Practical Software Analysis
Analysing Analog Core

Lukas Lykkeskov Hjelmstrand
luhj@itu.dk

Michael Jeppesen
micje@itu.dk

Omid Sabihi Marfavi
omma@itu.dk

21st December 2025

Course code: KSPRSOA1KU

GitHub repository:
https://github.com/marfavi/analog-core-psa

Contents

1 Introduction

Analog Core [1] is an ASP.NET REST API that powers Café Analog's digital coffee card app. This backend is important to student life at ITU and to Café Analog's operations. In 2025 alone, users claimed over 50 000 drinks through the app, generating more than 600 000 DKK in annual revenue for the non-profit café.[1] The codebase for Analog Core is developed and maintained entirely by students (and occasionally alumni), with more than 500 commits made over seven years of development. One of the authors has contributed to the system, providing familiarity with its implementation and operational context.

We selected Analog Core as our system under test because it is actively used by the university community, the source code is free and open-source, and we had prior experience through earlier contributions. In addition, quick initial experiments revealed a server error, suggesting that the system is a relevant target for further exploration.

The Analog Core project currently has a test suite with 192 unit tests and 6 integration tests, achieving a line coverage of 42.3% (see code coverage report in appendix B.1). The project follows a common ASP.NET Core architecture: API endpoints are defined in controller classes, which typically contain minimal logic and delegate processing to services. These services are defined in separate classes and are injected into the controllers via dependency injection. This low coupling benefits testing because service classes can easily be swapped with mock implementations (for instance, when a service interacts with an external API such as a payment gateway).

In this project, we supplement the existing test suite by generating fuzzed inputs for the public-facing Web API using techniques from *The Fuzzing Book* [2]. The goal is to assess whether random and heuristically crafted HTTP requests can trigger unexpected behavior, including unhandled exceptions. Since the API is the primary interface through which Analog Core is exposed to the world, defects reachable via this surface are particularly critical. We evaluate the effectiveness of the fuzzing strategy using code coverage, both to assess exploration of execution branches and to compare against the existing test suite.

1.1 Related Work

RESTler [3] is a REST API fuzzer that we applied as an initial experiment. Running RESTler out-of-the-box against a local instance of Analog Core with seed data revealed one endpoint returning an unexpected HTTP 500 response, which we classify as a bug. After configuring RESTler to include a valid authentication token in its API calls, it revealed two additional endpoints with unexpected HTTP 500 responses. These findings both motivated our choice of system and highlighted the importance of handling authentication when fuzzing Analog Core.

[1]Figure based on internal Café Analog operational data accessed by Omid Sabihi Marfavi (maintainer), 2025.

2 Process

In this section, we describe how we prepared Analog Core to run locally in a controlled environment, how we implemented and configured our grammar-based API fuzzer, and how we collected code coverage to evaluate effectiveness. Our overall goal was to preserve production behaviour for core request handling and business logic while safely replacing external services and making experiments repeatable across runs.

2.1 Local test environment

Analog Core depends on several external components: a relational database, an email-sender service, and a MobilePay integration. Furthermore, most endpoints require authentication. To fuzz input for the system reliably, we therefore needed a local setup that preserves production-like behaviour for core API logic while safely replacing external services.

Containerised setup

To ensure we could reproduce the same environment across runs, we used Docker containers. The most important element of the container setup is the database, because many endpoints depend on persistent state and because API requests can be destructive (e.g., creating, updating, or deleting entities).

Initially, we attached a persistent volume to the database container and manually imported seed data whenever we wanted to reset the state. For fuzzing, it proved more useful to always start from a known initial state. We therefore removed the volume and added an initialisation script that restores the seed data automatically when the database container starts. After starting the containers, Analog Core itself and the fuzzer need to be started manually, but the database state is reset to the same baseline.

In addition, the repository is configured to run inside a *Dev Container* [4], which starts the Docker containers and helps ensure that all required fuzzing tools and coverage-tracking dependencies are installed consistently.

Database and seeding

Because Analog Core relies heavily on database state (and because fuzzing often explores edge cases in persistence and validation logic), we considered a real database essential in our local environment.

We initially attempted to run Analog Core with an in-memory database. However, this introduced some issues and removed others compared to the behaviour we observed when running RESTler in a more realistic setup. To avoid artifacts from an unrealistic persistence layer, we switched to Microsoft SQL Server (MSSQL), matching what Analog Core uses normally.

Many endpoints require existing entities to be present in order to succeed. For example, account registration requires selecting an education programme, which must exist in the database for the request to proceed beyond input validation. To obtain seed data, Omid (who has access to the Analog test server) created a backup of the database used in Café Analog's internal testing environment. To reduce privacy concerns, the email and name of real test users were replaced with dummy data before we used the backup to seed our local MSSQL instance. The seeded baseline is restored at the start of each fuzzing run via the database container initialisation script.

Mocking external services

We replaced the email-sender service and MobilePay integration with local substitutes to avoid external dependencies and unintended side effects during fuzzing.

Email-sender. We replaced the email-sender with a method that prints email contents to the console. This can be done with low risk of introducing behavioural changes because the original email service does not return values; it only performs the side effect of sending an email. Some emails contain links used to complete registration or login flows, and with our setup these links require manual action to follow.

MobilePay. The MobilePay service consisted of three parts that needed replacement: `AccessToken`, `EPayment`, and `Webhook`. We replaced these with empty stubs. This removes some functionality (notably completing purchases), but after manual testing and fuzzing we did not observe that it introduced additional failures in the remaining API surface.

Authentication

Analog Core requires authentication for the vast majority of endpoints. User-facing endpoints (intended for the coffee card app) authenticate through a JWT bearer token, while some service-oriented endpoints (e.g., health checks) use a fixed API key configured in a configuration file. Furthermore, the user-facing API enforces different authorisation levels based on user status, so reaching different branches requires being able to authenticate as different user roles.

To enable systematic fuzzing of authenticated endpoints, we initially modified the system under test so that any login attempt using the password `"impersonate"` succeeds; all other login attempts run as normal. Upon successful login, the system returns a JWT bearer token, which we use for subsequent API calls. To ensure we also exercise the unmodified authentication logic (and do not only cover the added `"impersonate"` branch), we additionally obtain tokens by calling the v1 login endpoint (`POST /api/v1/Account/login`) with credentials that are known to exist in the seed database.

We therefore use multiple authentication contexts: (1) an `"impersonate"` login to obtain a token for a seeded user, (2) a login with real seeded credentials to cover the original authentication flow, and (3) a login with the credentials of a board member (the highest-privilege user group) to access endpoints gated by higher authorisation. For API-key-protected endpoints, we pass an `X-API-Key` header value (`local-development-apikey`) as configured for local execution.

2.2 Fuzzing methodology

Analog Core generates an OpenAPI specification when the web API is running.[2] We fetch the OpenAPI JSON documents directly from the running instance (locally during fuzzing), using the endpoints `/swagger/v2/swagger.json` and `/swagger/v1/swagger.json`. The specification defines all endpoints, the expected parameters, and their schemas, and thus serves as a machine-readable contract for constructing requests. This makes it a strong foundation for grammar-based fuzzing.

