Report for assignment 3

Project

Name: echo
URL: https://github.com/labstack/echo

Echo is a high-performance, extensible and minimalist web framework for Go, designed for building scalable and efficient web applications. It provides features like middleware support, request routing and built-in data binding while maintaining low memory footprint and high speed.

Onboarding experience

The Echo repository was very easy to clone and run. The only prerequisite was installing Go, the programming language used in the project. Once Go was installed, testing the project was straightforward—simply running go test ./... executed all tests without any issues.

We specifically chose a project written in Go because it has built-in testing tools, eliminating the need for additional dependencies or external testing frameworks. This made the setup process much simpler and reduced the overhead of installing extra tools.

The project was well-documented, with a dedicated documentation page for developers using the project. Additionally, most of the logic in the code was clearly explained, which made writing tests and refactoring more manageable. The build process concluded automatically without errors, and all necessary components were handled seamlessly by the Go toolchain.

We plan to continue working with this project. Although it already has a high test coverage of around 92%, we identified areas where improvements can be made, making this a valuable project to contribute to and enhance further.

Complexity

Tool	LOC	CC	CC (manually)	Function
Lizard	115	46	42	bindData()
Lizard	145	49	48	LoggerWithConfig()
Lizard	113	29	28	insertNode()
Lizard	133	34	32	Find()
Lizard	40	15	15	addMethod()

bindData()

1.

Tool	LOC	CC	CC (manually)	Function
Lizard	115	46	43	bindData()

2. BindData is both very long and also complex. There is a lot of different cases when binding data and therefore it's much logic to handle these cases.

3. BindData is used to bind HTTP headers to a Go interface, mapping the values from the request headers into the fields of the destination struct that have the specified tag. It ensures that only the explicitly tagged fields are populated with the corresponding data.

4. The function handles different kinds of input and if the input is invalid the function returns an error.

5. The code is well documented, almost every if-statement has a commented explaining what it checks.

LoggerWithConfig()

1.

Tool	LOC	CC	CC (manually)	Function
Lizard	145	49	48	LoggerWithConfig()

LoggerWithConfig() is long and complex. There are many ifs and switch-cases statements and this leads to high cyclomatic complexity. If we use the tool Lizard we will detect 49 as CCN, while a manual count derived from the the law CCN = E - N + 2P leads to 48, a very similar result. Following McCabe's rule, however, would lead to a different number much lower. Despite some differences it's still clear the high complexity of this function which makes it hard to draw and build the graph of the flow control of the function itself. It's important to notice the anonymous functions inside the function (which are separated by the Lizard tool)

2. The function is quite long (145 NLOC in total) as well as complex.

3. The LoggerWithConfig() function is a middleware function for Echo. Its purpose is to configure a custom logger that is executed for every HTTP request.

4. In our manual counting, the exceptions don't directly influence the value of CCN. It's clear, however, that for a correct exception handling many if structures are needed and this increases the cyclomatic complexity.

5. The code is poorly documented, but for the most difficult operations some comments provide a clear explanation.

insertNode()

1.

Tool	LOC	CC	CC (manually)	Function
Lizard	113	29	13	insertNode()

2. The insertNode() function is long at 113 lines, and the cyclomatic complexity is quite high with 29. This is due to the function handling many different cases. For example, searching for common prefixes of URL, splitting nodes at specific points and inserting a new child node.

3. The inserNode is responsible for adding a new route into the router's tree structure. It compares the new route parth with existing routes. Then it splits nodes if the paths partially overlap. For example, WIth registered node ../settings/user and new node ../settings/profile, the new node is added under ../settings parent node. It is for routing trees to correctly reflect the structure for URL matching.

4. The insertNode has different inputs. It must handle situations where the new route partially matches an existing route, or when it should create a new branch.

5. insertNode function is well documented. Every major decision point includes comments explaining what the code is checking for and why that branch exists which helps readers to easily understand the complex codes.

Find()

1.

