Software Quality Deliverable 2.3

**Parking Garage** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Fixed/Improved Analysis Design

*[we used the same design*​*!*]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Quality Report

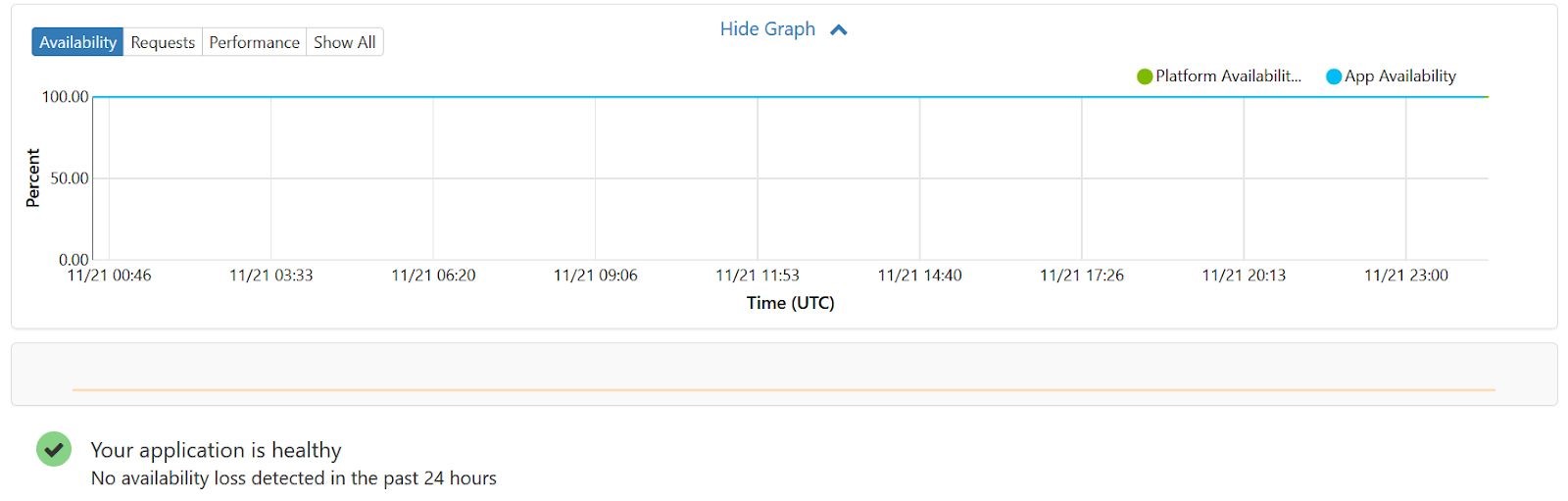
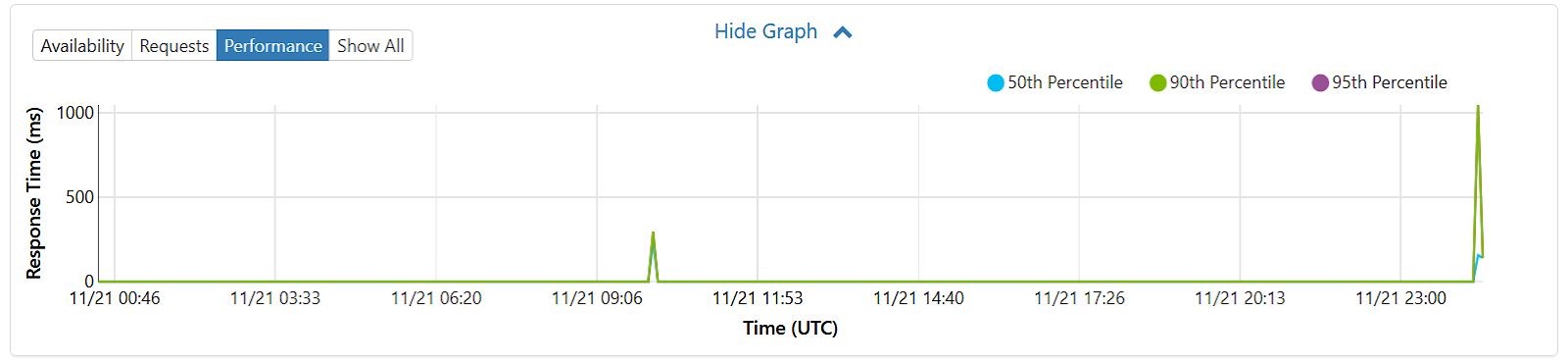
**Product Quality Metrics**