We translate the OpenAPI specification into grammars in the format used by *The Fuzzing Book* [2], allowing us to utilise its Python fuzzing library. Our grammars and fuzzing logic are contained in a single Jupyter notebook available in our repository.[3]

Pipeline overview

At a high level, our fuzzing workflow consists of: (1) downloading the OpenAPI JSON document from a running instance, (2) translating each endpoint into an endpoint-specific grammar, (3) applying manual heuristic overrides where useful, (4) generating concrete request strings using `GeneratorGrammarFuzzer`, (5) executing the corresponding HTTP requests under a chosen authorisation context, (6) comparing observed response codes against documented OpenAPI response codes, and (7) collecting code coverage using `dotnet-coverage` during fuzzing.

Generating grammars from OpenAPI

We wrote code to parse the OpenAPI JSON document and convert it into grammars. Rather than generating one global grammar for the entire API, we generate one grammar per endpoint and represent each endpoint explicitly (Figure 1). This choice improves usability and control:

- **Maintainability and heuristics.** Per-endpoint grammars give a clearer overview and make manual editing for heuristics easier.
- **Correct request context.** When generating input for a specific endpoint, the HTTP method and documented response codes are known.
- **Controlled sequencing.** Per-endpoint execution allows us to decide how much to exercise each endpoint and to treat endpoints with stateful or destructive effects with more care.
- **Performance.** Endpoints with deterministic behaviour (e.g., deprecated endpoints consistently returning `410 Gone`) can be assigned small budgets to avoid wasting executions.

Splitting grammars by endpoint can still represent the same overall request space as one large grammar. In practice, however, it enables targeted manual configuration

[2]Swagger UI for the production OpenAPI specification: https://core.prd.analogio.dk/swagger/index.html?urls.primaryName=v2

[3]https://github.com/marfavi/analog-core-psa/blob/main/fuzz/_swagger_fuzzer.ipynb

that improves coverage using system knowledge. Consequently, while our fuzzer can generate requests from OpenAPI alone, our approach is not intended as a plug-and-play black-box fuzzer like RESTler [3]; instead, we prioritise targeted coverage improvements through endpoint-specific configuration.

```python
class Endpoint:
    def __init__(self, method: str, grammar: Grammar,
  response_codes: list[str]):
        self.method = method.upper()
        self.grammar = trim_grammar(grammar)
        assert is_valid_grammar(self.grammar)
        self.response_codes = response_codes

```

Figure 1: Class we use to represent an endpoint to be fuzzed.

OpenAPI-to-grammar translation details

Our converter translates the OpenAPI document into one grammar per endpoint by mapping path templates, query/path parameters, and JSON request bodies into grammar nonterminals.

Paths and methods. For each OpenAPI path template (e.g., `/api/v2/menuitems/{id}`), we create a `<start>` expansion that produces the concrete URL and store the HTTP method in endpoint metadata.

Query and path parameters. For path parameters, we replace `{name}` placeholders with a corresponding nonterminal (e.g., `<id.integer>`) and define its expansion based on the parameter type. For query parameters, we first construct individual `name=value` fragments and then generate query strings by enumerating all combinations of these fragments (including the empty query string). The fuzzer then samples among these query shapes during execution, which systematically explores both minimal requests (no query parameters) and maximal requests (many query parameters) without assigning explicit per-parameter probabilities.

Request bodies and optional fields. For request bodies, we translate OpenAPI object schemas into JSON fragments. Required fields are always included. Optional fields are handled by enumerating all combinations of optional properties (including the case where only required fields are present) and sampling among these request shapes. This strategy is particularly useful for endpoints with many optional fields because it covers a broad range of valid request shapes without hand-crafting each combination.

Enums, arrays, and examples. Enumerations (`enum`) are translated into a choice among the documented enum values. Arrays are supported by generating list expansions, including empty lists. When the OpenAPI schema provides an `example` value, we include it as an additional expansion to increase the likelihood of producing semantically valid inputs.

Response codes as an oracle

The OpenAPI specification includes documented response status codes annotated by developers in the C# source. We store these codes per endpoint and compare them against observed responses during fuzzing. Responses with undocumented status codes are flagged as *unexpected*. Certain status codes are broadly plausible in many contexts, such as 401 *Unauthorized* or 429 *Too Many Requests*. In other cases, undocumented codes can indicate documentation drift or faulty behaviour. This is particularly true for undocumented 5xx responses, which often indicate an unhandled exception and may also leak internal error details back to the client.

Default expansions for primitive types

For primitive data types, we defined default expansions that are reused across endpoints.

Strings. Strings include both the empty string and a randomly generated string with length up to 100 000 characters. String lengths are sampled using a geometric distribution to favour shorter strings while still occasionally generating very large values. This increases the likelihood of exercising both typical validation paths and boundary cases (e.g., maximum-length handling).

Integers. Integer generation includes common small values (`1`, `0`, `-1`) as well as a generator that produces random 32-bit integers. With 10% probability, the generator returns the minimum or maximum 32-bit integer to stress numeric bounds; otherwise, it samples uniformly within the range. For parameters that semantically represent identifiers or counts, we also use a variant that generates only positive integers.

Domain-shaped primitives. We additionally define generators for email-shaped strings and dates. These are used for endpoints whose schemas expect formats such as email addresses or date-like values.

Manual heuristic overrides

Once grammars were generated from the OpenAPI specification, we reviewed endpoints and applied manual heuristics where appropriate. Figure 2 shows an example grammar for fuzzing the v2 login endpoint, which sends a login link to the user's email address.

```
endpoint.grammar = {
    '<start>': [
      '/api/v2/account/login requestBody: <UserLoginRequest>'
    ],
    '<UserLoginRequest>': [
      '{ "email": <email.string>, "loginType": <LoginType> }'
    ],
    '<LoginType>': ['"Shifty"', '"App"'],
    '<email.string>': ['<string>', '"john@doe.com"',
      ('"random@email.com"', opts(pre=random_email))
    ],
    '<string>': ['""',
      ('"random string"', opts(pre=lambda: random_string(1,
   100000)))
    ]
}
```

Figure 2: Example of grammar for a single endpoint.

A request generated by this grammar could be:

```
/api/v2/account/login requestBody: {
    "email": "amiaktuutp@svprwq.com",
    "loginType": "Shifty"
}
```

In Figure 2, we manually added `"john@doe.com"` as a possible expansion for `<email.string>`, based on our knowledge that it corresponds to an existing user in the seed database. This increases the probability of reaching deeper logic than purely random emails would. We applied similar heuristics to other endpoints where meaningful behaviour depends on referencing valid identifiers (e.g., `productId` values that exist in the seeded data). For some endpoints, we also ensured that at least some requests succeed (or fail in a controlled way), because both outcomes can be necessary to cover different branches. One example is attempting to register an account using an email known to already exist in the seed database.

Executing endpoint fuzzing

After constructing grammars, we fuzz endpoints from the notebook by calling `endpoint.test(n, token, ...)`. The `Endpoint.test` method generates request strings using `GeneratorGrammarFuzzer`, extracts the URL and optional request body from the generated string, and sends an HTTP request using the fixed method stored in the endpoint metadata. Requests are sent with an `Authorization` header when an authentication token is provided.