Tool	LOC	CC	CC (manually)	Function
Lizard	133	34		Find()

2.The function is both lengthy and complex. The router tree is implemented using a Longest Common Prefix (LCP) array, and Find() includes an advanced search mechanism that utilizes backtracking.

3. The purpose of the Find function in the Router struct is to look up a registered handler for a given HTTP method and path. It also extracts any path parameters from the URL and loads them into the provided Context.

4. The function have return statements that is used to exit the if it encounters any problem.

5. Find() function is well documented and there is comments related to all logic in the function.

addMethod()

Tool	LOC	CC	CC (manually)	Function
Lizard	40	15	15	addMethod()

2. The function is high in cyclomatic complexity, but is not particularly long with its 40 lines of code. The high CC is due to the fact that the function has a large switch case statement, and not from an overly long function.

3. The function addMethod is responsible for associating an HTTP method with a corresponding handler in a routing structure. It assigns handlers to the appropriate method field (e.g., GET, POST) within the node object and updates routing metadata (updateAllowHeader, isHandler). It also handles custom or unrecognized methods via a fallback mechanism (anyOther map).

4. No, because the function doesn’t include any error handling.

5. There is a clear lack of documentation for this function. Even though the functionality can be clear to some people due to the variable names, the function could be made more obvious with better documentation. The function only has one comment, which explains that it doesn't require any additional processing or assignment of handlers when the RouteNotFound case is encountered.

Refactoring

Plan for refactoring complex code:

Refactor Plan for bindData()

The bindData() function currently contains two large and complex sections of code: one handling map bindings and the other handling struct bindings. This makes the function difficult to read, maintain, and debug.

To improve clarity and maintainability, we will separate these concerns into two dedicated helper functions:

- handleMapBinding() – Responsible for processing map-based bindings.
- handleStructBinding() – Responsible for handling struct-based bindings.

Benefits of Refactoring:
- Improved Readability
- Better Maintainability
- Easier Debugging

After refactoring, bindData() will delegate work to these helper functions, significantly reducing its complexity. By dividing the function into two smaller parts, the overall CC is estimated to decrease by approximately 50%. Each new function will be roughly half the size of the original, making the code easier to read and maintain.

After completing the refactoring, I can present the following results:

Tool	**LOC**	**CC**	**Function**
Lizard	115	46	bindData() before
Lizard	18	10	bindData() after
Lizard	22	9	handleMapBinding() helper function
Lizard	81	29	handleStructBinding() helper function

The refactoring makes bindData() more readable and manageable by offloading complexity to handleStructBinding() and handleMapBinding().

Refactor Plan for insertNode()

The current insertNode function has two major logics:

1. It splits a node when the new route's path only partially matches the current node's path
2. It creates a new child node when no matching child exists for the rest of the path

Having both tasks in a single function is hard to read and fix.

To make the code much simpler and easier to work with, we propose an idea to split these tasks into two helper functions:

- splitNode() - this function will handle splitting a node including
- createChildNode() - this function will handle creating a new child node

Breaking the code into smaller parts makes the amin loop insertNode'(much easier to understand. It is also easier to find and fix errors in splitting.

After refactoring, the complexity of insertnode() will drop by 00%, as helper function will do one simple task.

Refactor Plan for addMethod()

The addMethod() function suffers from a high cyclomatic complexity number (CCN) of 15 due to a large switch case statement. To reduce the CCN, a possible refactor would be to replace this switch statement with a map-based lookup. This change would lower the CCN dramatically by eliminating multiple decision nodes and also potentially making the function more readable. A potential drawback would be that it might affect the consistency of the code style in the project, if switch statements are the go-to solution in these problems.

After refactoring, the CCN went from 15 to 5, a reduction of 67%. This was done according to the plan mentioned above, where a map was used to store key-value pairs instead of a switch statement. Map has a lookup time of O(1) which most likely also improved the speed of the function, as the pre-refactor function potential had to go through multiple switch cases to find its match.

