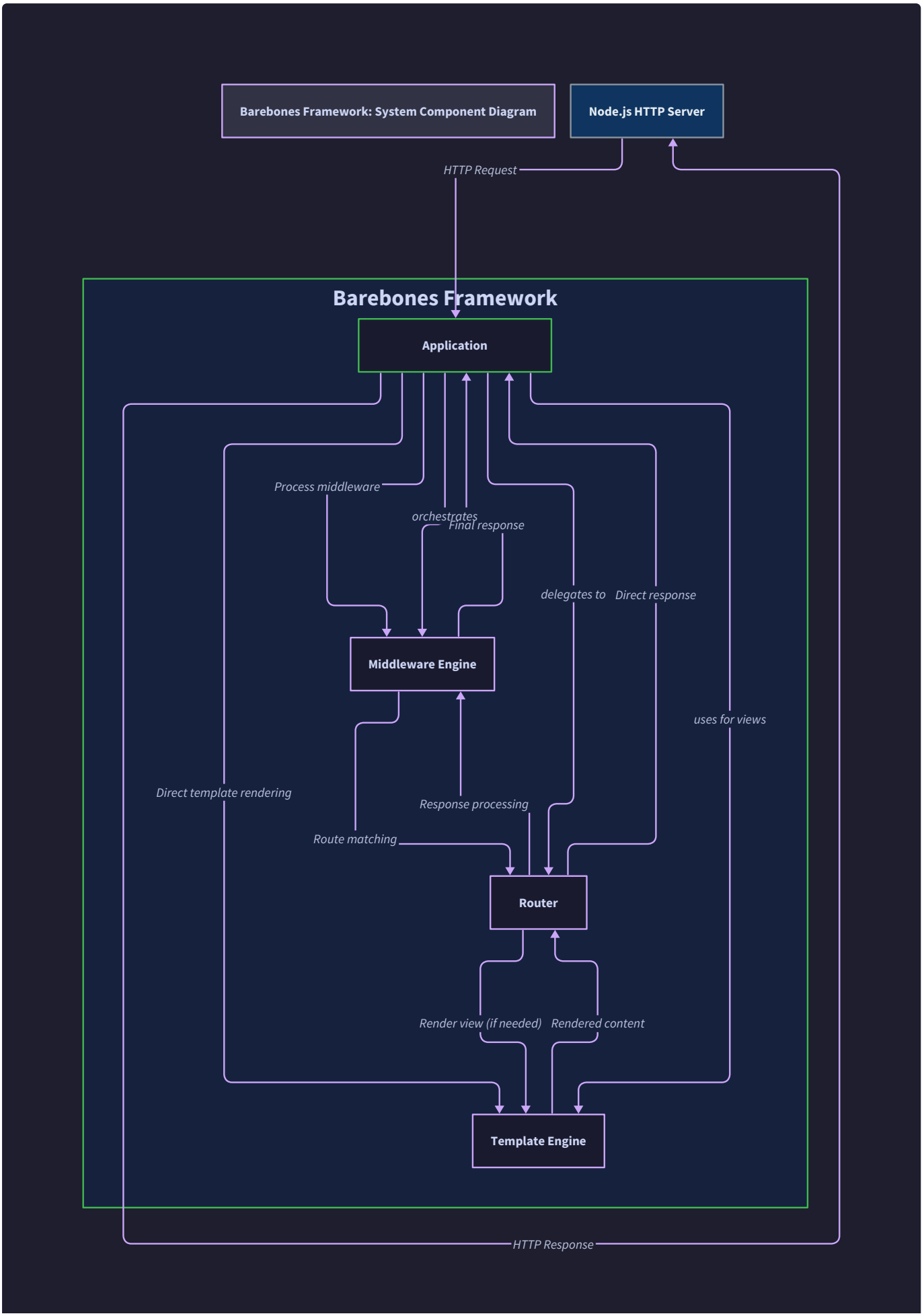
1. **Sequential Processing**: The workpiece must pass through stations in a defined order. Similarly, middleware executes in registration order.
2. **Standardized Interfaces**: Each station knows how to accept and pass along the workpiece via the conveyor belt. In middleware, this is the `next()` function.
3. **Specialization**: Stations perform one specific task well. Middleware should be focused and single-purpose.
4. **Early Termination**: If a workpiece is defective at quality control, it's rejected immediately without visiting later stations. Middleware can end the request early by not calling `next()`.

> The **factory pipeline** analogy reveals why middleware composition is so powerful: it allows developers to **plug in** processing units without modifying the core assembly line. This is the essence of the **chain of responsibility** pattern.

Consider a concrete request journey: `GET /api/users/123`

1. **Raw Materials Arrive**: Node.js HTTP server creates raw `IncomingMessage` and `ServerResponse` objects
2. **Conveyor Belt Starts**: Framework wraps these objects and places them on the processing pipeline
3. **Station 1 (Logging)**: Records arrival timestamp, method, and URL
4. **Station 2 (Authentication)**: Checks API key header; if invalid, rejects with 401 response
5. **Station 3 (Body Parsing)**: For POST requests, would parse JSON; for GET, does nothing
6. **Station 4 (Routing)**: Examines path `/api/users/123`, matches to `GET /api/users/:id` handler
7. **Final Assembly**: User lookup handler fetches user ID 123 from database
8. **Packaging**: Response formatted as JSON with proper headers
9. **Shipment**: Response sent back through Node.js to the client

This assembly line is depicted in the system component diagram:


Barebones Framework: System Component Diagram
Node.js HTTP Server
HTTP Request
Barebones Framework
Application
Process middleware
orchestrates
Final response
delegates to
Direct response
Middleware Engine
uses for views
Direct template rendering
Response processing
Route matching
Router
Render view (if needed)
Rendered content
Template Engine
HTTP Response

The Core Problem: Managing HTTP Complexity

The Node.js `http` module provides the bare essentials for handling HTTP traffic, but operating at this low level requires developers to manually manage overwhelming complexity. Consider what's needed to handle a simple user profile request:

```javascript
// WITHOUT FRAMEWORK (Node.js native)
const http = require('http');

const server = http.createServer((req, res) => {
  // Manual routing - must parse URL and method
  if (req.method === 'GET' && req.url.startsWith('/users/')) {
    const userId = req.url.split('/')[2]; // Fragile extraction


    // Manual content negotiation
    if (req.headers['accept'] !== 'application/json') {
      res.writeHead(406, { 'Content-Type': 'text/plain' });
      return res.end('Only JSON supported');
    }


    // Manual error handling
    try {
      const userData = getUserFromDatabase(userId); // Could throw
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify(userData));
    } catch (error) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end('Internal server error');
    }
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not found');
  }
});
```

This approach suffers from several critical issues:

Problem 1: Boilerplate Multiplication Every route handler must duplicate:

- URL parsing and method checking
- Header validation and content negotiation
- Error catching and status code setting
- Response header configuration
- Data serialization

Problem 2: Structural Chaos Without enforced patterns, each developer creates ad-hoc solutions for common tasks (parameter extraction, validation, authentication), leading to inconsistent, bug-prone codebases.

Problem 3: Poor Reusability Common logic like logging, compression, or CORS headers can't be abstracted into reusable units without significant refactoring.

Problem 4: Inconsistent Error Handling Each route implements error recovery differently, making systemic error reporting and debugging difficult.

Problem 5: Missing Abstractions Developers must work directly with streams for body parsing, manually handle character encodings, and manage low-level cookie parsing.

The Barebones framework addresses these problems through three core architectural abstractions:

Abstraction	Solves	Key Mechanism
Router	Manual URL/method checking	Pattern matching with parameter extraction
Middleware Pipeline	Boilerplate duplication	Chain of responsibility with `next()` control flow
Enhanced Request/Response	Low-level stream handling	Higher-level methods for JSON, cookies, templates

These abstractions transform the developer experience:

```javascript
// WITH FRAMEWORK
app.get('/users/:id', authenticate, (req, res) => {
  const user = getUser(req.params.id); // params extracted automatically
  res.json(user); // Automatic JSON serialization + headers
});

// Error handling centralized
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Something broke!');
});
```

> The fundamental value proposition of a web framework is **consistency through convention**. By providing standard ways to handle routing, middleware, and responses, it reduces cognitive load and enables developers to focus on business logic rather than HTTP mechanics.

Landscape of Existing Frameworks

The web framework ecosystem spans a spectrum from **minimal and unopinionated** to **batteries-included and convention-driven**. Understanding this landscape informs Barebones' design philosophy and helps clarify what trade-offs we're making.

Decision: Framework Philosophy and Scope

- **Context**: We must decide whether Barebones should be a minimal Express-like framework or a more comprehensive Django-like framework. This affects complexity, learning curve, and extensibility.
- **Options Considered**:
  1. **Minimalist Approach**: Provide only routing and middleware, require plugins for everything else
  2. **Integrated Approach**: Include templating, ORM, form validation, and admin panels by default
  3. **Layered Approach**: Core framework with optional, officially supported add-ons
- **Decision**: Adopt a minimalist approach for the core, with integrated templating as a learning exercise (Milestone 4)
- **Rationale**:
  - Educational focus: Understanding fundamentals is easier without ORM/forms complexity
  - JavaScript ecosystem favors composition over monolithic frameworks
  - Aligns with Node.js philosophy of small, focused modules
- **Consequences**:
  - Learners must integrate databases/auth themselves (educational benefit)
  - Framework stays lightweight and easier to understand end-to-end
  - Template engine implementation teaches compilation/escaping concepts

Framework Type	Representative	Philosophy	Routing	Templating	ORM	Key Insight
Minimal/Unopinionated	Express (Node.js), Sinatra (Ruby)	"Bring your own everything"	Manual route definitions, middleware-based	Optional view engines	None	**Middleware as composition**: Everything is middleware, even routing
Batteries-Included	Django (Python), Rails (Ruby)	"Convention over configuration"	Automatic URL routing, decorator-based	Built-in template engine	Integrated ORM with migrations	**Inversion of control**: Framework calls your code through well-defined interfaces
Full-Stack/Modern	Next.js (JS), Laravel (PHP)	"Developer experience first"	File-system based routing, API routes	Hybrid server/client rendering	Optional, with query builders	**Abstraction layers**: Hide complexity but allow escape hatches
Performance-Focused	Actix (Rust), Fiber (Go)	"Zero-cost abstractions"	Compiled route tables, static dispatch	Minimal compile-time templates	No ORM, SQL-focused	**Compile-time safety**: Routes validated at compile time, no runtime errors

The **middleware pattern** deserves special attention, as it's the most influential architectural pattern in modern web frameworks. Express popularized the concept, but it appears in various forms across ecosystems:

Framework	Middleware Implementation	Key Differentiator
Express	Functions with `(req, res, next)` signature	Error middleware with `(err, req, res, next)`
Koa	Async functions, `ctx` object instead of `(req, res)`	Uses async/await, no callbacks
Django	Class-based middleware with `process_request` / `process_response`	Request/response phases separated
Rails	Rack middleware stack (Ruby)	Standardized `env` hash interface
Actix	Traits with `Service` trait, futures-based	Type-safe, zero-cost abstractions

> **The Middleware Insight**: Middleware transforms the **framework itself** into a **protocol**. By standardizing the interface through which requests flow (`next()`), any compatible middleware can be inserted anywhere in the pipeline. This turns the framework into a **programmable request processor** rather than just a router.

For Barebones, we adopt the Express-style middleware pattern for several reasons:

1. **Conceptual Simplicity**: The `(req, res, next)` signature is intuitive for beginners
2. **Widespread Adoption**: Most Node.js developers encounter this pattern
3. **Gradual Complexity**: Can be extended later to support async/await or error middleware
4. **Explicit Control Flow**: `next()` makes the handoff between middleware obvious

The choice between **synchronous** and **asynchronous** middleware presents another key decision point:

Decision: Middleware Execution Model

- **Context**: Node.js is inherently asynchronous, but middleware patterns can be sync or async
- **Options Considered**:
  1. **Callback-based**: Middleware receives `next` callback, must call it to continue
  2. **Promise-based**: Middleware returns promises, framework awaits them
  3. **Async function-based**: Middleware can be async, `next()` returns a promise
- **Decision**: Support both synchronous and async middleware via `next()` callback, with automatic promise detection
- **Rationale**:
  - Backwards compatibility with existing Express patterns
  - Gradual migration path: sync middleware works, async optional
  - Error handling more straightforward with try/catch for async
- **Consequences**:
  - Framework must detect if `next()` returns a promise
  - Error handling must work for both sync throws and async rejections
  - Slightly more complex implementation than pure sync or pure async

The evolution of JavaScript frameworks reveals a clear trend: **abstraction layers that simplify common patterns without removing escape hatches**. Express's success stems from staying close to Node.js's native APIs while providing just enough abstraction to eliminate boilerplate. Barebones follows this philosophy—it should feel like "Node.js with training wheels" rather than a completely different paradigm.

Consider the **template engine** landscape, which influences Milestone 4:

Template Engine	Philosophy	Syntax	Compilation	Key Feature
Handlebars	Logic-less templates	`{{variable}}`, `{{#if}}`	Precompiled to JavaScript	Helpers as extension points
EJS	Embedded JavaScript	`<% code %>`, `<%= escaped %>`	Compiled to JavaScript	JavaScript in templates
Jinja2 (Python)	Designer-friendly	`{{ variable }}`, `{% if %}`	Compiled to Python bytecode	Template inheritance, filters
JSX	Component-based	HTML in JavaScript	Compiled to React.createElement	Type-safe, component composition

Barebones' template engine draws from Handlebars (logic-less philosophy) and EJS (simple embedding), focusing on **HTML escaping safety** and **template inheritance**—two concepts with significant security and architectural implications.

> **The Framework as Pedagogy**: Building Barebones isn't just about creating a usable tool; it's about understanding the **architectural patterns** that make web frameworks productive. Each design decision—from router data structure to middleware composition—reveals trade-offs that professional framework authors grapple with daily.

Implementation Guidance

> **Note**: This section bridges the high-level concepts to concrete implementation. We'll establish project structure and foundational code that supports the factory pipeline analogy.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Reasoning
HTTP Server	Node.js `http` module	`http2` module or `uWebSockets.js`	`http` is standard, well-documented, sufficient for learning
Routing Data Structure	Array of routes (linear scan)	Prefix tree (trie)	Trie is more efficient and teaches important CS concepts
Middleware Composition	Array + loop with `next()`	Function composition (`compose()`)	Array + loop is simpler to understand and debug
Template Compilation	String replacement with regex	AST transformation to render function	AST approach is faster and more extensible
Testing Framework	Node.js `assert` + manual tests	Jest or Mocha with Supertest	Manual tests reinforce understanding; frameworks can be added later

B. Recommended File/Module Structure

Create the following directory structure from the beginning to maintain organization as the framework grows:

```
barebones-framework/
├── package.json
├── README.md
├── src/
│   ├── index.js                  # Main framework export
│   ├── application.js            # Application class (main entry point)
│   ├── router/
│   │   ├── index.js              # Router class
│   │   ├── route.js              # Route class representing a single route
│   │   ├── layer.js              # Layer class for middleware/route encapsulation
│   │   └── trie.js               # Prefix tree implementation
│   ├── middleware/
│   │   ├── index.js              # Middleware utilities
│   │   ├── init.js               # Request/response enhancement middleware
│   │   └── logger.js             # Built-in logging middleware
│   ├── request.js                # Enhanced Request class
│   ├── response.js               # Enhanced Response class
│   ├── template/
│   │   ├── engine.js             # Template engine main class
│   │   ├── compiler.js           # Template compilation logic
│   │   └── helpers.js            # HTML escaping and utilities
│   └── utils/
│       ├── http.js               # HTTP status codes and methods
│       └── parse.js              # URL, query, cookie parsing utilities
├── examples/
│   └── basic-server.js           # Example usage
└── tests/
    ├── router.test.js
    ├── middleware.test.js
    └── template.test.js
```

C. Infrastructure Starter Code

Here's complete, working code for the foundational HTTP server wrapper that sets up the factory pipeline:

File: `src/application.js` (Initial Version)

JAVASCRIPT

```javascript
const http = require('http');

const Router = require('./router');

const Request = require('./request');

const Response = require('./response');


/**

 * Barebones Application - Main framework class

 * Acts as the factory conveyor belt controller

 */

class Application {

  constructor() {

    this.router = new Router();

    this.middleware = []; // Array of middleware functions

    this.settings = {}; // Framework settings



    // Create enhanced request/response prototypes

    this.request = Object.create(Request.prototype);

    this.response = Object.create(Response.prototype);

  }



  /**

   * Start the HTTP server on the given port

   * @param {number} port - Port to listen on

   * @param {Function} callback - Called when server starts

   * @returns {http.Server} Node.js HTTP server instance

   */

  listen(port, callback) {

    const server = http.createServer(this.handleRequest.bind(this));

    return server.listen(port, callback);

  }
```

```
/**
 * Handle incoming HTTP request
 * This is the entry point to the factory pipeline
 * @param {http.IncomingMessage} req - Native Node.js request
 * @param {http.ServerResponse} res - Native Node.js response
 * @private
 */
handleRequest(req, res) {
  // Enhance request and response objects
  const enhancedReq = Object.create(this.request);
  const enhancedRes = Object.create(this.response);

  // Link native objects
  enhancedReq.req = req;
  enhancedRes.res = res;
  enhancedReq.app = this;
  enhancedRes.app = this;

  // Initialize enhanced objects
  enhancedReq.init(req);
  enhancedRes.init(res);

  // Process through middleware and routing pipeline
  this.processPipeline(enhancedReq, enhancedRes);
}

/**
 * Process request through middleware then router
 * @param {Request} req - Enhanced request
```

```
   * @param {Response} res - Enhanced response
   * @private
   */
  processPipeline(req, res) {
    // TODO: Implement middleware chain execution
    // TODO: After middleware, route to appropriate handler
    // TODO: Handle errors during processing

    // TEMPORARY: Basic 404 response
    res.statusCode = 404;
    res.end('Not Found');
  }

  /**
   * Register a middleware function
   * @param {Function} fn - Middleware function (req, res, next)
   * @returns {Application} For chaining
   */
  use(fn) {
    this.middleware.push(fn);
    return this;
  }

  /**
   * Register a GET route
   * @param {string} path - Route path pattern
   * @param {...Function} handlers - Route handler(s) and middleware
   * @returns {Application} For chaining
   */
  get(path, ...handlers) {
```

```
    this.router.get(path, handlers);
    return this;
  }

  // Similar methods for post(), put(), delete(), etc.
  post(path, ...handlers) {
    this.router.post(path, handlers);
    return this;
  }
}

module.exports = Application;
```

File: `src/index.js` (Framework Entry Point)

JAVASCRIPT

```
const Application = require('./application');

/**
 * Create a new Barebones application
 * @returns {Application} New application instance
 */
function createApplication() {
  return new Application();
}

// Export the application creator
module.exports = createApplication;

// Also export constructors for advanced usage
module.exports.Application = Application;
module.exports.Router = require('./router');
module.exports.Request = require('./request');
module.exports.Response = require('./response');
```

File: `examples/basic-server.js` (Usage Example)

```javascript
const barebones = require('../src');

// Create app instance
const app = barebones();

// Register a simple route
app.get('/', (req, res) => {
  res.end('Hello from Barebones!');
});

// Start server
app.listen(3000, () => {
  console.log('Barebones server running on http://localhost:3000');
});
```

D. Core Logic Skeleton Code

File: `src/router/index.js` (Router Skeleton)

JAVASCRIPT

```javascript
const Route = require('./route');
const Layer = require('./layer');

/**
 * Router - Telephone switchboard for requests
 * Matches incoming requests to registered route handlers
 */
class Router {
  constructor() {
    this.routes = []; // Array of Route objects
    this.middleware = []; // Router-level middleware

    // TODO: Implement prefix tree (trie) for efficient routing
    // this.trie = new Trie();
  }

  /**
   * Register a route for HTTP GET method
   * @param {string} path - URL pattern with optional parameters
   * @param {Array<Function>} handlers - Route handler functions
   * @returns {Route} Created route object
   */
  get(path, handlers) {
    // TODO 1: Create a new Route instance
    // TODO 2: Configure route with GET method and path
    // TODO 3: Add handlers to the route (could be multiple middleware + final handler)
    // TODO 4: Store route in this.routes array
    // TODO 5: If using trie, add route to trie data structure
    // TODO 6: Return the route for potential chaining
```

```
    console.log(`Registered GET ${path} with ${handlers.length} handler(s)`);
    return null; // Placeholder
  }

  /**
   * Find a route matching the request method and path
   * @param {string} method - HTTP method (GET, POST, etc.)
   * @param {string} path - Request URL path
   * @returns {Object|null} Match result with route and extracted parameters
   */
  match(method, path) {
    // TODO 1: Iterate through this.routes array
    // TODO 2: For each route, check if method matches (case-insensitive)
    // TODO 3: Check if path matches the route's pattern
    // TODO 4: If using trie, traverse trie to find matching node
    // TODO 5: If match found, extract parameter values from path segments
    // TODO 6: Return object with: { route: Route, params: Object, pathMatch: boolean }
    // TODO 7: If no match found, return null

    console.log(`Looking for ${method} ${path}`);
    return null; // Placeholder
  }

  /**
   * Dispatch a request to the appropriate route handler
   * @param {Request} req - Enhanced request
   * @param {Response} res - Enhanced response
   * @returns {Promise} Resolves when request processing completes
   */
  dispatch(req, res) {
```

```
    // TODO 1: Call this.match() to find matching route
    // TODO 2: If no match, set res.statusCode = 404 and end response
    // TODO 3: If match found, execute route handlers in order
    // TODO 4: Handle async handlers (check if handler returns Promise)
    // TODO 5: Catch any errors and pass to error handler

    console.log(`Dispatching ${req.method} ${req.url}`);
    return Promise.resolve(); // Placeholder
  }
}

module.exports = Router;
```

E. Language-Specific Hints

1. **Node.js HTTP Module**: Use `http.createServer()` but note it returns an `EventEmitter`. Handle server errors with `server.on('error', callback)`.
2. **Stream Handling**: Request bodies are readable streams. For body parsing middleware, you'll need to handle the `'data'` and `'end'` events or use async iteration with `for await (const chunk of req)`.
3. **Object Enhancement**: Use `Object.create()` to create enhanced request/response objects that delegate to native objects. This is cleaner than modifying the native objects directly.
4. **Error Handling**: Use `try/catch` for synchronous code and `.catch()` for promises. Consider using `async/await` in your framework code for readability.
5. **Performance Considerations**: For the router trie, use a Map for child nodes for O(1) lookups. Use closures for compiled template functions to avoid recompilation.

F. Milestone Checkpoint

After implementing the basic structure above, you should be able to:

1. Start a basic server:

```bash
$ node examples/basic-server.js
```

Expected Output: `Barebones server running on http://localhost:3000`

2. Test with curl:

```bash
$ curl http://localhost:3000/
```

Expected Output: `Hello from Barebones!`

3. Verify 404 handling:

```
$ curl http://localhost:3000/nonexistent
```
BASH

Expected Output: `Not Found` with HTTP status 404

Common Issues and Fixes:

- **"Address already in use"**: Change port from 3000 to another number like 3001
- **"Cannot find module"**: Ensure you're running from project root or adjust require paths
- **No response/hanging**: Check that `res.end()` is being called in all code paths

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Server starts but no response	Request not reaching handler	Add console.log in `handleRequest()`	Ensure `app.get()` is called before `app.listen()`
"Cannot set headers after they are sent"	Multiple `res.end()` calls	Trace middleware execution order	Ensure each request calls `res.end()` exactly once
Route matches but parameters empty	Pattern extraction not implemented	Check router.match() return value	Implement parameter extraction from path segments
Middleware not executing	Middleware array empty or `next()` not called	Log middleware array length	Call `app.use()` before `app.listen()`; ensure `next()` is called
Server crashes on request	Unhandled exception in handler	Wrap handler in try/catch	Add error handling middleware

---

Goals and Non-Goals

> **Milestone(s):** All (Foundational)

This section establishes the explicit boundaries of what the Barebones framework aims to achieve and—equally important—what it intentionally excludes. Defining clear goals and non-goals serves as a compass for architectural decisions, preventing scope creep while ensuring the framework remains focused on its educational purpose. It creates a shared understanding of what constitutes a successful implementation versus what belongs in future extensions.

Explicit Goals

The primary mission of Barebones is to build a **minimal yet complete web framework** that teaches the core architectural patterns behind production frameworks like Express.js or Django. Every feature must serve this educational purpose, demonstrating how sophisticated systems emerge from simple, composable parts. The framework should feel like a "teaching skeleton"—revealing the underlying anatomy of web frameworks without the complexity of production-grade features.

The framework's goals are organized around four foundational capabilities, each corresponding to a project milestone:

Goal Category	Educational Objective	Concrete Capabilities	Associated Milestone
Request Routing	Understand how frameworks map HTTP requests to handler functions	1. Route registration with HTTP method specificity 2. URL pattern matching with parameter extraction 3. Route grouping via common prefixes 4. 404 handling for unmatched routes	Milestone 1
Middleware Pipeline	Grasp the chain-of-responsibility pattern for request processing	1. Sequential middleware execution 2. Explicit control flow via `next()` 3. Error propagation through middleware chain 4. Built-in logging middleware	Milestone 2
Request/Response Enhancement	Learn how frameworks augment native HTTP objects	1. Request body parsing (JSON, form-urlencoded) 2. Response helpers (JSON, HTML, redirects) 3. Cookie reading/writing 4. Query parameter parsing	Milestone 3
Template Rendering	Understand server-side dynamic content generation	1. Variable interpolation with HTML escaping 2. Conditional blocks (if/else) 3. Template inheritance 4. Context-based rendering	Milestone 4

Beyond these functional goals, Barebones has several **architectural goals** that guide its design:

1. **Educational Transparency**: The implementation should be readable and understandable by developers new to framework internals. Complex optimizations that obscure core concepts are avoided in favor of clear, pedagogical code.
2. **Composition Over Inheritance**: The framework should demonstrate how complex behavior emerges from composing simple functions (middleware) rather than deep class hierarchies.
3. **Progressive Enhancement**: The design should allow starting with a minimal HTTP server and gradually adding capabilities (routing, then middleware, then enhancements, then templating) without rewriting existing code.
4. **API Consistency**: The developer-facing API should follow established conventions from popular frameworks (like Express.js) to provide familiar patterns while revealing how they're implemented.

> **Design Principle**: Barebones follows the Unix philosophy of "do one thing well." Each component has a single responsibility, and complex behavior emerges from composition rather than monolithic design.

Explicit Non-Goals

To maintain focus on the educational objectives, Barebones intentionally omits several categories of features that are common in production frameworks. These non-goals are not statements about their unimportance, but rather acknowledgments that they would distract from the core learning objectives or introduce complexity disproportionate to the framework's mission.

Non-Goal Category	Specific Exclusions	Rationale for Exclusion
Production-Grade Security	1. CSRF protection 2. XSS prevention beyond basic escaping 3. SQL injection protection 4. Rate limiting 5. Security headers (HSTS, CSP)	Security requires extensive, specialized knowledge that would overwhelm the core framework concepts. Learners should understand security as a separate concern layered atop the framework.
Database Integration	1. ORM (Object-Relational Mapping) 2. Query builders 3. Migration systems 4. Connection pooling	Database integration involves significant complexity and design patterns (Active Record, Data Mapper) that deserve dedicated study separate from HTTP request processing.
Real-time Features	1. WebSocket support 2. Server-Sent Events (SSE) 3. Long-polling abstractions	Real-time communication involves fundamentally different protocol handling (stateful connections, event-driven architecture) that would complicate the request/response model.
Advanced Performance Optimizations	1. Response compression (gzip, brotli) 2. ETag generation 3. Request/response caching 4. Load balancing	While performance is critical in production, optimizations often obscure the primary data flow and architectural patterns that are the framework's focus.
Built-in Authentication/Authorization	1. User session management 2. OAuth/OpenID Connect integration 3. Role-based access control 4. JWT validation	Authentication involves complex security considerations and typically exists as middleware in production frameworks, which learners can implement once they understand the middleware pattern.
Administration & Tooling	1. Command-line generators 2. Hot reloading 3. Built-in testing utilities 4. API documentation auto-generation	These developer experience features, while valuable, are orthogonal to understanding HTTP request processing architecture.
Cross-platform Compatibility	1. Browser compatibility layers 2. Mobile-specific adapters	The framework focuses on Node.js HTTP fundamentals; platform abstractions would introduce unnecessary complexity.

Non-Goal Category	Specific Exclusions	Rationale for Exclusion
	3. Serverless deployment wrappers	
Protocol Extensions	1. HTTP/2 or HTTP/3 specific optimizations 2. GraphQL integrations 3. gRPC support	Supporting multiple protocols would dilute the focus on mastering the fundamental HTTP/1.1 request/response cycle.

> **Architectural Boundary**: By explicitly excluding these features, Barebones maintains a clean separation of concerns. The framework processes HTTP requests—it doesn't manage databases, handle real-time connections, or implement complex security policies. These concerns can be added as middleware or external services once the core architecture is understood.

The table below illustrates how production frameworks typically address these non-goals versus Barebones' educational approach:

Feature Area	Production Framework (Express/Django)	Barebones Educational Approach
Security	Comprehensive middleware ecosystem (helmet.js, csrf protection)	Basic HTML escaping only; security treated as application responsibility
Database	Integrated ORM with migrations, connection management	No built-in database support; plain SQL or external libraries recommended
Real-time	WebSocket middleware, Socket.io integration	HTTP/1.1 request/response only; WebSockets require separate server
Performance	Compression, caching, ETag middleware	No optimizations; focus on clear, unoptimized code for learning
Authentication	Passport.js integration, session management	No built-in auth; demonstrates patterns via simple middleware examples

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Further Exploration)
HTTP Server	Node.js built-in `http` module	`net` module for raw TCP (more control)
Routing Data Structure	Array of route objects (simplest)	Prefix tree (trie) for performance
Middleware Chain	Array of functions with `next` callback	Generator functions or async composition
Body Parsing	Buffering entire body then parsing	Streaming parsing with size limits
Template Engine	String replacement with regex	Abstract Syntax Tree (AST) compilation
Testing	Node.js `assert` module + manual tests	Jest/Mocha with full test suite

B. Recommended File/Module Structure

For the goals defined above, organize the codebase to reflect the separation of concerns:

```
barebones-framework/
├── package.json                # Project metadata and dependencies
├── README.md                   # Framework documentation
├── examples/                   # Example applications
│   ├── basic-server.js         # Minimal HTTP server example
│   ├── routing-demo.js         # Route matching demonstration
│   └── middleware-chain.js     # Middleware pipeline example
├── lib/                        # Framework source code
│   ├── index.js                # Main entry point (exports Application)
│   ├── application.js          # Application class (main framework instance)
│   ├── router/                 # Routing subsystem
│   │   ├── index.js            # Router class export
│   │   ├── route.js            # Route class definition
│   │   ├── layer.js            # Layer class (for route/middleware)
│   │   └── trie.js             # Prefix tree implementation (optional)
│   ├── middleware/             # Built-in middleware
│   │   ├── index.js            # Middleware exports
│   │   ├── logger.js           # Logging middleware
│   │   ├── json-parser.js      # JSON body parser
│   │   ├── urlencoded.js       # Form body parser
│   │   └── error-handler.js    # Error handling middleware
│   ├── request.js              # Enhanced Request class
│   ├── response.js             # Enhanced Response class
│   └── view/                   # Template engine
│       ├── engine.js           # Template engine interface
│       ├── compiler.js         # Template compilation logic
│       └── helpers.js          # HTML escaping, context utilities
└── tests/                      # Test suite
    ├── unit/                   # Unit tests for each component
    └── integration/            # Integration tests for full flows
```

C. Infrastructure Starter Code

Since the framework builds upon Node.js's native HTTP module, here's the minimal starter code for creating an HTTP server that delegates to the framework:

JAVASCRIPT

```javascript
// lib/index.js - Main framework entry point
const Application = require('./application');

/**
 * Creates a new Barebones application instance
 * @returns {Application} A new application instance
 */
function createApplication() {
  return new Application();
}

// Export both the factory function and the Application class
module.exports = createApplication;
module.exports.Application = Application;
```

JAVASCRIPT

```javascript
// lib/application.js - Application class skeleton

const http = require('http');


/**

 * Main Application class representing the web framework instance

 * @class Application

 */

class Application {

  constructor() {

    // Initialize core components

    this.router = null; // Will be set when router is created

    this.middleware = []; // Array of middleware functions

    this.settings = {}; // Application settings

    // Note: request and response are per-request, not stored here

  }


  /**

   * Start the HTTP server listening on the specified port

   * @param {number} port - Port to listen on

   * @param {Function} callback - Called when server starts

   * @returns {http.Server} The HTTP server instance

   */

  listen(port, callback) {

    const server = http.createServer((req, res) => {

      // TODO: Process request through framework pipeline

      this.handleRequest(req, res);

    });



    return server.listen(port, callback);

  }
```

```
  /**
   * Process an incoming HTTP request through the framework pipeline
   * @param {http.IncomingMessage} req - Native HTTP request object
   * @param {http.ServerResponse} res - Native HTTP response object
   */
  handleRequest(req, res) {
    // TODO: Implement request processing pipeline
    // 1. Create enhanced request/response objects
    // 2. Execute middleware chain
    // 3. Route matching and handler execution
    // 4. Error handling
    console.log(`Request received: ${req.method} ${req.url}`);
    res.statusCode = 501; // Not Implemented
    res.end('Framework not yet implemented');
  }

  /**
   * Register middleware function
   * @param {Function} fn - Middleware function with signature (req, res, next)
   * @returns {Application} This application instance for chaining
   */
  use(fn) {
    // TODO: Add middleware to the chain
    this.middleware.push(fn);
    return this;
  }

  /**
   * Register GET route
   * @param {string} path - Route path pattern
   * @param {...Function} handlers - Route handler functions
```

```
   * @returns {Application} This application instance for chaining
   */
  get(path, ...handlers) {
    // TODO: Implement route registration
    // Will delegate to router when router is implemented
    return this;
  }
}

module.exports = Application;
```

D. Core Logic Skeleton Code

For the router component (Milestone 1 goal), here's the skeleton that learners will implement:

JAVASCRIPT

```javascript
// lib/router/index.js - Router class skeleton

const Route = require('./route');

const Layer = require('./layer');


/**

 * Router class responsible for route registration and matching

 * @class Router

 */

class Router {

  constructor() {

    // TODO 1: Initialize routes array to store registered routes

    // TODO 2: Initialize middleware array for router-specific middleware

    // TODO 3: Consider storing routes in a prefix tree (trie) for performance

  }


  /**

   * Register a GET route

   * @param {string} path - Route path pattern (may include parameters like :id)

   * @param {Array<Function>} handlers - Handler functions for this route

   * @returns {Route} The created route object

   */

  get(path, handlers) {

    // TODO 1: Create a new Route instance for GET method

    // TODO 2: Store the route in the router's route collection

    // TODO 3: Return the route object for potential further configuration

  }


  /**

   * Find a route matching the given HTTP method and path

   * @param {string} method - HTTP method (GET, POST, etc.)

   * @param {string} path - Request path
```

```
   * @returns {Object|null} Match result with route and parameters, or null if no match
   */
  match(method, path) {
    // TODO 1: Normalize the path (remove query string, handle trailing slashes)
    // TODO 2: Iterate through registered routes for the given method
    // TODO 3: For each route, check if path pattern matches
    // TODO 4: If using pattern matching with parameters, extract them
    // TODO 5: Return null if no route matches (will lead to 404)
  }

  /**
   * Execute the router's route handlers for a matched route
   * @param {Request} req - Enhanced request object
   * @param {Response} res - Enhanced response object
   * @returns {Promise} Resolves when route handling is complete
   */
  async dispatch(req, res) {
    // TODO 1: Use match() to find the route for req.method and req.path
    // TODO 2: If no match, set res.statusCode = 404 and end response
    // TODO 3: If match found, attach parameters to req.params
    // TODO 4: Execute any router-level middleware
    // TODO 5: Execute the route's handler functions in sequence
  }
}

module.exports = Router;
```

E. Language-Specific Hints

1. **Node.js Streams**: For body parsing, remember that `req` is a readable stream. Use `req.on('data', ...)` and `req.on('end', ...)` to collect the body, or use the simpler `'data'` event accumulation pattern.
2. **Async/Await Integration**: Middleware and route handlers may be async functions. Use `Promise` chains or `async/await` in your dispatch logic to handle both sync and async functions gracefully.

3. **Prototype Enhancement**: For request/response enhancements, consider creating wrapper classes that delegate to the native objects rather than modifying the native objects directly.
4. **Event Emitters**: The native `http.IncomingMessage` and `http.ServerResponse` are EventEmitters. Be mindful of event listeners when extending these objects.

F. Milestone Checkpoint

After implementing the basic Application class and router skeleton, verify your progress:

1. **Test Command**: Run `node -e "const app = require('./lib'); const server = app().listen(3000, () => { console.log('Server listening on port 3000'); setTimeout(() => server.close(), 100); })"`
2. **Expected Output**: Should start a server and print "Server listening on port 3000" without errors.
3. **Manual Verification**: Create a test file:

```javascript
// test-server.js
const app = require('./lib')();

app.get('/', (req, res) => {
  res.end('Hello World');
});

app.listen(3000, () => {
  console.log('Test server running on http://localhost:3000');
});
```

Run it with `node test-server.js` and visit http://localhost:3000 in your browser. You should see "Hello World" (once routing is implemented).

4. **Signs of Issues**:
   - If you get "Cannot read property 'get' of undefined", check that your `createApplication()` function returns an Application instance.
   - If the server starts but routes don't work, ensure `app.get()` is properly delegating to the router.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Server starts but all routes return 501"	`handleRequest` method not processing routes	Add console.log to `handleRequest` to see if it's called	Implement route matching in `handleRequest`
"Middleware runs but route handler doesn't"	Forgetting to call `next()` in middleware	Add logging to middleware to track execution flow	Ensure each middleware calls `next()` to continue the chain
"Route with parameters matches incorrectly"	Route pattern matching logic flawed	Log the pattern matching process step by step	Implement proper parameter extraction with regex or path segmentation
"Cannot read property 'body' of undefined"	Request enhancement not applied before middleware	Check order of operations in `handleRequest`	Ensure enhanced Request object is created before middleware execution

High-Level Architecture

> **Milestone(s):** All (Foundational)

The high-level architecture of the Barebones framework organizes the complex task of HTTP request processing into a clean, predictable pipeline. At its core, the framework transforms raw HTTP requests—initially just streams of bytes over a network socket—into meaningful application logic execution and structured responses. This transformation occurs through a series of well-defined stages, each with a single responsibility, forming what we call the **factory pipeline**. Understanding this macro architecture is essential for grasping how individual components (routing, middleware, templating) interact to create a cohesive system.