Budgets and input-space size. We assign a request budget per endpoint. In our experiments, budgets were chosen from the set $\{1, 2, 10, 100, 400, 1000\}$, with 100 as the baseline for most endpoints. Smaller budgets (1–10) were used for smoke checks and for endpoints with deterministic behaviour (e.g., deprecated v1 endpoints consistently returning `410 Gone`), while larger budgets were used for endpoints with larger input spaces (e.g., endpoints with many optional query parameters). For example, we assigned a budget of 400 requests to `GET /api/v2/account/search`, which combines multiple optional query parameters, while `GET /api/v1/Leaderboard` received a minimal budget because it consistently returns `410 Gone`. A wide variety of observed status codes is a useful (though coarse) indicator that multiple execution paths are being exercised; different branches often manifest as different categories of responses.

Authorisation contexts. Most endpoints require authorisation, and some endpoints require specific roles. We therefore execute requests under different authorisation contexts depending on the endpoint: (1) anonymous (no token), (2) a JWT bearer token obtained via the v1 login endpoint, and (3) an API key for service endpoints such as health checks. In practice, the board-member JWT token is frequently used because it grants broader access and therefore reaches more branches.

Observed outcomes and debugging output. For each endpoint run, we print the documented response codes from OpenAPI and then execute the generated requests. The notebook prints a distribution of observed HTTP status codes. For debugging, the `Endpoint.test` method supports printing request/response pairs for responses matching a status-code pattern such as `4xx` or `5xx`.

2.3 Measuring coverage

To measure coverage we used the `dotnet-coverage` [5] utility, which can run an arbitrary .NET process while recording executed lines and branches and then write a coverage report when the process stops. It captures coverage from a given list of DLL assemblies, so it can collect comparable coverage whether a test suite is run with `dotnet test` or the application runs from its main entrypoint and is tested from the outside (for example, with our Python-driven fuzzer).

Instrumentation workflow

To collect coverage for fuzzed API calls, we run the Analog Core web API process under `dotnet-coverage` while the notebook executes requests against it. Once fuzzing finishes, we stop the instrumented process to flush the coverage report to disk. The resulting coverage artefact is then rendered into an interactive HTML report using *ReportGenerator* [6], which shows coverage at both file- and line-level granularity.

We use the following two commands to run Analog Core under `dotnet-coverage` and to generate the HTML report, respectively:

```
dotnet-coverage collect -f cobertura \
  -if bin/Debug/net8.0/CoffeeCard.Library.dll \
  -if bin/Debug/net8.0/CoffeeCard.WebApi.dll \
  dotnet bin/Debug/net8.0/CoffeeCard.WebApi.dll
```

```

reportgenerator -reports:output.cobertura.xml \
    -targetdir:/workspace/coverage-report \
    -classfilters:"-CoffeeCard.Library.Migrations.*;-*Mock;-*
   CoffeeCardContext" -riskhotspotassemblyfilters:"-*"
```

Assembly selection and comparability

Coverage collection is configured to include the assemblies that implement the API and its business logic (e.g., `CoffeeCard.WebApi` and `CoffeeCard.Library`). When generating our coverage reports, we have ignored some parts of the assemblies: the `DbContext` (which defines the database design), the database migrations (which are auto-generated code from the `DbContext`), and our own created Mock classes. The same inclusion/exclusion rules are used when comparing fuzzing coverage with existing test suite coverage. This ensures that any differences in reported coverage can be attributed to the testing approach rather than the measurement configuration.

Coverage summaries and links to full reports are provided in the appendices.

3 Results

3.1 Found bugs

We evaluated the API using the OpenAPI specification as an oracle for expected behaviour: for each endpoint, we compared the HTTP status codes observed during fuzzing to the status codes documented for that endpoint. Responses with undocumented status codes were flagged as unexpected. In practice, not all unexpected status codes indicate implementation faults; a handled 4xx response may simply be undocumented, whereas 5xx responses are strong indicators of unhandled exceptions.

Across 64 endpoints in the v1 and v2 of Analog Core APIs, we triggered undocumented status codes in 13 endpoints. For a subset of these endpoints, the undocumented codes were non-5xx responses (see Appendix A.1). These cases reflect documentation drift: the API returns a sensible and handled error, but the OpenAPI specification does not list it as a possible outcome. More critically, for 7 endpoints we constructed requests that resulted in HTTP 500 responses (see Appendix A.2). Since HTTP 500 indicates a server-side failure, these outcomes suggest that certain inputs can reach code paths where exceptions are not handled properly.

A representative example is the `leaderboard/top` endpoint shown in Figure 3. Café Analog's coffee card app has a leaderboard where customers compete based on purchases. This endpoint returns an ordered list of the top customers, can be called anonymously, and accepts a parameter specifying how many users to return. We found that supplying a negative number for this parameter causes the underlying SQL query to fail, producing an unhandled exception and returning HTTP 500. Notably, the response includes the database engine's error message verbatim, which may constitute an information leak.[4]

```
GET /api/v2/leaderboard/top?top=-2147483648 (Anonymous)
Response: 500 {"message":"A TOP N or FETCH rowcount value may not
    be negative."}
```

Figure 3: Request which gives response with status code 500.

Reproducing individual failures can depend on system state, particularly database contents that may be affected by earlier requests. However, because our setup resets the seed database at the beginning of each run, rerunning the fuzzing suite from a fresh seeded instance improves reproducibility. Consequently, the requests documented in Appendix A.2 are reproducible when executed against a freshly seeded instance, although some failures may require certain preconditions (such as the presence of specific entities created by earlier fuzzed requests). Overall, these findings demonstrate the grammar-based fuzzing can expose both documentation drift (undocumented responses) and issues of higher severity (unhandled exceptions) in Analog Core's public API surface.

3.2 Code coverage

To evaluate how thoroughly our fuzzing explores Analog Core, we measured line and branch coverage for the `CoffeeCard.Library` and `CoffeeCard.WebApi` assemblies. We focus on these assemblies because they contain the core logic that serves the mobile application. The `WebApi` project defines controllers and endpoint wiring, the `Library` project contains service implementations, database access, and most business logic. Since the `Library` project contains substantially more branching logic than the controller layer, it is a more likely place for bugs to emerge.

Figure 4 compares coverage achieved by (1) the existing automated test suite, (2) our fuzzing suite, and (3) both combined by merging coverage reports. The existing test suite, which is composed of 192 unit tests and 6 integration tests, achieves 48.0% line coverage for `CoffeeCard.Library` and 31.8% for `CoffeeCard.WebApi`, coresponding to 42.3% total line coverage across both assemblies (full summary in Appendix B.1). Using our fuzzing approach, we achieve 65.6% line coverage for `CoffeeCard.Library` and 74.0% for `CoffeeCard.WebApi`, corresponding to 68.5% total line coverage (full summary in Appendix B.2).

The larger increase in `WebApi` coverage is expected: fuzzing acts as integration testing and directly exercises controller endpoints, which then invoke services. Therefore, `Library` coverage is constrained by which service paths are reachable through public endpoints; some library code may not be reachable from the public API surface alone.

