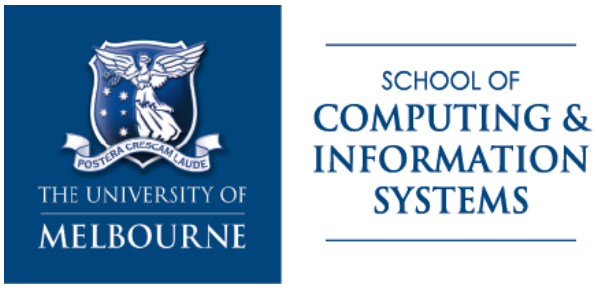
Performance Report

SWEN90007

Marketplace System
Team: Agora

Student Name	Student ID	UniMelb Username	GitHub Username	Email
Daniel Blain	831953	dblain	djblain	dblain@student.unimelb.edu.au
Christopher Byrnes	747295	byrnesc	chrisbyrnes	byrnesc@student.unimelb.edu.au
Gonzalo Molina	1085253	gmolina	GonzMol	gmolina@student.unimelb.edu.au
Mengjiao Wei	1242147	mengjiaow1	mengjiaowei	mengjiaow1@student.unimelb.edu.au



Revision History

Date	Version	Description	Author
25/10/2022	04.00-D01	Initial test plan	Mengjiao Wei
26/10/2022	04.00-D02	Added to test plan	Mengjiao Wei
26/10/2022	04.00-D03	Refined test plan	All
29/10/2022	04.00-D04	Added testing results	Gonzalo Molina
29/10/2022	04.00-D05	Added draft implemented patterns	Christopher Byrnes
1/11/2022	04.00-D06	Refined testing results	Gonzalo Molina, Daniel Blain
1/11/2022	04.00-D07	Added draft patterns not implemented	Gonzalo Molina, Daniel Blain
1/11/2022	04.00-D08	Refined draft implemented patterns	Christopher Byrnes
3/11/2022	04.00-D09	Finalised implemented patterns	Christopher Byrnes
3/11/2022	04.00-D10	Finalised patterns not implemented	Gonzalo Molina, Daniel Blain
4/11/2022	04.00-D11	Finalised document content	All
4/11/2022	04.00-D12	Proofreading	All

Table of Contents

1. Introduction

1.1. Purpose of Document

This document describes the performance of the various functionalities of the Agora Marketplace system, including a discussion of the patterns and principles followed and suggestions for alternative solutions.

1.2. Target Users

This document is aimed at the team developing the Agora Marketplace system as well as the teaching staff responsible for assessing the system and work done.

1.3. Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	Description
Admin	Special User responsible for overseeing the system's use and managing Listings and Users; short-hand for Administrator
Auction Good	A Good which multiple site users may place a bid on to try and acquire the item; the Good is sold to the user who places the highest bid
Bidding	The act of placing a bid on an Auction good
Buyer	A Marketplace User who purchases Goods
Co-seller	A nominated User which co-manages a Listing created by a Seller
Fixed Price Good	A Good, or Goods, to be sold at a single, non-negotiable price
Good	A product detailed for sale in a Listing
Listing	An entry for a product to be sold as it appears on the Marketplace site
Marketplace	The digital storefront proposed in this document, for buying and selling physical Goods
Order	A purchase, complete or incomplete, of a Good, detailing payee and shipping details
Seller	A Marketplace User who creates Listings for Goods to be sold
User	An individual, or business, registered with the Marketplace system, who can use its services to purchase or sell Goods

1.4. Actors

Actor	Description
Administrator	Manages User accounts and Listings; only one Administrator exists for the Marketplace system
All users	Refers to Standard users and Administrators collectively
Standard User	Standard users can create and purchase listings on the system.

2. Performance Testing

Detailed in the table below is a summary of timings when calling the various endpoints of the Agora Marketplace system. These tests were performed using JMeter, and note also that redirects were disabled so that only the timings of the specific endpoints were accounted for in each test.

Endpoint	Method	Samples	Min (ms)	Max (ms)	Average (ms)	Throughput	Error (%)	Notes
/account-detail	GET	1000	98	1383	551	41.7 / sec	0	
/all-accounts	GET	500	55	92	58	16.6 / sec	0	Admin feature tested without concurrent users
/cancel-order	POST	1000	1870	5160	2884	3.2 / sec	96	High error rate after failed update due to rollback, which is expected behaviour
/checkout	GET	1000	338	4740	1671	14.3 / sec	0	
/checkout	POST	1000	241	4224	2059	11.5 / sec	0	
/create-auction-listing	GET	1000	59	1074	430	53.1 / sec		
/create-auction-listing	POST	1000	71	1066	380	57.4 / sec	0	
/create-fixed-price-listing	GET	1000	95	1131	437	51.3 / sec	0	
/create-fixed-price-listing	POST	1000	62	922	414	54.2 / sec	0	
/deactivate-account	POST	500	572	964	641	0.93 / sec	0	Admin feature tested without concurrent users
/deactivate-listing	POST	500	206	480	235	1.4 / sec	0	Admin feature tested without concurrent users