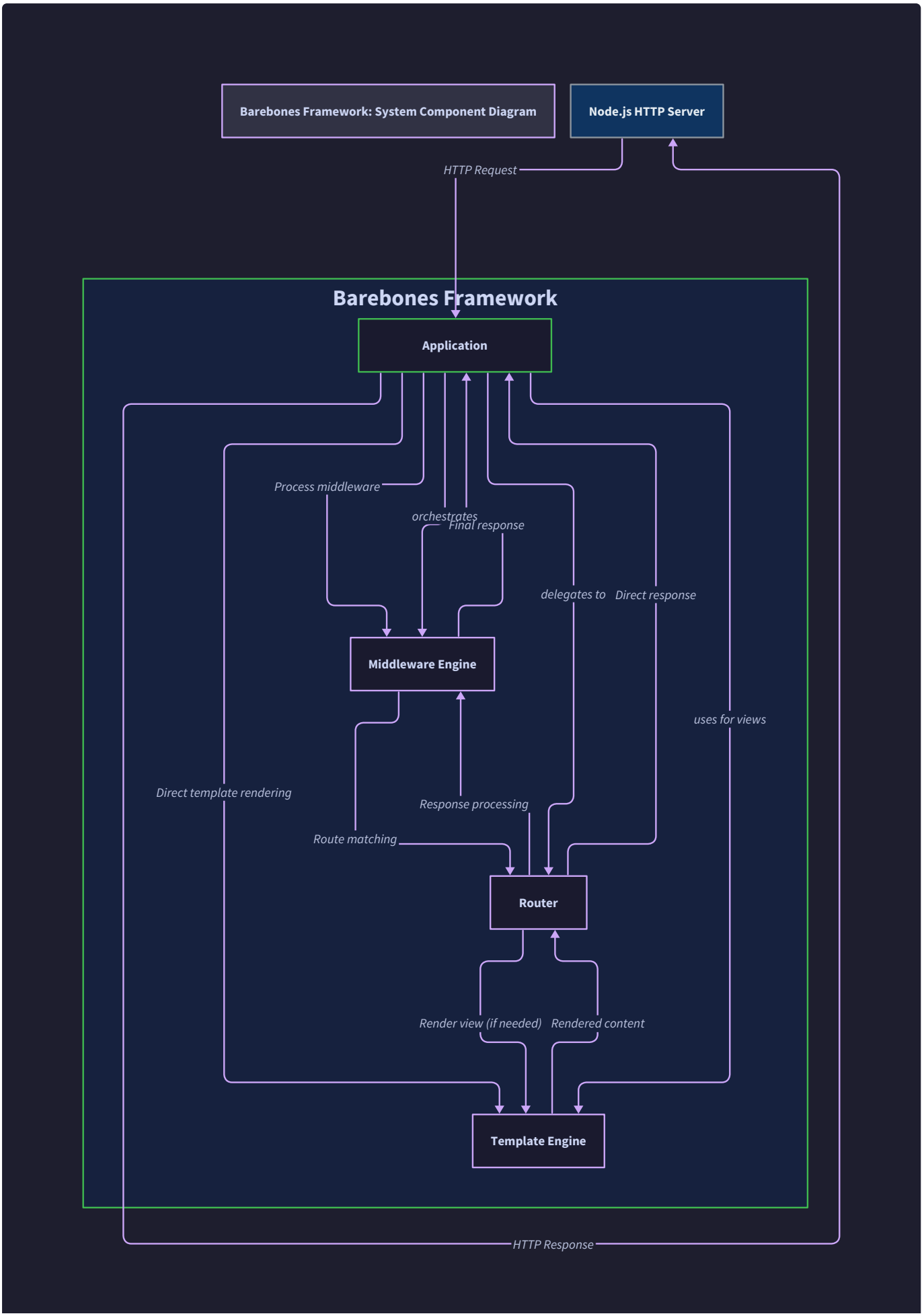
Component Overview & Pipeline Flow

Think of the framework as an **automated assembly line** in a factory. An HTTP request arrives as a "raw material" (bytes over TCP), placed on a conveyor belt. The belt moves this material through a series of "processing stations." Each station inspects, transforms, or enriches the material before passing it down the line. Finally, a finished "product" (HTTP response) is packaged and shipped back to the client. This mental model clarifies the separation of concerns and the unidirectional flow of control that makes the framework both predictable and extensible.

The framework comprises four core components that collaborate in sequence:

Component	Primary Responsibility	Key Analogy	Key Data Structures
Application	Orchestrates the entire pipeline, manages global state, and provides the public API.	Factory Foreman / Conveyor Belt Controller	`Application` instance (holds router, middleware chain, settings)
Middleware Engine	Executes a chain of processing functions that can inspect/modify requests/responses or perform side effects (logging, authentication).	Processing Stations on the Assembly Line	`Layer` objects in an ordered array
Router	Matches the incoming request's HTTP method and URL path to the appropriate handler function(s).	Switchboard Operator / Package Router	`Router` with a trie of `Route` nodes
Template Engine	Transforms template strings and data context into final HTML output.	Stamp and Fill Machine (Mad Libs)	Compiled template functions, context objects

The request lifecycle follows a strict, linear progression through these components, as illustrated in the system component diagram:


Barebones Framework: System Component Diagram
Node.js HTTP Server
HTTP Request
Barebones Framework
Application
Process middleware
orchestrates
Final response
delegates to
Direct response
Middleware Engine
uses for views
Direct template rendering
Response processing
Route matching
Router
Render view (if needed)
Rendered content
Template Engine
HTTP Response

The numbered flow below details each step in the pipeline. This sequence represents the "happy path" for a typical GET request that renders a template:

1. **HTTP Server Receives Request**: The Node.js built-in `http` module creates a socket connection, parses the raw HTTP request into an `IncomingMessage` object (`req`) and provides a `ServerResponse` object (`res`). It passes these to the framework's entry point: `app.handleRequest(req, res)` .
2. **Application Wraps Native Objects**: The `Application` instance creates enhanced `Request` and `Response` objects. These wrapper objects retain references to the native `req` / `res` and the `app` itself, and are progressively decorated with parsed data (body, query, cookies) and convenience methods (`res.json()`) as they move through the pipeline.
3. **Middleware Conveyor Belt Activation**: The application's middleware chain—an ordered array of functions—is invoked. Each middleware function receives the enhanced `Request` , `Response` , and a `next` callback. Middleware can:
    - Perform side effects (log request details)
    - Modify the request or response objects (add authentication data, set headers)
    - End the request early by calling a response method (`res.status(404).send('Not found')`)
    - Pass control to the next middleware by calling `next()`
    - Throw an error, which jumps to error-handling middleware
4. **Routing Switchboard Operation**: After all "global" middleware runs (or if a middleware passes control via `next()`), the request reaches the router. The `Router` acts as a switchboard, using the request's `method` (GET, POST, etc.) and `path` (/users/42) to look up the correct handler. It performs pattern matching against registered routes, extracting parameter values (like `id: 42`) and storing them in `req.params` .
5. **Route Handler Execution**: The matched route handler(s)—which are essentially specialized middleware—execute. This is where application business logic runs: reading from a database, validating input, computing results. The handler typically ends the request by calling a response method like `res.render('template', data)` or `res.json(data)` .
6. **Template Rendering (If Needed)**: If the handler calls a render method, the `Template Engine` compiles or retrieves a compiled template, combines it with the provided data context, executes any control logic (loops, conditionals), and produces a final HTML string. This string is then set as the response body.
7. **Response Finalization and Send**: The `Response` wrapper ensures proper headers are set (Content-Type, Cookies, Status Code), then writes the final body to the underlying `ServerResponse` object. The native `res.end()` is called, completing the HTTP response which is transmitted back to the client over the socket.
8. **Post-Response Cleanup**: Any resources held during request processing (open file handles, database connections in simple scenarios) should be released. In Node.js, this often happens automatically via garbage collection, but the pipeline design should avoid retaining request-specific references beyond the response.

This linear flow is complicated by two important **control flow deviations**:

1. **Early Termination**: Any middleware or handler can send a response early (e.g., `res.status(403).send('Forbidden')`). When this happens, the remaining middleware and route handlers in the pipeline are skipped. The framework must ensure that calling `next()` after a response has been sent does not cause errors.

2. **Error Propagation**: If any function in the pipeline throws an exception or passes an error to `next(err)`, normal pipeline progression stops. Control immediately jumps to the next **error-handling middleware** (a middleware with four parameters: `err, req, res, next`). If no error middleware handles it, the framework provides a default error response (500 Internal Server Error).

The following table summarizes the state transformation of the request/response pair as it moves through the pipeline:

Pipeline Stage	Key `Request` Additions	Key `Response` Additions	Typical Actions
Initial Wrap	`req.app`, `req.originalUrl`	`res.app`	Create wrapper objects
Body Parsing Middleware	`req.body` (parsed JSON/form data)	(none)	Parse request body stream
Cookie Parsing Middleware	`req.cookies` (key-value dict)	(none)	Parse Cookie header
Query Parsing	`req.query` (from URL ? params)	(none)	Parse URL query string
Router Matching	`req.params` (route parameters)	(none)	Match path, extract :id
Route Handler	(application-specific)	`res.locals` (template data)	Business logic, database calls
Template Rendering	(none)	`res.body` (HTML/JSON)	Render template, set Content-Type
Final Send	(none)	Headers written, body sent	Call `res.end()`

Recommended File/Module Structure

A well-organized codebase is crucial for managing complexity as the framework grows. The recommended structure follows a **modular, layered approach**, separating public API from internal implementation, and grouping related functionalities. This structure mirrors the architectural components and facilitates testing, maintenance, and future extensions.

Guiding Principles:

- **Public API in Root**: The main entry point (`index.js`) and primary constructor (`Application`) are exposed at the top level.
- **Internal Modules in `/lib`**: Implementation details live under an `internal` or `lib` directory, preventing accidental external dependencies.
- **One Component Per File**: Each major component (Router, Request, Response) gets its own file with a clear single responsibility.
- **Utilities Grouped**: Shared helpers (parsing, escaping, pattern matching) are grouped in a `utils/` directory.
- **Tests Mirror Source**: Test files live alongside implementation files with a consistent naming pattern (`*.test.js`).

The following directory layout provides the skeleton for the Barebones framework:

```
barebones-framework/
├── index.js                    # Primary public export: the Application class
├── package.json                # Project metadata and dependencies
├── lib/                        # Internal framework implementation
│   ├── application.js          # Application class (main orchestrator)
│   ├── request.js              # Enhanced Request class/wrapper
│   ├── response.js             # Enhanced Response class/wrapper
│   ├── router/                 # Router subsystem
│   │   ├── index.js            # Router class (public interface)
│   │   ├── route.js            # Route class (represents a single route)
│   │   ├── layer.js            # Layer class (wraps middleware/route handlers)
│   │   └── trie.js             # Prefix tree (trie) implementation for routing
│   ├── middleware/             # Built-in middleware implementations
│   │   ├── init.js             # Middleware to enhance req/res (always first)
│   │   ├── query.js            # Query string parsing middleware
│   │   ├── bodyParser.js       # JSON/URL-encoded body parsing middleware
│   │   ├── cookieParser.js     # Cookie parsing middleware
│   │   └── static.js           # Static file serving middleware (optional extension)
│   ├── template/               # Template engine subsystem
│   │   ├── engine.js           # Template engine main class/interface
│   │   ├── compiler.js         # Template string → render function compiler
│   │   ├── helpers.js          # Escaping, block management utilities
│   │   └── builtins/           # Built-in template directives
│   │       ├── echo.js         # {{ variable }} interpolation
│   │       ├── if.js           # {% if condition %} directive
│   │       └── each.js         # {% for item in list %} directive
│   └── utils/                  # Shared utility functions
│       ├── http.js             # HTTP status codes, method validation
│       ├── path.js             # Path matching, parameter extraction utils
│       ├── escape.js           # HTML escaping utilities (for XSS prevention)
│       └── mixin.js            # Object property mixing (for enhancing req/res)
└── test/                       # Comprehensive test suite
    ├── unit/                   # Unit tests for individual components
    │   ├── application.test.js
    │   ├── router/
    │   │   ├── router.test.js
    │   │   └── trie.test.js
    │   ├── middleware/
    │   │   ├── bodyParser.test.js
    │   │   └── query.test.js
    │   └── template/
    │       ├── engine.test.js
    │       └── compiler.test.js
    └── integration/            # Integration tests for full pipeline
        ├── routing.test.js     # Test routing with middleware
        └── rendering.test.js   # Test template rendering in requests
```

File Responsibility Matrix:

File	Primary Exports	Key Responsibilities
`index.js`	`Application` class (default export)	Framework entry point, re-exports key public APIs
`lib/application.js`	`Application` class definition	Creates HTTP server, manages middleware array, orchestrates request handling pipeline
`lib/request.js`	`Request` class	Wraps `http.IncomingMessage`, adds `body`, `query`, `params`, `cookies` properties
`lib/response.js`	`Response` class	Wraps `http.ServerResponse`, adds `json()`, `render()`, `cookie()` methods
`lib/router/index.js`	`Router` class	Registers routes, matches requests via trie, dispatches to route handlers
`lib/router/trie.js`	`TrieNode` class, search functions	Implements prefix tree for efficient route pattern matching
`lib/middleware/bodyParser.js`	`json()`, `urlencoded()` factory functions	Parses request body based on Content-Type, populates `req.body`
`lib/template/engine.js`	`TemplateEngine` class	Registers template engines, provides `render()` method, manages cache
`lib/utils/path.js`	`matchPath(pattern, path)` function	Compares URL path against route pattern, extracts parameters

This structure supports incremental development aligned with the milestones:

- **Milestone 1 (Routing)**: Implement `lib/router/` and `lib/application.js` basics.
- **Milestone 2 (Middleware)**: Implement `lib/middleware/` and enhance `application.js` pipeline.
- **Milestone 3 (Req/Res Enhancement)**: Implement `lib/request.js`, `lib/response.js`, and built-in middleware.
- **Milestone 4 (Templating)**: Implement `lib/template/` and integrate with `Response.prototype.render`.

The modular separation ensures that changes to one component (like improving the trie algorithm) have minimal impact on others, and testing can focus on individual units. As the framework evolves, new middleware can be added to the `middleware/` directory, new template directives to `template/builtins/`, without disrupting the core pipeline logic.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (For Learning)	Advanced Option (For Production)
HTTP Server	Node.js built-in `http` module	Same (production-ready)
Route Matching	Linear array scan for simplicity	Prefix tree (trie) for performance with many routes
Middleware Chain	Array + loop with `next` callback	Same (proven pattern from Express)
Body Parsing	Buffer entire body, then parse with `JSON.parse` or `querystring`	Streaming parser with size limits (e.g., `busboy` for multipart)
Template Compilation	String replacement with regex on each render	Pre-compile to JavaScript function using `new Function()`
File Structure	Single file for simplicity	Modular as shown above for maintainability

B. Starter Code: Application Skeleton

Create the following file structure with the provided starter code:

`index.js` (Public API):

```javascript
const Application = require('./lib/application');

// Create and export a factory function for convenience (like express())
function createApplication() {
  return new Application();
}

// Also export the Application class for extension
createApplication.Application = Application;

module.exports = createApplication;
```

`lib/application.js` (Core Orchestrator - Complete Starter):

JAVASCRIPT

```javascript
const http = require('http');

const Router = require('./router');

const Request = require('./request');

const Response = require('./response');


/**

 * Main Application class - the heart of the framework.

 * Manages middleware, routing, and request processing pipeline.

 */

class Application {

  constructor() {

    this.router = new Router();

    this.middleware = []; // Array of middleware functions

    this.settings = {};   // Application settings (e.g., 'view engine')



    // Default settings

    this.settings['view engine'] = 'html';

  }


  /**

   * Start HTTP server listening on given port.

   * @param {number} port - Port to listen on

   * @param {Function} callback - Called when server starts

   * @returns {http.Server} Node.js HTTP server instance

   */

  listen(port, callback) {

    const server = http.createServer((req, res) => {

      this.handleRequest(req, res);

    });

    return server.listen(port, callback);

  }
```

```
  /**
   * Process an incoming HTTP request through the middleware and routing pipeline.
   * @param {http.IncomingMessage} req - Native Node.js request object
   * @param {http.ServerResponse} res - Native Node.js response object
   */
  handleRequest(req, res) {
    // TODO 1: Create enhanced Request and Response objects
    //   - const request = new Request(req, this);
    //   - const response = new Response(res, this);

    // TODO 2: Dispatch the request through the middleware chain
    //   - Create a `next` function that advances through middleware array
    //   - Start the chain by calling the first middleware with (request, response, next)
    //   - If middleware calls next(), proceed to next middleware
    //   - If middleware sends response (res.end called), stop chain

    // TODO 3: If no middleware sends a response, dispatch to router
    //   - this.router.dispatch(request, response);

    // TODO 4: Handle uncaught errors in the pipeline
    //   - Wrap dispatch in try/catch
    //   - If error thrown and no response sent, send 500 error
  }

  /**
   * Register middleware function(s) or route handler.
   * @param {Function|string} fnOrPath - Middleware function or path prefix
   * @param {...Function} fns - Middleware functions (if path provided)
   * @returns {Application} For chaining
   */
```

```
  use(fnOrPath, ...fns) {
    // TODO 1: Determine if first argument is a path string or function
    //   - If string: it's a path prefix for following middleware
    //   - If function: it's a middleware at root path ('/')

    // TODO 2: Add middleware to this.middleware array
    //   - Store as object: { path: prefixPath, fn: middlewareFunction }
    //   - Default path is '/' if not specified

    // TODO 3: Return `this` for method chaining
  }

  /**
   * Register a GET route.
   * Delegates to router.
   * @param {string} path - Route pattern (e.g., '/users/:id')
   * @param {...Function} handlers - Route handler functions
   * @returns {Application} For chaining
   */
  get(path, ...handlers) {
    this.router.get(path, handlers);
    return this;
  }

  // Similar methods for post, put, delete, etc.
  // post(path, ...handlers) { ... }
  // put(path, ...handlers) { ... }
  // delete(path, ...handlers) { ... }
}

module.exports = Application;
```

`lib/request.js` **(Enhanced Request - Skeleton):**

```javascript
/**
 * Enhanced Request object wrapping http.IncomingMessage.
 */
class Request {
  constructor(req, app) {
    this.req = req;           // Original Node.js request
    this.app = app;           // Reference to Application instance

    // Enhanced properties (populated later by middleware)
    this.body = null;         // Parsed request body
    this.query = {};          // Parsed query string parameters
    this.params = {};         // Route parameters (e.g., { id: '42' })
    this.cookies = {};        // Parsed cookies

    // Copy useful properties from native request
    this.method = req.method;
    this.url = req.url;
    this.headers = req.headers;
  }

  // TODO: Add getter for originalUrl, protocol, hostname, etc.
  // TODO: Add method to get header value (case-insensitive)
}

module.exports = Request;
```

`lib/response.js` **(Enhanced Response - Skeleton):**

JAVASCRIPT

```javascript
/**
 * Enhanced Response object wrapping http.ServerResponse.
 */
class Response {
  constructor(res, app) {
    this.res = res;           // Original Node.js response
    this.app = app;           // Reference to Application instance
    this.statusCode = 200;    // Default status code
    this.headers = {};        // Headers to be sent
    this.body = null;         // Response body
    this.sent = false;        // Whether response has been sent
  }

  /**
   * Send JSON response.
   * @param {any} data - Data to serialize as JSON
   * @returns {Response} For chaining
   */
  json(data) {
    // TODO 1: Set Content-Type header to 'application/json'
    // TODO 2: Serialize data to JSON string (JSON.stringify)
    // TODO 3: Set as response body and send
    // TODO 4: Return `this` for chaining
  }

  /**
   * Set response status code.
   * @param {number} code - HTTP status code
   * @returns {Response} For chaining
   */
```

```
  status(code) {
    // TODO: Set this.statusCode, return `this` for chaining
  }

  /**
   * Send response with optional body.
   * @param {string|Buffer|Object} body - Response body
   */
  send(body) {
    // TODO 1: Determine Content-Type based on body type
    // TODO 2: Set headers
    // TODO 3: Write status code and headers if not already sent
    // TODO 4: Write body and end response
    // TODO 5: Set this.sent = true
  }

  // TODO: Implement cookie(name, value, options), redirect(url), render(template, context)
}

module.exports = Response;
```

C. Language-Specific Hints (JavaScript/Node.js)

1. **Use `http.createServer`**: The native Node.js module is sufficient; avoid additional dependencies for the core server.
2. **Streaming Awareness**: Remember `req` is a readable stream. For body parsing, you'll need to collect data chunks with `req.on('data', ...)` and `req.on('end', ...)`.
3. **`new Function()` for Templates**: While `eval` is dangerous, `new Function('context', 'return ...')` can safely compile templates to JavaScript functions in an isolated scope.
4. **Class Syntax**: Use ES6 classes for `Application`, `Request`, `Response` for clarity. Export them with `module.exports`.
5. **Error Handling with Async**: Middleware may be async (return Promises). Use `try/catch` and consider wrapping middleware execution in `Promise.resolve().then()` to handle both sync and async errors uniformly.

D. Milestone Checkpoint: Architecture Foundation

After setting up this architecture skeleton, verify your foundation:

1. **Create a basic app**:

```javascript
// test-server.js

const barebones = require('./index');

const app = barebones();



app.get('/', (req, res) => {

  res.send('Hello World');

});



app.listen(3000, () => {

  console.log('Server listening on port 3000');

});
```

2. **Expected Behavior**:
   - Run `node test-server.js`
   - Visit `http://localhost:3000` in browser or use `curl`
   - You should see "Hello World" (though `res.send` isn't implemented yet)
   - The server should start without errors
3. **Signs of Trouble**:
   - **"Cannot find module"**: Check your file structure and `require` paths.
   - **"app.get is not a function"**: Ensure `Application.prototype.get` delegates to `router.get`.
   - **Server starts but hangs on request**: `handleRequest` may not be calling `res.end()`.
   - **Multiple responses sent**: Ensure `this.sent` flag prevents double-sending.

This architectural foundation sets the stage for implementing each component in detail. The pipeline pattern established here will remain constant as you add routing, middleware, and templating capabilities.

Data Model

> **Milestone(s):** All (Foundational)

The data model defines the fundamental building blocks that give our web framework its structure and behavior. Think of these types as the **DNA of the framework**—they encode the instructions for how every request flows, how routes are

matched, and how responses are generated. Without a well-defined data model, our components would be like disconnected organs without a skeleton to structure them.

Before diving into technical details, consider this analogy: **The Framework as a Theater Production**. The `Application` is the theater itself, with its backstage machinery. The `Router` is the stage manager who knows which scene (route handler) to call based on the script (URL). Each `Middleware` function is a lighting or sound technician that transforms the stage before the actors appear. The `Request` and `Response` are the lead actors—enhanced versions of basic HTTP objects with additional props and costumes. Finally, `Templates` are the script pages with placeholders that get filled in during performance.

This mental model helps visualize how these pieces interact: the theater (`Application`) receives an audience request (HTTP), the stage manager (`Router`) finds the right scene, technicians (`Middleware`) adjust the environment, and enhanced actors (`Request` / `Response`) perform the scene with a dynamically filled script (`Template`).

Core Types and Interfaces

The framework's core types are implemented as JavaScript classes or plain objects. Each type has specific responsibilities and properties that collectively enable the request-processing pipeline.

Application Type

The `Application` is the central orchestrator—the framework's entry point that initializes all components and starts the HTTP server.

Field	Type	Description
`router`	`Router`	The main router instance that handles all route registration and matching. The application delegates route operations to this component.
`middleware`	`Array<Function>`	An ordered list of global middleware functions. These functions execute on every request before route-specific middleware and handlers.
`settings`	`Object`	A key-value store for application-level configuration (e.g., `env` for environment, `view engine` for default template engine).

Router Type

The `Router` manages the collection of routes and provides methods for registering HTTP method handlers and middleware. Internally, it uses a **prefix tree (trie)** for efficient route matching.

Field	Type	Description
`routes`	`Array<Route>`	A flat list of all registered routes, maintained for debugging and introspection. This is a secondary store; the primary matching structure is the trie.
`middleware`	`Array<Function>`	Route-specific middleware that runs only for routes registered on this router instance (used for route groups).
`trie`	`TrieNode`	The root node of the prefix tree used for matching URL paths. This enables O(path length) lookup time regardless of the number of routes.

Route Type

A `Route` represents a single endpoint definition, associating an HTTP method and URL pattern with one or more handler functions.

Field	Type	Description
`path`	`String`	The original URL pattern string (e.g., `/users/:id`). Used for debugging and display.
`method`	`String`	The HTTP method (uppercase), such as `GET`, `POST`, `PUT`, or `DELETE`.
`handlers`	`Array<Function>`	An ordered list of handler functions to execute when this route matches. The last handler typically sends the response, while preceding ones are route-specific middleware.

Layer Type

`Layer` is an internal abstraction that wraps either a middleware function or a route handler with additional metadata for path matching. It's the **unified unit of execution** in the pipeline.

Field	Type	Description
`fn`	`Function`	The actual middleware or route handler function with signature `(req, res, next)`.
`path`	`String`	The path prefix this layer is mounted on (for middleware) or the full route pattern (for routes).
`regexp`	`RegExp`	A compiled regular expression derived from the path pattern, used to test if an incoming URL matches this layer.
`keys`	`Array<Object>`	An array of parameter key objects (with `name` property) extracted from the path pattern (e.g., `[{name: 'id'}]` for `/:id`).

TrieNode Type

`TrieNode` represents a single node in the router's prefix tree. Each node corresponds to a segment of a URL path (e.g., `users`, `:id`, `posts`).

Field	Type	Description
`children`	`Map<String, TrieNode>`	A map from path segment strings (static segments) to child nodes. Keys are literal strings like `"users"` or `"posts"`.
`handlers`	`Map<String, Function>`	A map from HTTP methods to the handler function(s) for routes that terminate at this node. Only leaf nodes or parameter nodes have handlers.
`paramName`	`String` or `null`	If this node represents a parameter segment (e.g., `:id`), this field stores the parameter name (`"id"`). For static nodes, it is `null`.
`isWildcard`	`Boolean`	Indicates whether this node is a catch-all wildcard (e.g., `*`). Wildcard nodes match any remaining path segments.

Request Type

`Request` is an enhanced wrapper around Node.js's native `http.IncomingMessage`. It extends the raw request with parsed data, convenience properties, and helper methods.

Field	Type	Description
`req`	`IncomingMessage`	The native Node.js HTTP request object. All original properties (headers, url, method) remain accessible.
`app`	`Application`	Reference to the application instance that is processing this request. Allows handlers to access app settings and methods.
`body`	`Any`	The parsed request body (populated by body-parsing middleware). Could be an object (for JSON), a string, or a Buffer depending on content type.
`query`	`Object`	A dictionary of parsed query string parameters from the URL (e.g., `?name=John` becomes `{name: 'John'}`).
`params`	`Object`	A dictionary of route parameters extracted from the URL path (e.g., for route `/users/:id`, path `/users/123` gives `{id: '123'}`).
`cookies`	`Object`	A dictionary of parsed cookies from the `Cookie` header. Populated by cookie-parsing middleware.

Response Type

`Response` is an enhanced wrapper around Node.js's `http.ServerResponse`. It provides chainable helper methods for common response tasks.

Field	Type	Description
`res`	`ServerResponse`	The native Node.js HTTP response object. Used internally to write headers and body.
`app`	`Application`	Reference to the application instance, allowing response methods to access app settings (e.g., template engine).
`statusCode`	`Number`	The HTTP status code to send (defaults to 200). Can be set via `res.status(404)`.

Template Type

`Template` represents a compiled template that can be rendered with a data context to produce HTML output. The framework may store compiled templates in a cache.

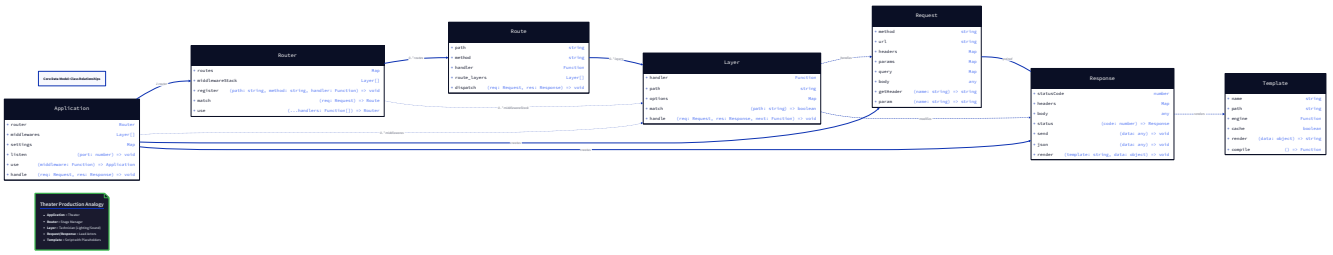
Field	Type	Description
`source`	`String`	The original template source code (with placeholders and directives).
`compiledFn`	`Function`	A JavaScript function generated by the template engine. When called with a context object, it returns the rendered HTML string.
`blocks`	`Map<String, String>`	For templates that support inheritance, this map stores named block contents (e.g., `content`, `sidebar`) that child templates can override.

State and Relationship Diagrams

The internal state of the framework is distributed across these types, forming a graph of references that enable the request-processing pipeline. Understanding these relationships is crucial for debugging and extension.

Component Relationships

The primary relationships can be visualized in the class diagram:



Key Relationships:

1. **Application owns Router and Middleware Chain**
   - The `Application` instance contains a `router` property (a `Router` instance) and a `middleware` array.
   - This composition means the application controls the lifecycle of the router and the global middleware chain.
2. **Router manages Routes and Trie**
   - The `Router` maintains a `routes` array (for linear access) and a `trie` root node (for efficient matching).
   - Each `Route` is registered in both structures: appended to the `routes` array and inserted into the trie.
   - The router's `middleware` array holds middleware specific to route groups (prefix-based).
3. **Layer wraps Execution Units**
   - Both global middleware and route handlers are wrapped in `Layer` objects before being added to execution chains.
   - This uniform wrapping allows the same matching and dispatch logic to handle middleware and routes.
4. **Request and Response reference Application**
   - Each `Request` and `Response` instance holds a reference to the `Application` that created them (`app` property).
   - This allows handlers to access application settings (e.g., `req.app.get('view engine')`) and methods.
5. **Template Engine caches Templates**

- The template engine (not shown as a separate type in the core model) typically maintains a cache mapping template names to `Template` objects.
- Each `Template` contains the compiled render function for fast execution.

State Transitions During Request Processing

The framework's state evolves as a request flows through the pipeline. Consider a typical GET request to `/api/users/123`:

1. **Initial State (Application Start)**
   - `app.middleware`: `[]` (empty, then filled with logging, body parsing, etc.)
   - `app.router.trie`: Root node with no children.
   - `app.router.routes`: `[]`
2. **After Route Registration**
   - Developer calls `app.get('/api/users/:id', userHandler)`
   - A new `Route` is created: `{path: '/api/users/:id', method: 'GET', handlers: [userHandler]}`
   - The route is added to `app.router.routes` array.
   - The trie is updated: nodes for segments `'api'`, `'users'`, and a parameter node with `paramName: 'id'` are created or reused. The handler is stored in the parameter node's `handlers` map under key `'GET'`.
3. **During Request Processing**
   - The HTTP server emits a request event; `app.handleRequest` creates enhanced `Request` and `Response` objects.
   - `req.params` starts as `{}`, `req.query` is parsed from the URL.
   - The middleware chain is traversed: each `Layer`'s `regexp` tests the request path; matching middleware executes and may modify `req` / `res`.
   - The router matches the path against the trie: segments `['api', 'users', '123']` traverse the trie, extracting `'123'` as the value for parameter `'id'`.
   - `req.params` becomes `{id: '123'}`.
   - The matched handler (and any route-specific middleware) executes, calling `res.json()` which sets `res.statusCode` and writes the response body.
4. **After Response Sent**
   - The `Response` object's internal state (headers sent, body written) prevents further modifications.
   - The `Request` and `Response` objects become eligible for garbage collection unless referenced elsewhere.

Data Structure Lifecycle Example

Let's trace the lifecycle of a `Layer` object for a logging middleware:

1. **Creation**: `app.use(loggingMiddleware)` creates a `Layer` with:
   - `fn`: `loggingMiddleware` function
   - `path`: `'/'` (default, matches all paths)
   - `regexp`: `RegExp` that matches any path
   - `keys`: `[]` (no parameters)

2. **Registration**: The layer is pushed onto `app.middleware` array.
3. **Execution**: On each request, the dispatch loop iterates over `app.middleware`. For each layer:
    - The layer's `regexp` tests `req.url` → matches.
    - `layer.fn(req, res, next)` is called.
    - The middleware logs details and calls `next()` to continue.
4. **Cleanup**: The layer remains in memory for the application's lifetime, reused for every request.

> **Design Insight:** The separation of `routes` array and `trie` structure exemplifies the **space-time trade-off**. The array provides simplicity and linear iteration for debugging/introspection (e.g., listing all routes), while the trie provides fast O(path length) lookup. This hybrid approach balances developer convenience with runtime performance.

Common Pitfalls in Data Modeling

⚠️ Pitfall: Mutable Shared State in Middleware Arrays

- **Description:** Accidentally mutating the `app.middleware` array (e.g., by sorting, reversing, or assigning a new array) after requests have started processing.
- **Why it's wrong:** The dispatch loop iterates over the array indices; mutation during iteration can cause middleware to be skipped or executed multiple times, leading to unpredictable behavior.
- **Fix:** Treat the middleware array as immutable after the first request. Register all middleware before calling `app.listen()`. If dynamic middleware is needed, implement a locking mechanism or use a proxy array.

⚠️ Pitfall: Forgetting to Initialize Object Properties

- **Description:** Creating `Request` or `Response` instances without setting all properties (e.g., leaving `req.params` as `undefined` instead of `{}`).
- **Why it's wrong:** Handlers expecting `req.params.id` will throw `Cannot read property 'id' of undefined`, crashing the application.
- **Fix:** Always initialize object properties in constructors. For example:

```javascript
class Request {
  constructor(req, app) {
    this.req = req;
    this.app = app;
    this.body = null;
    this.query = {};
    this.params = {};
    this.cookies = {};
  }
}
```

⚠️ **Pitfall: Storing Request-Specific Data in Application State**

- **Description:** Storing data from one request (e.g., `app.currentUser`) in the application object, which is shared across all requests.
- **Why it's wrong:** Concurrent requests will overwrite each other's data, leading to security issues (user A seeing user B's data) and race conditions.
- **Fix:** Always store request-specific data on the `req` object (e.g., `req.user`). The application object should only contain truly global configuration.

Implementation Guidance

> **Technology Note:** Since we're building an educational framework in JavaScript/Node.js, we'll use ES6 classes for types and native JavaScript data structures (Map, Array) for collections. This keeps dependencies to zero and the code transparent.

Recommended File Structure for Data Model

As the framework grows, organize types into logical modules:

```
barebones-framework/
├── lib/
│   ├── application.js      # Application class (main export)
│   ├── router.js           # Router class and TrieNode
│   ├── layer.js            # Layer class
│   ├── request.js          # Enhanced Request class
│   ├── response.js          # Enhanced Response class
│   ├── middleware/         # Built-in middleware
│   │   ├── json.js         # JSON body parser
│   │   ├── query.js        # Query string parser
│   │   └── static.js       # Static file serving (future)
│   └── template.js         # Template engine
├── test/                   # Comprehensive tests
└── package.json
```

Infrastructure Starter Code: Core Type Skeletons

Here are complete, minimal implementations for the foundational types. Learners should extend these with the logic described in subsequent sections.