[4]This was one of the errors that weren't triggered in our local setup when using an in-memory database (compared to using a MSSQL database, which is being used in production).

	CoffeeCard.Library		**CoffeeCard.WebApi**		**Total**	
	Line	Branch	Line	Branch	Line	Branch
Existing	48.0%	42.4%	31.8%	36.8%	42.3%	40.8%
Fuzzing	65.6%	53.1%	74.0%	63.8%	68.5%	56.1%
Merged	81.1%	73.1%	74.3%	65.9%	79.1%	71.0%

Figure 4: Comparison of code coverage between the existing test suite and our fuzzing suite.

Compared to the existing test suite, fuzzing increases total coverage from 42.3/40.8% line/branch coverage to at least 68.5/56.1%. However, this does not imply a straightforward increase of 26.2% in line coverage, as the fuzzing and existing test suites may cover different regions of the codebase to varying degrees. To measure the overlap, we merged the coverage reports using `dotnet-coverage merge`. The merged report yields 79.1/71.0% line/branch coverage (see the full summary in Appendix B.3), indicating that the two approaches complement each other well. Notably, the merged coverage for `CoffeeCard.Library` increases to 81.1% line coverage, suggesting that fuzzing reaches code paths not covered by existing tests and vice versa.

When generating coverage reports, we excluded parts of the assemblies that are not meaningful targets for runtime coverage evaluation: the `DbContext` (schema definition), database migrations (auto-generated from the `DbContext`), and our own mock classes. These exclusions were applied consistently so that the different coverage measurements remain comparable.

The full interactive coverage reports are available at `https://marfavi.github.io/analog-core-psa/`.

3.3 Future work

Greybox (coverage-guided) fuzzing

Our current fuzzer uses a fixed number of requests per endpoint and relies on manually added heuristics to increase the probability of executing meaningful code paths. A substantial improvement would be to incorporate *greybox* (coverage-guided) fuzzing, where coverage feedback is used to retain inputs that increase coverage and to prioritise them for further mutation [2].

In our setup, the fuzzer interacts with the Analog Core process via HTTP, so coverage guidance requires an explicit feedback channel from the server. Since `dotnet-coverage` can run in server mode, the fuzzer could query coverage during execution and treat requests that increase coverage as new seeds. Concretely, the fuzzer could (1) start from a small set of valid requests per endpoint, (2) generate or mutate requests while preserving schema constraints, (3) query coverage after each request (or in batches), and (4) keep and prioritise only those requests that increase coverage. This would reduce reliance on manual heuristics and increase the likelihood of exploring rare branches, thereby improving both bug discovery and overall coverage [2].

Add more heuristics

We currently include heuristics in our grammars based on known values in the seed database, such as email addresses corresponding to existing users. This could be further improved by dynamically sampling values from the database at runtime (e.g., randomly selecting a valid user email or product identifier). Doing so would reduce coupling between the grammar and any specific seed data, making the approach more feasible to run against datasets or environments.

Docker and replayability

Our repository is configured to run in a *Dev Container* [4], ensuring that relevant tools and seed data are available consistently. At present, Analog Core and the fuzzer are started manually. A Docker-only configuration could automate the full pipeline by starting all dependencies (including Analog Core in coverage mode), running the fuzzer, and ending with the generation of a coverage report. Such a setup could also emit explicit replay steps (sequence of HTTP requests) to reproducing failures discovered during fuzzing. Since the seed database is reset at the start of each run, this pipeline would improve the reproducibility of results.

In addition, we could enhance the determinism of the fuzzing campaign by enabling users to specify a fixed random seed to drive all random choices made by the fuzzer. This would enable the fuzzer to be rerun and the same request sequence regenerated, supporting the reproduction and debugging of observed failures.

Capturing emails sent by Analog Core and following links automatically

Several flows in Analog Core send emails containing links used to complete actions such as registration and account deletion. Our current setup prints email contents to the console, requiring manual intervention to follow links. Capturing outgoing emails programmatically and extracting URLs would allow the fuzzer to complete these workflows automatically and could increase reachable coverage in authentication-related code paths.

Mutation testing as an effectiveness measure

A complementary way to evaluate the effectiveness of fuzzing would be with mutation testing, for example using *Stryker.Net* [7]. Stryker.Net works nicely out-of-the-box with test suites runnable via `dotnet test`, repeatedly executing the tests while injecting mutants and producing a mutation score. It would therefore be valuable to translate our fuzzing into a deterministic test suite. One potential approach is to execute the Jupyter notebook once, record all requests and responses, and then replay the same sequence from a .NET test framework while asserting expected responses (assuming that the API was bug-free at the time of recording). However, mutation testing would require fast and deterministic state reset across many test runs. Our current reset strategy restores the database by dropping and reloading from a backup, which would be very slow for mutation testing at scale. It may also be necessary to mock additional sources of nondeterminism such as system time.

3.4 Conclusion

In this project, we explored the Analog Core web API using grammar-based fuzzing techniques covered in the course. We implemented a tool that converts Analog Core's OpenAPI specification into grammars compatible with the fuzzing library from *The Fuzzing Book*, and then manually refined endpoint grammars using knowledge of the system under test to increase meaningful coverage.

Using this approach, we identified undocumented responses in 13 endpoints and triggered HTTP 500 responses (unhandled exceptions) in 7 endpoints. We then evaluated effectiveness using code coverage metrics and compared our results to the existing automated test suite. Our fuzzing achieved 68.5% total line coverage across the primary assemblies, compared to 42.3% from the existing tests. Notably, merging the two coverage reports increased total line coverage further to 79.1%, showing that fuzzing and conventional automated testing complement each other by covering different regions of the codebase.

In summary, the project shows that course techniques can be transferred effectively to a production codebase and evaluated in a controlled local setup, yielding concrete findings and measurable improvements in test coverage.

References

[1] AnalogIO. ‘Analogio/analog-core: .net 8 backend for cafe analog’s coffee card app,’ Accessed: 21st Nov. 2025. [Online]. Available: https://github.com/AnalogIO/analog-core.

[2] A. Zeller, R. Gopinath, M. Böhme, G. Fraser and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024, Retrieved 2024-07-01 16:50:18+02:00. Accessed: 1st Dec. 2025. [Online]. Available: https://www.fuzzingbook.org/.

[3] V. Atlidakis, P. Godefroid and M. Polishchuk, ‘Restler: Stateful rest api fuzzing,’ in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE’19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083.

[4] Microsoft. ‘Dev containers,’ Accessed: 17th Dec. 2025. [Online]. Available: https://containers.dev.

[5] Microsoft. ‘Dotnet-coverage code coverage utility,’ Accessed: 15th Dec. 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/additional-tools/dotnet-coverage.

[6] D. Palme. ‘Reportgenerator,’ Accessed: 10th Dec. 2025. [Online]. Available: https://reportgenerator.io/.

[7] S. Team. ‘Stryker.net,’ Accessed: 19th Dec. 2025. [Online]. Available: https://stryker-mutator.io/docs/stryker-net/introduction/.