**Quality:** ​Availability

**Goal:**​ The system shall work continuously.

**Metrics:**​ The system should have 99% availability on the day.

**Validation Results:** ​According to the Azure portal, there were no failed requests, either to the web front-end or to the SQL database backend. Most responses were returned from the server on the order of milliseconds, though there were some latency spikes on the order of seconds at the 95th percentile. No timeouts were observed under normal load. Additionally, Azure is configured to dynamically scale up or down various resources under load. Alert emails are sent to service maintainers for upcoming maintenance and high CPU usage.

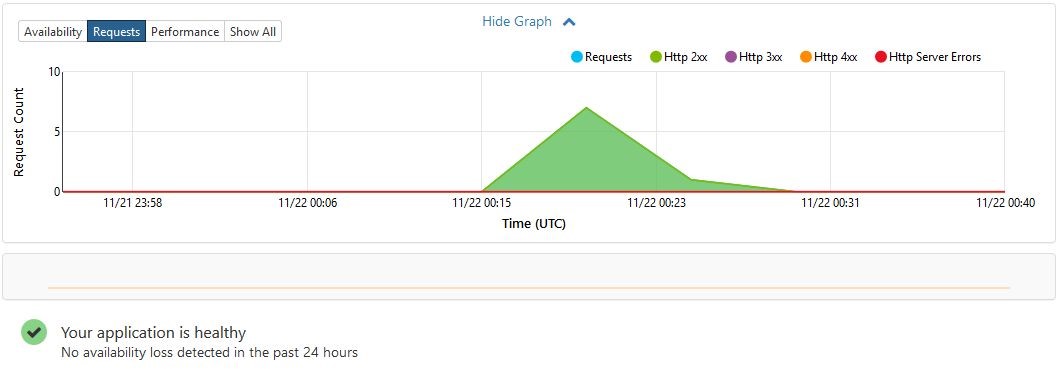


**Quality:** ​Reliability

**Goal:** ​The system shall work correctly at all times.

**Metric:**​ The error rate for operations should be below 1%.

**Validation Results:** Azure web portal shows 100% availability for the backend SQL database. For the web app itself, no requests returned internal service errors.



During the prototyping phase, some user queries incorrectly returned HTTP 400 error codes due to a bug in how the current list of checked out books were retrieved from the database. Once that bug was fixed, the 400 errors stopped.

**Quality:**​ ​Robustness

**Goal:** ​The system should handle a variety of inputs.

**Metric:** ​There should be zero errors that are a function of input data.

**Validation results:** All stories where a user can enter data were tested on Unicode inputs by hand and in test, with no failures. This was expected, given that C# strings support Unicode by default and all string fields in the database are specified as NVarChars which also support Unicode. Excessively long input strings could generate failures for librarians adding a book to the database - some validation logic should be added for such inputs.

**Quality**​: ​Learnability

**Goal:** ​The system should be easy to learn and use.

**Metric:**​ A new user should be able to use the system effectively in under a day.

**Validation results:** ​Potential users of the system learned in approximately ten minutes how to use all of the functionality of the prototype: namely, searching for books, checking them out, and placing holds.

**Quality:**​ ​Usability

**Goal:** The system should be user-friendly.​

**Metric:** There should be no more than three complaints from users when trying to learn the system.

**Validation results:** Users were dissatisfied with the lack of detail in the UI, particularly with respect to due dates and details in the hold emails. These complaints were consistent across the testers. Work should be scheduled to add the necessary details.

**Quality:**​ ​Efficiency

**Goal:**​ The system should feel responsive.

**Metric:** ​The system should return search results in less than one second at the 95th percentile.

**Validation results:** As mentioned in the section on availability, there were five minute intervals during test where latency exceeded one second. Such spikes were rare, on the order of once per day. Profiling these spikes indicated that the spikes correlate with operations that result in multiple calls to the database in series, like checking out books. There are certain operations in the DatabaseAccessor that could be done asynchronously where they are currently synchronous which could improve end to end latencies.