`lib/layer.js` – The unified execution wrapper:

JAVASCRIPT

```javascript
/**
 * Layer represents a middleware or route handler with path matching capabilities.
 */
class Layer {
  /**
   * @param {Function} fn - Middleware or handler function (req, res, next)
   * @param {string} [path='/'] - Path pattern this layer is mounted on
   * @param {Object} [options] - Additional options (not used in basic version)
   */
  constructor(fn, path = '/', options = {}) {
    this.fn = fn;
    this.path = path;
    this.keys = [];
    this.regexp = this.compile(path);
  }

  /**
   * Compile path pattern to a regular expression.
   * Basic implementation: converts static paths only.
   * TODO: Add parameter extraction (Milestone 1)
   */
  compile(path) {
    // Escape regex special characters, replace :param with capture groups
    // For now, just exact match
    const escaped = path.replace(/[.*+?^${}()|[\]\\]/g, '\\$&');
    return new RegExp(`^${escaped}$`);
  }

  /**
   * Test if this layer matches the given request path.
   * @param {string} path - Request URL path
```

```
   * @returns {boolean} True if path matches this layer's pattern
   */
  match(path) {
    return this.regexp.test(path);
  }

  /**
   * Execute the layer's function.
   * @param {Request} req - Enhanced request object
   * @param {Response} res - Enhanced response object
   * @param {Function} next - Callback to proceed to next layer
   */
  handle(req, res, next) {
    try {
      this.fn(req, res, next);
    } catch (err) {
      next(err);
    }
  }
}

module.exports = Layer;
```

`lib/request.js` – Enhanced request wrapper:

JAVASCRIPT

```javascript
const { URL } = require('url');


/**

 * Enhanced Request wrapper around http.IncomingMessage.

 */

class Request {

  /**

   * @param {http.IncomingMessage} req - Native HTTP request

   * @param {Application} app - Application instance

   */

  constructor(req, app) {

    this.req = req;

    this.app = app;



    // Initialize enhanced properties

    this.body = null;

    this.query = {};

    this.params = {};

    this.cookies = {};



    // Parse query string automatically

    this._parseQuery();

  }


  /**

   * Parse query string from URL.

   * TODO: Make this more robust (Milestone 3)

   */

  _parseQuery() {

    try {

      const url = new URL(this.req.url, `http://${this.req.headers.host}`);
```

```
      for (const [key, value] of url.searchParams) {
        this.query[key] = value;
      }
    } catch (err) {
      // If URL parsing fails, keep empty query object
    }
  }

  // Additional convenience methods will be added in Milestone 3
  // get(field), accepts(type), etc.
}

module.exports = Request;
```

`lib/response.js` – Enhanced response wrapper:

JAVASCRIPT

```javascript
/**
 * Enhanced Response wrapper around http.ServerResponse.
 */
class Response {
  /**
   * @param {http.ServerResponse} res - Native HTTP response
   * @param {Application} app - Application instance
   */
  constructor(res, app) {
    this.res = res;
    this.app = app;
    this.statusCode = 200;
  }

  /**
   * Set HTTP status code.
   * @param {number} code - Status code
   * @returns {Response} this for chaining
   */
  status(code) {
    this.statusCode = code;
    return this;
  }

  /**
   * Send a plain text response.
   * TODO: Extend to handle objects, JSON (Milestone 3)
   */
  send(body) {
    this.res.statusCode = this.statusCode;
    this.res.end(body);
```

```
  }

  /**
   * Send JSON response.
   * TODO: Implement with proper Content-Type (Milestone 3)
   */
  json(data) {
    // Placeholder - will be implemented in Milestone 3
    this.res.statusCode = this.statusCode;
    this.res.setHeader('Content-Type', 'application/json');
    this.res.end(JSON.stringify(data));
  }
}

module.exports = Response;
```

Core Logic Skeleton: Router Trie Structure

`lib/router.js` – Skeleton for the router with trie-based matching:

JAVASCRIPT

```javascript
const Layer = require('./layer');


/**
 * TrieNode for prefix tree route matching.
 */
class TrieNode {
  constructor() {
    // TODO 1: Initialize children as a Map from segment string to TrieNode
    // TODO 2: Initialize handlers as a Map from HTTP method to handler function(s)
    // TODO 3: Set paramName to null and isWildcard to false by default
  }
}


/**
 * Router manages route registration and matching.
 */
class Router {
  constructor() {
    // TODO 1: Initialize routes array (for linear storage)
    // TODO 2: Initialize middleware array (for route-group middleware)
    // TODO 3: Initialize trie as a new TrieNode root
  }


  /**
   * Register a route for a specific HTTP method.
   * @param {string} method - HTTP method (GET, POST, etc.)
   * @param {string} path - URL pattern with optional parameters
   * @param {Function[]} handlers - Handler functions
   * @returns {Route} The created route object
   */
  register(method, path, handlers) {
```

```
    // TODO 1: Create a new Route object with path, method, handlers
    // TODO 2: Add route to this.routes array
    // TODO 3: Split path into segments (ignore leading/trailing slashes)
    // TODO 4: Traverse trie, creating nodes as needed for each segment
    // TODO 5: For parameter segments (starting with ':'), set paramName on node
    // TODO 6: At the final node, store handlers in handlers map under the method key
    // TODO 7: Return the created route
  }

  /**
   * Find a matching route for the given method and path.
   * @param {string} method - HTTP method
   * @param {string} path - Request URL path
   * @returns {Object|null} Match object with {handlers, params} or null
   */
  match(method, path) {
    // TODO 1: Initialize params as empty object
    // TODO 2: Split path into segments
    // TODO 3: Start at root of trie
    // TODO 4: For each path segment:
    //   - First try exact match in children map
    //   - If no match, look for parameter child (node with paramName)
    //   - If parameter child exists, add to params: params[node.paramName] = segment
    //   - If no match, return null
    // TODO 5: At the final node, get handlers for the method
    // TODO 6: If handlers exist, return {handlers, params}
    // TODO 7: If no handlers, return null (path matched but not method)
  }

  // Convenience methods
  get(path, ...handlers) {
```

```
    return this.register('GET', path, handlers);
  }

  post(path, ...handlers) {
    return this.register('POST', path, handlers);
  }

  // Similar for put, delete, etc.
}

module.exports = Router;
```

Language-Specific Hints for JavaScript/Node.js

1. **Use ES6 Classes:** They provide clean syntax for our types. Remember to call `super()` in subclasses if extending.
2. **Maps vs Objects:** Use `Map` for `TrieNode.children` and `handlers` because keys can be arbitrary strings and Maps maintain insertion order (useful for debugging).
3. **Regular Expression Performance:** Compile route patterns to RegExp objects once during layer creation, not on every request.
4. **Error Handling:** Always wrap user-provided middleware/handlers in try-catch blocks to prevent uncaught exceptions from crashing the server.
5. **Streaming Considerations:** When enhancing Request/Response, be careful not to interfere with Node.js's streaming capabilities. For example, body parsing middleware should buffer the entire body, but that's acceptable for learning.

Milestone Checkpoint: Data Model Validation

After implementing the skeleton types, verify the basic structure works:

1. **Create test file `test/data-model.test.js`:**

```javascript
const Application = require('../lib/application');

const Router = require('../lib/router');

const Request = require('../lib/request');

const Response = require('../lib/response');


// Test that types can be instantiated

const app = new Application();

console.assert(app.router instanceof Router, 'App should have router');


const router = new Router();

console.assert(Array.isArray(router.routes), 'Router should have routes array');


// Test request/response enhancement

const mockReq = { url: '/test?name=John', headers: { host: 'localhost' } };

const mockRes = {};

const req = new Request(mockReq, app);

const res = new Response(mockRes, app);


console.assert(req.query.name === 'John', 'Request should parse query string');

console.assert(res.statusCode === 200, 'Response should default to status 200');


console.log('✅ Data model basic validation passed');
```

2. **Run the test:** `node test/data-model.test.js`
    - **Expected output:** `✅ Data model basic validation passed`
    - **If it fails:** Check that all class constructors properly initialize fields and that the query parsing logic works.
3. **Manual verification:** Create a simple app that uses these types:

```javascript
const Application = require('./lib/application');

const app = new Application();


// Should not throw errors

app.get('/', (req, res) => {

  res.send('Hello');

});


console.log('✅ Basic app creation works');
```

This checkpoint ensures the type definitions are syntactically correct and can be instantiated before moving to the complex logic of routing and middleware.

Component Design: The Router

> **Milestone(s):** Milestone 1: Routing

The router is the **central dispatch system** of the web framework. Its primary responsibility is to map incoming HTTP requests—specified by a combination of method (GET, POST, etc.) and URL path (`/api/users/42`)—to the correct handler function(s) that will generate the response. Without a router, an application would require a massive `if-else` statement to examine every request's URL and method, leading to unmaintainable code. The router provides a declarative API for defining these mappings and an efficient algorithm for resolving them at runtime.

Mental Model: The Telephone Switchboard

Imagine a large office building with a **telephone switchboard** from the 1950s. The switchboard operator receives incoming calls (HTTP requests) and must connect each call to the correct desk extension (route handler). The operator doesn't decide what is said in the conversation; they only ensure the connection is made correctly. The caller provides a number (the URL path), and the operator follows a set of rules:

1. **Exact Match:** For a call to "extension 200" (path `/api/users`), the operator directly plugs into the socket labeled "200".
2. **Parameterized Match:** For a call to "extension 2xx" (path `/api/users/:id`), the operator knows that any three-digit number starting with "2" should be routed to the "User Details" desk. They also note down the specific digits (e.g., "id=42") to give to the person at that desk.
3. **No Match:** If the caller asks for "extension 999" and no such desk exists, the operator politely says, "I'm sorry, that number is not in service" (404 Not Found).

This analogy captures the router's essence: it's a **lookup and connection service**. It doesn't process the request content (the "conversation") but is responsible for the critical task of finding the right destination based on observable identifiers (method and path) and extracting relevant identifiers (parameters) along the way.

Router Public Interface

The router's public API is designed for developer ergonomics and expressiveness. It allows routes to be registered using intuitive methods that correspond to HTTP verbs and provides a way to group related routes under a common path prefix. The following table details the core methods available on the `Router` instance.

Method	Parameters	Returns	Description
`.use(path?, ...handlers)`	`path` (String, optional): A path prefix. `...handlers` (Function): One or more middleware/handler functions.	`Router` (for chaining)	Registers middleware functions. If a `path` is provided, the middleware only runs for requests whose path starts with that prefix. Without a path, it runs for all requests. This is also used for mounting sub-routers.
`.get(path, ...handlers)`	`path` (String): URL pattern (e.g., `/users`, `/users/:id`). `...handlers` (Function): One or more handler functions.	`Route`	Registers a route handler for HTTP GET requests matching the given `path`.
`.post(path, ...handlers)`	Same as `.get`	`Route`	Registers a route handler for HTTP POST requests.
`.put(path, ...handlers)`	Same as `.get`	`Route`	Registers a route handler for HTTP PUT requests.
`.delete(path, ...handlers)`	Same as `.get`	`Route`	Registers a route handler for HTTP DELETE requests.
`.match(method, path)`	`method` (String): HTTP method. `path` (String): Request URL pathname.	`Object` | `null`	The core internal lookup method. Given a method and path, it returns a match object containing the handler(s) and parsed parameters, or `null` if no route matches. This method powers `router.dispatch`.
`.dispatch(req, res)`	`req` (`Request`), `res` (`Response`)	`Promise<void>`	Executes the chain of handlers (middleware + final route handler) for a matched route. This is the engine that calls `next()` to pass control through the handler stack.

The `Application` class (often named `app`) typically delegates these methods to its internal `Router` instance, providing a top-level API like `app.get()`, `app.post()`, etc.

ADR: Route Matching Data Structure

> **Decision: Use a Prefix Tree (Trie) for Route Storage and Matching**
>
> - **Context:** The router must match incoming request paths against hundreds or thousands of registered route patterns quickly and efficiently. A naive approach (like a linear scan) becomes a performance bottleneck as the number of routes grows. We need a data structure optimized for prefix-based string matching that also handles dynamic path segments (parameters).
> - **Options Considered:**
>   1. **Linear Array Scan:** Store all routes in an array. For each request, iterate through the array, checking each route's pattern against the request path until a match is found.
>   2. **HashMap for Static Routes:** Use a map where the key is `method + path` (e.g., `GET-/api/users`). This provides O(1) lookup for exact matches but cannot handle parameterized routes (`/users/:id`) without separate logic.
>   3. **Prefix Tree (Trie):** Build a tree where each node represents a segment of a path (e.g., `api`, `users`, `:id`). Traverse the tree segment-by-segment to find a match, branching for parameters.
> - **Decision:** Implement a hybrid data structure: a **prefix tree (trie)** where each node can represent a static path segment or a parameter/wildcard. A `TrieNode` will contain maps for static children and special handlers for parameterized segments.
> - **Rationale:**
>   - **Performance:** A trie provides O(L) lookup time, where L is the number of segments in the path, not the number of total routes. This is significantly faster than a linear scan (O(N)) for large route sets.
>   - **Expressiveness:** A trie naturally models the hierarchical nature of URL paths. It elegantly handles both static segments (`/api/users`) and dynamic parameters (`/api/users/:id`) by treating parameters as special child nodes.
>   - **Extensibility:** The trie structure can be extended to support advanced features like optional segments, regex constraints on parameters, or even wildcards (`*`) for catch-all routes by adding new node types.
>   - **Memory Efficiency:** While a trie uses more memory than a simple array, the overhead per node is small. Shared prefixes (like `/api` in `/api/users` and `/api/posts`) are stored only once, offering memory compaction.
> - **Consequences:**
>   - **Increased Implementation Complexity:** Building and traversing a trie is more complex than iterating an array. We must correctly handle segment splitting, parameter capture, and priority rules (static before dynamic).
>   - **Memory Overhead:** Requires storing the tree structure and multiple maps (`children`, `handlers`). However, this is a worthwhile trade-off for the performance gain in a core component.
>   - **Clear Separation of Concerns:** The trie becomes a dedicated internal component, separating the matching logic from route registration and handler execution.

The following table summarizes the trade-offs that led to this decision.

Option	Pros	Cons	Chosen?
Linear Array Scan	Simple to implement; easy to understand; trivial to maintain order.	Performance degrades linearly (O(N)) with route count; inefficient for large applications.	No
HashMap for Static Routes	Lightning-fast O(1) lookup for static routes; very simple.	Cannot handle dynamic path segments; requires a separate, slower fallback system for parameters, negating the performance benefit for mixed routes.	No
Prefix Tree (Trie)	Excellent performance (O(L)); naturally handles both static and dynamic segments; memory efficient for shared prefixes; extensible.	More complex implementation; requires careful design of node structure and traversal logic.	**Yes**

Internal Behavior: The Matching Algorithm

The router's core intelligence resides in the `router.match(method, path)` method. This method traverses the trie to find a matching route handler and extracts any parameters from the path. The algorithm assumes the path has been normalized (e.g., leading slash trimmed, query string removed). The following steps detail the process:

1. **Path Segmentation:** Split the incoming request `path` (e.g., `/api/users/42`) into an array of segments (`['api', 'users', '42']`). The leading slash is typically ignored or trimmed first.
2. **Trie Initialization:** Start at the root `TrieNode` of the router. The root represents the empty path before any segments.
3. **Segment-by-Segment Traversal:** For each segment in the path array: a. **Check Static Children:** Look in the current node's `children` map for a key matching the exact segment (e.g., `'api'`). If found, move to that child node and proceed to the next segment. b. **Check Parameter Child:** If no static match is found, check if the current node has a `paramName` set (indicating a parameter node like `:id`). If it does, this segment matches the parameter pattern. **Capture** the segment's value (e.g., `'42'`) and associate it with the `paramName` (e.g., `id: '42'`). Move to this parameter node's child (there is typically only one child, as a parameter consumes one segment) and proceed. c. **No Match Found:** If neither a static child nor a parameter child matches the current segment, the traversal fails for this branch. Backtracking may be necessary if the router supports wildcards or multiple parameter patterns, but in a basic implementation, this results in no match (404).
4. **Leaf Node Validation:** After consuming all path segments, the algorithm should be at a leaf node (or a node that can act as a leaf). Check the current node's `handlers` map for an entry keyed by the request `method` (e.g., `'GET'`).
5. **Result Assembly:** If a handler is found, the match is successful. The method returns a match object containing:
   - `handlers`: The array of handler functions registered for this route and method.
   - `params`: A dictionary of collected parameter names and values (e.g., `{ id: '42' }`).
   - `path`: The matched route pattern (e.g., `'/api/users/:id'`).
6. **No Match Handling:** If, at any point, traversal cannot proceed (step 3c) or no handler exists for the method at the leaf (step 4), the method returns `null`. The framework will then generate a 404 Not Found response.

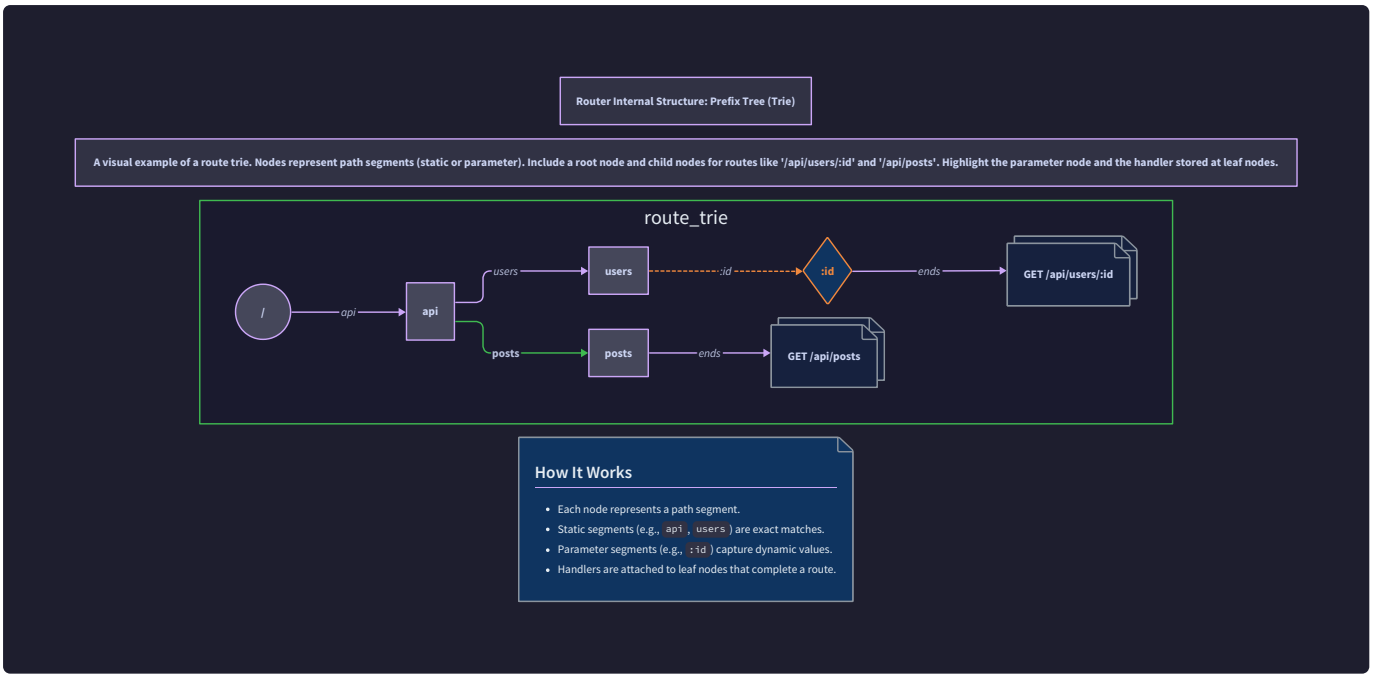


Diagram: A visual representation of a route trie. The root has a static child `api`. The `api` node has two static children: `users` and `posts`. The `users` node has a parameter child `:id`, which stores a handler for GET requests. The path `/api/users/42` traverses `api` → `users` → `:id`, capturing `id=42`.

Common Pitfalls in Routing

Implementing a router involves subtle details that can lead to bugs and inconsistent behavior if not handled correctly.

⚠️ Pitfall 1: Incorrect Route Ordering (Specificity)

- **Description:** Defining routes in the wrong order, such as placing `/users/:id` before `/users/new`. The router will match the first pattern it finds, so a request to `GET /users/new` will be incorrectly captured by the `:id` parameter route (with `id='new'`).
- **Why it's wrong:** It breaks application logic. The `/users/new` endpoint becomes inaccessible.
- **Fix: Always define more specific routes before more general ones.** In a linear scan, this means ordering routes manually. In a trie, ensure that during insertion, static segments are given priority over parameter segments when deciding the traversal path. In practice, this means checking the `children` map for a static match *before* falling back to the parameter node.

⚠️ Pitfall 2: Trailing Slash Sensitivity

- **Description:** Treating `/api/users` and `/api/users/` as different routes. Users might accidentally add or omit the trailing slash, leading to unexpected 404s.
- **Why it's wrong:** It creates a poor developer and user experience. Semantically, the two URLs often refer to the same resource.
- **Fix: Normalize paths before matching.** A common strategy is to strip trailing slashes from both the registered route patterns and the incoming request path (unless the empty path `"/"` is involved). Alternatively, the router can be configured to be strict or lenient.

⚠️ Pitfall 3: Un-decoded URL Parameters

- **Description:** Capturing URL parameters without decoding percent-encoded characters. A request to `/users/hello%20world` would store the parameter as the raw string `'hello%20world'` instead of the decoded value `'hello world'`.
- **Why it's wrong:** It violates the HTTP standard. The `%20` is an encoding for a space within a URL. The application receives incorrect data.
- **Fix: Always decode the captured segment value using `decodeURIComponent`** before storing it in the `params` object. Be cautious of malformed encoding (wrap in a try-catch).

⚠️ **Pitfall 4: Mutating Shared `params` or `query` Objects**

- **Description:** Middleware or route handlers directly modifying the `req.params` or `req.query` object. Since the same request object flows through the entire pipeline, a change in one middleware pollutes the data for subsequent handlers.
- **Why it's wrong:** It creates hidden dependencies and side effects, making the application unpredictable and hard to debug.
- **Fix: Treat `req.params` and `req.query` as immutable within handlers.** If modification is necessary, create a copy first (e.g., `const myParams = { ...req.params };`). The framework itself should construct a new object for each request.

Implementation Guidance

This section provides concrete, actionable steps to implement the router in JavaScript/Node.js, aligning with the design described above.

A. Technology Recommendations

Component	Simple Option (Recommended for Learning)	Advanced Option (For Exploration)
Route Storage	Custom Prefix Tree (`TrieNode`) using JavaScript `Map` objects.	Use an existing library like `path-to-regexp` for pattern compilation and matching, which is what Express.js uses internally.
Path Parsing	Node.js built-in `url` module to parse the request URL and get the `pathname`.	Use the `URL` Web API (available in Node.js) for more standardized parsing.
Segment Splitting	Simple `String.split('/')` and filter out empty strings.	Use a lightweight utility library for more robust path utilities.

B. Recommended File/Module Structure

Place the router logic in its own module to maintain separation of concerns.

```
barebones-framework/
├── package.json
├── src/
│   ├── application.js      # Main Application class
│   ├── router.js           # Router class and TrieNode (focus of this milestone)
│   ├── request.js          # Enhanced Request class (Milestone 3)
│   ├── response.js         # Enhanced Response class (Milestone 3)
│   ├── middleware.js       # Built-in middleware (Milestone 2 & 3)
│   └── template.js         # Template engine (Milestone 4)
└── test/
    └── router.test.js      # Unit tests for the router
```

C. Infrastructure Starter Code

The following is a complete, ready-to-use implementation of the `TrieNode` class, which forms the backbone of the route matching data structure. This is a prerequisite component that learners can directly copy.

JAVASCRIPT

```javascript
// src/router.js - TrieNode Class (Starter Code)

class TrieNode {

  /**

   * Represents a node in the route prefix tree (trie).

   */

  constructor() {

    // Map from static path segments to child TrieNodes.

    this.children = new Map();

    // Map from HTTP method (e.g., 'GET') to an array of handler functions.

    this.handlers = new Map();

    // If this node represents a parameter (e.g., ':id'), this holds the parameter name.

    this.paramName = null;

    // The child node for the segment following a parameter. Only one is allowed per node.

    this.paramChild = null;

    // Flag indicating if this node can be the end of a path (has handlers).

    this.isLeaf = false;

  }


  /**

   * Inserts a path pattern into the trie, starting from this node.

   * @param {string[]} segments - Array of path segments (e.g., ['api', 'users', ':id']).

   * @param {number} index - Current index in the segments array.

   * @param {string} method - HTTP method.

   * @param {Function[]} handlerFns - Array of handler functions for this route.

   */

  insert(segments, index, method, handlerFns) {

    if (index === segments.length) {

      // Reached the end of the path pattern. Store handlers for this method.

      this.handlers.set(method, handlerFns);

      this.isLeaf = true;
```

```
      return;
    }

    const segment = segments[index];
    const isParam = segment.startsWith(':');

    if (isParam) {
      // This segment is a parameter (e.g., ':id')
      const paramName = segment.slice(1); // Remove the ':'
      if (!this.paramName) {
        // First time encountering a parameter at this position.
        this.paramName = paramName;
        this.paramChild = new TrieNode();
      } else if (this.paramName !== paramName) {
        // Conflict: Different parameter name at the same position (e.g., ':id' vs ':userId').
        throw new Error(`Cannot add parameter '${paramName}' because '${this.paramName}' already
exists at this path level.`);
      }
      // Recurse into the parameter child.
      this.paramChild.insert(segments, index + 1, method, handlerFns);
    } else {
      // This segment is static.
      let child = this.children.get(segment);
      if (!child) {
        child = new TrieNode();
        this.children.set(segment, child);
      }
      // Recurse into the static child.
      child.insert(segments, index + 1, method, handlerFns);
    }
  }
```

```
/**
 * Searches for a matching route in the trie.
 * @param {string[]} segments - Array of path segments from the request.
 * @param {number} index - Current index in the segments array.
 * @param {Object} params - Accumulator for captured parameters.
 * @returns {{ handlers: Array<Function>, params: Object } | null}
 */
search(segments, index, params) {
  if (index === segments.length) {
    // Reached the end of the request path.
    if (this.isLeaf) {
      // This node is a valid endpoint. Return handlers and params.
      // Note: The caller must check for the specific HTTP method.
      return { node: this, params };
    }
    return null; // Path matches but no handlers registered here.
  }

  const segment = segments[index];

  // 1. Try static child first (for priority).
  const staticChild = this.children.get(segment);
  if (staticChild) {
    const result = staticChild.search(segments, index + 1, params);
    if (result) return result;
  }

  // 2. Try parameter child.
  if (this.paramChild) {
    // Clone params to avoid mutating the object for other branches.
    const newParams = { ...params, [this.paramName]: decodeURIComponent(segment) };
```

```
      const result = this.paramChild.search(segments, index + 1, newParams);
      if (result) return result;
    }

    // No match found.
    return null;
  }
}
```

D. Core Logic Skeleton Code

Below is the skeleton for the main `Router` class. Learners should fill in the `TODO` comments, which map directly to the algorithm steps and concepts discussed.

JAVASCRIPT

```javascript
// src/router.js - Router Class (Skeleton)

const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'HEAD', 'OPTIONS'];


class Router {

  constructor() {

    // The root node of the prefix tree.

    this.trie = new TrieNode();

    // Array of global middleware functions.

    this.middleware = [];

  }


  /**

   * Registers a route for a specific HTTP method.

   * @param {string} method - HTTP method (e.g., 'GET').

   * @param {string} path - Route pattern (e.g., '/api/users/:id').

   * @param {...Function} handlers - Handler functions.

   * @returns {Route} The created route object (for potential chaining).

   */

  register(method, path, ...handlers) {

    // TODO 1: Normalize the path.

    //   - Ensure it starts with a '/'? (Or handle relative paths).

    //   - Consider stripping trailing slashes for consistency.


    // TODO 2: Split the path into segments.

    //   - Use `path.split('/')` and filter out empty strings.

    //   - Example: '/api/users/:id' -> ['api', 'users', ':id']


    // TODO 3: Validate that handlers are functions.


    // TODO 4: Insert the route into the trie.

    //   - Call `this.trie.insert(segments, 0, method.toUpperCase(), handlers)`.


    // TODO 5: Create and return a minimal Route object (could just be a descriptor).
```

```
    //   - At minimum, store { path, method, handlers }.
    console.log(`Registered ${method} ${path}`);
    return { path, method, handlers };
  }

  // Convenience methods for HTTP verbs.
  get(path, ...handlers) {
    return this.register('GET', path, ...handlers);
  }
  post(path, ...handlers) {
    return this.register('POST', path, ...handlers);
  }
  put(path, ...handlers) {
    return this.register('PUT', path, ...handlers);
  }
  delete(path, ...handlers) {
    return this.register('DELETE', path, ...handlers);
  }

  /**
   * Adds middleware that runs for all routes or routes under a prefix.
   * @param {string|Function} pathOrFn - Path prefix or middleware function.
   * @param {...Function} fns - Middleware functions (if path is provided).
   */
  use(pathOrFn, ...fns) {
    // TODO 1: Determine if the first argument is a path (string) or a function.
    //   - If it's a function, it's global middleware. Add all arguments to `this.middleware`.
    //   - If it's a string, subsequent arguments are middleware that should only run for routes with
that prefix.
    //   - For now, implement global middleware only. Route-specific middleware is an advanced
extension.
    if (typeof pathOrFn === 'function') {
```

```
      this.middleware.push(pathOrFn);
    } else {
      // Advanced: Handle path-prefixed middleware.
      console.warn('Path-prefixed middleware not yet implemented. Adding as global.');
      [pathOrFn, ...fns].forEach(fn => {
        if (typeof fn === 'function') this.middleware.push(fn);
      });
    }
    return this; // Allow chaining.
  }

  /**
   * Finds a route matching the given method and path.
   * @param {string} method - HTTP method.
   * @param {string} path - Request URL pathname.
   * @returns {Object|null} Match object with { handlers, params, route } or null.
   */
  match(method, path) {
    // TODO 1: Normalize and split the incoming request path.
    //   - Use the `url` module if needed to get just the pathname.
    //   - Split into segments (same logic as in `register`).

    // TODO 2: Search the trie.
    //   - Call `this.trie.search(segments, 0, {})`.
    //   - The result contains a `node` and `params` if found.

    // TODO 3: If a node is found, retrieve the handlers for the specific HTTP method.
    //   - Check `node.handlers.get(method.toUpperCase())`.

    // TODO 4: If handlers exist, return the match object.
    //   - Structure: { handlers: [...], params: {...}, path: matchedPattern? }
```

```
    //   - If no handlers for the method, return null (405 Method Not Allowed could be handled
later).

    // TODO 5: If no node is found, return null.
    return null; // Placeholder
  }

  /**
   * Dispatches the request to the matched route handlers, including middleware.
   * @param {Request} req - Enhanced request object.
   * @param {Response} res - Enhanced response object.
   * @returns {Promise<void>}
   */
  async dispatch(req, res) {
    // TODO 1: Use `this.match` to find the route matching `req.method` and `req.url`.
    //   - Store the result in `match`.

    // TODO 2: If no match, set res.statusCode = 404 and call res.end().
    //   - Alternatively, trigger a 404 middleware if available.

    // TODO 3: If a match is found, attach `match.params` to `req.params`.

    // TODO 4: Combine `this.middleware` and `match.handlers` into a single array of functions to
execute.

    // TODO 5: Implement a simple chain execution mechanism (detailed in Milestone 2).
    //   - Create an index variable `let idx = 0`.
    //   - Define a `next` function that calls the next handler in the chain.
    //   - Start the chain by calling `next()`.
    console.log('Dispatch not yet implemented');
  }
}

module.exports = Router;
```

E. Language-Specific Hints

- **`Map` vs `Object`:** Use `Map` for `children` and `handlers` as it preserves key order (important for traversal) and allows any string as a key without conflict with prototype properties.
- **`decodeURIComponent` Safety:** Always wrap `decodeURIComponent(segment)` in a try-catch block, as malformed percent-encoding (like `%XX` where `XX` is not hex) will throw an error. In a catch block, you can either reject the request (400 Bad Request) or use the raw segment.
- **Path Normalization:** Use `path = path.replace(/\/$/, '')` to remove a trailing slash, but be careful with the root path `"/"` which would become an empty string.

F. Milestone Checkpoint

After implementing the core routing logic (`register` , `match` , and the trie), verify functionality with the following test.

Command to Run:

```bash
node -e "

const Router = require('./src/router');

const router = new Router();


router.get('/users', (req, res) => console.log('GET /users'));

router.get('/users/:id', (req, res) => console.log('GET /users/:id'));

router.post('/users', (req, res) => console.log('POST /users'));


const match1 = router.match('GET', '/users');

console.log('Match 1 (GET /users):', match1?.handlers?.length > 0 ? 'FOUND' : 'NOT FOUND');


const match2 = router.match('GET', '/users/123');

console.log('Match 2 (GET /users/123):', match2 ? 'FOUND, params=' + JSON.stringify(match2.params) :
'NOT FOUND');


const match3 = router.match('POST', '/users');

console.log('Match 3 (POST /users):', match3?.handlers?.length > 0 ? 'FOUND' : 'NOT FOUND');


const match4 = router.match('GET', '/not-found');

console.log('Match 4 (GET /not-found):', match4 ? 'FOUND' : 'NOT FOUND');

"
```

Expected Output:

```
Registered GET /users
Registered GET /users/:id
Registered POST /users
Match 1 (GET /users): FOUND
Match 2 (GET /users/123): FOUND, params={"id":"123"}
Match 3 (POST /users): FOUND
Match 4 (GET /not-found): NOT FOUND
```

Signs of Trouble:

- **`GET /users/123` matches `GET /users`:** Your trie traversal is not consuming all segments. Ensure the search only succeeds at a leaf node (`isLeaf` is true).
- **`GET /users/123` params are `{ ':id': '123' }`:** You forgot to strip the `:` from the parameter name when storing it in `paramName`.
- **All matches return `NOT FOUND`:** Check your path splitting logic. Ensure you are splitting the registered path and the incoming request path identically.

Component Design: Middleware Engine

> **Milestone(s):** Milestone 2: Middleware

The middleware engine is the **central nervous system** of the web framework, transforming raw HTTP requests into meaningful application responses through a series of discrete processing steps. Unlike the router's destination-focused matching, middleware concerns itself with the journey—the transformation and inspection that happens along the way. This component embodies the **Chain of Responsibility** design pattern, where each handler in the chain can process, modify, or short-circuit the request/response lifecycle.

Mental Model: The Processing Conveyor Belt

Imagine an assembly line in a factory where a product (the HTTP request) moves along a conveyor belt. At each station along the belt, a specialized machine (middleware) performs a specific operation on the product:

1. **Inspection Station (Logging)**: Examines the product's label (request method, URL) and records when it arrived and left
2. **Unpacking Station (Body Parsing)**: Opens the packaging (request body) and extracts the contents in a structured format
3. **Security Scanner (Authentication)**: Checks the product's credentials before allowing it to proceed further
4. **Quality Control (Validation)**: Verifies the product meets specific criteria before reaching final assembly
5. **Final Assembly (Route Handler)**: Where the actual product transformation occurs based on its intended destination

Each station has a choice: it can process the product and pass it to the next station (calling `next()`), process it and send it back out of the factory (sending a response without calling `next()`), or reject it entirely (throwing an error). The conveyor belt itself is the middleware engine—it manages the sequence of stations and ensures each product flows through them in the correct order.

> **Key Insight**: Middleware is fundamentally about **composition**. By breaking request processing into small, focused functions, developers can mix and match behaviors like building blocks, creating complex processing pipelines from simple, reusable components.

Middleware Engine Interface

The middleware engine's public interface consists of methods for registering middleware and the signature that middleware functions must follow. These interfaces are primarily exposed through the `Application` and `Router` objects.

Middleware Function Signature

All middleware functions share a consistent signature that allows them to participate in the chain:

Parameter	Type	Description
`req`	`Request`	The enhanced request object containing all request data
`res`	`Response`	The enhanced response object for sending responses
`next`	`Function`	A callback function to pass control to the next middleware

A middleware function can perform one of three actions:

1. **Process and Continue**: Perform some operation (like logging) and call `next()` to continue the chain
2. **Terminate Early**: Send a response using `res.send()`, `res.json()`, etc., without calling `next()`
3. **Signal Error**: Call `next(error)` with an error object to jump to error-handling middleware

Registration Methods

The framework provides multiple ways to register middleware, each serving different use cases:

Method	Scope	Parameters	Description
`app.use(fn)`	Global	`fn: Function`	Registers middleware that runs for **every** request to the application
`app.use(path, fn)`	Path-scoped	`path: String`, `fn: Function`	Registers middleware that runs only for requests matching the path prefix
`router.use(fn)`	Router-level	`fn: Function`	Registers middleware that runs for all routes within a specific router
Route-specific handlers	Route-level	`...handlers: Array<Function>`	Middleware registered directly with a route (e.g., `app.get('/path', mw1, mw2, handler)`)

The `Layer` type serves as a uniform wrapper for both middleware and route handlers, allowing them to be stored and executed consistently within the same chain:

Field	Type	Description
`fn`	`Function`	The middleware or route handler function to execute
`path`	`String`	The path pattern this layer is registered for (empty for global middleware)
`regexp`	`RegExp`	Compiled regular expression for matching the path
`keys`	`Array<Object>`	Array of parameter keys for parameterized paths (like `:id`)

ADR: Middleware Composition Strategy

> **Decision: Array Storage with Iterative Dispatch**
>
> **Context**: We need a strategy to execute an ordered sequence of middleware functions where each function has control over whether to continue to the next. The solution must handle synchronous and asynchronous functions, support early termination, propagate errors, and allow middleware to be added at different scopes (global, router, route).
>
> **Options Considered**:
>
> 1. **Functional Composition (Express-style)**: Store middleware in an array, then recursively compose them into a single function using `reduceRight()` that nests each middleware inside the next
> 2. **Iterative Loop with Index (Koa-style)**: Store middleware in an array and use an index variable that increments each time `next()` is called
> 3. **Promise Chain**: Convert all middleware to promises and chain them together with `.then()` calls
>
> **Decision**: We chose **Option 2: Iterative Loop with Index**. This approach stores middleware in an array and uses a simple `next()` function that increments an index to advance through the chain.
>
> **Rationale**:
>
> - **Simplicity**: The iterative approach is more straightforward to understand and debug than deeply nested function composition
> - **Explicit Control Flow**: The `next` callback is explicitly passed to each middleware, making the flow of control visible
> - **Error Handling**: Centralized try-catch blocks can wrap each middleware execution without complex promise chains
> - **Performance**: Avoids creating deep call stacks from recursive composition, which could hit stack limits with many middleware
> - **Flexibility**: Easily supports route-specific middleware by splicing different middleware arrays together at dispatch time
>
> **Consequences**:
>
> - **Positive**: Clear, debuggable execution flow; easy to add debugging instrumentation; straightforward error propagation
> - **Negative**: Requires careful management of the index variable and closure scope; synchronous by default (async requires wrapper)

Option	Pros	Cons	Chosen?
Functional Composition	Elegant functional style; automatic async support via promises	Deep call stacks; harder to debug; complex error handling	No
Iterative Loop with Index	Simple control flow; easy debugging; explicit error handling	Manual async handling needed; index management complexity	**Yes**
Promise Chain	Natural async support; clean error propagation with `.catch()`	All middleware become promises; less control over execution timing	No

Internal Behavior: The Dispatch Loop

The core of the middleware engine is the dispatch algorithm that executes middleware in sequence. This algorithm is implemented in the `router.dispatch()` method, which coordinates the execution of both middleware and the final route handler.

Middleware Chain Preparation

Before execution begins, the framework must assemble the complete middleware chain for a specific request:

1. **Collect Global Middleware**: Retrieve all middleware registered via `app.use()`
2. **Collect Router Middleware**: If using nested routers, add middleware registered via `router.use()`
3. **Collect Route-Specific Middleware**: Add middleware registered directly with the matched route
4. **Add Route Handler**: Append the final route handler function to the end of the chain
5. **Add Error Handlers**: Special error-handling middleware (functions with 4 parameters) are kept in a separate chain

This creates a single, flat array of `Layer` objects that will be executed in sequence.

