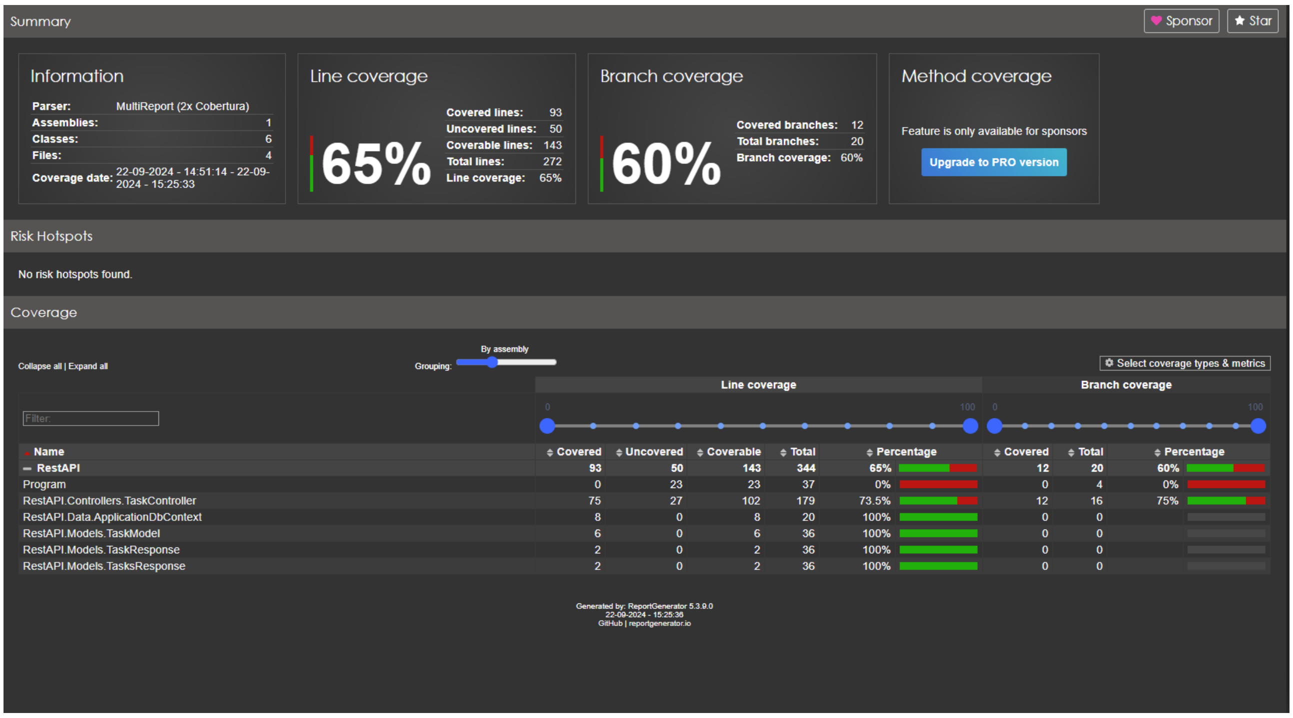
Code coverage

Managed to get the controller to 73.5% code line coverage.



Summary

Sponsor | Star

Information

Parser:	MultiReport (2x Cobertura)
Assemblies:	1
Classes:	6
Files:	4
Coverage date:	22-09-2024 - 14:51:14 - 22-09-2024 - 15:25:33

Line coverage

65%

Covered lines:	93
Uncovered lines:	50
Coverable lines:	143
Total lines:	272
Line coverage:	65%

Branch coverage

60%

Covered branches:	12
Total branches:	20
Branch coverage:	60%

Method coverage

Feature is only available for sponsors

Upgrade to PRO version

Risk Hotspots

No risk hotspots found.

Coverage

Collapse all | Expand all

Grouping: By assembly

Select coverage types & metrics

	Line coverage					Branch coverage		
Name	Covered	Uncovered	Coverable	Total	Percentage	Covered	Total	Percentage
RestAPI	93	50	143	344	65%	12	20	60%
Program	0	23	23	37	0%	0	4	0%
RestAPI.Controllers.TaskController	75	27	102	179	73.5%	12	16	75%
RestAPI.Data.ApplicationDbContext	8	0	8	20	100%	0	0	
RestAPI.Models.TaskModel	6	0	6	36	100%	0	0	
RestAPI.Models.TaskResponse	2	0	2	36	100%	0	0	
RestAPI.Models.TasksResponse	2	0	2	36	100%	0	0	

Generated by: ReportGenerator 5.3.9.0
22-09-2024 - 15:25:36
GitHub | reportgenerator.io

Software Review

The code performs the CRUD operations as expected, handles exceptions and return the correct HTTP status codes for the different errors and success. One improvement I could make to the code would be to create a repository and a service. The repository would handle the data access, the service would handle the business logic and the controller would handle the REST API. The reason I haven't done this so far is because it's a quite small and simple application with only one model task and it usually when applications get big and complex that using a repository-service-controller pattern makes it easier to understand the code. There is quite a lot of things going on in my controller though, so it is something to consider. I could also add some logging inside the catch blocks which would help with debugging and add timestamps to the responses.

Reflection on testing and Code Quality

Static code analysis tools like FxCop help improve code quality by helping detect issues like potential bugs and inefficient code patterns. By using it in our code we can ensure that code quality is continuously monitored and reducing the chance of bugs. I have used it to help avoid parallel executions of queries and adding ConfigureAwait(false) which can help improve performance and avoid deadlocks.

Mocking allows you to simulate the behavior external systems ensuring the tests focus solely on the logic within the function being tested. This isolation ensures that tests remain fast and reliable. Mocking also helps test edge cases such as handling third-party API failures or network latency. I have not used the Moq framework for my unit tests since I had some issues with it and couldn't get it to work. Instead, I used an in-memory database to simulate the actual database.

Code reviews and software reviews have an important role in ensuring high code quality through collaboration and knowledge sharing. They are used as a secondary check for potential errors and inefficiencies that automated tools might miss. Software reviews guarantee that the application meets the business requirements. Since I am working alone, I couldn't do the peer code review part, but I tried my best to review the code myself.

Equivalence Partitioning and Boundary Value Analysis are valuable techniques when designing tests. Boundary Value Analysis was used to create a test that minimum length set for the description in the controller works. Equivalence Partitioning tests check that the model has all the required values that it needs to have to be inserted into the database.