Yu Nong 3/30

**Quality:**​ ​Integrity

**Goal:** They data should be integrated for more than 3 years

**Metric:**​ The data leakage accidents should be less than 1 in 3 years

**Validation results:** There has not been full audit done. The connection between the web server and the database management system is based on TLS 1.3, which strongly encrypts the data between the web server and the database management system. However, the password for logging in the database system is short and easy-to-guess. Moreover, the web application in our system is based on HTTP protocol directly without any encryption. All the packets sent between the clients and the server can be easily captured and parsed. The system has no SQL injection checking to the content of submitted forms. Thus, An SQL injection attacking is possible to our system.

**Quality:**​ ​Portability

**Goal:** The system should be easily moved to a new platform

**Metric:**​ The system should work correctly on different devices, operating systems, database systems.

**Validation results:** This is a web-based application. All devices and operating systems which have a web browser can access the system without any issues. All queries and operations to the database system are based on standard SQL statements. There is no issue for different relational database systems. However, the system cannot moved to non-relational database systems without modifying the database query and operation statements.

**Process Quality Metrics**

**Quality:**​ ​Maintainability

**Goal:** ​The system should be easy to maintain.

**Metric:** The source code should not contain lexical and design anti-patterns[3][6], and it should be well-documented and readable [1][2].

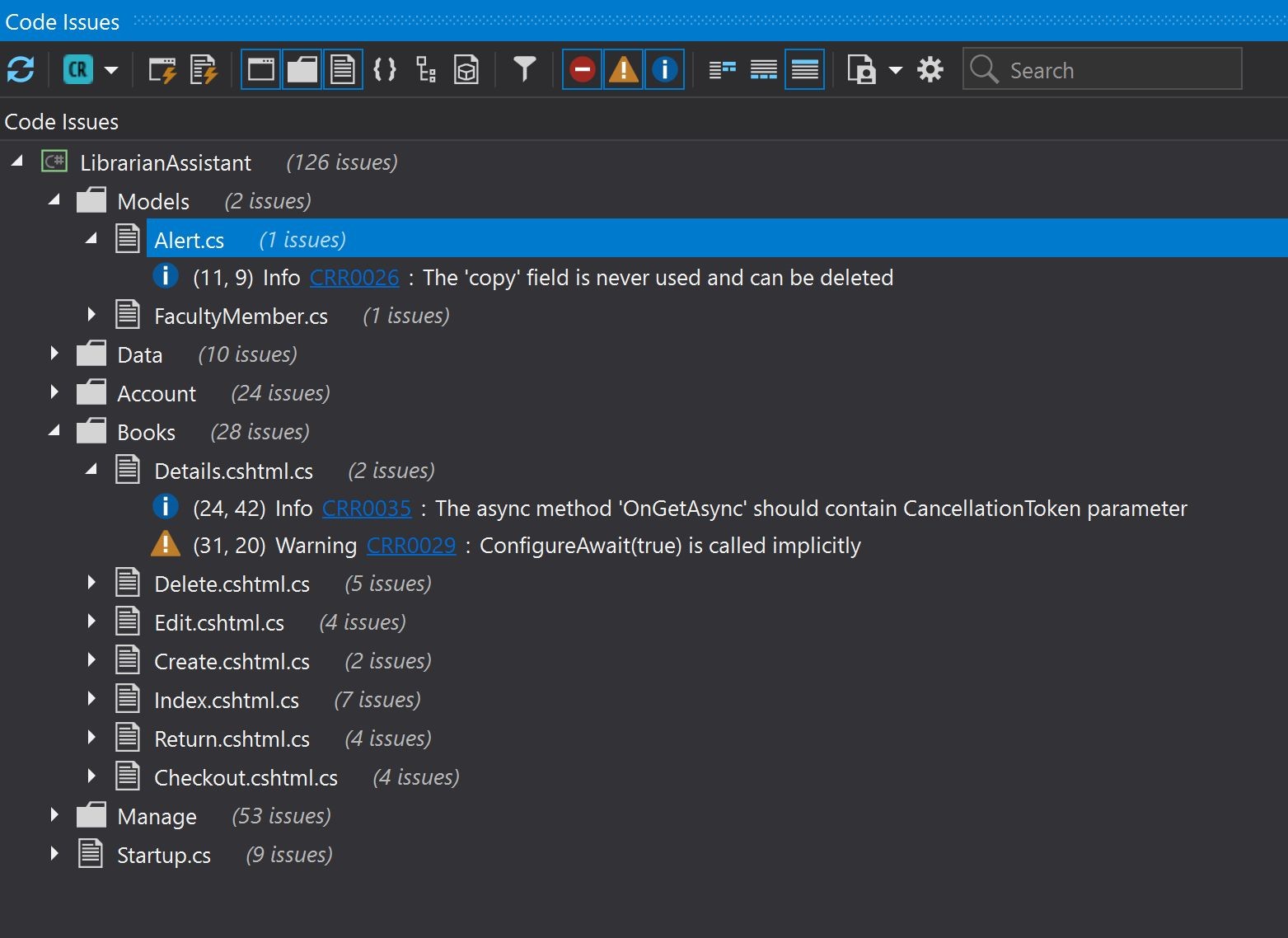
**New Metric:** The development team made the decision to only commit untangled changes. This makes the review of pull requests and reverting changes much easier. Overall the maintenance of the system is improved.