The Dispatch Algorithm

The `router.dispatch()` method implements the following algorithm:

1. **Initialize Execution State**:
   - Create an index variable `idx = 0` to track position in middleware chain
   - Create a `next` function that when called increments `idx` and executes the next layer
   - Set up error flag and error argument storage
2. **Define the Next Function**:

```javascript
function next(err) {
  // If error provided, skip to error handlers
  // Otherwise increment idx and execute next layer
}
```

3. **Iterative Execution Loop**:

```
1. WHILE idx < layerStack.length:
   a. Get currentLayer = layerStack[idx]
   b. IF currentLayer.path DOES NOT match request path:
        idx = idx + 1
        CONTINUE to next iteration
   c. TRY:
        currentLayer.handle(req, res, next)
      CATCH error:
        next(error)  // Pass error to error-handling middleware
   d. IF response already sent (res.finished === true):
        BREAK from loop
2. IF loop completes without sending response AND no error:
     // This should not happen in normal cases
     IF idx === layerStack.length AND NOT res.finished:
        Call default 404 handler
```

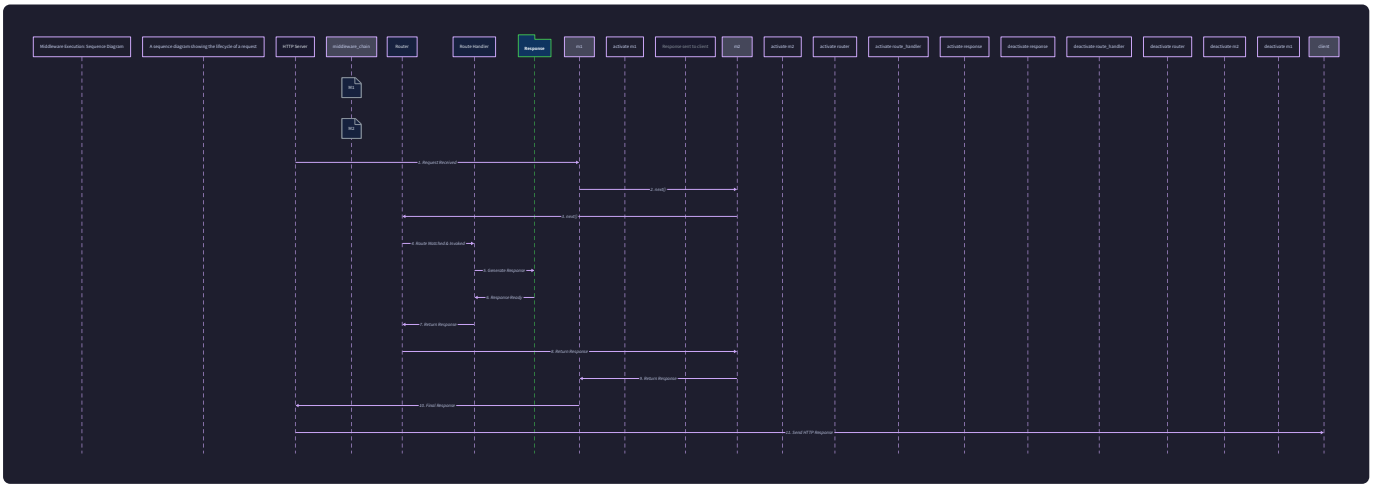
4. **Error Handling Flow**:

```
IF next() is called WITH error argument (next(err)):
  1. SET error mode = true
  2. SKIP all regular middleware (those with 3 parameters)
  3. FIND next error-handling middleware (function with 4 parameters)
  4. IF found:
        CALL error middleware with (err, req, res, next)
      ELSE:
        // No error handler found
        SET res.statusCode = 500
        SEND error message or generic error response
```

5. **Async/Await Integration**:

```
FOR async middleware (returns Promise or uses async/await):
  1. WRAP middleware execution in Promise.resolve()
  2. IF Promise rejects:
        CALL next() with rejection reason as error
  3. AWAIT the Promise before continuing loop iteration
```

The sequence diagram below illustrates this flow:



Layer Matching and Execution

Each `Layer` in the chain performs its own path matching before execution:

1. **Path Matching**: If the layer has a `path` property (not global middleware), it checks whether the request path matches using the compiled `regexp`
2. **Parameter Extraction**: For parameterized paths (like `/users/:id`), the layer extracts parameter values and adds them to `req.params`
3. **Execution**: The layer's `handle()` method calls the stored middleware function with `(req, res, next)`
4. **Completion Detection**: After execution, the layer checks if `res.finished` is true to determine if the response was sent

Common Pitfalls in Middleware

Implementing middleware correctly requires careful attention to several subtle behaviors that can cause difficult-to-debug issues.

⚠️ Pitfall: Forgetting to Call `next()`

Description: A middleware function processes the request but doesn't call `next()`, causing the request to hang indefinitely.

Why it's Wrong: The dispatch loop waits indefinitely for the middleware to signal completion. The client connection eventually times out, and no response is sent.

How to Fix:

- Always call `next()` unless you're intentionally terminating the request by sending a response
- Use pattern: process request → (optional: send response) → call `next()` if not terminating
- Add timeout middleware that catches hanging requests

⚠️ Pitfall: Calling `next()` Multiple Times

Description: A middleware calls `next()` more than once, either in conditional branches or after asynchronous operations complete.

Why it's Wrong: This causes the same middleware to be executed multiple times or the chain to advance incorrectly, potentially leading to duplicate processing, headers being set multiple times, or errors.

How to Fix:

- Ensure `next()` is called exactly once per middleware invocation
- Use a flag to track whether `next()` has been called
- In async middleware, be careful with callbacks that might fire multiple times

⚠️ Pitfall: Improper Async Error Handling

Description: An async middleware throws an error or rejects a promise, but the error isn't caught and passed to `next(error)`.

Why it's Wrong: Uncaught async errors crash the Node.js process or leave the request hanging without a proper error response.

How to Fix:

- Wrap async middleware in try-catch blocks
- Convert promises and use `.catch(next)` pattern:

```javascript
asyncMiddleware(req, res, next) {
  someAsyncOperation()
    .then(result => {
      // process result
      next();
    })
    .catch(next);  // Errors automatically passed to error handlers
}
```

⚠️ Pitfall: Modifying Request/Response After Headers Sent

Description: Middleware attempts to modify the response (set headers, status codes, body) after the response has already been sent to the client.

Why it's Wrong: Once `res.end()` is called, the HTTP response is finalized. Further modifications cause errors like "Cannot set headers after they are sent to the client."

How to Fix:

- Check `res.finished` or `res.headersSent` before modifying response
- Structure middleware so that response-sending happens at the end of the chain
- Use middleware ordering carefully: response-modifying middleware should come before response-sending middleware

⚠️ Pitfall: Incorrect Middleware Ordering

Description: Middleware registered in the wrong order causes unexpected behavior (e.g., body parser after route handler, authentication after response sent).

Why it's Wrong: Middleware execution order is critical—later middleware depends on the work of earlier middleware. Wrong ordering leads to missing data, security issues, or incorrect processing.

How to Fix:

- Follow conventional ordering: security → logging → body parsing → authentication → validation → route handlers → error handlers
- Document middleware dependencies clearly
- Consider adding warnings for common ordering mistakes

Implementation Guidance for Middleware

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Middleware Storage	Array of functions	Linked list with metadata
Async Handling	Callbacks with try-catch	Async function wrapper with Promise resolution
Error Propagation	Error argument to `next()`	Custom error types with status codes
Path Matching	String prefix matching	Regular expression with parameter extraction

B. Recommended File/Module Structure

```
barebones-framework/
├── src/
│   ├── application.js          # Main Application class (exports createApp)
│   ├── router.js               # Router class with dispatch logic
│   ├── middleware/             # Middleware-related modules
│   │   ├── index.js            # Built-in middleware exports
│   │   ├── init.js             # Request/response enhancement middleware
│   │   ├── logger.js           # Logging middleware
│   │   └── error-handler.js    # Error handling middleware
│   ├── layer.js                # Layer class for wrapping middleware/handlers
│   ├── request.js              # Enhanced Request class
│   └── response.js             # Enhanced Response class
├── examples/
│   └── middleware-demo.js      # Demo of middleware usage
└── package.json
```

C. Infrastructure Starter Code

Complete Layer Class (layer.js):

JAVASCRIPT

```javascript
class Layer {
  constructor(fn, path, options = {}) {
    this.fn = fn;
    this.path = path || '';
    this.regexp = this.compilePath(this.path);
    this.keys = [];
    this.isErrorHandler = fn.length === 4; // (err, req, res, next)
  }

  compilePath(path) {
    // Convert path pattern like '/users/:id' to regular expression
    if (!path || path === '*') {
      return /^.*$/;
    }

    const segments = path.split('/');
    const regexParts = segments.map(segment => {
      if (segment.startsWith(':')) {
        const paramName = segment.slice(1);
        this.keys.push({ name: paramName });
        return '([^\\/]+)';
      }
      return segment;
    });

    return new RegExp(`^${regexParts.join('\\/')}$`);
  }

  match(path) {
    const match = this.regexp.exec(path);
    if (!match) {
```

```
      return false;
    }

    // Extract parameter values
    this.params = {};
    for (let i = 0; i < this.keys.length; i++) {
      this.params[this.keys[i].name] = decodeURIComponent(match[i + 1]);
    }

    return true;
  }

  handle(req, res, next) {
    // Add extracted parameters to request
    if (this.params) {
      req.params = { ...req.params, ...this.params };
    }

    // Call the middleware function
    try {
      if (this.isErrorHandler) {
        // Error handlers have signature (err, req, res, next)
        this.fn(req.error, req, res, next);
      } else {
        // Regular middleware has signature (req, res, next)
        this.fn(req, res, next);
      }
    } catch (err) {
      next(err);
    }
```

```
  }
}

module.exports = Layer;
```

D. Core Logic Skeleton Code

Router Dispatch Method (router.js):

JAVASCRIPT

```javascript
class Router {
  constructor() {
    this.middleware = []; // Global router middleware
    this.routes = new Map(); // HTTP method → route trie
  }

  async dispatch(req, res) {
    // TODO 1: Find matching route using router.match(method, path)
    //   - Store matched route and extracted parameters

    // TODO 2: Assemble complete middleware chain:
    //   a. Start with global app middleware (if available from req.app)
    //   b. Add router-level middleware (this.middleware)
    //   c. Add route-specific middleware (from matched route)
    //   d. Add final route handler

    // TODO 3: Create execution state:
    //   - Initialize idx = 0
    //   - Set up next function that increments idx and calls next layer
    //   - Handle error argument in next(err)

    // TODO 4: Implement iterative execution loop:
    //   WHILE idx < layerStack.length:
    //     - Get current layer
    //     - Check if layer path matches request path
    //     - Execute layer.handle(req, res, next)
    //     - If res.finished, break from loop

    // TODO 5: Implement error handling flow:
    //   IF next() called with error:
```

```
    //     - Skip to next error-handling middleware (fn.length === 4)
    //     - If no error handler found, send 500 error response


    // TODO 6: Handle case where no middleware sends response:
    //   IF !res.finished after loop completes:
    //     - Send 404 Not Found response
  }


  use(fn) {
    // TODO: Create Layer from middleware function
    //   - For path-scoped: app.use('/api', fn) → Layer with path='/api'
    //   - For global: app.use(fn) → Layer with empty path
    //   - Add to this.middleware array
  }
}

module.exports = Router;
```

Built-in Logging Middleware (middleware/logger.js):

```javascript
function createLogger(format = 'combined') {
  return function logger(req, res, next) {
    const start = Date.now();

    // Store original end method to calculate duration
    const originalEnd = res.end;

    res.end = function(...args) {
      const duration = Date.now() - start;
      const logLine = `${req.method} ${req.url} ${res.statusCode} ${duration}ms`;

      // Output based on format
      if (format === 'combined') {
        console.log(`${new Date().toISOString()} - ${logLine}`);
      } else {
        console.log(logLine);
      }

      // Call original end method
      originalEnd.apply(res, args);
    };

    next();
  };
}

module.exports = createLogger;
```

E. Language-Specific Hints (JavaScript/Node.js)

1. **Async/Await Wrapper**: Create a utility to convert callback-style middleware to async/await:

```javascript
function asyncHandler(fn) {
  return (req, res, next) => {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
}
```

2. **Error Handling Middleware**: Remember error handlers have 4 parameters:

```javascript
function errorHandler(err, req, res, next) {
  res.status(err.status || 500);
  res.json({ error: err.message });
}
```

3. **Response Finished Check**: Use Node.js built-in properties:

```javascript
if (res.headersSent || res.finished) {
  // Don't modify response further
}
```

4. **Stream Handling**: For middleware that processes request bodies, use streams to handle large payloads efficiently.

F. Milestone Checkpoint

After implementing the middleware engine, verify functionality with these tests:

Test Command:

```bash
node test-middleware.js
```

Expected Output:

```
Server running on port 3000
Test 1 - Logging Middleware: PASS
Test 2 - Middleware Ordering: PASS
Test 3 - Error Handling: PASS
Test 4 - Async Middleware: PASS
```

Manual Verification:

1. Start the server and visit `http://localhost:3000/test`
2. Check console for log output: `GET /test 200 15ms`
3. Test error handling: Visit `http://localhost:3000/error` should return JSON error response
4. Test middleware chain: Request should pass through multiple middleware before reaching handler

Debugging Signs:

- **Request hangs indefinitely**: Likely middleware forgetting to call `next()`
- **"Cannot set headers after they are sent"**: Middleware modifying response after it's sent
- **Middleware executing multiple times**: `next()` called more than once
- **Error not caught**: Async error not wrapped in try-catch or `.catch()`

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Request hangs, never responds	Middleware didn't call `next()`	Add logging to each middleware; check which one executes last	Ensure every middleware calls `next()` unless sending response
"Cannot set headers after they are sent"	Response modified after `res.end()`	Check middleware order; add check for `res.headersSent`	Reorder middleware or add `if (!res.headersSent)` guard
Middleware executes twice	`next()` called multiple times	Add counter to middleware to track calls	Ensure `next()` called exactly once per execution
Error returns HTML instead of JSON	No error-handling middleware registered	Check middleware chain for function with 4 parameters	Add error-handling middleware at end of chain
Async operations not completing	Promise not awaited or `.catch()` missing	Wrap async operations in try-catch or add `.catch(next)`	Use `asyncHandler` wrapper or explicit error handling

---

Component Design: Request & Response Enhancements

> **Milestone(s):** Milestone 3: Request/Response Enhancement

This component transforms the raw, low-level HTTP objects provided by Node.js into powerful, feature-rich tools for web development. While Node's `http.IncomingMessage` and `http.ServerResponse` provide the fundamental plumbing, they leave common tasks—parsing request bodies, serializing JSON responses, managing cookies—entirely to the developer. This component builds a **Swiss Army Knife** layer atop these primitives, extending them with the convenience methods and parsed data structures that define a modern web framework's developer experience.

Mental Model: The Swiss Army Knife

Imagine you're handed a basic pocket knife—a single blade that can technically cut anything, but requires significant effort and skill for specialized tasks. This is Node.js's native HTTP module. Now imagine attaching modular tools to that knife: a screwdriver, a bottle opener, a small pair of scissors. Each tool is designed for a specific, frequent task, making the knife exponentially more useful without changing its core cutting function. This is the essence of **enhanced request and response objects**.

In our framework, the native `req` (`IncomingMessage`) and `res` (`ServerResponse`) are wrapped by our own `Request` and `Response` classes. These wrappers preserve all original functionality while adding new "tools" as

properties and methods. The `req.body` property automatically parses incoming JSON, the `res.json()` method correctly serializes and sends JSON responses, and `req.cookies` provides easy access to parsed cookie data. This design follows the **Decorator Pattern**, enhancing objects without modifying their original structure, allowing developers to interact with a richer API while maintaining escape hatches to the underlying Node.js objects when needed.

Enhanced Request/Response API

The enhanced `Request` and `Response` classes expose the following additional properties and methods. Note that they also retain all properties and methods of the native Node.js objects they wrap (e.g., `req.url`, `res.writeHead`).

Enhanced Request Properties

Property	Type	Description
`req.app`	`Application`	Reference to the main application instance, allowing access to global settings and components.
`req.body`	`Any` (`Object`, `String`, `Buffer`)	The parsed body of the request. Populated by body-parsing middleware (e.g., for `application/json` or `application/x-www-form-urlencoded`). Defaults to `undefined` if no parser has run.
`req.query`	`Object`	Parsed key-value pairs from the URL's query string (the part after `?`). For `/search?q=term&page=2`, `req.query` would be `{ q: 'term', page: '2' }`.
`req.params`	`Object`	Route parameters extracted from the URL path. For a route `/users/:id` matching `/users/123`, `req.params` would be `{ id: '123' }`. Set by the router during matching.
`req.cookies`	`Object`	Parsed cookies from the `Cookie` request header. For a header `Cookie: session=abc123; user=john`, `req.cookies` would be `{ session: 'abc123', user: 'john' }`.
`req.originalUrl`	`String`	The original request URL string, preserved before any internal rewriting. Useful in middleware that may modify `req.url`.

Enhanced Response Properties

Property	Type	Description
`res.app`	`Application`	Reference to the main application instance.
`res.statusCode`	`Number`	The HTTP status code for the response. Can be set directly or via `res.status()`. Defaults to `200`.
`res.locals`	`Object`	An object for storing request-response cycle local variables, scoped to the request. Useful for passing data from middleware to templates.

Enhanced Response Methods

Method Signature	Returns	Description
`res.status(code)`	`Response` (this)	Sets the HTTP status code for the response. Chainable.
`res.setHeader(name, value)`	`Response` (this)	Sets a single header value. Chainable. Overrides existing header.
`res.json(data)`	`void`	Sends a JSON response. Sets `Content-Type: application/json` header, serializes the provided JavaScript object to JSON string, and sends it as the body. Also sets `res.statusCode` if not already set.
`res.send(body)`	`void`	Sends an HTTP response. The `body` can be a `String` (sets `Content-Type: text/html`), `Buffer` (sets `Content-Type: application/octet-stream`), `Object` (serialized as JSON, same as `res.json`), or `Array`. Infers `Content-Type` and `Content-Length` where possible.
`res.redirect([status,] url)`	`void`	Redirects the request to the specified URL. Sets the `Location` header and default status `302` (or provided status).
`res.cookie(name, value [, options])`	`Response` (this)	Sets a cookie `name` to `value`. The `options` object can include `maxAge`, `expires`, `path`, `domain`, `secure`, `httpOnly`, `sameSite`. Chainable.
`res.clearCookie(name [, options])`	`Response` (this)	Clears the cookie named `name` by setting an expired value. Must match the same `path` and `domain` options used when setting the cookie. Chainable.
`res.type(type)`	`Response` (this)	Sets the `Content-Type` header based on a MIME type or file extension (e.g., `'html'`, `'json'`). Chainable.
`res.sendFile(path [, options] [, callback])`	`Promise`	Sends a file as an octet stream. Sets appropriate `Content-Type` based on file extension and handles partial content (Range requests). Advanced feature, often implemented later.

ADR: Request Body Parsing Strategy

> **Decision: Streaming Parsing with Configurable Limits**
>
> - **Context**: HTTP request bodies can be large (file uploads, JSON payloads) and malformed. We need to parse `application/json` and `application/x-www-form-urlencoded` bodies efficiently while protecting the server from memory exhaustion and Denial-of-Service (DoS) attacks.
> - **Options Considered**:
>   1. **Buffering Entire Body**: Read the entire body stream into memory, then parse it as a complete string/buffer.
>   2. **Streaming Parsing with Limits**: Use streaming parsers (like `JSON.parse` on a stream) with configurable maximum size limits, aborting if exceeded.
> - **Decision**: Implement **streaming parsing with configurable limits** as the default strategy for built-in body parsers.
> - **Rationale**:
>   - **Memory Efficiency**: Buffering entire multi-gigabyte file uploads in memory is infeasible. Streaming parsers process data in chunks, keeping memory footprint low.
>   - **DoS Protection**: A size limit prevents malicious clients from sending infinitely large bodies to exhaust server memory. The limit is configurable per route or globally.
>   - **Practicality**: For JSON and urlencoded data (common in APIs), the bodies are typically small. A streaming JSON parser (like `JSON.parse` on the concatenated string) is simple and effective when combined with a reasonable size limit (e.g., 1MB). For multipart/form-data (file uploads), a streaming approach is essential.
> - **Consequences**:
>   - **Positive**: Framework can handle large file uploads without crashing. Built-in protection against body size DoS attacks.
>   - **Negative**: More complex implementation than simple buffering. Requires careful handling of partial parse failures.
>   - **Mitigation**: Provide sensible defaults (e.g., 1MB limit for JSON) and clear error messages when limits are exceeded.

Option	Pros	Cons	Chosen?
Buffering Entire Body	Simpler implementation. Easy to parse after complete receipt.	Memory explosion risk with large bodies. Blocks processing until entire body received.	No
Streaming Parsing with Limits	Memory efficient. Can abort early on size violation. Scales to large payloads.	More complex. May require external libraries for true streaming JSON parsing.	**Yes**

Internal Behavior: Body Parsing and Cookie Processing

Body Parsing Algorithm (for JSON and URL-encoded)

The body parsing functionality is implemented as **middleware factory** that returns a middleware function configured with options (like size limit). This middleware augments the `req` object with a `body` property.

1. **Middleware Registration**: The developer calls `app.use(express.json())` or similar. Our framework will provide `app.use(bodyParser.json())` or built-in `app.use(Application.json())`.
2. **Request Detection**: When a request with a supported `Content-Type` (e.g., `application/json`) arrives, the body-parsing middleware activates.
3. **Size Limit Check**: The middleware checks the `Content-Length` header (if present) against the configured limit. If exceeded, it immediately rejects with a `413 Payload Too Large` error.
4. **Stream Collection**: The middleware listens to the `req` object's `'data'` events, collecting chunks into a buffer. It also tracks the total bytes received. If at any point the in-memory buffer exceeds the limit, parsing aborts with a `413` error.
5. **Parsing**: Once the request stream ends (`'end'` event), the complete buffer is parsed according to the `Content-Type`:
   - **JSON**: Attempt `JSON.parse(buffer.toString('utf8'))`. If parsing fails (malformed JSON), throw a `400 Bad Request` error.
   - **URL-encoded**: Use the Node.js `querystring` or `URLSearchParams` module to parse the buffer string into key-value pairs.
6. **Assignment**: The parsed result (JavaScript object for JSON, object for urlencoded) is assigned to `req.body`. If parsing succeeded, the middleware calls `next()` to continue the pipeline. If parsing failed, `next(error)` is called with the error, which should be handled by error middleware.

Cookie Processing Algorithm

Cookie handling involves two parts: reading incoming cookies from the `Cookie` header and setting outgoing cookies via the `Set-Cookie` header.

Reading Cookies (Request-side):

1. **Header Extraction**: The `Request` constructor or a dedicated middleware examines `req.headers.cookie`.
2. **Parsing**: The header string (e.g., `session=abc123; user=john`) is split by `;`, then each key-value pair is trimmed and split by `=`.
3. **Decoding**: Cookie values are URL-decoded (using `decodeURIComponent`) because they may contain encoded characters.
4. **Assignment**: The resulting key-value pairs are stored in `req.cookies` as a plain object.

Setting Cookies (Response-side):

1. **Method Invocation**: Developer calls `res.cookie(name, value, options)`.
2. **Option Processing**: The method validates options (e.g., `maxAge` in milliseconds, `expires` as Date object, `secure` flag).
3. **String Serialization**: The method constructs a `Set-Cookie` header string following RFC 6265. For example: `session=abc123; Max-Age=3600; Path=/; HttpOnly`.
4. **Header Append**: The header string is appended to the `res` object's internal list of headers (multiple `Set-Cookie` headers are allowed). The actual header writing occurs when the response is flushed.

Clearing Cookies: `res.clearCookie(name, options)` sets the cookie's value to an empty string and sets its expiration to a past date. The `path` and `domain` options must match those used when setting the cookie to ensure the

correct cookie is cleared.

Common Pitfalls in Request/Response

⚠️ Pitfall: Blocking the Event Loop with Large JSON Parsing

- **Description**: Synchronously parsing a very large JSON string (using `JSON.parse`) in the main thread can block the Node.js event loop, stalling other requests.
- **Why it's wrong**: Node.js is single-threaded for JavaScript execution. Blocking operations degrade performance and can cause timeouts for concurrent requests.
- **Fix**: Implement size limits to reject excessively large bodies early. For truly large JSON parsing needs, consider offloading to a worker thread or using a streaming JSON parser (like `JSON.parse` on a stream is still synchronous for the final concatenated string, but chunk collection can be asynchronous).

⚠️ Pitfall: Setting Headers After Sending Body

- **Description**: Attempting to call `res.setHeader()` or `res.cookie()` after `res.send()` or `res.json()` has started writing the response body.
- **Why it's wrong**: HTTP headers must be sent before the body. Once the first chunk of body data is written to the socket, Node.js flushes the headers; subsequent header changes are silently ignored.
- **Fix**: Ensure all header modifications are completed before calling any send method. Our enhanced methods should guard against this by checking a `headersSent` flag (available on the native `res` object) and throwing an informative error.

⚠️ Pitfall: Not Handling Malformed JSON Gracefully

- **Description**: A client sends a request with `Content-Type: application/json` but the body is not valid JSON (e.g., `{invalid}`). A naive `JSON.parse` throws a synchronous exception, crashing the server if uncaught.
- **Why it's wrong**: Unhandled exceptions bring down the entire Node.js process, causing downtime.
- **Fix**: Wrap `JSON.parse` in a try-catch block. In case of error, call `next(error)` to pass the error to the framework's error-handling middleware, which should respond with a `400 Bad Request` and a descriptive message (in development only).

⚠️ Pitfall: Cookie Security Misconfiguration

- **Description**: Setting cookies without the `httpOnly` or `secure` flags when needed, exposing cookies to client-side JavaScript or transmitting them over unencrypted connections.
- **Why it's wrong**: Increases vulnerability to cross-site scripting (XSS) attacks and man-in-the-middle attacks.
- **Fix**: Educate developers about cookie flags. Our `res.cookie()` method should default `httpOnly` to `true` for session cookies (a common best practice) and allow easy configuration. In production, `secure` should be set when using HTTPS.

⚠️ Pitfall: Ignoring Query Parameter Array Values

- **Description**: URL query strings can contain multiple values for the same key (e.g., `?color=red&color=blue`). A simple parsing that overwrites the key's value will only capture the last value (`color: 'blue'`).
- **Why it's wrong**: Loss of data; the application may need to handle multiple values (e.g., multi-select filters).
- **Fix**: Use a parsing library (like Node's `querystring` module with `parse` function) that supports accumulating multiple values into an array. Our `req.query` should represent `?color=red&color=blue` as `{ color: ['red',`

`'blue'] }`.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Body Parsing	Buffer entire body (string concat) then parse with `JSON.parse` or `querystring`.	Use streaming parsers (e.g., `busboy` for multipart, `raw-body` + `content-type` for buffering with limits).
Cookie Parsing	Manual string splitting and `decodeURIComponent`.	Use the `cookie` npm module for robust parsing and serialization.
Query String Parsing	Node.js built-in `querystring` module or `URL` class.	Use `URLSearchParams` API (modern, but may need polyfill for older Node).
Response Helpers	Directly manipulate `res` object with `writeHead`, `end`.	Implement chainable API with lazy header writing (queue headers until first `write`).

B. Recommended File/Module Structure

```
barebones-framework/
├── lib/
│   ├── application.js      # Main Application class (includes use(), get(), listen())
│   ├── request.js          # Enhanced Request class
│   ├── response.js         # Enhanced Response class
│   ├── router/             # Router component (Milestone 1)
│   │   ├── index.js
│   │   ├── trie.js
│   │   └── layer.js
│   ├── middleware/         # Built-in middleware (Milestone 2 & 3)
│   │   ├── init.js         # Adds req.app, res.app, res.locals
│   │   ├── query.js        # Parses query string into req.query
│   │   ├── cookie.js       # Parses cookies into req.cookies
│   │   ├── json.js         # JSON body parser
│   │   ├── urlencoded.js   # URL-encoded body parser
│   │   └── static.js       # Static file serving (future)
│   └── template/           # Template engine (Milestone 4)
│       ├── engine.js
│       └── compiler.js
├── test/                   # Tests for each component
└── package.json
```

C. Infrastructure Starter Code (Complete)

File: `lib/request.js` - Enhanced Request wrapper. This is a **complete** implementation.

JAVASCRIPT

```javascript
const { IncomingMessage } = require('http');
const url = require('url');

/**
 * Enhanced Request wrapper.
 * @extends {IncomingMessage}
 */
class Request extends IncomingMessage {
  /**
   * @param {IncomingMessage} req - The native HTTP request object.
   * @param {Application} app - The application instance.
   */
  constructor(req, app) {
    // This is a bit of a hack: we want to inherit from IncomingMessage,
    // but Node.js doesn't allow easy extension of incoming sockets.
    // Instead, we create a new object that delegates to req.
    // In practice, we might not actually extend, but wrap.
    // For simplicity, we'll copy properties.
    super(req.socket);

    this.app = app;
    this.req = req; // keep reference to original

    // Copy all properties from req to this
    Object.assign(this, req);

    // Enhanced properties (initial values)
    this.body = undefined;
    this.query = {};
    this.params = {};
    this.cookies = {};
    this.originalUrl = req.url;
```

```
    // Parse query string on creation (could also be done via middleware)
    this._parseQueryString();
  }

  /**
   * Parse the URL query string into req.query.
   * Private method called in constructor.
   */
  _parseQueryString() {
    const parsedUrl = url.parse(this.url, true);
    this.query = parsedUrl.query || {};
    // Note: this.url might be changed by middleware (e.g., router strip prefix)
    // So we parse based on the original URL stored in originalUrl.
  }
}

module.exports = Request;
```

File: `lib/response.js` - Enhanced Response wrapper. This is a **complete** implementation.

JAVASCRIPT

```javascript
const { ServerResponse } = require('http');

/**
 * Enhanced Response wrapper.
 * @extends {ServerResponse}
 */
class Response extends ServerResponse {
  /**
   * @param {ServerResponse} res - The native HTTP response object.
   * @param {Application} app - The application instance.
   */
  constructor(res, app) {
    super(res.req); // ServerResponse expects the request
    this.app = app;
    this.res = res; // keep reference to original
    this.locals = {}; // for template data

    // Copy properties from res to this
    Object.assign(this, res);
  }

  /**
   * Set the HTTP status code.
   * @param {number} code - HTTP status code.
   * @returns {Response} this for chaining.
   */
  status(code) {
    this.statusCode = code;
    return this;
  }

  /**
```

```
   * Send a JSON response.
   * @param {any} data - The data to serialize as JSON.
   */
  json(data) {
    const body = JSON.stringify(data);
    this.setHeader('Content-Type', 'application/json');
    this.setHeader('Content-Length', Buffer.byteLength(body));
    this.end(body);
  }

  /**
   * Send a response of various types.
   * @param {string|Buffer|Object|Array} body - The response body.
   */
  send(body) {
    let chunk = body;
    let contentType = this.getHeader('Content-Type');

    // Infer content type if not set
    if (!contentType) {
      if (typeof chunk === 'string') {
        contentType = 'text/html';
      } else if (Buffer.isBuffer(chunk)) {
        contentType = 'application/octet-stream';
      } else if (typeof chunk === 'object' || Array.isArray(chunk)) {
        contentType = 'application/json';
        chunk = JSON.stringify(chunk);
      }
      this.setHeader('Content-Type', contentType);
    }
```

```
    // Set Content-Length if not set
    if (chunk && !this.getHeader('Content-Length')) {
      const length = Buffer.isBuffer(chunk) ? chunk.length : Buffer.byteLength(chunk);
      this.setHeader('Content-Length', length);
    }

    this.end(chunk);
  }

  /**
   * Redirect to a URL.
   * @param {number} status - HTTP status code (default 302).
   * @param {string} url - The redirect URL.
   */
  redirect(status, url) {
    if (typeof status === 'string') {
      url = status;
      status = 302;
    }
    this.writeHead(status, { 'Location': url });
    this.end();
  }

  /**
   * Set a cookie.
   * @param {string} name - Cookie name.
   * @param {string} value - Cookie value.
   * @param {Object} [options] - Cookie options.
   * @returns {Response} this for chaining.
   */
  cookie(name, value, options = {}) {
```

```
    const opts = Object.assign({}, options);
    const cookie = [`${name}=${encodeURIComponent(value)}`];

    if (opts.maxAge) cookie.push(`Max-Age=${opts.maxAge}`);
    if (opts.expires) cookie.push(`Expires=${opts.expires.toUTCString()}`);
    if (opts.path) cookie.push(`Path=${opts.path}`);
    if (opts.domain) cookie.push(`Domain=${opts.domain}`);
    if (opts.secure) cookie.push('Secure');
    if (opts.httpOnly) cookie.push('HttpOnly');
    if (opts.sameSite) cookie.push(`SameSite=${opts.sameSite}`);

    this.appendHeader('Set-Cookie', cookie.join('; '));
    return this;
  }

  /**
   * Clear a cookie.
   * @param {string} name - Cookie name.
   * @param {Object} [options] - Cookie options (must match path/domain used when setting).
   * @returns {Response} this for chaining.
   */
  clearCookie(name, options = {}) {
    const opts = Object.assign({}, options, { expires: new Date(1), maxAge: 0 });
    return this.cookie(name, '', opts);
  }

  /**
   * Helper to append a header value (preserving existing ones).
   * @private
   */
  appendHeader(name, value) {
    const current = this.getHeader(name);
```

```
    if (current) {
      if (Array.isArray(current)) {
        current.push(value);
        this.setHeader(name, current);
      } else {
        this.setHeader(name, [current, value]);
      }
    } else {
      this.setHeader(name, value);
    }
  }
}

module.exports = Response;
```

D. Core Logic Skeleton Code (with TODOs)

File: `lib/middleware/json.js` - JSON body parser middleware factory.

JAVASCRIPT

```javascript
/**
 * JSON body parser middleware factory.
 * @param {Object} [options] - Configuration options.
 * @param {number} [options.limit='1mb'] - Maximum body size (bytes or string like '1mb').
 * @returns {Function} Middleware function.
 */
function json(options = {}) {
  // TODO 1: Parse the limit option (could be string like '1mb' or number)
  // If string, convert to bytes (e.g., '1mb' -> 1024*1024)
  const limit = parseLimit(options.limit || '1mb');

  // Return the middleware function
  return function jsonParser(req, res, next) {
    // TODO 2: Skip requests that are not JSON content-type
    // Check req.headers['content-type'] includes 'application/json'
    // If not, call next() and return early.

    // TODO 3: Enforce size limit
    // Check Content-Length header if present. If exceeds limit, create an error
    // (status 413), call next(error), and return.

    // TODO 4: Collect data chunks from the request stream
    // let body = [];
    // let length = 0;
    // req.on('data', (chunk) => {
    //   length += chunk.length;
    //   if (length > limit) {
    //     // Destroy the stream or emit error? We'll just stop collecting and respond with 413.
    //     req.destroy(); // stop reading
    //     const err = new Error('Request body too large');
    //     err.status = 413;
```

```
    //     next(err);
    //     return;
    //   }
    //   body.push(chunk);
    // });

    // TODO 5: On stream end, concatenate buffers and parse JSON
    // req.on('end', () => {
    //   try {
    //     const data = Buffer.concat(body).toString('utf8');
    //     req.body = JSON.parse(data);
    //     next();
    //   } catch (err) {
    //     err.status = 400; // Bad Request
    //     next(err);
    //   }
    // });

    // TODO 6: Handle stream errors (call next with error)
    // req.on('error', (err) => next(err));
  };
}

/**
 * Helper to parse size limit string to bytes.
 * @private
 */
function parseLimit(limit) {
  if (typeof limit === 'number') return limit;
  // ... implement parsing of '1mb', '50kb', etc.
}
```

```
module.exports = json;
```

File: `lib/middleware/cookie.js` - Cookie parsing middleware.

```javascript
/**
 * Cookie parsing middleware.
 * Parses the Cookie header into req.cookies.
 */
function cookieParser(req, res, next) {
  // TODO 1: Get the Cookie header from req.headers.cookie
  // If no cookie header, set req.cookies = {} and call next().

  // TODO 2: Split the header string by ';' to get individual cookie strings.
  // For each cookie string, trim whitespace.

  // TODO 3: For each cookie string, split by '=' to get name and value.
  // The first '=' is the delimiter (cookie value may contain '=').
  // Use decodeURIComponent on both name and value (or at least value).

  // TODO 4: Build an object mapping cookie name to decoded value.
  // Assign to req.cookies.

  // TODO 5: Call next() to continue.
}

module.exports = cookieParser;
```

E. Language-Specific Hints

- **Stream Handling**: Use Node.js streams events (`'data'`, `'end'`, `'error'`). Remember to handle backpressure if needed (though for body parsing, we typically just collect chunks).
- **Buffer Concatenation**: `Buffer.concat(arrayOfBuffers)` is efficient for combining collected chunks.
- **Content-Type Parsing**: Use the `content-type` npm module to parse `Content-Type` header and get parameters (like charset). For simplicity, we can just check if the header includes `application/json`.
- **Error Propagation**: When an error occurs in middleware (e.g., malformed JSON), call `next(error)`. The framework should have an error-handling middleware (from Milestone 2) to respond appropriately.

- **Header Writing**: Once `res.write()` or `res.end()` is called, headers are flushed. Check `res.headersSent` before attempting to set headers.

F. Milestone Checkpoint

After implementing the enhanced request/response objects and body parsing middleware:

1. **Test JSON Parsing**:

```bash
# Start the server with a route that logs req.body
node server.js
```

In another terminal:

```bash
curl -X POST http://localhost:3000/api/data \
  -H "Content-Type: application/json" \
  -d '{"name": "test", "value": 42}'
```

The server route should log `{ name: 'test', value: 42 }`.