Appendices

A Bugs and Undocumented responses

A.1 Responses with Undocumented Status Codes

```
PATCH /api/v2/account/-1/user-group (Authorized as Board Member)
RequestBody: { "userGroup": "Barista" }
Response: 403
---
POST /api/v2/menuitems (Authorized as Board Member)
RequestBody: { "name": "Espresso" }
Response: 409 {"message":"Menu item already exists with name
    Espresso"}
---
PUT /api/v2/menuitems/0 (Authorized as Board Member)
RequestBody: { "name": "Espresso", "active": false }
Response: 404 {"message":"No menu item was found by Menu Item Id
    0"}
---
PUT /api/v2/menuitems/17 (Authorized as Board Member)
RequestBody: { "name": "Espresso", "active": true }
Response: 409 {"message":"Menu item already exists with name
    Espresso"}
---
POST /api/v2/products (Authorized as Board Member)
RequestBody: { "price": 10, "numberOfTickets": 10, "name": "
    Espresso", "description": "Voucher codes for intro week   ", "
    visible": false, "allowedUserGroups": [], "menuItemIds":
    [888389011, 1, 0] }
Response: 409 {"message":"Product already exists with name Espresso
     and price of 10"}
---
POST /api/v2/purchases (Authorized as Board Member)
RequestBody: { "productId": -1, "paymentType": "FreePurchase" }
Response: 404 {"message":"No product was found by Product Id: -1"}
---
PUT /api/v2/purchases/0/refund (Authorized as Board Member)
RequestBody: None
Response: 404 {"message":"No purchase was found by Purchase Id: 0"}
---
POST /api/v2/vouchers/ESP-VYKZ1C/redeem (Authorized as Board Member
    )
RequestBody: None
Response: 409 {"message":"Voucher \u0027ESP-VYKZ1C\u0027 has
    already been redeemed"}
---
POST /api/v1/Tickets/use (Authorized as Board Member)
RequestBody: { "productId": -1 }
Response: 404 {"message":"Product not found"}
---
POST /api/v1/Tickets/use (Authorized as Board Member)
RequestBody: { "productId": 1 }
Response: 403 {"message":"User has no tickets for this product"}
```

A.2 Responses with 500 Status Codes

```
GET /api/v2/account/search?pageNum=1826788232
(Authorized as Board Member)
RequestBody: None
Response: 500 {"message":"The offset specified in a OFFSET clause
    may not be negative."}
---
POST /api/v2/account/login
(Anonymous)
RequestBody: { "email": "john@doe.com", "loginType": "App" }
Response: 500 {"message":"The method or operation is not
    implemented."}
---
GET /api/v2/leaderboard/top?top=-2147483648
(Anonymous)
RequestBody: None
Response: 500 {"message":"A TOP N or FETCH rowcount value may not
    be negative."}
---
POST /api/v2/products
(Authorized as Board Member)
RequestBody: { "price": 10, "numberOfTickets": 10, "name": "
    KbASiFIXQkkkojJlfvVVvDKtOikbBLuNfAeWXjPhgDf", "description": "
    wMUVjFkhdgPHkpJwTezWOBQ", "visible": false, "allowedUserGroups":
     ["Manager", "Customer", "Customer", "Board", "Customer"], "
    menuItemIds": [1, 2] }
Response: 500 {"message":"The instance of entity type \
    u0027ProductUserGroup\u0027 cannot be tracked because another
    instance with the same key value for {\u0027ProductId\u0027, \
    u0027UserGroup\u0027} is already being tracked. When attaching
    existing entities, ensure that only one entity instance with a
    given key value is attached. Consider using \
    u0027DbContextOptionsBuilder.EnableSensitiveDataLogging\u0027 to
     see the conflicting key values."}
---
PUT /api/v2/products/975523553
(Authorized as Board Member)
RequestBody: { "price": 10, "numberOfTickets": 1743392294, "name":
    "gNmQChUC", "description": "Voucher codes for intro week   ", "
    visible": true, "allowedUserGroups": ["Manager", "Board"], "
    menuItemIds": [] }
Response: 500 {"message":"Object reference not set to an instance
    of an object."}
---
PUT /api/v2/products/1
(Authorized as Board Member)
RequestBody: { "price": 2022080462, "numberOfTickets": 1, "name": "
    Espresso", "description": "UNGgmcUiMFdHFURC", "visible": true, "
    allowedUserGroups": ["Barista", "Barista", "Manager", "Customer
    "], "menuItemIds": [] }
Response: 500 {"message":"The instance of entity type \
    u0027ProductUserGroup\u0027 cannot be tracked because another
    instance with the same key value for {\u0027ProductId\u0027, \
    u0027UserGroup\u0027} is already being tracked. When attaching
    existing entities, ensure that only one entity instance with a
    given key value is attached. Consider using \
    u0027DbContextOptionsBuilder.EnableSensitiveDataLogging\u0027 to
     see the conflicting key values."}
---
POST /api/v2/purchases
(Authorized as Board Member)
RequestBody: { "productId": 1, "paymentType": "MobilePay" }
Response: 500 {"message":"Object reference not set to an instance
    of an object."}
---
GET /api/v1/Ping
(Anonymous)
RequestBody: None
Response: 500 {"message":"The method or operation is not
    implemented."}
```

B Coverage

B.1 Code Coverage of Existing Test Suite

Interactive at: https://marfavi.github.io/analog-core-psa/existing-tests-report/

Summary

Generated on:	12/18/2025 - 7:38:44 PM
Parser:	Cobertura
Assemblies:	2
Classes:	72
Files:	72
Line coverage:	42.3% (1561 of 3684)
Covered lines:	1561
Uncovered lines:	2123
Coverable lines:	3684
Total lines:	7108
Branch coverage:	40.8% (208 of 509)
Covered branches:	208
Total branches:	509

Coverage

Name	**Covered**	**Uncovered**	**Coverable**	**Total**	**Line**	**Branch**
CoffeeCard.Library	**1152**	**1248**	**2400**	**3752**	**48%**	**42.4%**
CoffeeCard.Library.Services.AccountService	131	152	283	388	46.2%	39.2%
CoffeeCard.Library.Services.EmailService	14	101	115	186	12.1%	
CoffeeCard.Library.Services.HashService	6	7	13	26	46.1%	
CoffeeCard.Library.Services.LoginLimiter	39	1	40	89	97.5%	91.6%
CoffeeCard.Library.Services.MailgunEmailSender	0	25	25	39	0%	0%