**Validation results:** The first run of ReSharper’s style checker on the solution revealed 3896 “issues.” Of these, 1 represented an actual logical issue in the code base. Other issues with the codebase are the tight coupling of all classes with the DatabaseAccessor class and the design of the database tables themselves, which would be better designed as a relational model.

There is currently no tool support for detecting linguistic antipatterns in c# code. So we ask an expert to review the source code for any lexical code smells. A few minor issues arose with documentation, but otherwise there were no lexical code smells.

We use static analysis tool CodeRush

(​<https://marketplace.visualstudio.com/items?itemName=DevExpress.CodeRushforRoslyn>​[)](https://marketplace.visualstudio.com/items?itemName=DevExpress.CodeRushforRoslyn) to analyze the source code for issues.



CodeRush is incredibly helpful and detected 126 source code issues. We address the majority of these issues. Some issues had to do with code cleanup, like removing unused fields. Other issues were related to making the project more fail-safe! CodeRush cleans the project automatically and removes all unused references.

**Quality:**​ ​Testability

**Goal:** The system should be easily testable.​

**Metric:** Every class must have both white and black box testing, with integration/mutation testing where possible.[11]

**Validation results:** The system is not easily testable at all. It turns out that mocking databases in ASP.NET is nontrivial, because adding new fields to a class that do not correspond to a database entry causes exceptions to be thrown on binding. Luckily, most of the functionality for the app is contained in the DatabaseAccessor class, which can be unit tested directly. It is also difficult to test UI elements automatically, though frameworks like Selenium [(](https://www.seleniumhq.org/)​<https://www.seleniumhq.org/>​) do exist to try to make this easier. We implemented a potion of the tests using Selenium, but full code coverage is not yet achieved.

*Planned future tests:*

Code Coverage tests

Mutation Testing

# Quality Goals

*See the table of quality goals below.*

|  |  |  |  |
| --- | --- | --- | --- |
| Product Quality | Quality Goals | Quality Metrics | Strategy |
| Product Operation |  |  |  |
| Correctness | The number of correctness issues should be less than 5 | The functionality issue reports should less than 5 in the whole lifecycle | Provide beta version to allow customers to experience and provide feedback |
| Reliability | The system shall work 99% accurately | The crashing frequency should be less than 3 per year | Stress testing |
| Usability | A new user can learn it in one day | A new user should be able to start using it with less than a day’s training | Provide beta version to allow customers to experience and provide feedback |
| Integrity | They data should be integrated for more than 3 years | The data leakage accidents should be less than 1 in 3 years | Unit test on normal cases and boundary cases |
| Efficiency | The system should solve an intended problem in one second | The system should finish all the data query and manipulation in less than one second | Unit test on normal cases and boundary cases |
| Product Revision |  |  |  |
| Maintainability | A bug or an error can be fixed in one day | A developer should be able to trace back to the documents in one day. | Code Review |
| Flexibility | The system can be easily changed | The changing in some parts should not affect the whole running. | Objected Oriented Design |
| Testability | The system should be testable | Every class must available for white and black box testing. | Mutation and Unit testing tools |
| Product Transition |  |  |  |
| Reusability | Parts of the system should be able to be reused in another system | Modules in the system can be used in another system in less than 1 week configuring | Standard interface design and Objected Oriented Design |
| Portability | The system should be easily moved to a new platform | The system should work correctly on different devices, operating systems, database systems | Standard interface design |
| Interoperability | The system should be able to interaction with other systems | The system should be able to work with other data management systems | Standard interface design |