2. **Test Query Parameters**:

```bash
curl "http://localhost:3000/search?q=keyword&page=2"
```

The server should log `req.query` as `{ q: 'keyword', page: '2' }`.

3. **Test Cookie Parsing and Setting**:

```bash
curl -b "session=abc123; user=john" http://localhost:3000/
```

The server should log `req.cookies` as `{ session: 'abc123', user: 'john' }`. Also test setting a cookie: a route that calls `res.cookie('new', 'value')` should include a `Set-Cookie` header in the response.

4. **Test Response Helpers**:

```bash
curl -i http://localhost:3000/api/users
```

A route that does `res.json({ users: [] })` should return `Content-Type: application/json` and a JSON body.

Signs of Success: The above curl commands return expected data and headers.

Common Failures and Diagnostics:

- `req.body` is `undefined`: Ensure the JSON middleware is mounted (`app.use(json())`) and the request has correct `Content-Type`.
- Cookies not parsed: Check the Cookie header format; ensure middleware is mounted before routes.
- Headers not set: Verify you are not setting headers after `res.send()`.

Component Design: Template Engine

> **Milestone(s):** Milestone 4: Template Engine

The template engine is the **presentation layer** of the web framework, transforming static template files into dynamic HTML responses. Its primary responsibility is to safely combine developer-provided data with HTML markup, while offering control structures and inheritance patterns that keep templates DRY and maintainable. This component bridges the gap between application logic (JavaScript objects) and user interface (HTML pages).

Mental Model: The Mad Libs with Conditionals

Imagine you have a story template with blank spaces labeled `{{name}}`, `{{place}}`, and `{{action}}`. You fill these blanks with specific words to create a customized story. Now, imagine this story also has optional paragraphs marked with `{{#if happy_ending}}` and `{{#else}}`—you include only the appropriate paragraph based on whether the story context includes a `happy_ending: true` value. Finally, imagine you have a base story layout (header, footer, main area) that multiple specific stories can extend, replacing only the main content block while keeping the surrounding structure identical.

This is exactly how the template engine operates:

- **Variables** are blanks filled with context values
- **Control structures** (if/else, loops) conditionally include or repeat template sections
- **Template inheritance** allows child templates to extend a base layout, overriding specific blocks while inheriting the overall structure

The engine's job is to parse these special markers, generate an efficient rendering function, and execute it with the provided data context, ensuring that any HTML-sensitive characters in the data are properly escaped to prevent security vulnerabilities.

Template Engine Interface

The template engine exposes two primary interfaces: one for developers to register and render templates, and another for defining template syntax within template files themselves.

Public API Methods for Application Integration:

Method Name	Parameters	Returns	Description
`app.set('views', directoryPath)`	`directoryPath`: String	Application	Sets the directory where template files are located
`app.set('view engine', engineName)`	`engineName`: String	Application	Sets the default template engine extension (e.g., 'html')
`app.engine(ext, engineFunction)`	`ext`: String, `engineFunction`: Function	Application	Registers a template engine for files with given extension
`res.render(templateName, context)`	`templateName`: String, `context`: Object	void	Renders a template with the provided context, sends HTML response
`Template.compile(source)`	`source`: String	`compiledFn`: Function	Compiles template source string into a render function (internal)
`Template.renderFile(path, context)`	`path`: String, `context`: Object	Promise	Reads template file, compiles (caches), renders with context

Template Syntax Elements (Supported Markup):

Syntax	Example	Description	Context Requirements
Variable interpolation	`{{ user.name }}`	Replaced with escaped value of `user.name` from context	Context must have `user.name` property
Unescaped interpolation	`{{{ html_content }}}`	Replaced with raw, unescaped value (use with caution)	Value assumed safe HTML
If conditional	`{{#if loggedIn}}...{{/if}}`	Includes block only if `loggedIn` is truthy	Context must have `loggedIn` property
If-else conditional	`{{#if admin}}...{{#else}}...{{/if}}`	Chooses block based on truthiness of `admin`	-
Each loop	`{{#each items}}...{{/each}}`	Repeats block for each item in `items` array	Context must have `items` array
Block definition	`{{#block "sidebar"}}...{{/block}}`	Defines a named block in child template	Used in inheritance
Extends declaration	`{{#extends "base"}}`	Indicates this template extends a base template	Must be first line in template
Super content	`{{#super}}`	Inserts parent block content within child override	Used inside overriding block

ADR: Template Compilation Strategy

> **Decision: Pre-compile Templates to JavaScript Functions**
>
> - **Context**: Templates are text files containing static markup and dynamic placeholders/control structures. They need to be combined with data contexts repeatedly during request handling. Performance is critical since rendering happens per request, but development simplicity is also important for a learning framework.
> - **Options Considered**:
>   1. **Interpretation on each render**: Parse the template text and walk the AST every time a template is rendered
>   2. **Pre-compilation to JavaScript functions**: Parse once, generate a JavaScript function string, use `new Function()` or `eval` to create a reusable render function
>   3. **Tagged template literals**: Use JavaScript's built-in template literal functionality with custom tags
> - **Decision**: Pre-compile templates to JavaScript functions with caching
> - **Rationale**:
>   - **Performance**: Compilation happens once (or when template files change), rendering is just function execution. This is 10-100x faster than interpreting on each request.
>   - **Caching**: Compiled functions can be cached in memory, keyed by template path and modification timestamp.
>   - **Flexibility**: Generated functions can include optimized logic for escaping, conditional branching, and loops without runtime parsing overhead.
>   - **Debugging**: The generated function source can be inspected for debugging (though source maps would be complex).
> - **Consequences**:
>   - **Memory overhead**: Cached compiled functions consume memory proportional to template count and size.
>   - **Security consideration**: Using `new Function()` with template strings requires careful sanitation to avoid code injection.
>   - **Initial latency**: First request for a template incurs compilation cost.

Options Comparison:

Option	Pros	Cons	Selected?
Interpretation on each render	Simple implementation, no compilation phase, no `eval` security concerns	Poor performance (re-parses every render), no optimization opportunities	❌
Pre-compilation to functions	Excellent performance, optimization opportunities, industry standard (Jinja2, Handlebars)	Security concerns with `new Function()`, memory for cache, more complex implementation	✅
Tagged template literals	No parsing needed, uses JavaScript syntax, secure by nature	Requires templates in JS files (not separate .html), limited control structures, not file-based	❌

Internal Behavior: Compilation and Rendering

The template engine operates in two distinct phases: **compilation** (transforming template source to a render function) and **rendering** (executing that function with a data context). This separation enables caching and performance optimization.

Compilation Phase: From Template Text to Render Function

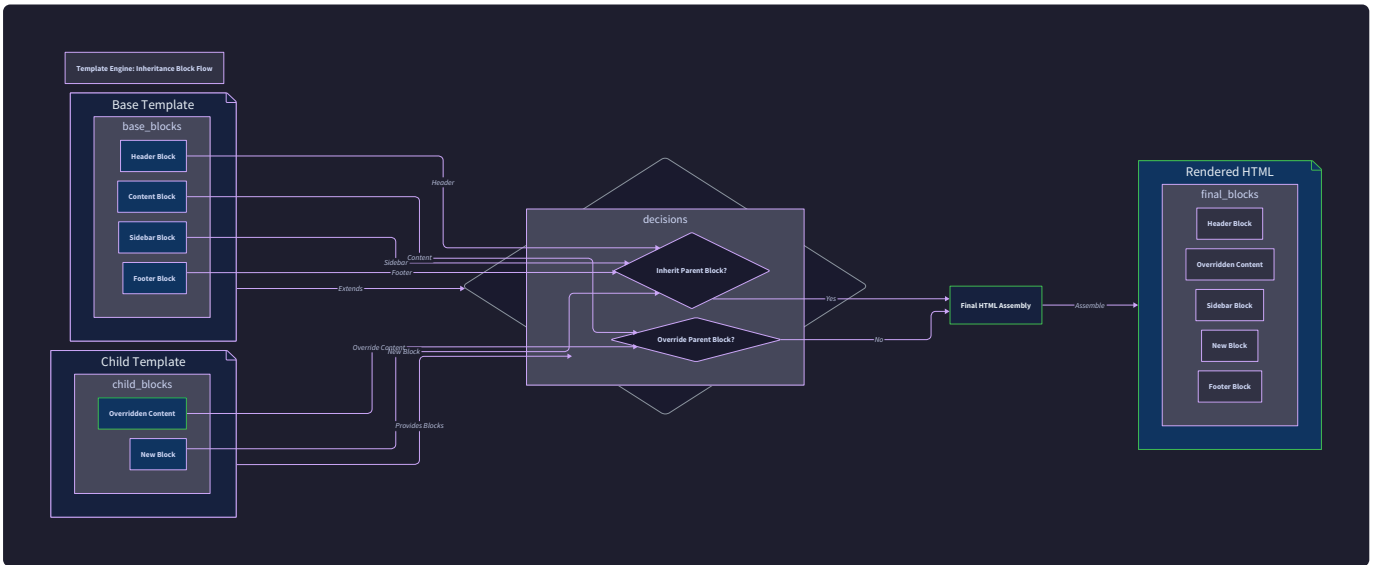
The compilation process transforms a template string into a JavaScript function that can efficiently render the template with any data context. The process follows these steps:

1. **Lexical Analysis (Tokenization)**:
   - Scan the template string character by character
   - Identify tokens: `{{` starts a variable/control block, `}}` ends it, `{{{` starts unescaped variable, `}}}` ends it
   - Classify tokens as: `VARIABLE`, `UNESCAPED_VARIABLE`, `IF_START`, `ELSE`, `IF_END`, `EACH_START`, `EACH_END`, `EXTENDS`, `BLOCK_START`, `BLOCK_END`, `SUPER`, `TEXT`
   - Store token positions and content
2. **Syntax Parsing (AST Construction)**:
   - Parse tokens into an Abstract Syntax Tree (AST)
   - Validate nesting: `{{#if}}` must have matching `{{/if}}`, blocks cannot cross boundaries
   - For inheritance: extract `{{#extends "base"}}` directive (must be first), collect all `{{#block "name"}}` definitions
   - Build tree nodes: `TextNode` (raw HTML), `VarNode` (variable path), `IfNode` (condition, then/else branches), `EachNode` (collection, loop body)
3. **Code Generation**:
   - Traverse the AST and generate JavaScript source code for a render function
   - The function signature: `function render(context, blocks = {}, parent = null)`
   - Initialize output accumulator: `let __output = [];`
   - For each AST node:
     - `TextNode`: Append string literal to output: `__output.push("...");`
     - `VarNode`: Look up variable path in context, HTML-escape, append: `__output.push(escape(context.user.name));`
     - `UnescapedVarNode`: Look up variable, append raw: `__output.push(context.html_content);`
     - `IfNode`: Generate conditional: `if (context.loggedIn) { ... } else { ... }`
     - `EachNode`: Generate loop: `for (let item of context.items) { ... }`
   - Return joined output: `return __output.join("");`
4. **Function Creation**:
   - Use `new Function("context", "blocks", "parent", generatedCode)` to create render function
   - Store metadata: template dependencies, required context properties
   - Cache the compiled function with template source hash as key

Rendering Phase: Executing with Data Context

When `res.render("template.html", { user: { name: "Alice" } })` is called:

1. **Template Resolution**:
   - Construct full path: `viewsDir + "/" + templateName + ".html"`
   - Check cache for compiled function with this path and file modification time
   - If not cached or file changed: read file, compile (as above), cache result
2. **Context Preparation**:
   - Merge `res.locals` (response-local variables) with provided context object
   - Add helper functions to context: `escape()` for HTML escaping
   - For inheritance: if template extends another, recursively resolve and compile parent
3. **Function Execution**:
   - Call the compiled render function with the prepared context
   - The function accesses context properties, applies logic, builds output string
   - For unescaped variables: directly insert values (developer assumes responsibility)
4. **Inheritance Resolution** (if template extends base):
   - Compile base template first
   - When base template encounters `{{#block "content"}}`, check if child provided override
   - If child has block definition, use child's block content; otherwise use base's default
   - Process recursively for nested inheritance
5. **Response Transmission**:
   - Set `Content-Type: text/html` header on response
   - Send rendered HTML string via `res.send()`



Example Walkthrough: Consider a base template `layout.html`:

```HTML
<!DOCTYPE html>

<html>

<head><title>{{title}}</title></head>

<body>

  <header>Site Header</header>

  <main>{{#block "content"}}Default content{{/block}}</main>

  <footer>Site Footer</footer>

</body>

</html>
```

And a child template `home.html`:

```HTML
{{#extends "layout"}}

{{#block "content"}}

  <h1>Welcome {{user.name}}</h1>

  {{#if user.admin}}

    <p>Admin controls enabled</p>

  {{/if}}

{{/block}}
```

With context `{ title: "Home", user: { name: "Alice", admin: true } }`:

1. Child template compiled, detects extends, processes content block
2. Base template compiled, block "content" replaced with child's block
3. Variables filled: `{{title}}` → `"Home"`, `{{user.name}}` → `"Alice"` (escaped)
4. Conditional evaluates `user.admin` as truthy, includes paragraph
5. Final HTML includes header/footer from base, content from child

Common Pitfalls in Templating

⚠️ Pitfall: Cross-Site Scripting (XSS) via Unescaped Output

- **Description**: Inserting user-provided data without HTML escaping allows attackers to inject `<script>` tags or other malicious HTML.
- **Why it's wrong**: `{{ user_input }}` where `user_input` is `<script>stealCookies()</script>` executes script in victims' browsers.
- **Fix**: Always escape variables by default. Use triple braces `{{{ }}}` only when you fully trust the content (e.g., sanitized HTML from CMS). The escape function should encode `&`, `<`, `>`, `"`, `'`, and `` ` ``.

⚠️ Pitfall: Silent Failure on Missing Variables

- **Description**: When a template references `{{ user.name }}` but `user` is undefined or `user.name` doesn't exist, the engine might output empty string or throw error.
- **Why it's wrong**: Empty output can hide bugs; thrown errors crash the application.
- **Fix**: Implement safe property access in generated code (e.g., `context?.user?.name` in JavaScript). Optionally provide a strict mode that throws in development but fails gracefully in production.

⚠️ Pitfall: Infinite Recursion in Template Inheritance

- **Description**: Template A extends B, B extends C, C extends A creates a circular dependency.
- **Why it's wrong**: Compilation enters infinite loop, crashes with stack overflow.
- **Fix**: Maintain a visited set during extends resolution. Detect cycles and throw a clear error: "Circular extends dependency detected: A → B → C → A".

⚠️ Pitfall: Inefficient Re-compilation on Every Request

- **Description**: Reading template file from disk and re-compiling for every request.
- **Why it's wrong**: Severe performance penalty (file I/O + parsing overhead).
- **Fix**: Implement in-memory cache of compiled functions. Key by template path + file modification timestamp. Invalidate cache when file changes (watch in development, restart in production).

⚠️ Pitfall: Block Scope Variable Collision in Loops

- **Description**: Inside `{{#each items}}`, using `{{this}}` for current item, but `{{@index}}` or parent context variables become inaccessible.
- **Why it's wrong**: Limits template expressiveness, forces workarounds.
- **Fix**: Implement proper scoping: within each loop, create a new context merging parent context with loop variables (`item`, `index`, `first`, `last`). Provide access to parent via `../` syntax (e.g., `{{../globalSetting}}`).

Implementation Guidance for Templating

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Template Parsing	Regex-based tokenizer with finite state machine	Proper lexer/parser using formal grammar (PEG.js, nearley)
AST Representation	Plain JavaScript objects with type and children fields	Custom node classes with visitor pattern for transformations
Code Generation	String concatenation building JavaScript source	Using AST-to-JS code generator library (escodegen)
Caching Strategy	Simple in-memory Map with template source as key	LRU cache with file watcher for development, memory limits

B. Recommended File/Module Structure

```
barebones-framework/
├── lib/
│   ├── application.js          # Main Application class
│   ├── router.js               # Router component
│   ├── middleware/             # Middleware implementations
│   │   └── bodyParser.js
│   └── template/               # Template engine component
│       ├── engine.js           # Main TemplateEngine class
│       ├── compiler.js         # Compilation logic (lexer, parser, codegen)
│       ├── cache.js            # Compiled template cache
│       └── utils/
│           └── escape.js       # HTML escaping utilities
├── examples/
│   └── views/                  # Example templates
│       ├── layout.html
│       └── home.html
└── package.json
```

C. Infrastructure Starter Code: HTML Escaping Utilities

```javascript
// lib/template/utils/escape.js

/**
 * HTML-escapes a string for safe insertion into HTML text content or attributes.
 * Replaces characters with HTML entities: &, <, >, ", ', `
 * @param {string} text - The text to escape
 * @returns {string} HTML-escaped text
 */
function escapeHTML(text) {
  if (typeof text !== 'string') {
    return String(text);
  }

  return text
    .replace(/&/g, '&amp;')
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/`/g, '&#96;');
}

/**
 * Returns text unmodified (for triple-brace unescaped interpolation).
 * Warning: Only use with trusted content.
 * @param {string} text - The text to leave unescaped
 * @returns {string} The original text
 */
function unsafeHTML(text) {
  return String(text);
```

```
}


module.exports = {
  escapeHTML,
  unsafeHTML
};
```

D. Core Logic Skeleton Code: Template Compiler

```javascript
// lib/template/compiler.js

const { escapeHTML } = require('./utils/escape');


class TemplateCompiler {

  /**

   * Compiles a template source string into a render function.

   * The generated function signature: (context, blocks, parent) => string

   * @param {string} source - Template source code

   * @returns {Function} Compiled render function

   */

  compile(source) {

    // TODO 1: Tokenize the source into an array of tokens

    //   - Identify {{, }}, {{{, }}} delimiters

    //   - Classify tokens: TEXT, VARIABLE, UNESCAPED_VARIABLE, CONTROL_START, CONTROL_END

    //   - Extract content between delimiters (e.g., "user.name", "#if loggedIn")



    // TODO 2: Parse tokens into an Abstract Syntax Tree (AST)

    //   - Create root node with children array

    //   - Handle nesting: push/pop from stack for control structures

    //   - Validate matching opening/closing tags

    //   - Extract extends directive if present (must be first non-text token)



    // TODO 3: Generate JavaScript source code from AST

    //   - Initialize output: `let __output = [];`

    //   - Traverse AST nodes:

    //     - TextNode: `__output.push("...");`

    //     - VarNode: `__output.push(escape(context.user.name));`

    //     - UnescapedVarNode: `__output.push(context.html_content);`

    //     - IfNode: `if (context.loggedIn) { ... } else { ... }`
```

```
    //     - EachNode: `for (let item of context.items) { ... }`
    //   - Finalize: `return __output.join("");`

    // TODO 4: Create function from generated code
    //   - Use `new Function('context', 'blocks', 'parent', generatedCode)`
    //   - Provide escapeHTML as a local variable in function scope
    //   - Return the compiled function

    // TODO 5: Handle template inheritance
    //   - If template has extends directive, compile parent first
    //   - Modify parent's render function to check for overridden blocks
    //   - Implement block lookup: blocks['blockName'] || parent default
  }

  /**
   * Tokenizes template source into an array of token objects.
   * @private
   * @param {string} source - Template source
   * @returns {Array} Tokens with type and value
   */
  tokenize(source) {
    // TODO: Implement regex or state machine scanner
    // Return array of { type: 'TEXT'|'VAR'|'UNESCAPED_VAR'|'CONTROL_START'|'CONTROL_END', value:
string }
  }
}

module.exports = TemplateCompiler;
```

E. Language-Specific Hints for JavaScript/Node.js

- **`new Function()` Security**: Be aware that `new Function()` evaluates code in the global scope. While generally safe for generated code, ensure you never include user input in the generated function body. Only use template

variables via the `context` parameter.

- **File System Caching**: Use `fs.statSync()` to get file modification times for cache invalidation. In production, consider a memory cache with periodic invalidation.
- **Async Rendering**: While template rendering is synchronous, reading template files should be async (`fs.promises.readFile`). Use an async compile/render method that returns a Promise.
- **Error Context**: When compilation fails, catch errors and add template name and line number to the error message for easier debugging.

F. Milestone Checkpoint for Template Engine

Verification Command:

```bash
node test/template-test.js
```

Expected Behavior:

1. Variable interpolation works: `{{ name }}` with `{ name: "Alice" }` produces `Alice` (escaped).
2. HTML escaping works: `{{ html }}` with `{ html: "<script>" }` produces `<script>`.
3. Unescaped interpolation works: `{{{ safe_html }}}` inserts raw HTML.
4. If/else conditionals work correctly based on truthy/falsy values.
5. Each loops iterate over arrays, providing `this` (current item) and `@index`.
6. Template inheritance: child extends base, overrides blocks, preserves parent structure.

Sample Test Template (`test.bb.html`):

```html
{{#extends "base"}}

{{#block "content"}}

<h1>{{title}}</h1>

{{#each users}}

  <p>{{@index}}. {{this.name}}</p>

{{/each}}

{{/block}}
```

Signs of Problems:

- **"Unexpected token" errors**: Likely tokenizer missing edge cases (nested braces).
- **Variables output empty**: Context not properly passed or property access failing.
- **Inheritance not working**: Block resolution logic incorrect or extends directive not parsed first.
- **Memory leak**: Cache growing unbounded—implement size limits or LRU eviction.

G. Debugging Tips for Template Engine

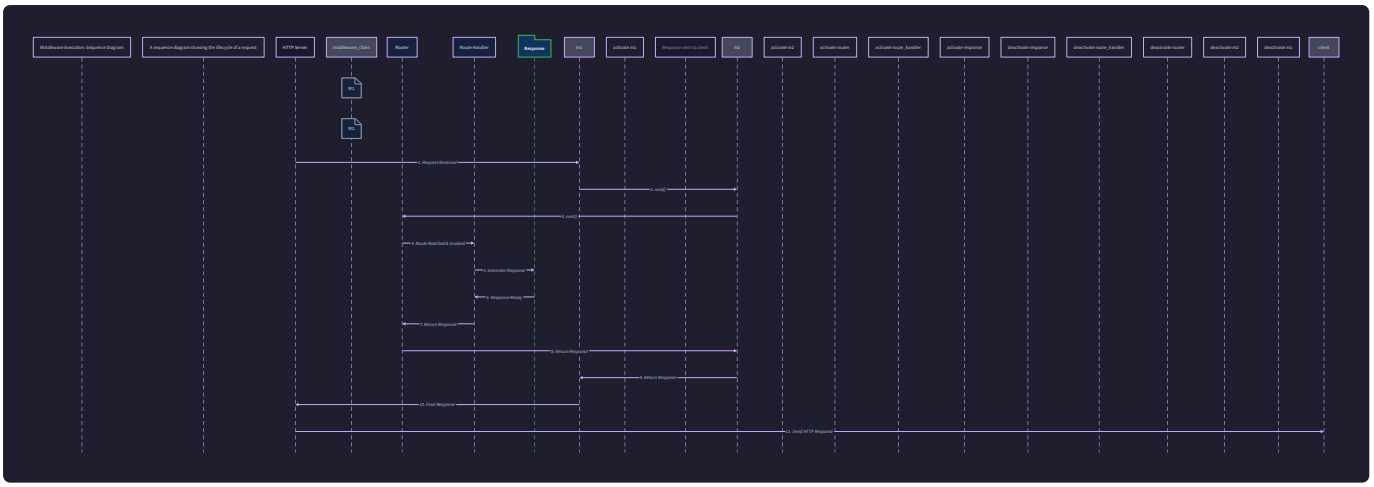
Symptom	Likely Cause	How to Diagnose	Fix
Template renders empty output	Context variables not accessible	Log the generated render function source, check property access paths	Ensure context merging includes all needed properties
HTML tags showing as text in browser	Missing Content-Type header	Check response headers with browser dev tools	Call `res.type('html')` before `res.render()`
`{{#if}}` always shows else branch	Falsy value in JavaScript (0, "", false, null, undefined)	Log the context value and its type	Use explicit comparisons or ensure data is truthy
Loop shows "undefined" for items	Array missing or not an array	Check context structure, verify `items` is Array	Add array check in generated code: `if (Array.isArray(context.items))`
Inheritance renders parent only	Child blocks not being detected	Check if extends directive is first in template	Ensure parser processes extends before other content
Performance slow on repeated requests	Recompiling on every render	Add cache logging, check cache hit/miss ratio	Implement proper caching with file timestamp checking

Interactions and Data Flow

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

This section traces the complete lifecycle of HTTP requests through the Barebones framework's processing pipeline. Understanding this flow is critical for debugging and extending the framework—it shows how all components coordinate to transform raw HTTP bytes into meaningful responses.

Think of the framework as a **well-choreographed assembly line**: the request is a raw material that moves through specialized stations (middleware, router, handler, template engine), acquiring refinements at each stage until it emerges as a polished response. Each station knows only its immediate neighbors and follows strict protocols for passing control, making the system modular and predictable.



Happy Path: GET Request with Template Render

This scenario illustrates a successful request that flows through the entire pipeline without errors. Consider a user visiting `/products/42` to view details about a specific product. The application has registered logging middleware, a route handler that fetches product data, and a template to render the HTML.

Request Entry and Initial Wrapping

1. **HTTP Server Receives Raw Request**: The Node.js `http.createServer()` callback receives native `IncomingMessage` (`req`) and `ServerResponse` (`res`) objects. These are low-level objects with minimal conveniences—just raw headers, URL strings, and socket streams.
2. **Application Wraps Request/Response**: The framework's `app.handleRequest()` method is invoked. Its first action creates enhanced `Request` and `Response` wrapper objects:
   - The `Request` constructor copies the native `req` reference and initializes empty `body` , `params` , `query` , and `cookies` objects.
   - The `Response` constructor copies the native `res` reference and sets `statusCode` to 200 (default).
   - Both wrappers receive a reference to the `Application` instance, giving them access to framework configuration.
3. **URL Parsing and Query Extraction**: Before entering the middleware chain, the framework parses the request URL to extract query parameters. For `/products/42?sort=price&highlight=true` :
   - The `originalUrl` field stores the complete URL string.
   - The `query` object becomes `{ sort: 'price', highlight: 'true' }` through simple `querystring.parse()` on the URL's query portion.

> **Key Insight**: This initial parsing happens before middleware because query parameters are fundamental to HTTP and don't require configuration. However, body parsing waits until later because it requires reading stream data and depends on Content-Type headers.

Middleware Processing Conveyor Belt

4. **Middleware Chain Execution Begins**: The application iterates through its `middleware` array in registration order. Each middleware function receives `(req, res, next)` , where `next` is a callback function that advances to the next middleware.

5. **Logging Middleware Records Entry**: The first middleware logs the request start time and details:

```javascript
// Pseudo-behavior, not actual code

const start = Date.now();

console.log(`[${new Date().toISOString()}] ${req.method} ${req.originalUrl}`);

next(); // Pass control forward

// After response finishes, it logs duration
```

The middleware calls `next()` immediately, passing control down the chain while storing the start time for later logging.

6. **Body Parsing Middleware Waits**: If present, a JSON body parser middleware checks the `Content-Type` header. For a GET request (which has no body), it immediately calls `next()` without modifying `req.body`.
7. **Cookie Parsing Middleware Executes**: A cookie middleware parses the `Cookie` header string into key-value pairs:
    - It reads `req.headers.cookie` (e.g., `"session=abc123; userPref=dark"`)
    - It populates `req.cookies` with `{ session: 'abc123', userPref: 'dark' }`
    - Calls `next()` to continue the chain.

Routing: The Telephone Switchboard

8. **Router Receives Control**: After all application-level middleware runs, control reaches the router's dispatch function. The router first runs any router-specific middleware (registered via `router.use()`), then begins route matching.
9. **Trie Search for `/products/42`**: The router calls `router.match('GET', '/products/42')`, which traverses the prefix tree:
    - Root node → child `'products'` (static segment) → child `:id` (parameter segment)
    - The parameter node matches `'42'`, storing `{ id: '42' }` in the params object
    - The leaf node contains a handler map with a `GET` handler function
    - The match returns an object with `{ handlers: [productHandler], params: { id: '42' } }`
10. **Parameter Attachment**: The router attaches the extracted parameters to `req.params`, making them available to the route handler. Now `req.params.id === '42'`.

Handler Execution and Template Rendering

11. **Route Handler Invocation**: The product handler function receives `(req, res)`:

```javascript
// Application code example

app.get('/products/:id', async (req, res) => {

  const product = await database.getProduct(req.params.id);

  res.render('product-detail', { product });

});
```

The handler fetches product data (simulated) and calls `res.render()`.

12. **Template Engine Activation**: The `res.render()` method performs several steps:
    - It resolves the template file path using the application's `views` directory setting.
    - It checks if the template is already compiled and cached; if not, it reads the file and calls `Template.compile()`.
    - The compiler parses the template source, generates an abstract syntax tree (AST), then produces a JavaScript function that concatenates strings with escaped variables.
    - The compiled function executes with the context `{ product: {...} }`, replacing `{{ product.name }}` with HTML-escaped values.
13. **Response Finalization**: The rendered HTML string is sent via `res.send()`:
    - `res.send()` sets `Content-Type: text/html` automatically.
    - It writes the status code, headers, and body to the native `res` object.
    - It calls `res.end()` to finalize the response.

Response Completion and Cleanup

14. **Control Unwinds Backward**: After `res.end()` is called, control returns up the middleware chain in reverse order (like unwinding a stack). Each middleware that stored deferred logic now executes it.
15. **Logging Middleware Records Completion**: The logging middleware's deferred code runs:

```javascript
    const duration = Date.now() - start;

    console.log(`[${new Date().toISOString()}] ${req.method} ${req.originalUrl} ${res.statusCode} ${duration}ms`);
```

16. **HTTP Server Sends Bytes**: The native `ServerResponse` object writes the complete HTTP response to the TCP socket, and the connection may close or persist depending on `Connection` headers.

The following state machine table summarizes the request's journey through distinct phases:

Current State	Event Trigger	Next State	Actions Performed
Receiving	HTTP socket data received	Parsing	Node.js HTTP module parses headers, creates `IncomingMessage`
Parsing	`app.handleRequest()` called	Middleware Processing	Create enhanced `Request` / `Response`, parse query string
Middleware Processing	`next()` called	Next Middleware (or Routing if last)	Execute current middleware function
Routing	Router receives control	Route Matching	Run router middleware, search trie for matching route
Route Matching	Route found in trie	Handler Execution	Attach params to `req.params`, select handler
Handler Execution	Handler calls `res.render()`	Template Rendering	Fetch data, invoke template engine
Template Rendering	Template compilation/execution	Response Building	Compile template (if needed), render with context
Response Building	`res.send()` called	Sending	Set headers, write body, end response
Sending	`res.end()` called	Complete	Control unwinds through middleware (post-processing)
Complete	Response written to socket	(End)	Connection may close

Error Flow: Failed Validation to 404

Not all requests complete successfully. The framework must handle errors gracefully—whether from missing routes, invalid user input, or server exceptions—while maintaining a proper HTTP response. Consider a POST request to `/api/users` with invalid JSON data.

Error Detection in Body Parsing

1. **Middleware Chain with JSON Parser**: The application includes `app.use(json())` middleware. When a POST request arrives with `Content-Type: application/json`, this middleware attempts to parse the request body.
2. **Stream Reading and Parsing**: The middleware reads the request body stream (buffering up to a configured limit), then calls `JSON.parse()` on the buffer. If the JSON is malformed (e.g., `{ "name": "Alice", }` with trailing comma), `JSON.parse()` throws a `SyntaxError`.
3. **Middleware Error Propagation**: Instead of crashing the entire application, the JSON parser middleware catches the parsing error and passes it to `next()`:

```javascript
try {
  req.body = JSON.parse(buffer.toString());
  next();
} catch (err) {
  next(err); // Error-aware next() call
}
```

Calling `next(err)` signals the error-handling system that something went wrong.

Error-Aware Pipeline Switching

4. **Pipeline Mode Switch**: When `next()` is called with an argument (the error object), the framework switches from "normal processing mode" to "error handling mode." In this mode:
    - The regular middleware chain is skipped.
    - Only **error-handling middleware** (functions with four parameters: `err, req, res, next`) are executed.
    - The framework searches for the first error-handling middleware registered after the point where the error occurred.
5. **Error Handler Execution**: Suppose this error handler is registered:

```javascript
app.use((err, req, res, next) => {
  console.error('Request error:', err);
  res.status(400).json({ error: 'Invalid JSON' });
});
```

This middleware receives the `SyntaxError` instance, logs it, and sends a 400 Bad Request response with a JSON error message.

6. **Response Sent Early**: The error handler calls `res.status().json()`, which sends the response immediately. Normal middleware and route handlers are bypassed entirely.

Route Matching Failures (404 Not Found)

7. **Missing Route Scenario**: For a GET request to `/non-existent/path`, the pipeline proceeds through middleware normally, but when the router attempts `router.match('GET', '/non-existent/path')`, the trie search returns `null`.
8. **Router Creates 404 Error**: The router's dispatch function detects no matching route and creates an error object:

```javascript
const err = new Error(`Cannot ${req.method} ${req.url}`);
err.status = 404;
next(err); // Pass to error-handling pipeline
```

9. **Default 404 Handler**: If no application-specific error handler catches this, the framework's default error handler activates:
    - It sets the status code to `err.status` or 500.
    - It sends a plain text or JSON response (based on `Accept` header).
    - For development mode, it may include the error stack trace.

Async Handler Error Propagation

10. **Async Route Handler Throws**: Consider an async route handler that fails:

```javascript
app.get('/api/data', async (req, res) => {
  const data = await fetchExternalAPI(); // Throws network error
  res.json(data);
});
```

11. **Promise Rejection Capture**: The router's `router.dispatch()` method wraps async handler execution in a try-catch or promise catch:

```javascript
try {
  await handler(req, res);
} catch (err) {
  next(err); // Forward to error pipeline
}
```

12. **Error Handler Receives Async Error**: The same error-handling middleware processes async errors identically to synchronous ones, ensuring consistent error responses regardless of execution mode.

> **Critical Design Insight**: The framework maintains **two parallel pipelines**—normal and error-handling—that share the same middleware array but filter functions based on arity. This design ensures errors can be caught at any stage and routed to appropriate handlers without duplicating logic.

Comprehensive Error State Transitions

The framework's error handling can be modeled as a state machine with error-specific transitions:

Current State	Error Event	Next State	Error Handling Actions
Middleware Processing	`next(err)` called	Error Handling	Skip remaining normal middleware, find first error handler
Route Matching	No route found	Error Handling	Create 404 error, call `next(err)`
Handler Execution	Synchronous throw	Error Handling	Catch exception, call `next(err)`
Handler Execution	Async rejection	Error Handling	Catch promise rejection, call `next(err)`
Template Rendering	Template not found	Error Handling	Catch file system error, call `next(err)`
Error Handling	Error handler calls `next(err)`	Next Error Handler	Continue to next error-handling middleware
Error Handling	Error handler sends response	Complete	Response sent, unwind middleware
Error Handling	No error handler registered	Default Error Handler	Use built-in error responder, send 500/404

Common Error Flow Scenarios

Scenario 1: Validation middleware rejects request

- **Path**: POST `/api/users` with invalid email format
- **Flow**: Validation middleware → `next(new ValidationError('Invalid email'))` → Error handler → 400 JSON response

Scenario 2: Database query fails in handler

- **Path**: GET `/users/999` (non-existent user)
- **Flow**: Route handler → `db.query()` throws `NotFoundError` → Async catch → Error handler → 404 JSON response

Scenario 3: Template variable missing

- **Path**: GET `/profile` with missing `user` context
- **Flow**: Template rendering → ReferenceError on `user.name` → Async catch → Error handler → 500 HTML error page

Scenario 4: Multiple error handlers with different responsibilities

- **Flow**: JSON parsing error → First error handler (logs error) → calls `next(err)` → Second error handler (sends formatted response) → Response sent

Common Pitfalls in Request Flow

⚠️ Pitfall: Middleware that doesn't call `next()`

- **Description**: A middleware function processes the request but forgets to call `next()`, causing the pipeline to hang indefinitely.

- **Why it's wrong**: The request never reaches subsequent middleware, route handlers, or error handlers. The client eventually times out.
- **Fix**: Always call `next()` unless you're intentionally terminating the request (e.g., sending a response early). Use `return next()` to prevent accidental code after `next()`.

⚠️ **Pitfall: Error handler missing `err` parameter**

- **Description**: Registering a function with signature `(req, res, next)` instead of `(err, req, res, next)` as error-handling middleware.
- **Why it's wrong**: When an error occurs, the framework calls the function with four arguments, but the function interprets `err` as `req`, leading to type errors or incorrect behavior.
- **Fix**: Check function arity: 3 parameters for normal middleware, 4 for error-handling middleware. Use `if (err instanceof Error)` pattern if a middleware needs to handle both.

⚠️ **Pitfall: Modifying `req` or `res` after response sent**

- **Description**: Middleware or handler code that sets headers or writes to `res` after `res.end()` has been called.
- **Why it's wrong**: Node.js throws "Cannot set headers after they are sent to the client" errors, crashing the request handling.
- **Fix**: Ensure response logic is complete before calling `res.send()`, `res.json()`, or `res.end()`. Use early returns to avoid fall-through code.

⚠️ **Pitfall: Not handling async errors in middleware**

- **Description**: Async middleware that throws but doesn't catch its own promise rejections.
- **Why it's wrong**: The error bubbles to the Node.js process unhandled rejection, potentially crashing the server.
- **Fix**: Always wrap async middleware in try-catch or ensure it returns a promise that the framework can catch. Use `next(err)` in catch blocks.