CoffeeCard.Library.Services.MapperService	0	64	64	97	0%	0%
CoffeeCard.Library.Services.ProductService	23	0	23	46	100%	
CoffeeCard.Library.Services.ProgrammeService	0	10	10	27	0%	
CoffeeCard.Library.Services.PurchaseService	0	88	88	133	0%	0%
CoffeeCard.Library.Services.SmtpEmailSender	2	16	18	34	11.1%	
CoffeeCard.Library.Services.TicketService	0	191	191	271	0%	0%
CoffeeCard.Library.Services.TokenService	80	6	86	153	93%	87.5%
CoffeeCard.Library.Services.v2.AccountService	202	66	268	392	75.3%	75%
CoffeeCard.Library.Services.v2.AdminStatisticsService	0	26	26	48	0%	
CoffeeCard.Library.Services.v2.EmailService	56	2	58	98	96.5%	25%
CoffeeCard.Library.Services.v2.LeaderboardService	52	0	52	91	100%	83.3%
CoffeeCard.Library.Services.v2.MenuItemService	0	69	69	116	0%	0%
CoffeeCard.Library.Services.v2.ProductService	83	17	100	151	83%	75%
CoffeeCard.Library.Services.v2.PurchaseService	263	103	366	523	71.8%	60.7%
CoffeeCard.Library.Services.v2.StatisticService	0	86	86	120	0%	0%
CoffeeCard.Library.Services.v2.TicketService	34	104	138	197	24.6%	16.6%
CoffeeCard.Library.Services.v2.TokenService	34	0	34	59	100%	100%
CoffeeCard.Library.Services.v2.VoucherService	55	0	55	98	100%	100%
CoffeeCard.Library.Services.v2.WebhookService	0	84	84	136	0%	0%
CoffeeCard.Library.Utils.ClaimsUtilities	16	20	36	63	44.4%	33.3%
CoffeeCard.Library.Utils.DateTimeConverter	12	0	12	36	100%	
CoffeeCard.Library.Utils.DateTimeProvider	3	0	3	12	100%	
CoffeeCard.Library.Utils.LeaderboardPresetExtensions	8	1	9	19	88.8%	75%
CoffeeCard.Library.Utils.ProductExtensions	27	0	27	50	100%	100%
CoffeeCard.Library.Utils.SemesterUtils	12	0	12	34	100%	100%
CoffeeCard.Library.Utils.StatisticUtils	0	9	9	20	0%	
CoffeeCard.WebApi	**409**	**875**	**1284**	**3356**	**31.8%**	**36.8%**
AspNetCoreGeneratedDocument.Pages_Recover	0	2	2	23	0%	
AspNetCoreGeneratedDocument.Pages_Result	0	7	7	19	0%	0%

AspNetCoreGeneratedDocument.Pages_VerifyDelete	0	2	2	7	0%	
AspNetCoreGeneratedDocument.Pages_VerifyEmail	0	2	2	7	0%	
CoffeeCard.WebApi.Controllers.AccountController	13	39	52	148	25%	50%
CoffeeCard.WebApi.Controllers.AppConfigController	0	4	4	38	0%	
CoffeeCard.WebApi.Controllers.CoffeeCardsController	0	7	7	44	0%	
CoffeeCard.WebApi.Controllers.LeaderboardController	0	4	4	38	0%	
CoffeeCard.WebApi.Controllers.MobilePayController	0	7	7	52	0%	
CoffeeCard.WebApi.Controllers.PingController	0	2	2	28	0%	
CoffeeCard.WebApi.Controllers.ProductsController	0	26	26	85	0%	
CoffeeCard.WebApi.Controllers.ProgrammesController	0	9	9	45	0%	0%
CoffeeCard.WebApi.Controllers.PurchasesController	0	15	15	82	0%	
CoffeeCard.WebApi.Controllers.TicketsController	0	24	24	100	0%	0%
CoffeeCard.WebApi.Controllers.v2.AccountController	43	44	87	285	49.4%	50%
CoffeeCard.WebApi.Controllers.v2.AdminStatisticsController	0	8	8	52	0%	
CoffeeCard.WebApi.Controllers.v2.AppConfigController	0	8	8	39	0%	
CoffeeCard.WebApi.Controllers.v2.HealthController	0	33	33	86	0%	0%
CoffeeCard.WebApi.Controllers.v2.LeaderboardController	0	15	15	72	0%	
CoffeeCard.WebApi.Controllers.v2.MenuItemsController	0	16	16	89	0%	
CoffeeCard.WebApi.Controllers.v2.MobilePayController	0	65	65	137	0%	0%
CoffeeCard.WebApi.Controllers.v2.ProductsController	0	26	26	124	0%	
CoffeeCard.WebApi.Controllers.v2.PurchasesController	0	34	34	144	0%	
CoffeeCard.WebApi.Controllers.v2.TicketsController	0	15	15	78	0%	
CoffeeCard.WebApi.Controllers.v2.VouchersController	0	17	17	90	0%	
CoffeeCard.WebApi.Controllers.v2.WebhooksController	0	8	8	50	0%	
CoffeeCard.WebApi.Helpers.ApiExceptionFilter	9	13	22	58	40.9%	33.3%
CoffeeCard.WebApi.Helpers.AuthorizeRolesAttribute	4	14	18	44	22.2%	0%
CoffeeCard.WebApi.Helpers.ConfigurationServiceCollectionExtension	24	0	24	48	100%	
CoffeeCard.WebApi.Helpers.PageUtils	0	28	28	67	0%	0%
CoffeeCard.WebApi.Helpers.ReadableBodyFilter	4	0	4	24	100%	

CoffeeCard.WebApi.Helpers.RequestLoggerMiddleware	0	48	48	89	0%	0%
CoffeeCard.WebApi.Helpers.Swagger.RemoveApiVersionProcessor	0	15	15	40	0%	0%
CoffeeCard.WebApi.Logging.Enricher	0	34	34	102	0%	0%
CoffeeCard.WebApi.Logging.LoggerConfigurationExtensions	0	5	5	24	0%	0%
CoffeeCard.WebApi.Pages.Recover	0	48	48	101	0%	0%
CoffeeCard.WebApi.Pages.Result	0	3	3	20	0%	
CoffeeCard.WebApi.Pages.VerifyDelete	0	17	17	50	0%	
CoffeeCard.WebApi.Pages.VerifyEmail	0	26	26	58	0%	0%
CoffeeCard.WebApi.Program	21	81	102	155	20.5%	25%
CoffeeCard.WebApi.Startup	291	104	395	514	73.6%	66.1%

B.2 Code Coverage of our Fuzzing

Interactive at: https://marfavi.github.io/analog-core-psa/fuzz-coverage-report/

Summary

Generated on:	12/18/2025 - 7:38:17 PM
Parser:	Cobertura
Assemblies:	2
Classes:	72
Files:	72
Line coverage:	68.5% (2529 of 3688)
Covered lines:	2529
Uncovered lines:	1159
Coverable lines:	3688
Total lines:	7108
Branch coverage:	56.1% (287 of 511)
Covered branches:	287
Total branches:	511

Coverage

Name	**Covered**	**Uncovered**	**Coverable**	**Total**	**Line**	**Branch**
CoffeeCard.Library	**1573**	**824**	**2397**	**3752**	**65.6%**	**53.1%**
CoffeeCard.Library.Services.AccountService	152	128	280	388	54.2%	41.3%
CoffeeCard.Library.Services.EmailService	79	36	115	186	68.6%	
CoffeeCard.Library.Services.HashService	13	0	13	26	100%	
CoffeeCard.Library.Services.LoginLimiter	30	10	40	89	75%	58.3%
CoffeeCard.Library.Services.MailgunEmailSender	0	25	25	39	0%	0%