Endpoint	Method	Samples	Min (ms)	Max (ms)	Average (ms)	Throughput	Error (%)	Notes
/edit-account	GET	1000	135	1935	860	28.2 / sec	0	
/edit-account	POST	1000	851	13937	11508	2.1 / sec	0	
/edit-listing	GET	1000	329	2393	1084	21.6 /sec	0	
/edit-listing	POST	1000	412	21469	1730	22 / sec	0	
/edit-order	GET	1000	307	3921	1841	13.0 / sec	0	
/edit-order	POST	1000	441	7726	3879	7.7/ sec	0	
/edit-password	GET	1000	3	920	207	90.0 / sec	0	
/edit-password	POST	1000	1064	7007	3740	6.4 / sec	0	
/index	GET	1000	106	1386	572	39.8 / sec	0	
/index	POST	1000	104	1230	561	40.4 / sec	0	
/login	GET	1000	577	4093	2139	10.6 / sec	0	
/login	POST	1000	433	3090	1759	12.7 / sec	0	
/logout	GET	1000	0	2	0	167.3/ sec	0	
/logout	POST	1000	0	2	0	168.0 / sec	0	
/order	GET	1000	298	3552	1865	12. 9 / sec	0	
/place-bid	POST	1000	19	4889	2627	15.8 / sec	15.8	Bids deleted after creation; also, since bid ordering cannot be guaranteed, the error rate indicates how often a lower

Endpoint	Method	Samples	Min (ms)	Max (ms)	Average (ms)	Throughput	Error (%)	Notes
								amount has been attempted as a bid and failed
/purchases	GET	1000	178	2077	1170	20.7 / sec	0	
/register	GET	1000	43	528	376	128.0 / sec	0	
/register	POST	1000	2410	5347	4046	11.5 / sec	0	
/sales	GET	1000	7687	10623	9608	2.6 / sec	0	Loading 10 sales
/listing	GET	1000	166	1472	608	36.9 / sec	0	
/all-listings	GET	500	489	861	568	1.7 / sec	0	Loading 10 listings; admin feature tested without concurrent users

3. Performance of Patterns & Principles Implemented

The following section comprises a discussion on the patterns and principles implemented that have an effect on performance. Any patterns or principles that do not have a significant effect on the performance of the system have been omitted.

3.1. Data Mapper

The data mapper is responsible for converting objects from the database into domain objects, and vice versa, with each mapper handling a specific conversion. Its impact on performance is therefore directly correlated to the difference between database and domain layer objects. In the Agora Marketplace system, there are several key differences between these two types of objects, and because of this, there is added complexity within the mappers. Most notably, there is no Co-Seller domain object; instead, co-sellers for a given listing are retrieved as Standard User objects in the domain. To achieve this, the Listing mapper must make an additional call to the database which queries the co-seller table joined with the user table, which is a more complex query. Additionally, there is only a single User table in the database, as opposed to the two kinds of User objects in the domain, but this conversion has simple logic and does not require additional database calls to resolve.

The obvious alternative to the data mappers would be to have identical domain and database object designs, meaning that no conversion logic would be necessary; however, this would result in extremely tight coupling and poor scalability. Other alternatives would be the table data gateway, row data gateway, or active record patterns. The first two, however, have compatibility issues with the domain model, leading to increased overheads with conversion of the data. Active record, meanwhile, while strongly coupled with both domain logic and the database, has scalability issues, such that it is difficult to use with more complex domain logic and also object-oriented design choices due to the database coupling. As such, performance-wise, the data mapper provides the best trade-off between complexity and compatibility, and since most of the domain model objects are reasonably similar to the database schemas in the Agora Marketplace system, conversion overheads are tolerable.

3.2. Unit of Work

Unit of work keeps track of changes to domain objects during a single business transaction, and when the transaction is committed, writes all changes to the database in one go. The unit of work pattern improves performance in a number of ways. Firstly, for transactions that only result in a few small changes to a large set of objects, only the objects with changes are updated, rather than the entire set. This results in fewer database calls, and hence, higher throughput. Secondly, a single database connection is used for the entire *commit* method, which is much more efficient than creating a new connection for each query. Furthermore, using a single database connection for the entire *commit* method means that if something goes wrong, the entire transaction can be undone using the *rollback* method, which is more efficient than manually tracking and undoing changes. Unit of work was implemented for creating and updating listings

and orders, as well as deactivating/cancelling users, listings and orders. Hence, the following endpoints utilise unit of work:

- POST /cancel-order
- POST /checkout
- POST /create-auction-listing
- POST /create-fixed-price-listing
- POST /deactivate-account
- POST /deactivate-listing
- POST /edit-listing
- POST /edit-order

As seen in section 2, these endpoints exhibit mixed performance; /deactivate-listing (POST) has an average response time of 235 ms, whereas /cancel-order (POST) has an average response time of 2884 ms. This is likely due to the increased complexity of cancelling an order, as it also requires updating the corresponding listing stock.