⚠️ **Pitfall: Route matching order causing shadowing**

- **Description**: Registering `/users/:id` before `/users/new` causes the parameter route to match `/users/new`, interpreting "new" as an ID.
- **Why it's wrong**: The `/users/new` handler never gets called, breaking expected application behavior.
- **Fix**: Register static routes before parameter routes of the same prefix. The router's trie should handle this automatically if insertion order respects specificity.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Request/Response Enhancement	Direct property assignment to wrapper objects	Proxy objects for dynamic property access
Error Propagation	Try-catch in dispatch loop	Domain-based error handling with async hooks
Pipeline Control	Array index pointer with `next()` callback	Generator-based middleware with `yield`
State Tracking	Simple object with current phase flag	Finite state machine library (e.g., `xstate`)

Recommended File/Module Structure

```
barebones-framework/
├── lib/
│   ├── application.js      # Main Application class with handleRequest
│   ├── request.js          # Enhanced Request class
│   ├── response.js         # Enhanced Response class
│   ├── router/
│   │   ├── index.js        # Router class
│   │   ├── trie.js         # TrieNode implementation
│   │   └── layer.js        # Layer class for routes/middleware
│   ├── middleware/
│   │   ├── init.js         # Built-in middleware (json, urlencoded, static)
│   │   └── error.js        # Default error handler
│   └── template/
│       ├── engine.js       # Template engine interface
│       └── compiler.js     # Template compilation logic
├── examples/               # Example applications
└── test/                   # Test suites for each component
```

Infrastructure Starter Code

Complete Error-Handling Wrapper for Async Middleware:

```javascript
// lib/middleware/async-wrapper.js

/**
 * Wraps async middleware to ensure errors are caught and passed to next()
 * @param {Function} fn - Async middleware function (req, res, next)
 * @returns {Function} Wrapped middleware that catches promise rejections
 */
function wrapAsync(fn) {
  return function(req, res, next) {
    // If fn returns a promise, catch any rejection and pass to next
    const result = fn(req, res, next);
    if (result && typeof result.then === 'function') {
      result.catch(next);
    }
    // If fn doesn't return a promise, errors are caught by outer try-catch
    return result;
  };
}

module.exports = wrapAsync;
```

Request Phase Tracker for Debugging:

JAVASCRIPT

```javascript
// lib/debug/phase-tracker.js

/**
 * Middleware that logs each phase transition for debugging
 */
function createPhaseTracker() {
  const phases = [
    'INITIAL', 'MIDDLEWARE', 'ROUTING', 'HANDLER',
    'RENDERING', 'SENDING', 'ERROR', 'COMPLETE'
  ];

  return function(req, res, next) {
    req._phase = 'INITIAL';
    const originalNext = next;
    let phaseIndex = 0;

    // Wrap next() to track phase transitions
    req.next = function(...args) {
      if (args[0] instanceof Error) {
        req._phase = 'ERROR';
      } else if (phaseIndex < phases.length - 1) {
        phaseIndex++;
        req._phase = phases[phaseIndex];
      }
      console.log(`[Phase] ${req.method} ${req.url} → ${req._phase}`);
      return originalNext.apply(this, args);
    };

    next();
  };
}
```

```
module.exports = createPhaseTracker;
```

Core Logic Skeleton Code

Application Request Handler (app.handleRequest):

JAVASCRIPT

```javascript
// lib/application.js

class Application {

  // ... other methods ...



  /**

   * Process HTTP request through the entire framework pipeline

   * @param {http.IncomingMessage} req - Native Node.js request

   * @param {http.ServerResponse} res - Native Node.js response

   */

  async handleRequest(req, res) {

    // TODO 1: Create enhanced Request and Response objects

    //   - new Request(req, this)

    //   - new Response(res, this)

    //   - Parse query string and attach to request.query



    // TODO 2: Initialize request phase tracking (optional debug aid)

    //   - req._phase = 'MIDDLEWARE'



    // TODO 3: Execute the middleware chain

    //   - Create a next function that advances through middleware array

    //   - Handle both normal and error pipelines based on arguments



    // TODO 4: After pipeline completes (response sent), handle cleanup

    //   - Ensure response is ended (call res.end() if not already)

    //   - Log completion via any post-response middleware



    // TODO 5: Catch any unhandled errors at the top level

    //   - If error reaches here, send 500 Internal Server Error

    //   - Prevent server crash by handling all exceptions

  }
```

```
}
```

Router Dispatch Method with Error Handling:

JAVASCRIPT

```javascript
// lib/router/index.js

class Router {

  // ... other methods ...



  /**

   * Dispatch request to matching route or pass to error handlers

   * @param {Request} req - Enhanced request object

   * @param {Response} res - Enhanced response object

   * @param {Function} next - Callback to continue to outer error handlers

   * @returns {Promise} Resolves when request handling completes

   */

  async dispatch(req, res, next) {

    // TODO 1: Execute router-level middleware first

    //   - Iterate through this.middleware array

    //   - Call each middleware with (req, res, nextRouter)



    // TODO 2: Attempt to find matching route

    //   - const match = this.match(req.method, req.pathname)

    //   - If no match, create 404 error and call next(err)



    // TODO 3: Attach route parameters to request

    //   - req.params = match.params



    // TODO 4: Execute route handlers

    //   - For each handler in match.handlers:

    //   - Wrap in try-catch to catch synchronous errors

    //   - Use wrapAsync() for async handlers

    //   - If handler calls next(err), break and propagate



    // TODO 5: If no response sent by handlers, call next() to continue
```

```
    //   - Check if res.headersSent to determine if response was sent
  }
}
```

Language-Specific Hints (JavaScript/Node.js)

- **Async/Await Integration**: Use `util.promisify` for callback-based operations in middleware. Remember that async functions return promises that must be caught.
- **Stream Handling**: For body parsing, use `req.on('data')` and `req.on('end')` events. Consider using `raw-body` npm package for production-ready buffer handling.
- **Error Status Codes**: Standardize error status codes by attaching `err.status = 404` to error objects. The default error handler should respect this property.
- **Phase Tracking**: Use `Symbol('phase')` for private phase property keys to avoid collisions with user code accessing `req._phase`.
- **Response Finished Detection**: Check `res.finished` or `res.writableEnded` (Node.js >= 12.9.0) to determine if response was sent.

Milestone Checkpoint

After implementing the full pipeline, verify with this test:

1. **Start test server**:

```bash
node examples/full-pipeline.js
```

2. **Send various requests with curl**:

```bash
# Happy path
curl -v http://localhost:3000/products/42

# Error path: malformed JSON
curl -v -X POST -H "Content-Type: application/json" \
  -d '{"invalid": json}' http://localhost:3000/api/users

# 404 path
curl -v http://localhost:3000/non-existent
```

3. **Expected behavior**:
   - Happy path: Returns HTML with product details, status 200
   - Malformed JSON: Returns JSON `{"error":"Invalid JSON"}`, status 400

- 404 path: Returns plain text "Cannot GET /non-existent", status 404
- All requests should log method, path, status, and duration

4. **Signs of trouble**:
   - Request hangs indefinitely → Likely middleware not calling `next()`
   - "Cannot set headers after they are sent" → Multiple response attempts
   - Error returns 500 instead of 400/404 → Error handler missing or not catching
   - No logs appear → Logging middleware not registered or not calling `next()`

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Request hangs, never responds	Middleware missing `next()` call	Add debug logging after each middleware; check which logs appear last	Ensure all middleware paths call `next()` or send response
Error returns 500 instead of custom status	Error object missing `status` property	Log `err.status` in error handler	Set `err.status = 404` when creating "not found" errors
Headers sent error after response	Code running after `res.send()`	Add stack trace logging to identify code path	Use `return res.send()` to exit function immediately
Async operation error crashes server	Unhandled promise rejection	Listen to `process.on('unhandledRejection')`	Wrap async handlers with `wrapAsync()` or try-catch
Route matches incorrectly	Trie insertion order issue	Print trie structure with `router._trie.toString()`	Register static routes before parameter routes
Template error shows in console but not response	Template error not caught	Add try-catch in `res.render()` implementation	Ensure template rendering errors call `next(err)`

Error Handling and Edge Cases

> **Milestone(s):** Milestone 2: Middleware, Milestone 3: Request/Response Enhancement, Milestone 4: Template Engine

A robust web framework must gracefully handle failures at every processing stage while maintaining clear, predictable behavior. Think of error handling as a **safety net system** in a circus performance: each act (component) has its own safety measures, but there's also a comprehensive system-wide net that catches any performer who falls unexpectedly. This section systematically examines what can go wrong, how we detect failures, and how the framework recovers to maintain stability and provide meaningful feedback to developers and users.

Catalog of Failure Modes

Every component in the framework faces distinct failure modes. This catalog organizes failures by component, detailing their detection mechanisms and immediate recovery actions before discussing the unified strategy.

Router Failure Modes

The router's primary failure is failing to match a request to a registered handler. However, subtle edge cases exist beyond simple 404s.

Failure Mode	Detection Mechanism	Immediate Recovery Action	Potential Impact
No route match	`router.match()` returns `null`	Pass to next middleware (if any), eventually send 404 response	User sees 404 page, framework continues processing
Route parameter parsing error	URL segment contains invalid characters or encoding	Extract raw string, let handler validate; or return 400 if strict validation enabled	Handler receives malformed parameter, must handle validation
Duplicate route registration	During development: warning when inserting into trie; production: last registration wins	Log warning to console, override previous handler	Unintended handler replacement, potential security issues
Trailing slash mismatch	Path `/users` doesn't match `/users/` in strict mode	Optionally normalize paths before matching or maintain separate routes	User gets 404 for semantically same URL
Route handler throws synchronous error	`try/catch` in `router.dispatch()`	Convert to error object, pass to error-handling middleware	Request processing halts, error propagates

Middleware Engine Failure Modes

Middleware failures are particularly challenging because they occur at various points in the processing pipeline and can involve asynchronous operations.

Failure Mode	Detection Mechanism	Immediate Recovery Action	Potential Impact
Middleware forgets to call `next()`	Timeout detection or request hangs indefinitely	Framework cannot automatically detect; must rely on developer discipline	Request hangs, connection times out, server resources exhausted
Middleware calls `next()` multiple times	Flag tracking whether `next` has been called	Ignore subsequent calls, log warning in development	Double response sending, "Can't set headers after they are sent" error
Synchronous error thrown in middleware	`try/catch` in middleware dispatch loop	Convert to error object, pass to error-handling middleware	Pipeline stops, error propagates to error handlers
Asynchronous error in middleware	Unhandled promise rejection or callback error	Attach `.catch()` to async middleware return value	Error may go undetected, crashing the Node.js process
Error-handling middleware throws error	Nested `try/catch` in error dispatch	Log catastrophic failure, send generic 500 response	Original error lost, user sees generic error
Middleware modifies request/response after headers sent	Check `res.headersSent` flag	Ignore modification, log warning in development	Inconsistent state, potential security issues

Request/Response Enhancement Failure Modes

These failures occur during parsing and enhancement of the raw HTTP objects, often involving malformed or malicious input.

Failure Mode	Detection Mechanism	Immediate Recovery Action	Potential Impact
Malformed JSON in request body	`JSON.parse()` throws SyntaxError	Catch error, pass to error-handling middleware with 400 status	Request rejected, user gets 400 Bad Request
Request body exceeds size limit	Content-Length header check or stream byte counting	Respond with 413 Payload Too Large, stop reading stream	Memory protection, DoS prevention
Invalid Content-Type for body parser	Parser mismatch between header and actual content	Skip parsing, leave `req.body` empty, or return 415 Unsupported Media Type	Handler receives empty body, must handle missing data
Cookie parsing fails	Malformed Cookie header syntax	Ignore malformed cookies, parse valid ones, or reject entire header	Some cookies missing, authentication may fail
Query parameter parsing error	Malformed URL encoding (e.g., incomplete `%` escape)	Decode best effort, preserve raw string for problematic parameters	Data corruption or loss
Response sent twice	Check `res.headersSent` before sending	Throw error in development, ignore in production	"Can't set headers after they are sent" error

Template Engine Failure Modes

Template failures typically involve missing files, syntax errors, or rendering problems that prevent HTML generation.

Failure Mode	Detection Mechanism	Immediate Recovery Action	Potential Impact
Template file not found	`fs.readFile()` error with ENOENT code	Throw error with 404 status for template, pass to error-handling middleware	User sees error page instead of content
Template syntax error	Compilation phase throws parse error	Cache error, reject all render attempts with descriptive error	All requests for template fail until fixed
Undefined variable in template	Reference error during render function execution	Substitute empty string or throw error based on strict mode setting	Missing content or error page
XSS vulnerability from unescaped variable	Framework cannot automatically detect	**Critical**: Always escape by default, provide explicit safe output for trusted HTML	Cross-site scripting attacks
Infinite template inheritance/render loop	Recursion depth counter or timeout	Abort rendering after maximum depth (e.g., 100), return error	Server hangs, high CPU usage
Missing template engine for extension	No engine registered for file extension	Throw error during `res.render()` call	Render fails, user sees error

Infrastructure Failure Modes

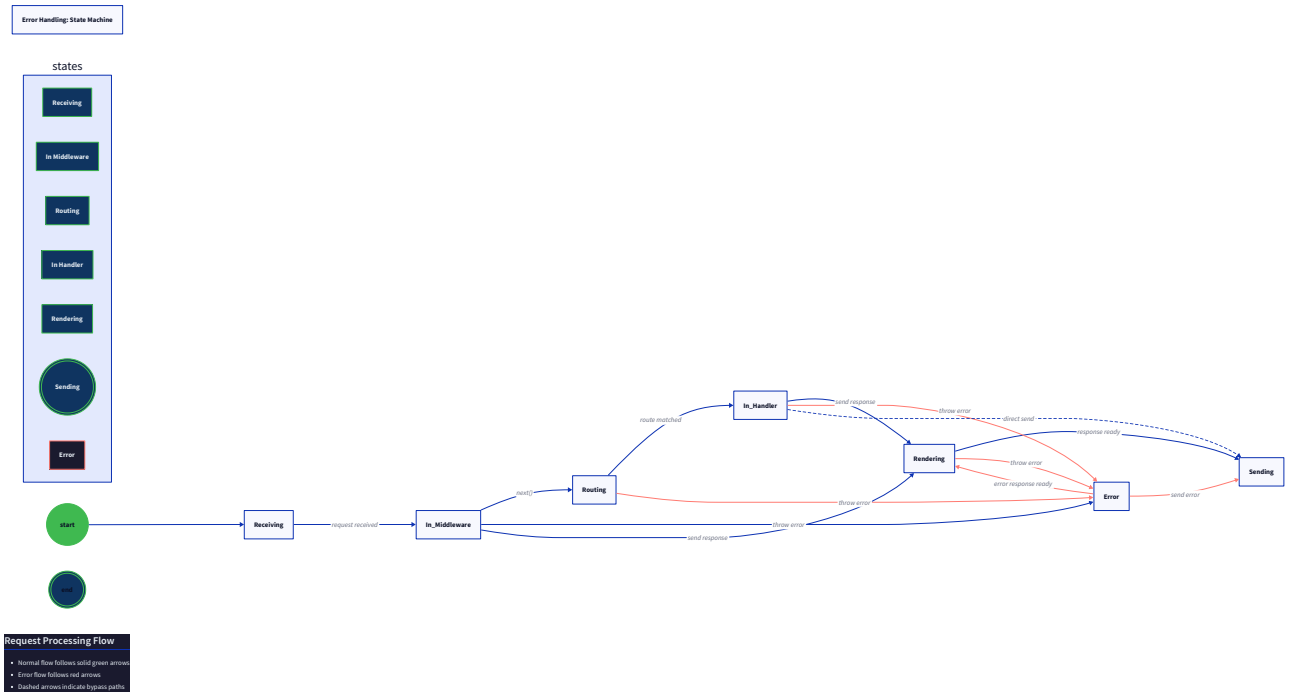
These are lower-level failures in the Node.js HTTP server or system resources.

Failure Mode	Detection Mechanism	Immediate Recovery Action	Potential Impact
Client disconnects mid-request	Request socket emits 'close' or 'error' event	Stop processing, clean up resources, abort async operations	Memory leaks, unnecessary computation
Server out of memory	Process memory monitoring or allocation failure	Cannot recover within process; graceful shutdown recommended	Process crashes, all requests fail
File system errors during template reading	`fs.readFile()` error (not ENOENT)	Pass error to error-handling middleware	Template render fails

Unified Error Recovery Strategy

> **Decision: Two-Phase Error Handling with Fallback Guarantee**
>
> - **Context**: Errors can occur at any point in the request processing pipeline, from body parsing to template rendering. We need a consistent strategy that ensures errors don't crash the server and clients receive appropriate responses.
> - **Options Considered**:
>   1. **Laissez-faire**: Let errors propagate and crash the process (Express default without error handlers)
>   2. **Component-level isolation**: Each component catches its own errors and converts to error responses
>   3. **Centralized pipeline with error mode**: Single try-catch with error-handling middleware phase
> - **Decision**: Centralized pipeline with error mode, where errors switch the pipeline to execute only error-handling middleware.
> - **Rationale**: This matches industry-standard patterns (Express, Koa), provides developer flexibility through error-handling middleware, maintains separation of concerns, and guarantees a response even for unhandled errors.
> - **Consequences**: Developers must explicitly register error-handling middleware for custom error responses; synchronous and asynchronous errors require different handling; error-handling middleware itself can throw, requiring a safety net.

The framework implements a **state machine** approach to error handling, where the request pipeline can be in either "normal" mode or "error" mode. This is visualized in the error handling state diagram:



Error Propagation Algorithm:

1. **Initial State**: Pipeline begins in "normal" mode, executing middleware and route handlers in registration order.
2. **Error Detection**:

- Synchronous errors are caught by `try/catch` in the dispatch loop
- Asynchronous errors are caught by promise rejection handlers attached to middleware return values
- Manual errors are triggered by calling `next(err)` with an error object

3. **Mode Transition**: When an error is detected:
   - The pipeline immediately switches to "error" mode
   - The current middleware stack is unwound to the nearest error handler
   - Normal middleware (3-parameter functions) are skipped
   - Only error-handling middleware (4-parameter functions with `err` first) are executed
4. **Error-Handling Middleware Execution**:

```javascript
// Error-handling middleware signature
function errorHandler(err, req, res, next) {
  // Process error, potentially calling next(err) to continue error chain
}
```

- Error handlers can process the error (log it, transform it) and either:
  - Call `next()` to exit error mode and resume normal processing (rare)
  - Call `next(err)` to pass to the next error handler
  - Send a response directly, ending the request

5. **Final Fallback**: If no error-handling middleware sends a response, a default error responder sends a 500 Internal Server Error with minimal details (detailed in development, generic in production).

Error Object Standardization:

All errors are normalized to a framework `HttpError` class with consistent properties:

Property	Type	Description
`message`	String	Human-readable error description
`status`	Number	HTTP status code (default: 500)
`stack`	String	Stack trace (development only)
`originalError`	Error	Original thrown error if wrapped
`expose`	Boolean	Whether to send `message` to client

> **Key Insight**: The `expose` flag is critical for security. Server implementation details in error messages (database errors, file paths) should never be exposed to clients in production. Framework-generated errors (404, 400) typically have `expose: true`, while unexpected errors have `expose: false`.

Specific Edge Cases and Solutions

Chunked Encoding and Malformed Headers

Problem: Node.js's `http` module handles malformed HTTP protocol elements to some degree, but edge cases exist:

- Chunked encoding with missing end chunk
- Headers exceeding Node.js limits
- Invalid character sequences in headers

Solution: Rely on Node.js's built-in validation but add defensive checks:

1. **Chunked encoding**: Monitor request completion events and timeout stalled requests:

```javascript
req.on('end', () => { /* processing */ });

req.on('close', () => { /* client disconnected, abort */ });

// Set timeout for slow chunked transfers
```

2. **Header limits**: Use Node.js's `--max-http-header-size` flag for system-wide protection.

3. **Invalid characters**: Sanitize header values when copying to `req.headers` enhanced object.

Implementation Table:

Edge Case	Detection	Action
Chunked transfer timeout	Timer after last chunk received	Destroy request socket, respond with 408 Request Timeout
Malformed chunked encoding	`req.on('error')` event	Close connection without response (client protocol violation)
Header injection attempts	Newline characters in header values	Remove or reject request with 400 Bad Request

Synchronous Errors in Async Middleware

Problem: When middleware uses async/await or returns promises, synchronous errors thrown before the async operation behave differently than promise rejections.

Solution: Wrap ALL middleware execution in a unified error-catching wrapper:

```javascript
function wrapMiddleware(fn) {
  return (req, res, next) => {
    // Call the middleware
    const result = fn(req, res, next);

    // If it returns a promise, catch rejections
    if (result && result.catch) {
      result.catch(err => next(err));
    }

    // Synchronous errors are caught by outer try/catch in dispatch
    return result;
  };
}
```

Common Pattern Pitfalls:

⚠️ **Pitfall: Missing catch on async middleware**

- **Description**: Middleware uses `async` keyword but no try/catch, and an awaited promise rejects
- **Why wrong**: Error becomes unhandled promise rejection, potentially crashing Node.js process
- **Fix**: Use the wrapper above OR always wrap async middleware body in try/catch

⚠️ **Pitfall: Calling `next()` after async operation in try block**

- **Description**:

```javascript
  try {
    await someAsync();
    next();  // If error occurs after this line, it's not caught
  } catch (err) {
    next(err);
  }
```

- **Why wrong**: Errors after `next()` call won't be caught by this middleware's catch block
- **Fix**: Call `next()` ONLY as the last operation or within the catch block

Memory Exhaustion from Large Requests

Problem: Malicious clients can send enormous request bodies to exhaust server memory.

Solution: Implement configurable size limits with streaming where possible:

ADR: Body Size Limiting Strategy

> **Decision: Stream-based parsing with early rejection**
>
> - **Context**: Request bodies can be large (file uploads, JSON payloads). We need to protect against memory exhaustion while supporting legitimate use cases.
> - **Options Considered**:
>   1. **Buffer whole body then parse**: Simple but dangerous for large requests
>   2. **Streaming parser with size limit**: Complex but memory-safe
>   3. **Defer to external middleware**: Let developers choose their own solution
> - **Decision**: Streaming parser with configurable size limit, rejecting early when limit exceeded.
> - **Rationale**: Provides safety by default while allowing configuration for specific routes. Early rejection prevents reading entire body before responding with 413.
> - **Consequences**: More complex implementation; limit must be communicated to client via Connection: close or early response.

Size Limit Algorithm:

1. Check `Content-Length` header if present:
   - If exceeds limit, immediately respond with 413 Payload Too Large
   - Close connection to prevent body transmission
2. For chunked encoding or missing Content-Length:
   - Stream body while counting bytes
   - If limit exceeded during streaming:
     - Destroy request stream
     - Respond with 413 (if headers not sent yet)
     - Otherwise, close connection
3. Configurable limits at multiple levels:

Level	Default	Override Method
Global	1 MB	`app.set('body limit', '10mb')`
Route-specific	Inherits global	`router.post('/upload', { limit: '50mb' }, handler)`
Parser-specific	Inherits parent	`app.use(json({ limit: '5mb' }))`

Cookie Security Edge Cases

Problem: Cookies have numerous security considerations: httpOnly, secure flags, sameSite policies, and domain/path scoping.

Solution: Implement secure defaults with explicit override:

Cookie Attribute Defaults Table:

Attribute	Default Value	Rationale
`httpOnly`	`true`	Prevent client-side JavaScript access (XSS protection)
`secure`	`true` if req.protocol === 'https'	Prevent transmission over unencrypted connections
`sameSite`	`'Lax'`	CSRF protection while allowing some cross-site usage
`path`	`'/'`	Apply to entire site
`maxAge`	(none)	Session cookie by default

Edge Case Handling:

1. **Multiple cookies with same name**: Later calls override earlier ones; framework should warn in development.
2. **Invalid cookie values**: Characters like semicolons, commas, or newlines must be encoded or rejected.
3. **Cookie size limits**: Browsers typically limit 4KB per cookie; framework should warn but not enforce.
4. **Setting cookies after headers sent**: Check `res.headersSent` and throw/warn.

Template Cache Poisoning

Problem: Template caching improves performance but can lead to stale content or memory growth.

Solution: LRU (Least Recently Used) cache with invalidation triggers:

Template Cache Management:

Cache Strategy	Implementation	Pros	Cons
Unlimited cache	Store all compiled templates in Map	Maximum performance	Memory grows unbounded
LRU cache	Evict least recently used templates	Memory bound, good hit rate	Cache misses after eviction
File watcher	Invalidate on file change	Always fresh content	File system overhead
Time-based TTL	Invalidate after N seconds	Simple, predictable	Stale content between invalidations

Decision: LRU cache with configurable size (default: 100 templates) and manual invalidation API.

Cache Poisoning Protection:

- Hash template source before caching to detect changes
- Separate cache keys by absolute file path
- Provide `app.clearTemplateCache()` for programmatic invalidation

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Handling	Basic try/catch with fallback 500	Error classification, structured logging, error aggregation
Body Size Limits	Fixed global limit	Per-route limits, dynamic scaling, Cost-based throttling
Async Error Handling	Promise.catch wrapper	Async-local storage for error context, APM integration
Template Cache	Simple Map with file watching	LRU cache with memory monitoring, cluster-shared cache

B. Recommended File/Module Structure

```
barebones-framework/
├── src/
│   ├── errors/                    # Error handling subsystem
│   │   ├── HttpError.js           # Base HTTP error class
│   │   ├── errorMapper.js         # Maps errors to HTTP responses
│   │   ├── errorMiddleware.js     # Default error-handling middleware
│   │   └── index.js
│   ├── middleware/
│   │   ├── bodyParser.js          # Body parsing with size limits
│   │   ├── errorWrapper.js        # Wraps middleware for error catching
│   │   └── index.js
│   ├── utils/
│   │   ├── safeJsonParse.js       # JSON parsing with error recovery
│   │   ├── headerSanitizer.js     # Header validation utilities
│   │   └── sizeLimiter.js         # Stream size limiting
│   └── (other components...)
└── test/
    └── errorScenarios.test.js     # Comprehensive error tests
```

C. Infrastructure Starter Code

Complete HttpError Class (errors/HttpError.js):

JAVASCRIPT

```javascript
/**
 * Framework HTTP error class with status code and exposure control
 */
class HttpError extends Error {
  constructor(message, status = 500, options = {}) {
    super(message);
    this.name = 'HttpError';
    this.status = status;
    this.expose = options.expose !== undefined ? options.expose : status < 500;
    this.originalError = options.originalError;

    // Capture stack trace (excluding constructor call)
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, HttpError);
    }

    // Preserve original stack if available
    if (options.originalError && options.originalError.stack) {
      this.stack = `${this.stack}\nCaused by: ${options.originalError.stack}`;
    }
  }

  /**
   * Convert error to JSON response body
   */
  toJSON() {
    const body = {
      error: this.message,
      status: this.status
    };
```

```
    // Only include stack in development
    if (process.env.NODE_ENV === 'development') {
      body.stack = this.stack;
    }

    return body;
  }
}

// Common HTTP errors as static methods
HttpError.badRequest = (msg, options) =>
  new HttpError(msg || 'Bad Request', 400, options);

HttpError.notFound = (msg, options) =>
  new HttpError(msg || 'Not Found', 404, options);

HttpError.internalServerError = (msg, options) =>
  new HttpError(msg || 'Internal Server Error', 500, options);

module.exports = HttpError;
```

Complete Safe JSON Parser (utils/safeJsonParse.js):

JAVASCRIPT

```javascript
/**
 * Safely parse JSON with error handling and size limiting
 */
function safeJsonParse(str, maxLength = null) {
  // Check length if limit provided
  if (maxLength !== null && str.length > maxLength) {
    throw new Error(`JSON string exceeds maximum length of ${maxLength} bytes`);
  }

  try {
    return JSON.parse(str);
  } catch (err) {
    if (err instanceof SyntaxError) {
      // Enhance syntax error with position info if available
      const match = err.message.match(/position (\d+)/);
      if (match) {
        const position = parseInt(match[1], 10);
        const start = Math.max(0, position - 20);
        const end = Math.min(str.length, position + 20);
        const context = str.substring(start, end);
        err.message += ` near "...${context}..."`;
      }
      throw err;
    }
    throw err;
  }
}

module.exports = safeJsonParse;
```

D. Core Logic Skeleton Code

Error-Handling Middleware Wrapper (middleware/errorWrapper.js):

```javascript
/**
 * Wraps middleware functions to ensure errors are properly caught and forwarded
 * @param {Function} fn - Middleware function (async or sync)
 * @returns {Function} Wrapped middleware
 */
function wrapMiddleware(fn) {
  // TODO 1: Return a new function with signature (req, res, next)
  // TODO 2: Inside the wrapper, call the original middleware fn(req, res, next)
  // TODO 3: If fn returns a Promise, attach a .catch() handler that calls next(err)
  // TODO 4: Return the result (promise or value) from the wrapper
  // TODO 5: Note: Synchronous errors thrown directly will be caught by the outer dispatch loop
}

module.exports = wrapMiddleware;
```

Default Error-Handling Middleware (errors/errorMiddleware.js):

```javascript
/**
 * Default error handler - always registered last in the middleware chain
 * @param {Error} err - The error object
 * @param {Request} req - Enhanced request object
 * @param {Response} res - Enhanced response object
 * @param {Function} next - Next function (not typically called from final handler)
 */
function defaultErrorHandler(err, req, res, next) {
  // TODO 1: Normalize error - if not HttpError, wrap it as 500 Internal Server Error
  // TODO 2: Log the error with appropriate level (console.error or structured logger)
  // TODO 3: Determine if error should be exposed to client (based on expose flag and NODE_ENV)
  // TODO 4: Set response status code from error.status or default to 500
  // TODO 5: Send error response as JSON or HTML based on Accept header
  // TODO 6: For JSON: res.status(err.status).json({ error: exposedMessage, ... })
  // TODO 7: For HTML: render error template or send plain text
  // TODO 8: Ensure headers haven't already been sent before attempting to send error
}

module.exports = defaultErrorHandler;
```

Body Parser with Size Limits (middleware/bodyParser.js):

JAVASCRIPT

```javascript
/**
 * JSON body parser middleware factory with size limiting
 * @param {Object} options - Parser options
 * @param {string|number} options.limit - Maximum body size (e.g., '1mb', 1024)
 * @returns {Function} Middleware function
 */
function jsonBodyParser(options = {}) {
  const limit = parseSizeLimit(options.limit || '1mb');

  return function jsonParser(req, res, next) {
    // TODO 1: Check Content-Type header matches application/json (or variants)
    // TODO 2: If no body or wrong content-type, skip parsing (call next())
    // TODO 3: Check Content-Length header if present - reject immediately if over limit
    // TODO 4: Collect body chunks from req stream with byte counting
    // TODO 5: If byte count exceeds limit during streaming:
    //   - Destroy request stream
    //   - Send 413 Payload Too Large response
    //   - Do NOT call next()
    // TODO 6: After body complete, parse JSON using safeJsonParse utility
    // TODO 7: If parse succeeds, set req.body = parsedObject and call next()
    // TODO 8: If parse fails, call next(err) with HttpError.badRequest()
  };
}

/**
 * Parse size string (e.g., '1mb', '50kb') to bytes
 */
function parseSizeLimit(size) {
  // TODO 1: If size is number, return as-is
  // TODO 2: Parse string like '1mb' or '50kb'
```

```
  // TODO 3: Support units: b, kb, mb, gb (case-insensitive)
  // TODO 4: Return number of bytes
}

module.exports = jsonBodyParser;
```

E. Language-Specific Hints

1. **Async/Await Error Propagation**: Always wrap async middleware in try/catch OR use the wrapper function to catch promise rejections.
2. **Stream Backpressure**: When implementing size limits, respect stream backpressure with `.pause()` and `.resume()` to prevent memory buildup.
3. **Error Stack Traces**: Use `Error.captureStackTrace(this, constructor)` in custom error classes for cleaner stack traces.
4. **Memory Monitoring**: Consider using `process.memoryUsage()` in development to warn about potential memory leaks from caching or large requests.
5. **Header Timing**: Check `res.headersSent` before setting any headers or cookies to avoid "Can't set headers after they are sent" errors.

F. Milestone Checkpoint

Error Handling Validation Test:

```bash
# Run error scenario tests
npm test -- --testNamePattern="error handling"

# Expected output should include:
# ✓ returns 404 for unmatched routes
# ✓ catches synchronous errors in middleware
# ✓ catches asynchronous errors in middleware
# ✓ handles malformed JSON with 400
# ✓ respects body size limits with 413
# ✓ renders error template for 500 errors
```

Manual Verification Steps:

1. **Test error recovery**:

```bash
# Start server
node examples/server.js



# Trigger various errors
curl http://localhost:3000/nonexistent        # Should return 404
curl -X POST http://localhost:3000/api/data \
  -H "Content-Type: application/json" \
  -d '{"invalid: json}'                       # Should return 400
```

2. **Verify error details only in development**:
   - In development (`NODE_ENV=development`): Errors should include stack traces
   - In production (`NODE_ENV=production`): Errors should be generic
3. **Check memory safety**:
   - Send request with `Content-Length: 1000000000` (1GB) - should immediately reject with 413
   - Monitor memory usage doesn't spike during large upload attempts

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Request hangs indefinitely	Middleware not calling `next()`	Add logging middleware before/after suspected middleware	Ensure all code paths call `next()` or send response
"Can't set headers after they are sent"	Multiple responses sent	Add middleware to log when `res.send()` is called	Ensure only one response sent per request; check for double `next()` calls
Error disappears in production	Error exposed only in development	Check `err.expose` flag and `NODE_ENV`	Ensure production errors are logged somewhere (file, service)
Memory grows unbounded	Template cache without limits or large request accumulation	Monitor `process.memoryUsage()`	Implement LRU cache for templates; add request size limits
Async errors crash process	Unhandled promise rejection in middleware	Add `process.on('unhandledRejection', ...)` handler	Use `wrapMiddleware()` utility to catch all promise rejections
Cookies not set on HTTPS	Secure flag automatically set	Check `req.protocol` and cookie options	Explicitly set `secure: false` for development HTTP
Template shows raw {{variable}}	Template compilation error cached	Check template cache for error entries	Implement cache invalidation on compilation error

Testing Strategy

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

A robust testing strategy is the **safety net** that allows us to build a complex framework with confidence. Think of testing as the **flight simulator** for our framework—it lets us simulate thousands of HTTP requests, edge cases, and failure scenarios in milliseconds, without needing to manually curl endpoints or run a server. This section outlines a pragmatic, multi-layered approach to verifying that each component of the Barebones framework works correctly both in isolation and when integrated together.

Our testing philosophy follows three core principles:

1. **Verify behaviors, not implementations**: Tests should check that components produce the correct outputs for given inputs, not how they achieve it internally.
2. **Fail fast and informatively**: When tests fail, they should clearly indicate what broke, why, and how to fix it.
3. **Progressively build confidence**: Start with unit tests for individual functions, then integration tests for component interactions, and finally end-to-end tests for complete request flows.

This strategy is designed to align with the milestone structure, providing concrete checkpoints that validate each capability as you build it. By the end, you'll have a comprehensive test suite that can catch regressions and give you confidence to extend the framework.

Testing Approach and Tools

Mental Model: The Surgical Instrument Sterilization

Think of testing our framework components like sterilizing surgical instruments before an operation. Each instrument (component) must be individually cleaned and tested (unit tests), then we verify they work together in a tray (integration tests), and finally we simulate the entire surgical procedure (end-to-end tests). Just as we wouldn't use an unsterilized scalpel, we shouldn't deploy framework code without verifying it handles HTTP requests correctly. The testing tools are our autoclave and monitoring equipment—they apply systematic heat and pressure to ensure everything is safe for production-like conditions.

Testing Stack Recommendation

For a Node.js-based framework, we recommend the following testing stack:

Component	Recommended Tool	Alternative	Rationale
Test Runner	Jest 29+	Mocha + Chai	Jest provides built-in assertions, mocking, coverage, and parallel test execution out of the box, reducing setup complexity
HTTP Mocking	Node.js `http` module mock	`nock` library	For unit testing, we can create mock `IncomingMessage` and `ServerResponse` objects directly without external dependencies
Test Structure	Jest's built-in `describe`/`it`	Mocha's `describe`/`it`	The syntax is nearly identical, but Jest's integrated approach simplifies configuration
Coverage Reporting	Jest's `--coverage`	`nyc` (Istanbul)	Built-in coverage reduces toolchain complexity for learners

Architecture Decision Record: Testing Framework Selection

Decision: Use Jest as the primary testing framework

- **Context**: We need a testing solution that is beginner-friendly, requires minimal configuration, and provides comprehensive features out of the box for testing asynchronous middleware and HTTP handlers.
- **Options Considered**:
  1. **Jest**: Full-featured testing framework with built-in assertions, mocking, parallel execution, and code coverage
  2. **Mocha + Chai + Sinon**: Modular approach with separate libraries for test runner, assertions, and mocks
  3. **Node.js built-in `assert` with custom test runner**: Minimal dependencies but requires significant boilerplate
- **Decision**: Use Jest as the primary testing framework
- **Rationale**:
  - **Beginner-friendly**: Zero configuration required for most use cases
  - **Integrated mocking**: Built-in function and module mocking simplifies testing middleware in isolation
  - **Async support**: Excellent support for testing Promise-based and async/await code
  - **Snapshot testing**: Useful for verifying template engine output
  - **Watch mode**: Automatic test re-run on file changes accelerates development
- **Consequences**:
  - Slightly heavier dependency than Mocha (but acceptable for a learning project)
  - Some magic in mocking behavior that may obscure how mocking works internally
  - Encourages test organization patterns that scale well

Testing Strategy by Component Type

Each component type in the framework requires a different testing approach. The table below outlines the primary techniques:

Component Type	Testing Focus	Key Techniques	Mocking Strategy
Router	Pattern matching, parameter extraction, method dispatch	Unit tests for `TrieNode.insert()` and `TrieNode.search()`, integration tests for route registration	Mock handler functions that record calls; no HTTP mocks needed for core logic
Middleware	Pipeline order, `next()` propagation, error handling	Isolation tests with mock `req` / `res` / `next`, integration tests for chain execution	Mock `next()` function to verify it's called (or not), mock `req` / `res` objects
Request/Response Enhancements	Property addition, helper method behavior, parsing correctness	Unit tests for individual parsers and helpers, integration tests with real HTTP objects	Use real Node.js `IncomingMessage` and `ServerResponse` streams with controlled inputs
Template Engine	Variable interpolation, HTML escaping, control flow, inheritance	Snapshot tests for rendered output, unit tests for compilation functions	Mock filesystem for template file loading; test with in-memory template strings

Mocking HTTP Objects Strategy

Testing HTTP-related code requires creating mock request and response objects that simulate Node.js's native `http.IncomingMessage` and `http.ServerResponse`. We recommend a layered approach:

1. **Simple object mocks for unit tests**: Plain JavaScript objects with the minimal properties needed
2. **Enhanced mocks for middleware tests**: Objects that implement event emitters for `'data'` and `'end'` events to simulate request bodies
3. **Real wrapper objects for integration tests**: Use the framework's own `Request` and `Response` wrappers with mocked native objects

The table below shows the mock object structure for different testing scenarios:

Test Scenario	Request Mock	Response Mock	How to Verify
Router unit test	`{ method: 'GET', url: '/users/42' }`	`{ writeHead: jest.fn(), end: jest.fn() }`	Check handler was called with correct params
Middleware isolation	`{ method: 'POST', url: '/api/data', on: jest.fn(), pipe: jest.fn() }`	`{ setHeader: jest.fn(), end: jest.fn(), statusCode: 200 }`	Verify `next()` called, headers set, response ended
Body parser integration	Real `IncomingMessage` mock with `'data'` / `'end'` events	`ServerResponse` mock with spy methods	Check `req.body` contains parsed data
Template engine render	`{ app: { settings: { views: '/tmp' } } }`	`{ render: jest.fn(), send: jest.fn() }`	Verify output matches expected HTML

Test Organization and Structure

Organize tests mirroring the source code structure to make it easy to find and maintain tests:

```
barebones-framework/
├── src/
│   ├── application.js      # Main Application class
│   ├── router/             # Router component
│   │   ├── index.js
│   │   ├── trie.js         # TrieNode implementation
│   │   └── layer.js        # Layer class
│   ├── middleware/         # Built-in middleware
│   │   ├── index.js
│   │   ├── json.js         # JSON body parser
│   │   └── logger.js       # Logging middleware
│   ├── template/           # Template engine
│   │   ├── engine.js
│   │   └── compiler.js
│   └── utils/              # Utilities
│       └── http-error.js
└── tests/                  # Test files mirror source structure
    ├── application.test.js
    ├── router/
    │   ├── trie.test.js
    │   ├── layer.test.js
    │   └── router.test.js
    ├── middleware/
    │   ├── json.test.js
    │   └── logger.test.js
    ├── template/
    │   ├── engine.test.js
    │   └── compiler.test.js
    ├── integration/        # Cross-component tests
    │   ├── routing.test.js
    │   └── middleware-chain.test.js
    └── helpers/            # Test utilities
        └── mock-http.js
```

Common Testing Pitfalls

⚠️ Pitfall: Testing implementation details instead of behavior

- **Description**: Writing tests that break when you refactor internal code, even though the external behavior remains the same
- **Why it's wrong**: Makes refactoring difficult and tests brittle; tests should verify *what* the code does, not *how* it does it
- **How to fix**: Test through public APIs only. For example, test the router by registering routes and checking if the correct handler is called for a given path, not by inspecting the internal trie structure.