CoffeeCard.Library.Services.MapperService	49	15	64	97	76.5%	25%
CoffeeCard.Library.Services.ProductService	23	0	23	46	100%	
CoffeeCard.Library.Services.ProgrammeService	7	3	10	27	70%	
CoffeeCard.Library.Services.PurchaseService	82	6	88	133	93.1%	75%
CoffeeCard.Library.Services.SmtpEmailSender	0	18	18	34	0%	
CoffeeCard.Library.Services.TicketService	154	37	191	271	80.6%	79.1%
CoffeeCard.Library.Services.TokenService	55	31	86	153	63.9%	37.5%
CoffeeCard.Library.Services.v2.AccountService	218	50	268	392	81.3%	75%
CoffeeCard.Library.Services.v2.AdminStatisticsService	26	0	26	48	100%	
CoffeeCard.Library.Services.v2.EmailService	57	1	58	98	98.2%	75%
CoffeeCard.Library.Services.v2.LeaderboardService	52	0	52	91	100%	83.3%
CoffeeCard.Library.Services.v2.MenuItemService	63	6	69	116	91.3%	70%
CoffeeCard.Library.Services.v2.ProductService	95	5	100	151	95%	87.5%
CoffeeCard.Library.Services.v2.PurchaseService	143	223	366	523	39%	27.4%
CoffeeCard.Library.Services.v2.StatisticService	46	40	86	120	53.4%	35.2%
CoffeeCard.Library.Services.v2.TicketService	117	21	138	197	84.7%	58.3%
CoffeeCard.Library.Services.v2.TokenService	25	9	34	59	73.5%	50%
CoffeeCard.Library.Services.v2.VoucherService	5	50	55	98	9%	0%
CoffeeCard.Library.Services.v2.WebhookService	0	84	84	136	0%	0%
CoffeeCard.Library.Utils.ClaimsUtilities	27	9	36	63	75%	75%
CoffeeCard.Library.Utils.DateTimeConverter	12	0	12	36	100%	
CoffeeCard.Library.Utils.DateTimeProvider	3	0	3	12	100%	
CoffeeCard.Library.Utils.LeaderboardPresetExtensions	8	1	9	19	88.8%	75%
CoffeeCard.Library.Utils.ProductExtensions	23	4	27	50	85.1%	87.5%
CoffeeCard.Library.Utils.SemesterUtils	9	3	12	34	75%	50%
CoffeeCard.Library.Utils.StatisticUtils	0	9	9	20	0%	
CoffeeCard.WebApi	**956**	**335**	**1291**	**3356**	**74%**	**63.8%**
AspNetCoreGeneratedDocument.Pages_Recover	0	2	2	23	0%	
AspNetCoreGeneratedDocument.Pages_Result	5	2	7	19	71.4%	50%

AspNetCoreGeneratedDocument.Pages_VerifyDelete	0	2	2	7	0%	
AspNetCoreGeneratedDocument.Pages_VerifyEmail	0	2	2	7	0%	
CoffeeCard.WebApi.Controllers.AccountController	42	10	52	148	80.7%	50%
CoffeeCard.WebApi.Controllers.AppConfigController	4	0	4	38	100%	
CoffeeCard.WebApi.Controllers.CoffeeCardsController	7	0	7	44	100%	
CoffeeCard.WebApi.Controllers.LeaderboardController	4	0	4	38	100%	
CoffeeCard.WebApi.Controllers.MobilePayController	7	0	7	52	100%	
CoffeeCard.WebApi.Controllers.PingController	2	0	2	28	100%	
CoffeeCard.WebApi.Controllers.ProductsController	24	2	26	85	92.3%	
CoffeeCard.WebApi.Controllers.ProgrammesController	9	0	9	45	100%	100%
CoffeeCard.WebApi.Controllers.PurchasesController	15	0	15	82	100%	
CoffeeCard.WebApi.Controllers.TicketsController	24	0	24	100	100%	100%
CoffeeCard.WebApi.Controllers.v2.AccountController	82	5	87	285	94.2%	50%
CoffeeCard.WebApi.Controllers.v2.AdminStatisticsController	8	0	8	52	100%	
CoffeeCard.WebApi.Controllers.v2.AppConfigController	8	0	8	39	100%	
CoffeeCard.WebApi.Controllers.v2.HealthController	28	5	33	86	84.8%	50%
CoffeeCard.WebApi.Controllers.v2.LeaderboardController	15	0	15	72	100%	
CoffeeCard.WebApi.Controllers.v2.MenuItemsController	16	0	16	89	100%	
CoffeeCard.WebApi.Controllers.v2.MobilePayController	0	65	65	137	0%	0%
CoffeeCard.WebApi.Controllers.v2.ProductsController	26	0	26	124	100%	
CoffeeCard.WebApi.Controllers.v2.PurchasesController	26	8	34	144	76.4%	
CoffeeCard.WebApi.Controllers.v2.TicketsController	14	1	15	78	93.3%	
CoffeeCard.WebApi.Controllers.v2.VouchersController	14	3	17	90	82.3%	
CoffeeCard.WebApi.Controllers.v2.WebhooksController	8	0	8	50	100%	
CoffeeCard.WebApi.Helpers.ApiExceptionFilter	19	3	22	58	86.3%	83.3%
CoffeeCard.WebApi.Helpers.AuthorizeRolesAttribute	18	0	18	44	100%	100%
CoffeeCard.WebApi.Helpers.ConfigurationServiceCollectionExtension	24	0	24	48	100%	
CoffeeCard.WebApi.Helpers.PageUtils	20	8	28	67	71.4%	50%
CoffeeCard.WebApi.Helpers.ReadableBodyFilter	4	0	4	24	100%	

CoffeeCard.WebApi.Helpers.RequestLoggerMiddleware	0	48	48	89	0%	0%
CoffeeCard.WebApi.Helpers.Swagger.RemoveApiVersionProcessor	0	15	15	40	0%	0%
CoffeeCard.WebApi.Logging.Enricher	34	0	34	102	100%	100%
CoffeeCard.WebApi.Logging.LoggerConfigurationExtensions	4	1	5	24	80%	50%
CoffeeCard.WebApi.Pages.Recover	0	48	48	101	0%	0%
CoffeeCard.WebApi.Pages.Result	3	0	3	20	100%	
CoffeeCard.WebApi.Pages.VerifyDelete	13	4	17	50	76.4%	
CoffeeCard.WebApi.Pages.VerifyEmail	0	26	26	58	0%	0%
CoffeeCard.WebApi.Program	90	15	105	155	85.7%	68.7%
CoffeeCard.WebApi.Startup	339	60	399	514	84.9%	73.5%

B.3 Union of Code Coverage of our Fuzzing and Existing Test Suite

Interactive at: https://marfavi.github.io/analog-core-psa/merged-report/

Summary

Generated on:	12/18/2025 - 8:12:58 PM
Parser:	Cobertura
Assemblies:	2
Classes:	72
Files:	72
Line coverage:	79.1% (2953 of 3729)
Covered lines:	2953
Uncovered lines:	776
Coverable lines:	3729
Total lines:	7108
Branch coverage:	71% (329 of 463)
Covered branches:	329
Total branches:	463