Refactor Plan for LoggerWithConfig() [Giacomo]

The LoggerWithData() function is a large complex function that deals with the logging with the right configuration. The length and high complexity of the function makes it hard to read, maintain, and debug.

To improve clarity and maintainability, we will separate these concerns into many dedicated subfunctions:

- processRequest()
- processTemplate()
- writeOutput()
- handleTag()
- getColoredStatus

Benefits of Refactoring:
- Improved Readability
- Better Maintainability
- Easier Debugging

After refactoring, LoggerWithConfig() will delegate work to functions, significantly reducing its complexity. Also, by dividing the function into smaller components the complexity of the function LoggerWithConfig is reduced a lot. Indeed, it goes from a CCN=49 to a CCN=5, spreading some complexity to other functions.

After completing the refactoring, I can present the following results:

Tool	**LOC**	**CC**	**Function**
Lizard	145	49	LoggerWithConfig() before
Lizard	21	5	LoggerWithConfig() after

The overall result makes it clear that, it's true that cyclomatic complexity is a structural characteristic of a function, however it's not difficult to reduce it drastically thanks to refactoring.

[Lin]

Note: my assigned function is originally *Find()*, but after carefully reading, I found it's a routing search algorithm to generate corresponding handlers by specific routes. It's more about logic algorithm rather than SE, and it contains many `goto` statements that disturb codeflow order, which makes it very hard to refactor. I extract all none-`goto` parts, but CCN only reduces from 34 to 30. So I try to refactor other functions, but they're all have too small CCN that hard to reduce further (like *key_auth* below). So I choose to refactor another teammate's assigned function *LoggerWithConfig()* but with a different implementation. To avoid file conflicts, all my refactored functions are kept in a separate branch, not merging into the main.

Refactor Plan for LoggerWithConfig() [Lin]

The high complexity in the original code is not necessary, as demonstrated by the refactored version. By using a map to associate extraction errors with corresponding error messages, the code eliminates redundant conditional checks, making it more concise and readable. This approach preserves backward compatibility while improving maintainability.

To further reduce complexity, the logic could be modularized by extracting the error handling into a dedicated function. This function would take the last extraction error as input and return the appropriate error message or the original error if no match is found. This way, the main code remains clean, and any future modifications to error handling can be done in a single place without affecting the core logic, reducing CCN from 17 to 14.

Refactor Plan for Line116-Line177 in middleware/key_auth [Lin]

This is another more extreme example of reducing CCN by replacing a large switch statement with a map for simple value assignments. This change improves readability and efficiency by centralizing lookups instead of using multiple conditional branches.

The switch statement is now reserved for cases requiring additional logic, making the code more structured and maintainable. The final CCN is 21 compared with original 38.

Refactor Plan for Find() [Lin]

This function has a high complexity (34 CC), but there is no necessity for refactoring. It leans more towards an algorithmic backend rather than the structural part of software engineering. Its main purpose is to take the given route path and execute a search algorithm starting from the root node, based on three custom node types, until either the entire tree is traversed or a fully matched route handler is found.

The algorithm extensively uses the `goto` keyword, which concentrates the complexity in the search process, making it non-sequential and difficult to refactor.

Nevertheless, I attempted to extract the non-`goto` parts into helper functions, but the results were unsatisfactory. The cyclomatic complexity (CC) only decreased from 34 to 30, clearly indicating that the complexity is primarily concentrated in the sections containing `goto`. Therefore, a complete refactor of this function is not feasible.

Coverage

Tools

Q1: Document your experience in using a "new"/different coverage tool.

The coverage tool used in this test is the built-in Go coverage tool, which was executed using the following commands:

1. `go test -coverprofile=coverage.out` - Generates a `coverage.out` file that contains raw coverage data.
2. `go tool cover -func=coverage.out > cover.txt` - Summarizes function-level coverage, which can be saved as `cover.txt`.
3. `go tool cover -html=coverage.out` - Produces an HTML file (`coverage.html`) for visual representation of coverage.

Q2: How well was the tool documented? Was it possible/easy/difficult to integrate it with your build environment?