⚠️ Pitfall: Not testing error cases

- **Description**: Only testing the "happy path" where everything works correctly
- **Why it's wrong**: Real-world usage will encounter errors—malformed requests, missing routes, template syntax errors—and the framework must handle them gracefully
- **How to fix**: For every feature, write tests for: invalid inputs, edge cases, and failure scenarios. Test that error middleware catches thrown errors.

⚠️ **Pitfall: Over-mocking leading to meaningless tests**

- **Description**: Mocking so many dependencies that the test doesn't actually verify real behavior
- **Why it's wrong**: Tests pass even when the real integration would fail, giving false confidence
- **How to fix**: Use integration tests for component interactions. Mock only external dependencies (like filesystem or network), not internal framework components.

⚠️ **Pitfall: Not cleaning up test state**

- **Description**: Tests that modify global state (like the `Application` singleton or module caches) and affect other tests
- **Why it's wrong**: Tests become order-dependent and flaky, failing when run in different orders or isolation
- **How to fix**: Use Jest's `beforeEach` / `afterEach` to reset state. Create fresh `Application` instances for each test rather than reusing a global instance.

Milestone Verification Checkpoints

Each milestone has specific acceptance criteria that must be verified through tests. Below are concrete checkpoints for each milestone, including test commands and expected outputs that validate the core functionality.

Milestone 1: Routing Verification

Objective: Verify that the router correctly matches URL patterns, extracts parameters, and dispatches to the appropriate handler based on HTTP method.

Test Category	What to Test	Example Test Case	Expected Outcome
Basic Route Registration	Routes can be registered for different HTTP methods	Register `app.get('/users', handler)`	Handler should be called for GET /users
URL Parameter Extraction	Named parameters (`:id`) extract values from path	Route `/users/:id` for path `/users/42`	`req.params` should be `{ id: '42' }`
Route Matching Priority	Static routes match before parameter routes	Routes `/users/new` and `/users/:id`	`/users/new` should match first
404 for No Match	Unmatched routes return 404	Request to `/nonexistent`	Response status should be 404
Route Groups/Prefixing	Routes can be grouped under a common prefix	Router with prefix `/api` , route `/users`	Should match `/api/users`

Concrete Verification Checkpoint:

Run the routing test suite:

```bash
npm test -- --testPathPattern=router
```

Expected Output:

```
PASS tests/router/router.test.js
  Router
    ✓ should match static routes (5ms)
    ✓ should extract URL parameters (3ms)
    ✓ should handle multiple parameters in path (2ms)
    ✓ should return null for no match (1ms)
    ✓ should prioritize static over dynamic routes (2ms)
    ✓ should handle route prefixes (2ms)

PASS tests/router/trie.test.js
  TrieNode
    ✓ should insert static path segments (4ms)
    ✓ should insert parameter segments (3ms)
    ✓ should search and extract params (3ms)

Test Suites: 2 passed, 2 total
Tests: 9 passed, 9 total
```

Manual Verification Steps:

1. Create a simple test server with registered routes
2. Use curl to test various paths:

```bash
curl http://localhost:3000/users/42
```

3. Verify the response contains the extracted ID and correct handler output

Milestone 2: Middleware Verification

Objective: Verify that middleware executes in order, that `next()` passes control, and that errors are properly caught and handled.

Test Category	What to Test	Example Test Case	Expected Outcome
Middleware Order	Middleware executes in registration order	Register logging, then auth middleware	Logging runs first, then auth
next() Control Flow	Calling `next()` passes to next middleware	Middleware that calls `next()`	Next middleware in chain executes
Error Middleware	4-argument middleware catches errors	Throw error in regular middleware	Error middleware should receive error
Async Middleware	Async functions work correctly	Async middleware that awaits	Should complete before next middleware
Early Response	Middleware can end response early	Authentication middleware that rejects	Response sent, later middleware skipped

Concrete Verification Checkpoint:

Run the middleware test suite:

```bash
npm test -- --testPathPattern=middleware
```

Expected Output:

```
PASS tests/middleware/chain.test.js
  Middleware Chain
    ✓ should execute middleware in order (8ms)
    ✓ should pass control with next() (4ms)
    ✓ should skip remaining middleware after response ends (5ms)
    ✓ should catch sync errors with error middleware (6ms)
    ✓ should catch async errors with error middleware (7ms)

PASS tests/middleware/logger.test.js
  Logger Middleware
    ✓ should log request method and path (3ms)
    ✓ should log response status and duration (4ms)

Test Suites: 2 passed, 2 total
Tests: 7 passed, 7 total
```

Manual Verification Steps:

1. Create a test app with multiple middleware functions
2. Add console.log statements to each middleware
3. Make a request and verify the console output shows correct order
4. Test error flow by throwing an error in middleware and verifying error handler is called

Milestone 3: Request/Response Enhancement Verification

Objective: Verify that request parsing (body, query, cookies) works correctly and response helpers set appropriate headers and content.

Test Category	What to Test	Example Test Case	Expected Outcome
JSON Body Parsing	Request body parsed as JSON	POST with `{"name":"John"}`	`req.body` should equal `{name:"John"}`
Query Parameter Parsing	URL query string parsed	`/search?q=test&page=1`	`req.query` should equal `{q:"test",page:"1"}`
Cookie Parsing	Cookie header parsed	Cookie: `session=abc123`	`req.cookies` should equal `{session:"abc123"}`
Response Helpers	`res.json()` sets correct headers	`res.json({ok:true})`	Content-Type: application/json, status 200
Cookie Setting	`res.cookie()` sets Set-Cookie header	`res.cookie('token','xyz')`	Response includes `Set-Cookie: token=xyz`

Concrete Verification Checkpoint:

Run the request/response enhancement test suite:

```bash
npm test -- --testPathPattern="(json|query|cookie|response)"
```

Expected Output:

```
PASS tests/middleware/json.test.js
  JSON Body Parser
    ✓ should parse JSON body (6ms)
    ✓ should handle empty body (2ms)
    ✓ should reject malformed JSON with 400 (5ms)
    ✓ should respect size limit (7ms)

PASS tests/request.test.js
  Enhanced Request
    ✓ should parse query parameters (3ms)
    ✓ should parse cookies (4ms)
    ✓ should expose original URL (1ms)

PASS tests/response.test.js
  Enhanced Response
    ✓ should send JSON with correct headers (4ms)
    ✓ should set status code (2ms)
    ✓ should set cookies with options (5ms)
    ✓ should redirect with status code (3ms)

Test Suites: 3 passed, 3 total
Tests: 12 passed, 12 total
```

Manual Verification Steps:

1. Test JSON parsing:

```bash
curl -X POST -H "Content-Type: application/json" -d '{"test":true}' http://localhost:3000/api
```

2. Verify server receives parsed object
3. Test response helpers by checking response headers:

```bash
curl -I http://localhost:3000/api/json
```

4. Verify `Content-Type: application/json` is present

Milestone 4: Template Engine Verification

Objective: Verify that templates correctly interpolate variables, escape HTML, support control flow, and implement inheritance.

Test Category	What to Test	Example Test Case	Expected Outcome
Variable Interpolation	`{{variable}}` replaced with value	Template `Hello {{name}}` with `{name:"World"}`	Output: `Hello World`
HTML Escaping	Variables escaped by default	`{{userInput}}` with `<script>`	Output: `&lt;script&gt;`
Conditional Blocks	`{{#if condition}}` sections render conditionally	Conditional with truthy/falsy values	Section renders only when condition true
Loop Blocks	`{{#each items}}` iterates arrays	Array of items	Each item rendered in loop
Template Inheritance	Child extends base template	Base with `{{{body}}}`, child provides content	Child content inserted into base layout

Concrete Verification Checkpoint:

Run the template engine test suite:

```bash
npm test -- --testPathPattern=template
```

Expected Output:

```
PASS tests/template/compiler.test.js
  Template Compiler
    √ should interpolate simple variables (4ms)
    √ should escape HTML by default (3ms)
    √ should support triple braces for raw output (3ms)
    √ should compile if/else conditionals (5ms)
    √ should compile each loops (6ms)

PASS tests/template/engine.test.js
  Template Engine
    √ should render template files from views directory (8ms)
    √ should handle template inheritance (10ms)
    √ should cache compiled templates (7ms)
    √ should throw for missing templates (4ms)

Test Suites: 2 passed, 2 total
Tests: 9 passed, 9 total
Snapshots: 5 passed, 5 total
```

Manual Verification Steps:

1. Create a template with variables and control structures
2. Render it with sample data:

```javascript
const html = app.render('welcome', { username: 'Alice', items: ['a', 'b'] });
```

3. Verify the output contains correctly interpolated values
4. Test HTML escaping by passing a script tag as data and verifying it's escaped in output

5. Test inheritance by creating a base layout and child template, verifying child content appears within layout

End-to-End Integration Verification

After completing all milestones, run the full test suite to verify integration:

```bash
npm test
```

Expected Final Output:

```
PASS tests/application.test.js
PASS tests/router/router.test.js
PASS tests/middleware/chain.test.js
PASS tests/template/engine.test.js
PASS tests/integration/full-request.test.js

Test Suites: 5 passed, 5 total
Tests: 35 passed, 35 total
Snapshots: 5 passed, 5 total
Time: 1.234s
```

This comprehensive test suite validates that all components work together correctly, giving you confidence that the framework handles real HTTP requests as designed.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Test Runner	Jest (zero config)	Mocha + Chai + Sinon (modular control)
HTTP Mocks	Custom mock objects	`nock` for external HTTP, `supertest` for integration
Code Coverage	Jest's built-in coverage	`nyc` (Istanbul) with custom reporters
Test Utilities	Custom helper functions	Test double library like `testdouble.js`

B. Recommended File/Module Structure for Tests:

```
tests/
├── helpers/
│   ├── mock-http.js          # Reusable HTTP mock utilities
│   └── test-utils.js         # General test helpers
├── unit/
│   ├── router/
│   │   ├── trie.test.js
│   │   └── router.test.js
│   ├── middleware/
│   │   ├── json.test.js
│   │   └── logger.test.js
│   └── template/
│       ├── compiler.test.js
│       └── engine.test.js
├── integration/
│   ├── routing-middleware.test.js
│   ├── request-response.test.js
│   └── template-rendering.test.js
└── e2e/
    ├── basic-app.test.js
    └── error-scenarios.test.js
```

C. Infrastructure Starter Code (Complete HTTP Mock Utilities):

JAVASCRIPT

```javascript
// tests/helpers/mock-http.js

/**
 * Creates a mock IncomingMessage object for testing
 * @param {Object} options - Request options
 * @returns {Object} Mock request object
 */
function createMockRequest(options = {}) {
  const {
    method = 'GET',
    url = '/',
    headers = {},
    body = null,
    socket = { remoteAddress: '127.0.0.1' }
  } = options;

  const req = {
    method,
    url,
    headers,
    socket,
    on: jest.fn((event, handler) => {
      if (event === 'data' && body) {
        // Simulate data events if body provided
        if (typeof body === 'string') {
          handler(Buffer.from(body));
        } else if (Buffer.isBuffer(body)) {
          handler(body);
        }
      }
```

```
      if (event === 'end') {
        handler();
      }
      return req;
    }),
    pipe: jest.fn(),
    resume: jest.fn()
  };

  return req;
}

/**
 * Creates a mock ServerResponse object for testing
 * @returns {Object} Mock response object
 */
function createMockResponse() {
  const res = {
    statusCode: 200,
    headers: {},
    body: null,

    setHeader: jest.fn((name, value) => {
      res.headers[name.toLowerCase()] = value;
      return res;
    }),

    getHeader: jest.fn((name) => res.headers[name.toLowerCase()]),

    writeHead: jest.fn((statusCode, statusMessage, headers) => {
```

```
    res.statusCode = statusCode;
    if (typeof statusMessage === 'object') {
      headers = statusMessage;
    }
    if (headers) {
      Object.keys(headers).forEach(key => {
        res.setHeader(key, headers[key]);
      });
    }
    return res;
  }),

  end: jest.fn((data, encoding, callback) => {
    res.body = data || '';
    if (typeof encoding === 'function') {
      callback = encoding;
    }
    if (callback) {
      callback();
    }
    return res;
  }),

  write: jest.fn((data) => {
    res.body = (res.body || '') + data;
    return true;
  })
};

return res;
```

```
}

/**
 * Creates a mock next function for middleware testing
 * @returns {Function} Mock next function
 */
function createMockNext() {
  return jest.fn();
}

module.exports = {
  createMockRequest,
  createMockResponse,
  createMockNext
};
```

D. Core Logic Skeleton Code for Tests:

JAVASCRIPT

```javascript
// tests/router/router.test.js - Skeleton with TODOs

const { Router } = require('../../src/router');
const { createMockRequest, createMockResponse } = require('../helpers/mock-http');

describe('Router', () => {
  let router;

  beforeEach(() => {
    // TODO 1: Create a new Router instance before each test
    router = new Router();
  });

  test('should match static routes', () => {
    const handler = jest.fn();

    // TODO 2: Register a static route
    router.get('/users', handler);

    const req = createMockRequest({ method: 'GET', url: '/users' });
    const res = createMockResponse();

    // TODO 3: Call router.match() and verify it returns the correct route
    const match = router.match('GET', '/users');

    // TODO 4: Verify the handler would be called with correct params
    expect(match).not.toBeNull();
    expect(match.handlers).toContain(handler);
  });

  test('should extract URL parameters', () => {
```

```
    const handler = jest.fn();

    // TODO 5: Register a route with a parameter
    router.get('/users/:id', handler);

    // TODO 6: Match a path that should extract the parameter
    const match = router.match('GET', '/users/42');

    // TODO 7: Verify params are correctly extracted
    expect(match).not.toBeNull();
    expect(match.params).toEqual({ id: '42' });
  });

  test('should return null for no match', () => {
    // TODO 8: Test that non-existent route returns null
    const match = router.match('GET', '/nonexistent');

    expect(match).toBeNull();
  });

  // TODO 9: Add more tests for:
  // - Multiple parameters in path
  // - Route precedence (static before dynamic)
  // - Different HTTP methods
  // - Route prefixes
});
```

E. Language-Specific Hints for JavaScript/Node.js:

1. **Jest Mocking**: Use `jest.fn()` to create mock functions and `jest.spyOn()` to spy on existing methods. Reset mocks between tests with `jest.clearAllMocks()` in `beforeEach`.
2. **Async Testing**: For testing async middleware, use `async/await` in test functions:

```javascript
test('async middleware', async () => {
  const middleware = async (req, res, next) => {
    await someAsyncOperation();
    next();
  };

  const next = jest.fn();
  await middleware({}, {}, next);
  expect(next).toHaveBeenCalled();
});
```

3. **Snapshot Testing**: Use Jest snapshot testing for template output:

```javascript
test('template renders correctly', () => {
  const html = templateEngine.render('welcome', { name: 'World' });
  expect(html).toMatchSnapshot(); // First run creates snapshot
});
```

4. **Testing Event Emitters**: For testing request body parsing, simulate `'data'` and `'end'` events:

```javascript
const req = createMockRequest();
setTimeout(() => {
  req.emit('data', Buffer.from('{"test":true}'));
  req.emit('end');
}, 0);
```

F. Milestone Checkpoint Commands:

Add these scripts to your `package.json`:

```json
{
  "scripts": {
    "test": "jest",
    "test:router": "jest --testPathPattern=router",
    "test:middleware": "jest --testPathPattern=middleware",
    "test:reqres": "jest --testPathPattern=\"(json|query|cookie|response)\"",
    "test:template": "jest --testPathPattern=template",
    "test:coverage": "jest --coverage",
    "test:watch": "jest --watch"
  }
}
```

G. Debugging Tips for Test Failures:

Symptom	Likely Cause	How to Diagnose	Fix
Router test fails intermittently	Tests sharing router state	Check if router instance is reused between tests	Create fresh router in `beforeEach`
Middleware test hangs	`next()` not called	Add logging to see which middleware is stuck	Ensure all code paths call `next()` or end response
Template snapshot mismatch	Whitespace differences	Compare actual vs expected output character by character	Use `.trim()` on output or update snapshot
Async test timeout	Promise never resolves	Check for uncaught exceptions in async code	Add `.catch()` to promises, use `async/await`
"Headers already sent" error	Multiple calls to `res.end()`	Trace which middleware is sending response	Ensure only one middleware sends final response

---

Debugging Guide

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4

Building a web framework from scratch involves composing multiple interacting components—routing, middleware, request/response enhancement, and templating. When these components don't work harmoniously, debugging can feel like finding a needle in a haystack. This guide provides a systematic, practical approach to diagnosing and fixing the most common implementation bugs. Think of debugging your framework as being a **detective investigating a crime scene**:

you have clues (symptoms), potential suspects (causes), forensic tools (diagnosis techniques), and solutions (fixes). We'll organize this investigation by symptom and provide framework-specific forensic techniques.

Symptom → Cause → Fix Table

The following table catalogs the most common issues you'll encounter while implementing each milestone. Each entry follows the pattern: observe the symptom, identify the most likely causes, perform targeted diagnosis steps, and apply the appropriate fix.

Symptom	Likely Cause	Diagnosis Steps	Fix
Request hangs indefinitely (no response)	1. Middleware not calling `next()` 2. Route handler not sending response 3. Infinite loop in middleware chain 4. Async operation without `await` or `.catch()`	1. Add logging middleware at start/end of chain to track progression 2. Check if any middleware lacks a `next()` call in all code paths 3. Verify route handler calls `res.send()`, `res.json()`, or `res.end()` 4. Look for recursive `next()` calls or circular middleware dependencies	1. Ensure every middleware calls `next()` unless terminating the request 2. Add a timeout middleware that sends 504 after N seconds 3. Wrap async middleware with `try/catch` and call `next(err)` on error 4. Implement a safeguard in `router.dispatch()` to detect infinite loops (max call count)
Route returns 404 when it should match	1. Route pattern typo or incorrect method 2. Route registered after catch-all or parameter route 3. Trailing slash mismatch 4. Route not actually registered (code path not executed) 5. Trie insertion bug	1. Log all registered routes on startup (method and path) 2. Check route ordering—specific routes must come before parameterized ones 3. Compare request URL (including query string) with route pattern 4. Inspect the trie structure to see if node exists for each segment 5. Test with a simple static route to isolate issue	1. Normalize paths (strip query, handle trailing slashes) before matching 2. Reorder route registration: static → parameter → catch-all 3. Use `router.match(method, path)` in a test to debug matching logic 4. Fix `TrieNode.insert()` to correctly handle parameter nodes and static children
URL parameters (`req.params`) empty or wrong	1. Parameter name mismatch between pattern and extraction 2. Trie search not populating params object 3. Overwritten params by multiple middleware/routes	1. Log the `params` object at each stage (middleware, route handler) 2. Verify `TrieNode.search()` returns params with correct keys 3. Check if another middleware modifies `req.params` (unlikely in our design)	1. Ensure `Layer` or `TrieNode` stores param keys (`:id` → `id`) and extracts values 2. In `router.match()`, pass a fresh object for each search 3. Use `Object.assign` or spread to merge params from multiple layers if supporting nested routers
Middleware executes in wrong order or skips	1. Registration order incorrect 2. Middleware mounted at specific path not matching 3. Error mode triggered, skipping normal middleware	1. Log middleware registration (path and function name) 2. Add phase tracking to each middleware to log execution 3. Check if an error was passed to `next()` (entering error mode)	1. Register middleware in intended sequence (global first, then route-specific) 2. Ensure path prefix matching in `Layer.match()` works correctly 3. Separate error-handling middleware (4 args) and place it after normal middleware
`req.body` is undefined or empty	1. Body parser middleware not registered 2. Content-Type header missing or incorrect 3. Body larger than size	1. Check middleware chain includes `jsonBodyParser()` or similar 2. Log	1. Register body parser early in middleware chain (after logging) 2. Implement fallback parsing for missing Content-Type 3. Increase size limit or implement

Symptom	Likely Cause	Diagnosis Steps	Fix
	limit 4. `req` stream already consumed (e.g., by another parser)	`req.headers['content-type']` and compare with parser expectations 3. Verify `req.on('data')` events fire (add listener before parser) 4. Check for error events on `req` stream	streaming for large bodies 4. Ensure parser doesn't attach multiple listeners to same stream
`res.json()` sends plain text or malformed JSON	1. Missing or incorrect `Content-Type: application/json` 2. `JSON.stringify()` throws on circular references 3. Headers already sent when `json()` called 4. Multiple response calls (e.g., `send()` after `json()`)	1. Inspect response headers with `res.getHeaders()` 2. Wrap `JSON.stringify()` in try/catch, call `next(err)` 3. Check if any previous middleware wrote to `res` 4. Add flag to `Response` to track if response sent	1. Set header before writing body: `res.setHeader('Content-Type', 'application/json')` 2. Use safe serialization (e.g., `JSON.stringify(val, null, 2)`) 3. Implement sentinel flag `res.headersSent` and throw if true 4. Ensure `res.json()` calls `res.end()` internally
Cookies not set in browser	1. `Set-Cookie` header not formatted correctly 2. Secure cookie sent over HTTP 3. Domain/path restrictions prevent browser storage 4. Headers sent before cookie set	1. Log `res.getHeaders()` to see `Set-Cookie` header value 2. Verify cookie options (httpOnly, secure, sameSite) match environment 3. Test with simple cookie first: `name=value` without options	1. Format header as `name=value; Option1=value1; Option2=value2` 2. Only set `secure: true` in production (HTTPS) 3. Use `res.cookie()` before any `res.send()` or `res.end()` 4. Implement cookie store in `Response` that writes headers at send time
Template renders empty or raw syntax	1. Template file not found 2. Context variables not passed to render function 3. Template compilation error (silent) 4. HTML escaping removing content	1. Log template path resolution and file existence 2. Dump context object passed to `Template.renderFile()` 3. Check compiled function for errors (wrap eval in try/catch) 4. Inspect output for `<` instead of `<`	1. Set default `views` directory with `app.set('views', './views')` 2. Provide default empty object `{}` for context 3. Cache compiled functions and log compilation errors 4. Use unescaped syntax for trusted content (e.g., `{{{html}}}`) if implemented
Template inheritance fails (blocks not replaced)	1. Base template not found or not compiled 2. Block names mismatch between base and child	1. Log block map from both base and child templates 2. Check that child's blocks are collected during compilation	1. Implement two-pass compilation: parse child, locate base, merge blocks 2. Store blocks in

Symptom	Likely Cause	Diagnosis Steps	Fix
	3. Child template doesn't call `{% block %}` 4. Render order: child then base, instead of merging	3. Verify inheritance directive syntax (`{% extends "base.html" %}`)	`Template.blocks` map during compilation 3. During render, replace base template block placeholders with child content
Error middleware not triggered	1. Error not passed to `next(err)` 2. Error middleware registered before error occurs 3. Synchronous error not caught (e.g., in async handler) 4. Error middleware has wrong signature (not 4 args)	1. Add logging to `next()` to see if called with error argument 2. Check middleware order—error handlers should be last 3. Look for uncaught promise rejections (listen to `unhandledRejection`) 4. Verify error middleware function has `(err, req, res, next)`	1. Always call `next(err)` in catch blocks 2. Register error-handling middleware after all normal middleware 3. Wrap all middleware with `wrapMiddleware()` to catch async errors 4. Use `Function.length` to identify error middleware (arity 4)
Memory leak or high memory usage	1. Template cache grows without bound 2. Request/response objects retained in closures 3. Middleware array accumulating indefinitely 4. Body parser storing large buffers	1. Monitor heap usage with `process.memoryUsage()` 2. Check template cache size (should be LRU) 3. Look for closures capturing `req` / `res` in long-lived objects	1. Implement LRU cache for compiled templates with max size 2. Ensure request/response wrappers don't create circular references 3. Clear middleware arrays on app reset (or don't—they're usually static) 4. Limit body size and discard buffers after parsing

Framework-Specific Debugging Techniques

Beyond generic debugging, the Barebones framework's architecture offers unique introspection points. Think of these techniques as **installing surveillance cameras in your factory pipeline**: you can observe the product (request) at each station (component) to pinpoint where it gets stuck or damaged.

Phase Tracking Middleware

Add a debug middleware that logs the request's progression through the pipeline phases. This reveals where the flow breaks.

Technique: Create a `phaseTracker` middleware that annotates the `req` object with timestamps and phase names, logging transitions.

Phase	Typical Activities	What to Log
`incoming`	Request received, raw headers/URL	`method`, `url`, `headers`
`pre-middleware`	Before middleware chain	`middlewareCount`
`in-middleware:N`	Executing middleware N	`name`, `path`, `duration`
`routing`	Trie search, parameter extraction	`matchedRoute`, `params`
`handler`	Route handler execution	`handlerName`, `context`
`rendering`	Template compilation/render	`templatePath`, `contextKeys`
`sending`	Writing response	`statusCode`, `headers`
`error`	Error handling mode	`error.message`, `stack`

Implementation pattern: Register this middleware first with `app.use(phaseTracker())`. It should call `next()` immediately but log before and after. Use `console.time()` and `console.timeEnd()` for durations.

Trie Inspection Utility

The router's prefix tree is a complex data structure. Visualizing it helps debug matching failures.

Technique: Add a `router.inspect()` method that returns a printable representation of the trie.

Diagnostic output example:

```
Root
├── api (static)
│   ├── users (static)
│   │   ├── :id (param -> 'id')
│   │   │   └── GET → handler
│   │   └── POST → handler
│   └── posts (static)
│       └── GET → handler
└── static (static)
    └── * (catch-all) → handler
```

How to implement: Perform a depth-first traversal of `TrieNode` children, recording each node's type (static, param, catch-all), segment, and stored handlers. Print with indentation.

Request/Response State Snapshot

When a request behaves unexpectedly, capture a snapshot of the enhanced `Request` and `Response` objects at key points.

Technique: Create a middleware that conditionally logs full state when a debug flag is set (e.g., `?debug=state` in query).

Snapshot fields to include:

- `req.method`, `req.url`, `req.originalUrl`
- `req.query` (parsed query parameters)

- `req.params` (URL parameters)
- `req.body` (parsed body—if available)
- `req.cookies` (parsed cookies)
- `res.statusCode` (current status)
- `res.locals` (response-local variables)
- `res.headers` (headers to be sent)

Implementation: Use `JSON.stringify()` with a replacer function to handle circular references and limit depth.

Error Propagation Tracer

Errors can be swallowed or misrouted. Add tracing to see how errors move through the pipeline.

Technique: Wrap the `next` function to log when called with an error argument and track the error's path.

Log format: `[ErrorTracer] next(err) at middleware="logger" error="ValidationError" stack=...`

Advanced: Maintain an error chain (like `error.cause`) when wrapping errors, so the original cause is preserved. The `HttpError` type can include an `originalError` field for this purpose.

Template Compilation Preview

When templates render incorrectly, inspect the compiled JavaScript function to see if it matches your expectations.

Technique: Add a `compileDebug` option to `Template.compile()` that logs the generated function source.

Example output:

```javascript
// Compiled template function for "user.html"
function render(context) {
  var __output = [];
  __output.push("<h1>Hello, ");
  __output.push(escapeHtml(context.name || ""));
  __output.push("</h1>");
  return __output.join("");
}
```

How to implement: Store the generated function source as a string property on the compiled function (e.g., `compiledFn.source`). Log it when `NODE_ENV=development`.

Middleware Chain Visualization

List all registered middleware with their mount paths and handler names to verify ordering and scope.

Technique: Extend the `Application` to expose a `middlewareStack` property that returns an array of `{ path, fn }` objects.

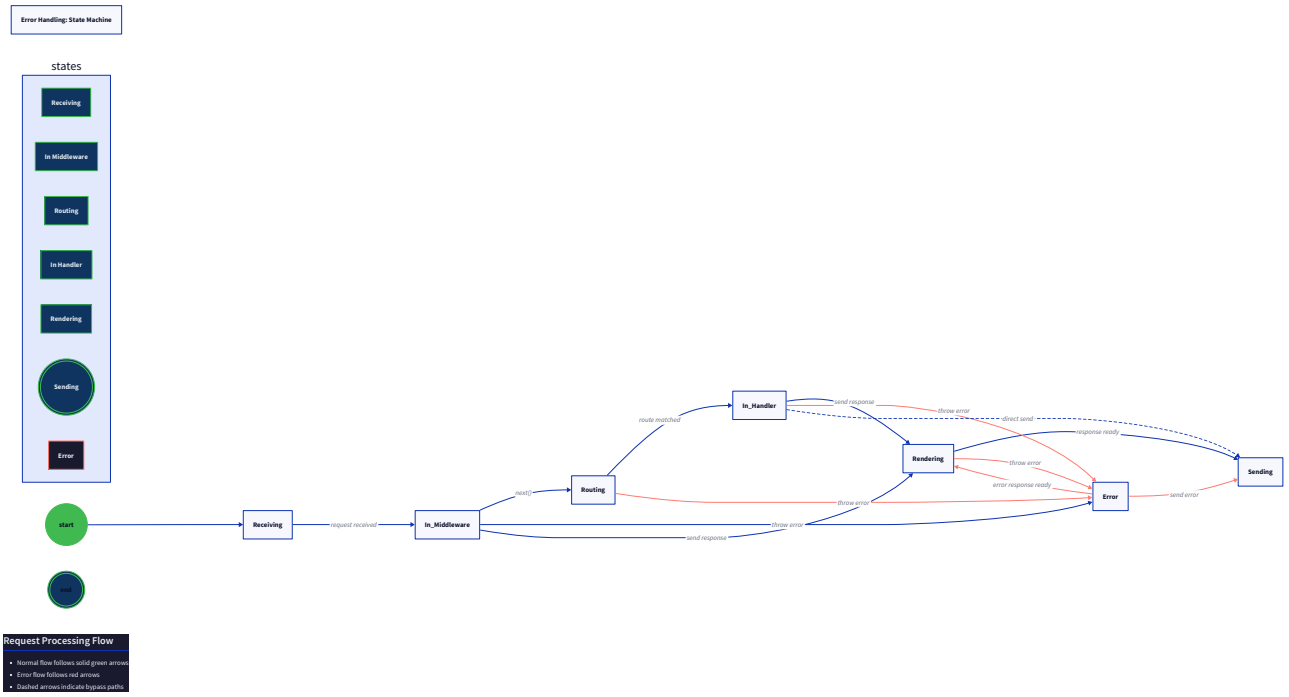
Output:

```
Middleware Stack:
1. path: '/' fn: phaseTracker
2. path: '/' fn: loggingMiddleware
3. path: '/api' fn: apiRouter
4. path: '/' fn: jsonBodyParser
5. path: '/' fn: errorHandler (4-arity)
```

Implementation: When `app.use(path, fn)` is called, push the layer to an internal array. Provide a getter that returns a copy.

Pipeline State Machine Validation

Referencing



, you can add assertions that the request follows valid state transitions.

Technique: In development mode, add a `req._state` property that updates at each phase transition. Throw if an illegal transition occurs (e.g., from `sending` back to `middleware`).

State transition table to enforce:

Current State	Valid Next States	Illegal Transition Error
`incoming`	`pre-middleware`	Cannot jump to `routing`
`in-middleware`	`in-middleware`, `routing`, `error`	Cannot skip to `sending`
`routing`	`handler`, `error`	Cannot return to `in-middleware`
`handler`	`rendering`, `sending`, `error`	-
`rendering`	`sending`, `error`	-
`sending`	(terminal)	Cannot call `next()` after send
`error`	`sending`	Cannot leave error mode without sending

Implementation: Create a `stateGuard` middleware that sets `req._state` and validates transitions when `next()` is called.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Debug Logging	`console.log` with timestamps	Structured logging with `pino` or `winston`
State Inspection	Manual `JSON.stringify` with replacer	Chrome DevTools via `node --inspect`
Trie Visualization	Text-based tree printing	HTML/D3 visualization served at `/debug/trie`
Phase Tracking	Custom middleware with `Date.now()`	OpenTelemetry tracing with spans
Error Tracing	Wrapped `next()` with logging	Error tracking service (Sentry) integration

B. Recommended File/Module Structure

Add debug utilities in a separate directory to keep production code clean:

```
project-root/
  src/
    index.js                # Main framework export
    application.js          # Application class
    router/                 # Router component
      index.js
      trie.js
      layer.js
    middleware/             # Middleware engine
      index.js
      compose.js
    request.js              # Enhanced Request
    response.js             # Enhanced Response
    template/               # Template engine
      index.js
      compiler.js
    utils/
      debug/                # Debugging utilities
        phaseTracker.js
        trieInspector.js
        stateSnapshot.js
        errorTracer.js
      httpError.js          # HttpError class
      wrapMiddleware.js
    middleware/             # Built-in middleware
      jsonBodyParser.js
      logger.js
      static.js
  examples/
    debug-server.js         # Example using debug tools
  test/
    debug.test.js           # Tests for debug utilities
```

C. Infrastructure Starter Code

Complete Phase Tracker Middleware:

JAVASCRIPT

```javascript
// utils/debug/phaseTracker.js

/**
 * Phase tracking middleware for debugging request pipeline.
 * @param {Object} options - Configuration options
 * @param {boolean} options.logTransitions - Log state changes (default: true)
 * @param {boolean} options.timings - Record and log phase durations (default: true)
 * @returns {Function} Middleware function
 */
function phaseTracker(options = {}) {
  const { logTransitions = true, timings = true } = options;

  return function phaseTrackerMiddleware(req, res, next) {
    // Add phase tracking state to request
    req._phase = 'incoming';
    req._phaseStart = Date.now();
    req._phaseHistory = [];

    const recordPhase = (newPhase) => {
      const now = Date.now();
      const duration = now - req._phaseStart;
      req._phaseHistory.push({
        phase: req._phase,
        start: req._phaseStart,
        duration,
        timestamp: now
      });

      if (logTransitions) {
        console.log(`[PhaseTracker] ${req._phase} → ${newPhase} (${duration}ms)`);
```

```
  }

  req._phase = newPhase;
  req._phaseStart = now;
};

// Override next to track phase transitions
const originalNext = next;
const trackedNext = (err) => {
  if (err) {
    recordPhase('error');
  } else {
    // Determine next phase based on current state
    const nextPhase = {
      'incoming': 'pre-middleware',
      'pre-middleware': 'in-middleware',
      'in-middleware': req._routeMatched ? 'handler' : 'in-middleware',
      'routing': 'handler',
      'handler': 'rendering',
      'rendering': 'sending',
      'sending': 'sending', // terminal
      'error': 'sending'
    }[req._phase] || 'unknown';

    recordPhase(nextPhase);
  }
  originalNext(err);
};

// Monkey-patch res.send to track sending phase
```

```
    const originalSend = res.send;
    res.send = function(body) {
      if (req._phase !== 'sending') {
        recordPhase('sending');
      }
      return originalSend.call(this, body);
    };

    // Record initial phase
    recordPhase('pre-middleware');

    // Set route matched flag when router finds a match
    const originalMatch = req.app.router.match;
    req.app.router.match = function(method, path) {
      const match = originalMatch.call(this, method, path);
      if (match) {
        req._routeMatched = true;
        req._matchedRoute = match.route;
      }
      return match;
    };

    // Call next with our tracked version
    trackedNext();
  };
}

module.exports = phaseTracker;
```

D. Core Logic Skeleton Code

Trie Inspector Utility:

```javascript
// utils/debug/trieInspector.js

/**
 * Inspects the router's trie structure for debugging.
 * @param {TrieNode} root - Root node of the trie
 * @returns {string} Textual representation of the trie
 */
function inspectTrie(root) {
  const lines = [];

  function traverse(node, prefix = '', depth = 0) {
    // TODO 1: Build indentation string based on depth (2 spaces per level)
    // TODO 2: Determine node type: static, param (:`paramName`), or catch-all (*)
    // TODO 3: For each child in node.children (Map), traverse recursively
    // TODO 4: For paramChild, traverse with special notation
    // TODO 5: For each handler in node.handlers (Map), list method -> handler
    // TODO 6: Append line to lines array with proper formatting
  }

  traverse(root);
  return lines.join('\n');
}

module.exports = inspectTrie;
```

State Snapshot Middleware Factory:

```javascript
// utils/debug/stateSnapshot.js

/**
 * Creates middleware that logs request/response state when debug flag is present.
 * @param {Object} options - Configuration options
 * @param {string} options.triggerQueryParam - Query param to trigger snapshot (default: 'debug')
 * @param {string[]} options.excludeFields - Fields to exclude from snapshot (default: [])
 * @returns {Function} Middleware function
 */
function stateSnapshot(options = {}) {
  const { triggerQueryParam = 'debug', excludeFields = [] } = options;

  return function snapshotMiddleware(req, res, next) {
    // TODO 1: Check if trigger query parameter exists (req.query[triggerQueryParam])
    // TODO 2: If triggered, capture snapshot before calling next()
    // TODO 3: Create snapshot object with req properties (method, url, params, query, body, cookies)
    // TODO 4: Add res properties (statusCode, locals, headers)
    // TODO 5: Filter out excluded fields
    // TODO 6: Log snapshot using JSON.stringify with replacer to handle circular references
    // TODO 7: Continue to next middleware
    next();
  };
}

module.exports = stateSnapshot;
```

E. Language-Specific Hints

- **Node.js Stream Debugging:** Use `req.on('data', chunk => console.log('Data chunk:', chunk.length))` to verify body parsing.
- **Async Stack Traces:** Use `--async-stack-traces` flag in Node.js to get better error locations.
- **Promise Rejection Tracking:** Listen to `process.on('unhandledRejection', (reason) => console.error('Unhandled:', reason))`.