Coverage

Name	Covered	Uncovered	Coverable	Total	Line	Branch
CoffeeCard.Library	**1965**	**436**	**2401**	**3752**	**81.8%**	**73.1%**
CoffeeCard.Library.Services.AccountService	203	81	284	388	71.4%	52.2%
CoffeeCard.Library.Services.EmailService	79	36	115	186	68.6%	
CoffeeCard.Library.Services.HashService	13	0	13	26	100%	
CoffeeCard.Library.Services.LoginLimiter	39	1	40	89	97.5%	91.6%
CoffeeCard.Library.Services.MailgunEmailSender	0	25	25	39	0%	0%

CoffeeCard.Library.Services.MapperService	49	15	64	97	76.5%	25%
CoffeeCard.Library.Services.ProductService	23	0	23	46	100%	
CoffeeCard.Library.Services.ProgrammeService	7	3	10	27	70%	
CoffeeCard.Library.Services.PurchaseService	82	6	88	133	93.1%	75%
CoffeeCard.Library.Services.SmtpEmailSender	2	16	18	34	11.1%	
CoffeeCard.Library.Services.TicketService	154	37	191	271	80.6%	79.1%
CoffeeCard.Library.Services.TokenService	83	3	86	153	96.5%	91.6%
CoffeeCard.Library.Services.v2.AccountService	267	1	268	392	99.6%	97.3%
CoffeeCard.Library.Services.v2.AdminStatisticsService	26	0	26	48	100%	
CoffeeCard.Library.Services.v2.EmailService	57	1	58	98	98.2%	75%
CoffeeCard.Library.Services.v2.LeaderboardService	52	0	52	91	100%	100%
CoffeeCard.Library.Services.v2.MenuItemService	63	6	69	116	91.3%	70%
CoffeeCard.Library.Services.v2.ProductService	100	0	100	151	100%	100%
CoffeeCard.Library.Services.v2.PurchaseService	325	41	366	523	88.7%	78.7%
CoffeeCard.Library.Services.v2.StatisticService	46	40	86	120	53.4%	35.2%
CoffeeCard.Library.Services.v2.TicketService	117	21	138	197	84.7%	58.3%
CoffeeCard.Library.Services.v2.TokenService	34	0	34	59	100%	100%
CoffeeCard.Library.Services.v2.VoucherService	55	0	55	98	100%	100%
CoffeeCard.Library.Services.v2.WebhookService	0	84	84	136	0%	0%
CoffeeCard.Library.Utils.ClaimsUtilities	27	9	36	63	75%	87.5%
CoffeeCard.Library.Utils.DateTimeConverter	12	0	12	36	100%	
CoffeeCard.Library.Utils.DateTimeProvider	3	0	3	12	100%	
CoffeeCard.Library.Utils.LeaderboardPresetExtensions	8	1	9	19	88.8%	
CoffeeCard.Library.Utils.ProductExtensions	27	0	27	50	100%	100%
CoffeeCard.Library.Utils.SemesterUtils	12	0	12	34	100%	100%
CoffeeCard.Library.Utils.StatisticUtils	0	9	9	20	0%	
CoffeeCard.WebApi	**988**	**340**	**1328**	**3356**	**74.3%**	**65.9%**
AspNetCoreGeneratedDocument.Pages_Recover	0	2	2	23	0%	
AspNetCoreGeneratedDocument.Pages_Result	5	2	7	19	71.4%	50%

AspNetCoreGeneratedDocument.Pages_VerifyDelete	0	2	2	7	0%	
AspNetCoreGeneratedDocument.Pages_VerifyEmail	0	2	2	7	0%	
CoffeeCard.WebApi.Controllers.AccountController	42	10	52	148	80.7%	
CoffeeCard.WebApi.Controllers.AppConfigController	4	0	4	38	100%	
CoffeeCard.WebApi.Controllers.CoffeeCardsController	7	0	7	44	100%	
CoffeeCard.WebApi.Controllers.LeaderboardController	4	0	4	38	100%	
CoffeeCard.WebApi.Controllers.MobilePayController	7	0	7	52	100%	
CoffeeCard.WebApi.Controllers.PingController	2	0	2	28	100%	
CoffeeCard.WebApi.Controllers.ProductsController	24	2	26	85	92.3%	
CoffeeCard.WebApi.Controllers.ProgrammesController	9	0	9	45	100%	100%
CoffeeCard.WebApi.Controllers.PurchasesController	15	0	15	82	100%	
CoffeeCard.WebApi.Controllers.TicketsController	24	0	24	100	100%	100%
CoffeeCard.WebApi.Controllers.v2.AccountController	84	3	87	285	96.5%	
CoffeeCard.WebApi.Controllers.v2.AdminStatisticsController	8	0	8	52	100%	
CoffeeCard.WebApi.Controllers.v2.AppConfigController	8	0	8	39	100%	
CoffeeCard.WebApi.Controllers.v2.HealthController	28	5	33	86	84.8%	50%
CoffeeCard.WebApi.Controllers.v2.LeaderboardController	15	0	15	72	100%	
CoffeeCard.WebApi.Controllers.v2.MenuItemsController	16	0	16	89	100%	
CoffeeCard.WebApi.Controllers.v2.MobilePayController	0	65	65	137	0%	0%
CoffeeCard.WebApi.Controllers.v2.ProductsController	26	0	26	124	100%	
CoffeeCard.WebApi.Controllers.v2.PurchasesController	26	8	34	144	76.4%	
CoffeeCard.WebApi.Controllers.v2.TicketsController	14	1	15	78	93.3%	
CoffeeCard.WebApi.Controllers.v2.VouchersController	14	3	17	90	82.3%	
CoffeeCard.WebApi.Controllers.v2.WebhooksController	8	0	8	50	100%	
CoffeeCard.WebApi.Helpers.ApiExceptionFilter	19	3	22	58	86.3%	83.3%
CoffeeCard.WebApi.Helpers.AuthorizeRolesAttribute	18	0	18	44	100%	100%
CoffeeCard.WebApi.Helpers.ConfigurationServiceCollectionExtension	24	0	24	48	100%	
CoffeeCard.WebApi.Helpers.PageUtils	20	8	28	67	71.4%	50%
CoffeeCard.WebApi.Helpers.ReadableBodyFilter	4	0	4	24	100%	

CoffeeCard.WebApi.Helpers.RequestLoggerMiddleware	0	48	48	89	0%	0%
CoffeeCard.WebApi.Helpers.Swagger.RemoveApiVersionProcessor	0	15	15	40	0%	0%
CoffeeCard.WebApi.Logging.Enricher	34	0	34	102	100%	100%
CoffeeCard.WebApi.Logging.LoggerConfigurationExtensions	4	1	5	24	80%	50%
CoffeeCard.WebApi.Pages.Recover	0	48	48	101	0%	0%
CoffeeCard.WebApi.Pages.Result	3	0	3	20	100%	
CoffeeCard.WebApi.Pages.VerifyDelete	13	4	17	50	76.4%	
CoffeeCard.WebApi.Pages.VerifyEmail	0	26	26	58	0%	0%
CoffeeCard.WebApi.Program	90	17	107	155	84.1%	71.4%
CoffeeCard.WebApi.Startup	369	65	434	514	85%	77.4%