The Go coverage tool is well-documented and straightforward to use. Since it is built into the Go toolchain, there was no need for additional installations or configurations, making it easy to integrate into the build and test environment. Running `go test` with the `-coverprofile` flag was sufficient to generate the required data.

Your own coverage tool

Q1: Show a patch (or link to a branch) that shows the instrumented code to gather coverage measurements.

The branch link is here.

Q2: What kinds of constructs does your tool support, and how accurate is its output?

Our tool results closely matches Go's built-in `go test -cover`, with minor discrepancies due to different counting methods. The table shows that our manual method provides comparable accuracy to `go test`.

	Manual	`go test`
bindData()	94.12%	97.5%

LoggerWithConfig	83.05%	90.6%
insertNode()	94.12%	98.7%
Find()	96.30%	99.1%
addMethod	97.37%	95.2%

Evaluation

1. How detailed is your coverage measurement?

Our tool provides branch-level coverage, tracking execution of `if`, `for`, `switch`, and other decision points. It offers fine-grained insights beyond simple statement coverage.

2. What are the limitations of your own tool?
- Requires manual instrumentation, making it labor-intensive.
- May miss some implicit branches handled by Go's built-in tool.
- Does not automatically generate visual reports like `go test -cover -html`.

3. Are the results of your tool consistent with existing coverage tools?

Yes, the results closely match Go's built-in `go test`, with only minor variations (within ~3%). This confirms our tool's accuracy in measuring branch coverage.

Coverage improvement

Report of old coverage:
https://drive.google.com/file/d/1BbkDX6VTywLqhkIDcCEWezbLmGiE7M6S/view?usp=sharing (Download html content and then open with browser)

Echo package coverage: 97.5%
Middleware package coverage: 92.7%

Report of new coverage:
https://drive.google.com/file/d/1wTD3LsivjdxI8wcuEKGU_IVrAO0AsE9p/view?usp=sharing

Echo package coverage: 97.6% (0.1 % units increase)
Middleware package coverage: 93.5% (0.8 % units increase)

Test cases added:
Max
- TestBasicAuth (BasicAuth 0.0% to 100.0%)
- TestBasicAuthWithConfig (BasicAuthWithConfig 86.7% to 90.0%)
- TestBind_Error (Bind 85.7% to 100.0%)
- TestBindPathParams_Error (BindPathParams 87.5% to 100.0%)

Lin

- TestGzipResponseWriter_CanPush
  TestGzipResponseWriter_CannotPush (Push 0% to 100%)
- TestRequestID_DefaultConfig (RequestId 0% to 100%)
- TestMethodOverride_SkipperAndEmptyMethod
  (MethodOverrideWithConfig 85.7% to 100%)
- TestTimeoutWithConfig_InvalidTimeout (TimeoutWithConfig 75% to 100%)

Giacomo

- TestLoggerCustomTagFunc2
- TestLoggerWithCustomHeader
- TestLoggerWithSkipper
- TestLoggerWithMalformedCustomTimeFormat

  Overall Test Coverage in file Logger.go before: 90.7%
  Overall Test Coverage in file Logger.go later: 93.0%

Simon

- TestGzipResponseWriter_Push_WithPusher
- TestGzipResponseWriter_Push_WithoutPusher (Push 0% to 100%)
- TestStatic_ServeIndexFile
- TestStatic_ServeStaticFile (Static 0% to 100%)

Kohei

-
- insertNodeTestInsertNodeMaxParamUpdate Before 94.2% to 98.7%
- TestInsertNodeSplitting Before 94.2% to 98.7%

- Overall Test Coverage in file router_test.go before: 94.12%
- Overall Test Coverage in file router_test.go before: 98.7%

Way of working

From the first moment, in the group it has been established a good channel of communication and
honesty that allowed us to collaborate efficiently despite our diverse backgrounds and initial challenges. We indeed had long discussions to organize our work, ensuring that everyone was aligned on the principles and constraints that would guide our efforts.