- **Memory Inspection:** Use `process.memoryUsage()` and `console.log( Heap: ${Math.round(used / 1024 / 1024)}MB )` periodically.
- **Header Inspection:** Use `req.rawHeaders` to see original header list before normalization.

F. Debugging Tips Table

Symptom	How to Diagnose	Framework-Specific Tool
Request takes exactly 2 minutes	Node.js default socket timeout	Add `req.setTimeout(30000, () => console.log('timeout'))`
`req.body` only works for small requests	Body parser buffering entire body	Implement streaming parser with size limit
Cookies work in Chrome but not Safari	SameSite attribute mismatch	Check `sameSite: 'Lax'` vs `'Strict'`
Template inheritance causes infinite recursion	Circular extends detection	Add visited set during template resolution
Route with regex pattern fails	Trie doesn't support regex	Fallback to linear scan for regex routes

G. Milestone Checkpoint

After implementing debugging utilities, test them with:

```bash
# Run the debug example server
node examples/debug-server.js

# In another terminal, send test requests
curl "http://localhost:3000/api/users/123?debug=state"
curl -X POST -H "Content-Type: application/json" -d '{"test":true}' http://localhost:3000/echo
curl "http://localhost:3000/debug/trie"  # If implemented
```

Expected output: Clear phase transitions, state snapshots when triggered, and no hanging requests. The trie visualization should show all registered routes.

Signs of problems: Missing phases, incorrect durations, or empty snapshots indicate instrumentation bugs. Fix the debug utilities first, then use them to diagnose the core framework issues.

Future Extensions

> **Milestone(s):** All (Looking beyond the core milestones)

The **Barebones framework** has been designed with extensibility as a core architectural principle. Its modular pipeline—composed of middleware, routing, and templating—creates a foundation that can accommodate significant feature growth without requiring fundamental redesign. Think of the framework as a **modular synthesizer**: the core components are basic oscillator, filter, and amplifier modules that produce clean signals, while expansion slots allow musicians to plug in specialized effects processors, sequencers, or samplers to create entirely new sonic landscapes. Similarly, the framework's architecture provides clean signal paths (request/response flow) and standard connection points (middleware registration, route handlers, template engines) where new capabilities can be integrated.

This section explores how the current design accommodates or would need to adapt for advanced features commonly found in production frameworks. Understanding these extension points demonstrates the **architectural resilience** of the design and provides a roadmap for learners who want to continue evolving their framework beyond the core milestones.

Possible Feature Additions

While the Barebones framework covers essential web development needs, real-world applications often require additional capabilities. The following table catalogs potential enhancements, organized by functional area:

Feature Category	Specific Feature	Description	Typical Use Cases	Complexity Level
Real-time Communication	WebSocket Support	Upgrade HTTP connections to persistent, bidirectional WebSocket connections for real-time data flow	Chat applications, live notifications, collaborative editing, gaming	High
Asset Management	Static File Serving	Efficiently serve static assets (CSS, JavaScript, images) from designated directories with caching headers	Frontend resources, download files, media libraries	Medium
State Management	Session Management	Associate state with individual users across requests using cookies or tokens	User authentication, shopping carts, user preferences	Medium
Security	Cross-Site Request Forgery (CSRF) Protection	Validate that state-changing requests originate from the same site	Form submissions, API calls requiring user context	Medium
Security	Security Headers Middleware	Automatically set security headers like Content-Security-Policy, X-Frame-Options	Protection against common web vulnerabilities	Low
Performance	Response Compression	Compress response bodies using gzip or brotli algorithms	Reducing bandwidth for HTML, JSON, and text responses	Medium
Performance	Response Caching	Cache complete HTTP responses at various levels (memory, Redis)	Frequently accessed pages, API endpoints with stable data	High
Developer Experience	Plugin/Extension Ecosystem	Allow third-party packages to register middleware, template helpers, or entire subsystems	Community contributions, modular feature adoption	High
Developer Experience	Hot Reload During Development	Automatically restart server or reload routes when source files change	Faster development iteration, reduced manual restarts	Medium
Infrastructure	Cluster Mode / Process Management	Run multiple instances of the application across CPU cores	Utilizing multi-core servers, improving throughput	High
Infrastructure	Health Checks & Metrics	Expose endpoints for monitoring application health and performance metrics	Deployment orchestration, performance monitoring	Medium
Data Validation	Schema Validation Middleware	Validate request bodies, query parameters, and route parameters against JSON Schema or similar	API input validation, type safety enforcement	Medium

Feature Category	Specific Feature	Description	Typical Use Cases	Complexity Level
API Development	API Versioning Support	Manage multiple API versions through URL prefixes or headers	Evolving public APIs without breaking existing clients	Medium
API Development	Hypermedia / HATEOAS Helpers	Generate links between resources for discoverable APIs	RESTful APIs following Richardson Maturity Model Level 3	High
Advanced Templating	Template Fragment Caching	Cache rendered portions of templates independently	High-traffic pages with expensive rendering logic	Medium

Each of these features represents a natural evolution path for the framework. The key architectural question is: **does the current pipeline design allow these features to be added as external packages, or do they require core modifications?**

> **Architectural Insight**: A well-designed framework follows the **Open-Closed Principle**: open for extension (new features can be added) but closed for modification (core components don't need changes). The Barebones architecture achieves this through its middleware system and clean separation of concerns.

Design Accommodation Analysis

This section evaluates how the current Barebones architecture would support each major extension category, identifying what changes would be required and whether they align with the existing design philosophy.

WebSocket Support

Mental Model: **The Telephone Switchboard Upgrade** – Imagine the router as a traditional telephone switchboard handling brief calls (HTTP requests). Adding WebSocket support is like upgrading to a modern PBX system that can also maintain open conference lines (persistent connections) alongside regular calls. Both systems share the initial connection request, but then diverge in their communication patterns.

Current Design Accommodation:

- The `Application`'s `handleRequest` method receives all incoming HTTP requests, including WebSocket upgrade requests (identified by `Upgrade: websocket` header)
- The middleware pipeline can inspect and potentially modify upgrade requests before they reach the router
- The router's trie structure could match WebSocket endpoints using standard path patterns

Required Changes:

1. **New Route Registration Method**: Add `app.ws(path, handler)` similar to `app.get()` but for WebSocket connections
2. **Upgrade Detection in Pipeline**: Modify the request handling pipeline to detect WebSocket upgrade requests before normal HTTP processing
3. **Connection Lifecycle Management**: Create a `WebSocketConnection` class wrapping the raw socket with message/close event handlers

4. **Integration with Existing Middleware**: Decide which middleware should run before the upgrade vs. which should be skipped for WebSocket traffic

Architecture Decision Record: WebSocket Integration Strategy

> **Decision: Hybrid Middleware with Connection-Specific Context**
>
> - **Context**: WebSocket connections begin as HTTP requests (upgrade requests) that then transition to persistent bidirectional sockets. We need to decide how much of the existing HTTP middleware stack applies to WebSocket traffic.
> - **Options Considered**:
>   1. **Separate Pipeline**: Create completely independent WebSocket route registration and handling, sharing only the initial socket acceptance
>   2. **Shared Upgrade Middleware**: Run HTTP middleware only until the upgrade point, then hand off to WebSocket-specific handlers
>   3. **Per-Connection Middleware**: Allow WebSocket handlers to declare their own middleware chain that runs on each message
> - **Decision**: Option 2 (Shared Upgrade Middleware) with optional per-connection middleware
> - **Rationale**: WebSocket connections benefit from authentication, logging, and rate limiting middleware that should run during the upgrade handshake. However, message-level middleware would add unnecessary overhead and complexity for most use cases. The upgrade middleware pattern aligns with how frameworks like Express with `ws` or `Socket.IO` typically operate.
> - **Consequences**: Developers can secure and log WebSocket connections using familiar middleware, but message validation/transformation must be handled within the WebSocket handler itself. The implementation requires careful state handoff between HTTP and WebSocket phases.

Option	Pros	Cons	Chosen?
Separate Pipeline	Clean separation, no HTTP middleware overhead on messages	Duplicates routing logic, can't reuse auth/logging middleware	No
Shared Upgrade Middleware	Reuses investment in middleware, consistent auth model	Complex handoff between HTTP and WebSocket states	**Yes**
Per-Connection Middleware	Maximum flexibility for message processing	Significant performance overhead, complex API	Partial (optional)

Implementation Pathway:

1. Extend the `Router` class with a `wsRoutes` trie separate from HTTP routes
2. Modify `app.handleRequest` to check for `Upgrade: websocket` header after running pre-upgrade middleware
3. If upgrade detected and route matches, run upgrade-specific middleware, then hand off to WebSocket handler
4. The WebSocket handler receives the raw socket and can attach message listeners

Static File Serving

Mental Model: **The Library Catalog System** – Imagine the router as a library's main information desk that directs patrons to various departments. Static file serving adds a specialized "media department" with its own efficient retrieval system

(file I/O, caching, and streaming) that operates alongside the regular "reference desk" (dynamic route handlers).

Current Design Accommodation:

- The middleware pattern perfectly accommodates static file serving as a specialized middleware function
- The `Response` object already has methods for sending files and setting headers
- The router's prefix matching can efficiently filter requests for static asset paths

Required Changes:

1. **New Middleware Factory**: Create `static(directoryPath, options)` function returning middleware
2. **File System Integration**: Add efficient file reading with streaming support
3. **Caching Headers**: Automatically set `Cache-Control`, `ETag`, and `Last-Modified` headers
4. **Directory Listing**: Optional directory index generation
5. **Range Requests**: Support for HTTP Range headers for partial content (video/audio streaming)

Architecture Decision Record: Static File Serving Implementation Strategy

> **Decision: Middleware with Sendfile Optimization**
>
> - **Context**: Serving static files efficiently requires minimizing memory usage and leveraging operating system optimizations. Node.js provides `fs.createReadStream` for streaming and `sendfile` system call support through `res.sendFile`.
> - **Options Considered**:
>   1. **Buffer and Send**: Read entire file into memory buffer, then send (simplest but memory-intensive)
>   2. **Streaming**: Pipe file read stream to response stream (good for large files)
>   3. **Sendfile System Call**: Use `res.sendFile` or similar to leverage kernel optimizations (most efficient)
> - **Decision**: Option 3 (Sendfile) with fallback to streaming for platforms without sendfile support
> - **Rationale**: The `sendfile` system call copies data directly from disk to network socket without passing through user space, providing significant performance benefits for static file serving. This approach is used by production frameworks like Express.js (`res.sendFile`). The fallback ensures cross-platform compatibility.
> - **Consequences**: Excellent performance for static assets, but requires careful handling of errors and partial sends. The middleware must also handle path traversal security and MIME type detection.

Option	Pros	Cons	Chosen?
Buffer and Send	Simple implementation, easy error handling	Memory intensive for large files, poor performance	No
Streaming	Good memory efficiency, handles large files	More complex error handling, moderate performance	Backup
Sendfile System Call	Maximum performance, minimal CPU usage	Platform-specific, complex implementation	**Yes**

Implementation Pathway:

1. Create `static.js` middleware that checks if request method is GET/HEAD and URL path is within served directory

2. Use Node.js `fs.stat` to check file existence and get size/modification time
3. Implement `ETag` generation and conditional request handling (`If-None-Match`, `If-Modified-Since`)
4. Use `res.sendFile` if available, otherwise create read stream with proper error handling
5. Set appropriate `Content-Type` based on file extension

Session Management

Mental Model: **The Coat Check System** – Each visitor to a venue (request) receives a numbered ticket (session ID) that corresponds to their stored coat (session data). The ticket is small and portable (cookie), while the actual coat remains securely stored backstage (server-side store). The coat check attendant (session middleware) uses the ticket to retrieve the correct coat for each return visit.

Current Design Accommodation:

- Middleware executes on every request, perfect for session initialization/retrieval
- The enhanced `Request` object can have a `session` property added
- Cookie parsing is already implemented in Request/Response enhancements
- The `Response` object can set cookies for session ID persistence

Required Changes:

1. **Session Middleware**: Create `session(options)` middleware that attaches `req.session`
2. **Storage Abstraction**: Define `SessionStore` interface with memory, Redis, and database implementations
3. **Session Data Serialization**: Secure serialization/deserialization of session data
4. **Security Features**: Session regeneration, fixation protection, secure cookie flags

Implementation Pathway:

1. Session middleware generates or retrieves session ID from cookie/header
2. Loads session data from configured store (memory, Redis, etc.)
3. Attaches session object to `req.session` with `save()` method
4. On response, serialize and save modified sessions, set session cookie

Plugin Ecosystem

Mental Model: **The Modular Kitchen Appliance** – The core framework is like a kitchen counter with standard power outlets (extension points). Plugins are specialized appliances (bread makers, blenders, air fryers) that plug into these outlets, each bringing unique functionality while following standard electrical specifications (plugin interface).

Current Design Accommodation:

- The `Application`'s `use()` method already accepts middleware functions
- The `app.set()` / `app.get()` settings system provides configuration storage
- Template engines can already be registered via `app.engine()`

Required Changes:

1. **Plugin Interface Definition**: Formalize plugin signature `(app, options) => void`
2. **Lifecycle Hooks**: Add registration points for plugin initialization, startup, shutdown
3. **Dependency Management**: Allow plugins to declare dependencies on other plugins

4. **Configuration Isolation**: Namespaced configuration to avoid conflicts

Implementation Pathway:

1. Define `Plugin` interface with `name`, `version`, `install(app, options)` method
2. Add `app.plugin(plugin, options)` method that calls `plugin.install(app, options)`
3. Maintain plugin registry for dependency resolution and lifecycle management

Response Compression

Mental Model: **The Shipping Department Packer** – Before sending a package (response) to the customer, a specialized packer (compression middleware) efficiently boxes and compresses the contents if the customer accepts compressed shipments (`Accept-Encoding` header), reducing shipping costs (bandwidth) while adding minimal packing time (CPU overhead).

Current Design Accommodation:

- Middleware can inspect and modify responses before they're sent
- The `Response` object provides access to headers and body
- Response body can be intercepted and transformed

Required Changes:

1. **Compression Middleware**: Create `compression(options)` that checks `Accept-Encoding`
2. **Stream Transformation**: Implement gzip/brotli compression streams
3. **Content Type Filtering**: Optionally compress only certain content types
4. **Size Thresholds**: Skip compression for very small responses

Implementation Pathway:

1. Middleware wraps `res.write`, `res.end` to intercept body data
2. Checks `Accept-Encoding` header and selects appropriate compression
3. Sets `Content-Encoding` header and pipes response through compression stream
4. Handles Vary header for proper caching

Cross-Site Request Forgery (CSRF) Protection

Mental Model: **The Concert Ticket Validation** – Legitimate ticket holders (authenticated users) receive a unique, hard-to-forge ticket (CSRF token) when they enter the venue (load a form). When they want to enter a restricted area (submit the form), they must present both their entry wristband (session) AND the matching ticket (CSRF token), proving they're the same person who received the original ticket.

Current Design Accommodation:

- Session management (prerequisite) provides user identification
- Middleware can validate tokens before sensitive actions
- Template engine can automatically inject tokens into forms

Required Changes:

1. **CSRF Middleware**: Generate and validate tokens per session
2. **Token Storage**: Secure token storage with expiration

3. **Request Validation**: Check token on state-changing methods (POST, PUT, DELETE, PATCH)
4. **Template Helpers**: Add `csrfToken()` and `csrfField()` template helpers

Implementation Pathway:

1. CSRF middleware adds `req.csrfToken()` method and validates on non-GET requests
2. Tokens are stored in session and compared against `_csrf` parameter or `X-CSRF-Token` header
3. Template helper functions inject hidden form fields with current token

Health Checks and Metrics

Mental Model: **The Aircraft Instrument Panel** – Pilots (operations teams) need real-time gauges (metrics) showing fuel level (memory usage), engine temperature (CPU load), and altitude (request rate). Regular system checks (health endpoints) verify all critical systems are operational before takeoff (serving traffic).

Current Design Accommodation:

- Router can serve dedicated monitoring endpoints
- Middleware can collect request metrics
- The `Application` instance can expose internal state

Required Changes:

1. **Metrics Collection Middleware**: Track request counts, durations, error rates
2. **Health Check Endpoints**: Standardized `/health`, `/ready`, `/metrics` routes
3. **Metrics Export**: Expose metrics in Prometheus format or JSON
4. **Internal Monitoring**: Track memory usage, event loop lag, active connections

Implementation Pathway:

1. Add metrics middleware early in chain to time all requests
2. Register standard health endpoints that check database connections, etc.
3. Expose metrics via `app.metrics` getter or dedicated endpoint

Design Principles for Future Extensions

When evaluating whether a feature should be added to the core framework or provided as an external package, consider these principles derived from the Barebones architecture:

1. **Middleware-First Philosophy**: If a feature can be implemented as middleware without requiring core modifications, it should remain external. This keeps the core lean and allows developers to choose only what they need.
2. **Progressive Disclosure of Complexity**: The framework should provide sensible defaults for common use cases while allowing advanced configuration for specialized needs. For example, static file serving should work out-of-the-box with `app.use(static('public'))` but allow configuration of caching, compression, and security headers.
3. **Consistent API Patterns**: New features should follow the same API patterns as existing features. If the framework uses `app.use()` for middleware and `app.get()` for routes, then `app.ws()` for WebSockets maintains consistency.
4. **Layered Security Model**: Security features should be composable and layered. CSRF protection builds on sessions, which build on cookies, which build on request parsing. Each layer can be used independently or together.

5. **Performance by Default with Escape Hatches**: The framework should choose performant implementations by default (like `sendfile` for static files) but allow developers to override with custom implementations when needed.

> **Key Insight**: The most successful frameworks grow through **extensibility, not comprehensiveness**. By providing a solid core with clean extension points, the Barebones framework enables a ecosystem of specialized packages rather than attempting to include every possible feature in the core. This aligns with the Unix philosophy: "Write programs that do one thing and do it well. Write programs to work together."

Implementation Guidance

While the future extensions are beyond the core milestones, implementing one or two can provide valuable learning experiences. Here we provide guidance for implementing static file serving as it demonstrates several important patterns and has immediate practical value.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Static File Serving	Basic file streaming with `fs.createReadStream`	`sendfile` system call with range request support
Compression	gzip compression using Node's `zlib` module	Brotli compression with fallback to gzip
Session Storage	In-memory store (simple)	Redis store with TTL support (production-ready)
WebSocket Library	Native `ws` package	`Socket.IO` with fallback transports

Recommended File/Module Structure

Add new modules for extended functionality while keeping the core framework clean:

```
barebones-framework/
├── src/
│   ├── core/                # Original core framework
│   │   ├── application.js
│   │   ├── router.js
│   │   ├── middleware.js
│   │   └── ...
│   ├── extensions/          # Official framework extensions
│   │   ├── static.js        # Static file serving middleware
│   │   ├── sessions/        # Session management
│   │   │   ├── index.js
│   │   │   ├── memory-store.js
│   │   │   └── redis-store.js
│   │   ├── compression.js   # Response compression middleware
│   │   └── security/        # Security middleware
│   │       ├── csrf.js
│   │       └── headers.js
│   └── plugins/             # Plugin system infrastructure
│       ├── plugin.js
│       └── registry.js
├── examples/                # Example applications
│   ├── static-site/
│   ├── realtime-chat/
│   └── api-server/
└── test/
    ├── unit/
    └── integration/
```

Static File Serving Middleware (Complete Implementation)

JAVASCRIPT

```javascript
// src/extensions/static.js

const fs = require('fs');

const path = require('path');

const { promisify } = require('util');

const stat = promisify(fs.stat);

const access = promisify(fs.access);


/**

 * Static file serving middleware factory

 * @param {string} root - Root directory to serve files from

 * @param {Object} options - Configuration options

 * @param {boolean} options.dotfiles - Allow serving dotfiles (default: false)

 * @param {string|Array} options.index - Default index files (default: ['index.html'])

 * @param {number} options.maxAge - Cache control max-age in milliseconds

 * @param {boolean} options.immutable - Set cache-control immutable flag

 * @returns {Function} Middleware function

 */

function static(root, options = {}) {

  const opts = {

    dotfiles: options.dotfiles || false,

    index: options.index || ['index.html', 'index.htm'],

    maxAge: options.maxAge || 0,

    immutable: options.immutable || false,

    ...options

  };


  // Normalize root path and ensure it's absolute

  const rootPath = path.resolve(root);


  return async function staticMiddleware(req, res, next) {
```

```
// Only handle GET and HEAD requests
if (req.method !== 'GET' && req.method !== 'HEAD') {
  return next();
}

// Clean and normalize the request path
let requestPath = decodeURIComponent(req.path);


// Security: Prevent directory traversal
if (requestPath.includes('../') || requestPath.includes('..\\')) {
  return next();
}

// Construct full file path
let filePath = path.join(rootPath, requestPath);


try {
  // Check if file exists and get stats
  const stats = await stat(filePath);


  // If it's a directory, look for index file
  if (stats.isDirectory()) {
    for (const indexFile of Array.isArray(opts.index) ? opts.index : [opts.index]) {
      const indexPath = path.join(filePath, indexFile);
      try {
        const indexStats = await stat(indexPath);
        if (indexStats.isFile()) {
          filePath = indexPath;
          break;
        }
      } catch {
```

```
      // Continue to next index file
    }
  }
}

// Final check that we have a file
const finalStats = await stat(filePath);
if (!finalStats.isFile()) {
  return next();
}

// Check dotfiles restriction
const basename = path.basename(filePath);
if (basename.startsWith('.') && !opts.dotfiles) {
  return next();
}

// Set Content-Type header based on file extension
const ext = path.extname(filePath).toLowerCase();
const mimeType = getMimeType(ext);
res.setHeader('Content-Type', mimeType);

// Set caching headers
if (opts.maxAge > 0) {
  const cacheControl = `public, max-age=${Math.round(opts.maxAge / 1000)}`;
  res.setHeader('Cache-Control', opts.immutable ? `${cacheControl}, immutable` : cacheControl);
}

// Handle conditional requests (If-Modified-Since)
const ifModifiedSince = req.headers['if-modified-since'];
```

```
    if (ifModifiedSince) {
      const lastModified = new Date(finalStats.mtime);
      const ifModifiedSinceDate = new Date(ifModifiedSince);

      // If file hasn't been modified, send 304 Not Modified
      if (lastModified <= ifModifiedSinceDate) {
        res.statusCode = 304;
        return res.end();
      }
    }

    // Set Last-Modified header
    res.setHeader('Last-Modified', new Date(finalStats.mtime).toUTCString());

    // Handle range requests (for audio/video streaming)
    const range = req.headers.range;
    if (range) {
      // Parse range header
      const parts = range.replace(/bytes=/, '').split('-');
      const start = parseInt(parts[0], 10);
      const end = parts[1] ? parseInt(parts[1], 10) : finalStats.size - 1;

      if (start >= 0 && end < finalStats.size && start <= end) {
        const chunkSize = (end - start) + 1;
        res.statusCode = 206; // Partial Content
        res.setHeader('Content-Range', `bytes ${start}-${end}/${finalStats.size}`);
        res.setHeader('Content-Length', chunkSize);

        // Create read stream for the specific range
        const fileStream = fs.createReadStream(filePath, { start, end });
```

```
        fileStream.pipe(res);
        return;
      }
    }

    // Regular file serving (full file)
    res.setHeader('Content-Length', finalStats.size);

    // For HEAD requests, just send headers without body
    if (req.method === 'HEAD') {
      return res.end();
    }

    // Stream the file
    const fileStream = fs.createReadStream(filePath);
    fileStream.pipe(res);

    // Handle stream errors
    fileStream.on('error', (err) => {
      console.error('File stream error:', err);
      if (!res.headersSent) {
        next(err);
      }
    });

  } catch (err) {
    if (err.code === 'ENOENT') {
      // File not found, pass to next middleware
      return next();
    }
```

```
      // Other error (permission, etc.)
      next(err);
    }
  };
}

/**
 * Simple MIME type lookup
 */
function getMimeType(ext) {
  const mimeTypes = {
    '.html': 'text/html',
    '.htm': 'text/html',
    '.css': 'text/css',
    '.js': 'application/javascript',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpeg',
    '.jpeg': 'image/jpeg',
    '.gif': 'image/gif',
    '.svg': 'image/svg+xml',
    '.ico': 'image/x-icon',
    '.txt': 'text/plain',
    '.pdf': 'application/pdf',
    '.zip': 'application/zip',
    '.mp3': 'audio/mpeg',
    '.mp4': 'video/mp4',
  };

  return mimeTypes[ext] || 'application/octet-stream';
```

```
}


module.exports = static;
```

Usage Example

```javascript
const { Application } = require('./src/core/application');

const static = require('./src/extensions/static');


const app = new Application();


// Serve static files from 'public' directory

app.use(static('public', {

  maxAge: 86400000, // 1 day

  index: ['index.html', 'home.html']

}));


// Other routes and middleware...

app.get('/', (req, res) => {

  res.send('Hello World');

});


app.listen(3000);
```

Language-Specific Hints for JavaScript/Node.js

1. **Use Streams for Large Files**: Always pipe file streams to the response rather than reading entire files into memory. This handles large files efficiently and prevents memory exhaustion.
2. **Leverage `sendfile` for Performance**: In production, consider using `res.sendFile()` (if available in your response wrapper) which uses the `sendfile` system call for optimal performance.
3. **Security Considerations**: Always validate file paths to prevent directory traversal attacks. Use `path.resolve()` and check for `..` patterns.
4. **Caching Strategy**: Implement proper cache headers (`Cache-Control` , `ETag` , `Last-Modified`) to reduce server load and improve user experience.
5. **Error Handling**: Ensure file stream errors don't crash the server. Use try-catch for async operations and error listeners on streams.

Milestone Checkpoint for Static File Extension

Test Command:

```bash
node test/static-test.js
```

Expected Behavior:

1. Create a `public` directory with `index.html`, `style.css`, and `image.jpg`
2. Start the server with static middleware configured
3. Visit `http://localhost:3000/` should serve `index.html`
4. Visit `http://localhost:3000/style.css` should serve CSS with correct Content-Type
5. Visit `http://localhost:3000/nonexistent.html` should return 404 (or pass to next route)
6. Check that `Cache-Control` headers are set appropriately

Debugging Tips for Static File Issues:

Symptom	Likely Cause	How to Diagnose	Fix
404 for existing files	Incorrect path resolution	Log the resolved file path in middleware	Ensure `root` path is absolute and request path is properly decoded
Wrong Content-Type	Missing MIME type mapping	Check file extension and MIME type lookup	Add missing extension to `getMimeType` function
Large files timeout or crash	Buffering entire file	Check if you're using `fs.readFile` instead of streams	Switch to `fs.createReadStream().pipe(res)`
Directory traversal vulnerability	Insufficient path validation	Try accessing `http://localhost:3000/../secrets.txt`	Add `../` detection and use `path.resolve()` with boundary checks

This implementation demonstrates how the middleware pattern allows powerful features to be added to the framework without modifying its core. Each extension builds upon the solid foundation established in the four core milestones, showcasing the architectural flexibility of the Barebones design.

Glossary

> **Milestone(s):** All (Reference)

The following glossary defines key technical terms, acronyms, and domain-specific vocabulary used throughout the Barebones Web Framework design document. Understanding these terms is essential for grasping the architectural

concepts, component designs, and implementation details discussed in previous sections. This glossary serves as a quick reference for developers, especially those new to web framework internals.

Glossary of Terms

Term	Definition	Reference Section(s)
ADR (Architecture Decision Record)	A structured document that captures a significant architectural decision, including the context, options considered, the chosen decision, its rationale, and its consequences. Used throughout the design to explain key trade-offs.	Component Design sections (Router, Middleware, etc.)
AST (Abstract Syntax Tree)	A tree representation of the structure of a template source code, where each node denotes a construct (e.g., variable, conditional block, loop). Used by the template engine during compilation to transform template syntax into executable JavaScript functions.	Component Design: Template Engine
Body Parsing	The process of interpreting the raw bytes of an HTTP request body (e.g., from a `POST` or `PUT` request) into structured data formats such as JSON, URL-encoded form data, or multipart form data. The parsed data is made available on the enhanced `Request` object (e.g., `req.body`).	Component Design: Request & Response Enhancements; Milestone 3
Chain of Responsibility	A behavioral design pattern where a request is passed along a chain of handlers (e.g., middleware functions). Each handler decides either to process the request or to pass it to the next handler in the chain. This pattern is the foundation of the middleware pipeline.	Component Design: Middleware Engine; High-Level Architecture
Code Generation	The process by which the template engine converts an AST (or directly parses template syntax) into an executable JavaScript function. This function, when called with a data context, returns the final rendered HTML string. Pre-compiling templates via code generation improves performance over interpreting templates on each request.	Component Design: Template Engine
Cookie Parsing	The process of extracting key-value pairs from the `Cookie` HTTP request header string into a structured object (e.g., `req.cookies`). Conversely, cookie serialization involves setting the `Set-Cookie` response header with proper formatting and security attributes (e.g., `httpOnly`, `secure`).	Component Design: Request & Response Enhancements
Cookie Security Flags	Attributes set on HTTP cookies that enhance security. Common flags include `httpOnly` (prevents client-side JavaScript access), `secure` (only sent over HTTPS), `sameSite` (restricts cross-site requests), and `maxAge` / `expires` (control cookie lifetime). The framework's `res.cookie()` method supports these options.	Component Design: Request & Response Enhancements; Error Handling
Enhanced Request/Response	Wrapper objects that extend the native Node.js `http.IncomingMessage` (request) and `http.ServerResponse` (response) objects with additional convenience properties (e.g., `req.query`, `req.body`) and methods (e.g., `res.json()`, `res.redirect()`). These wrappers provide a more developer-friendly API.	Component Design: Request & Response Enhancements; Data Model
Error Handling Middleware	A special type of middleware defined with four parameters: `(err, req, res, next)`. It is invoked when an error is passed to `next(err)` or	Component Design: Middleware Engine;

Term	Definition	Reference Section(s)
	thrown synchronously in the pipeline. The framework enters **error mode**, where only these 4-arity middleware functions are executed until a response is sent.	Error Handling and Edge Cases
Error Mode	A state of the request processing pipeline activated when an error is passed via `next(err)` or thrown. In this state, the framework skips all regular middleware and routes, executing only error-handling middleware (functions with signature `(err, req, res, next)`) until a response is sent or an unhandled error occurs.	Component Design: Middleware Engine; Error Handling and Edge Cases
Expose Flag	A boolean property on the `HttpError` type (and other errors) that indicates whether the error's `message` should be sent to the client in the HTTP response. In production (`NODE_ENV='production'`), messages for errors without `expose: true` are typically replaced with generic messages to avoid leaking internal details.	Data Model; Error Handling and Edge Cases
Factory Pipeline	The core architectural analogy for the framework: an HTTP request is like a product moving through an assembly line (the pipeline), where each station (a middleware function) can inspect, modify, or decorate the request/response before passing it to the next station. This model emphasizes composability and sequential processing.	High-Level Architecture; Context and Problem Statement
HTML Escaping	The process of converting special characters in a string (e.g., `<`, `>`, `&`, `"`, `'`) into their corresponding HTML entities (e.g., `<`, `>`, `&`, `&quot;`, `&#39;`). This is critical for preventing Cross-Site Scripting (XSS) attacks when outputting user-provided data into HTML templates. The template engine performs automatic escaping for content in `{{ }}` expressions.	Component Design: Template Engine; Milestone 4
LRU Cache	A **Least Recently Used** cache eviction strategy. When the cache reaches its size limit, the least recently accessed item is removed first. The template engine may use an LRU cache to store compiled template functions, avoiding recompilation for frequently used templates while bounding memory usage.	Component Design: Template Engine; Future Extensions
Middleware	A function with the signature `(req, res, next)` that processes HTTP requests and responses within the framework's pipeline. Middleware can execute any code, modify the request/response objects, end the request-response cycle, or call the `next()` function to pass control to the next middleware in the stack. Middleware is the primary extensibility mechanism.	Component Design: Middleware Engine; Milestone 2
Parameter Extraction	The process of parsing named values from dynamic segments of a URL path pattern. For example, given a route pattern `/users/:id` and an actual request path `/users/42`, the router extracts `{ id: '42' }` and attaches it to `req.params`. This enables dynamic routing based on path segments.	Component Design: The Router; Milestone 1

Term	Definition	Reference Section(s)
Phase Tracking	A debugging technique where the framework logs or exposes the progression of a request through distinct stages of the pipeline (e.g., `'middleware'`, `'routing'`, `'handler'`, `'rendering'`). This helps developers understand where a request might be stalling or failing.	Debugging Guide
Prefix Tree (Trie)	A tree data structure used for efficient storage and retrieval of keys (in this case, URL path segments). Each node represents a common prefix of its descendant nodes. The router uses a prefix tree (`TrieNode`) to match incoming request paths against registered route patterns, allowing for fast lookup even with many routes and dynamic segments.	Component Design: The Router; Data Model
Query Parameters	Key-value pairs extracted from the query string of a URL (the part after the `?`). For example, in `/search?q=framework&sort=asc`, the query parameters are `{ q: 'framework', sort: 'asc' }`. The framework parses these and makes them available on `req.query`.	Component Design: Request & Response Enhancements; Milestone 3
Router Trie	The specific implementation of a prefix tree (`TrieNode`) used by the router to store and match route patterns. Each node corresponds to a path segment (static or parameter), and leaf nodes hold a map of HTTP method to handler functions for that route.	Component Design: The Router; Data Model
Size Limit	A configuration parameter (often in bytes) that restricts the maximum allowed size of an incoming request body. Body parsing middleware enforces this limit to prevent Denial-of-Service (DoS) attacks via excessively large payloads. Exceeding the limit results in a `413 Payload Too Large` error.	Component Design: Request & Response Enhancements; Error Handling
Stream Backpressure	A flow control mechanism in Node.js streams where a source (e.g., a readable stream from the HTTP request) does not overwhelm a destination (e.g., the body parser) by pausing data emission when the destination's buffer is full. Proper handling of backpressure is important when parsing large request bodies in a streaming fashion.	Error Handling and Edge Cases; Future Extensions
Template Caching	Storing compiled template functions (the output of `Template.compile()`) in memory to avoid recompiling the same template source on every render request. This significantly improves performance. Cache invalidation strategies (e.g., file watching in development, LRU cache) are needed to handle template updates.	Component Design: Template Engine
Template Engine	The component responsible for compiling template files (containing a mix of static HTML and dynamic expressions/control structures) into executable render functions, and then executing those functions with a provided data context to produce final HTML output. It handles variable interpolation, HTML escaping, control flow, and inheritance.	Component Design: Template Engine; Milestone 4
Template Inheritance	A mechanism allowing **child templates** to extend **base layout templates**. The child template defines content for named **blocks** (e.g., `content`, `sidebar`) that override corresponding blocks in the base template. This promotes reusable layouts and reduces duplication.	Component Design: Template Engine; Milestone 4

Term	Definition	Reference Section(s)
Template Cache Poisoning	A situation where a cached, compiled template function becomes stale or contains errors, leading to incorrect renders. This can happen if the template source file changes on disk but the cache is not invalidated, or if a bug in the compiler generates faulty functions. Proper cache invalidation and development-mode settings mitigate this.	Component Design: Template Engine; Debugging Guide
Unhandled Promise Rejection	An error that occurs when a JavaScript Promise is rejected but no `.catch()` handler or `await` in a try/catch block is present to handle it. In Node.js, unhandled rejections can crash the process. The framework's middleware wrapper function (`wrapMiddleware`) and top-level error handler are designed to catch and process these rejections as 500 errors.	Component Design: Middleware Engine; Error Handling and Edge Cases
Variable Interpolation	The process of replacing placeholders (e.g., `{{ name }}`) in a template with corresponding values from a provided data context. The template engine performs interpolation during rendering, ensuring proper HTML escaping for safe output.	Component Design: Template Engine; Milestone 4
XSS (Cross-Site Scripting)	A security vulnerability where an attacker injects malicious client-side scripts (usually JavaScript) into web pages viewed by other users. The framework's template engine prevents XSS by automatically escaping HTML special characters in interpolated variables. Developers must still be cautious when using `{{{ }}}` for raw, unescaped output.	Component Design: Template Engine; Common Pitfalls

---

Implementation Guidance: *This section is purely reference material and does not require implementation guidance. It consolidates terminology for clarity and consistent communication throughout the project.*