Compromises , Work Distribution and Initial Challenges

One of the key aspects of our Way-of-Working was our ability to find good compromises as well as a balanced workload distribution. Each team member focused on their strengths, which allowed us to overcome individual limitations. Despite differences in skill levels, we ensured that every task was assigned in a way that maximized expertise. Our team was particularly variegated, consisting of members from different parts of the world, including two exchange students. Some of us had never used Git or Go or Java before, while others were

still in their undergraduate studies. These differences presented an initial challenge, as we were not used to the tools and conventions primarily used in this context. However, through support from one another, we adapted quickly to the unfamiliar software and methodologies.

Progression Through Way-of-Working States

We successfully progressed through the early states of Way-of-Working (as stated in the checklist,
p.60 of Essence):

- **Principles Established:** We set clear principles and constraints that were agreed upon by all team members. These included how to divide work fairly, how to make decisions collectively, and how to ensure both synchronous and asynchronous communication among the different group members.
- **Foundation Established:** We identified and integrated key practices and tools necessary for our workflow, ensuring a structured foundation for our work. We also analyzed gaps in knowledge and adjusted our expectations accordingly by writing a list of our competences. We managed to choose tools, software and programming language that each of us agreed to.
- **In Use:** We actively implemented our agreed-upon methods. Each member started using Git and Java and Go at their own pace, gaining familiarity with the required tools.
- In Place: Eventually, our Way-of-Working became the standard for all team members. Despite initial difficulties, everyone was effectively using the established processes and tools to accomplish their tasks. The adaptation to unfamiliar software and methodologies was well-accomplished validating this way our approach.
- **Working Well:** our team now is touching the working-well state. We are finally understanding each other's way of working. We know we can count on them. From the team perspective we absolutely don't have any complaints. We are used to concise but effective communication. We are all using independently the main tools required in these projects. We act both as independent developers(ex. work splitting) but also as a unified entity.

However, not all team members reached such a high level of proficiency with these tools. While we managed to use them effectively to complete the assignment, they did not become second nature for everyone in daily work. This is the main reason why we consider our Way-of-Working to have reached a premature state of "Working Well".

Statement of contributions

Onboarding experience - Max & Simon

Complexity:
bindData() - Max
addMethod() - Simon
LoggerWithConfig() - Giacomo
insertNode() - Kohei

find() - Lin

Refactoring:
bindData() - Max
addMethod() - Simon
LoggerWithConfig() - Giacomo
LoggerWithConfig() - Lin
insertNode() - Kohei
find() - Lin

Tests:
TestBasicAuth - Max
TestBasicAuthWithConfig - Max
TestBind_Error - Max
TestBindPathParams_Error - Max
TestGzipResponseWriter_CanPush - Lin
TestGzipResponseWriter_CannotPush - Lin
TestRequestID_DefaultConfig - Lin
TestMethodOverride_SkipperAndEmptyMethod - Lin
TestTimeoutWithConfig_InvalidTimeout - Lin
TestLoggerCustomTagFunc2 - Giacomo
TestLoggerWithCustomHeader - Giacomo
TestLoggerWithSkipper - Giacomo
TestLoggerWithMalformedCustomTimeFormat - Giacomo
TestGzipResponseWriter_Push_WithPusher - Simon
TestGzipResponseWriter_Push_WithoutPusher - Simon
TestStatic_ServeIndexFile - Simon
TestStatic_ServeStaticFile - Simon

Coverage:
Own tool - Lin

Coverage improvement report - Max
Overall experience - All

Overall experience

This was our first time exploring test coverage and code complexity, and we found it both interesting and insightful.

We also enjoyed working on an open-source project, as it gave us the opportunity to see how others write, structure, document, and test their code.

Comparing Go, which we used for this assignment, to Java, which we used in previous ones, we found Go much easier for testing. Its built-in testing and coverage tools made the process smoother and more efficient, and we really enjoyed using them.

Overall, this was a fun and valuable assignment!