The alternative to unit of work is delegating responsibility for updating objects to the controllers or service layer functions. This approach can lead to lots of small database calls, which can end up slowing down the transaction. Furthermore, it requires that a single database connection is maintained for the entire transaction, which can affect the performance of other transactions if they are waiting for a connection to become available.

3.3. Lazy Load

Lazy load aims to reduce the amount of data being read from the database by only reading a subset of the data initially, and then loading other fields only when they are needed. The main motivation for lazy load is to improve performance of the system, particularly on pages that load a large number of objects at once. The team implemented lazy load in the *Listing, Order* and *User* classes. Hence, the following endpoints make use of lazy loading:

- GET /all-accounts
- GET /index
- POST /index
- GET /purchases
- GET /sales
- GET /all-listings

Using lazy load for the above endpoints helps to reduce response time and increase throughput on the respective pages. Whilst most of these endpoints have fairly high response times, the cause of this is

discussed in section 3.5 and is not attributed to the lazy load implementation. Had lazy load not been used when loading these pages, the response times would likely be even higher.

The alternative to lazy load is to just load all of the data at once. Whilst this may be a better choice if lazy load is causing ripple loading, if ripple loading is not an issue, it will most definitely result in poorer performance.

3.4. Identity Field

The identity field pattern saves the database primary key(s) in domain objects to maintain a reference between in-memory objects and database rows. For all of the domain objects, a UUID is auto generated when calling the constructor to create a new object. This does not require any connections or calls to the database and is an extremely fast way to generate keys. The resulting keys are meaningless, simple and database-unique. The following endpoints create new domain objects and hence evoke this key creation method:

- POST /checkout
- POST /create-auction-listing
- POST /create-fixed-price-listing
- POST /place-bid
- POST /register

Alternative methods for key generation include allowing the database to auto-generate keys, conducting table scans or implementing a key table. Since all of these methods require calls to the database, they are likely to result in slower response times when creating new objects. Using meaningful keys could improve performance because they do not require the addition of an *id* field, however, within the Agora marketplace domain, most objects do not have fields that can be used as simple meaningful keys. Using compound keys would allow for the use of meaningful keys, but having to look up multiple fields to identify an object in the database is slower than looking up a single field. If the identity map pattern was implemented, the decision to use database-unique keys over table-unique keys may have provided further improved performance, as only a single identity map would be needed, rather than one per class.

3.5. Foreign Key Mapping

Foreign key mapping involves mapping associations between domain objects to foreign key references between tables in the database. There are a number of foreign key references in the Agora marketplace database, a couple of examples are: each *Listing* references a *User* (owner) and *Bids* (if it is an auction), and each *Order* references a *User* (buyer), a *Listing*, and *ShippingDetails* for the order. Currently when fetching an object from the database, whenever there are association relationships, the mapper for the object containing the reference calls the mapper(s) for the referenced object(s). To make matters worse, the current implementation does not maintain a single database connection while fetching all of the referenced objects, which is resulting in high response times due to the overhead of establishing database

connections. Furthermore, there is a limit on the number of connections available and when the system is under load, often this limit is reached and the database issues an error declaring there are too many client connections. The *OrderMapper* in particular is especially problematic as it currently establishes a minimum of five database connections, plus one for every co-seller and bid on the referenced listing. The effect of this can be seen in the performance testing (section 2) for the following endpoints, which fetch multiple orders:

- GET /purchases
- GET /sales

These endpoints have some of the highest response times (1170 ms and 9608 ms respectively) and lowest throughputs (20.7/s and 2.6/s respectively) in the whole system. To improve performance, the mappers should be modified to use a single database connection for association relationships. Another option is to use joins to fetch all of the data needed in a single query, however, this would lead to increased coupling, which is something the data mapper pattern aims to reduce.

There are not really any alternatives to foreign key mapping that are compatible with an SQL database and the domain model pattern.

3.6. Association Table Mapping

Association table mapping involves creating a dedicated table in the database to handle many-to-many relationships. Since each *Listing* can have multiple co-sellers and each *User* can be a co-seller for multiple listings, association table mapping was used to handle this relationship. The current implementation suffers from the same problems described in section 3.5; when fetching a listing, a new database connection is created for each co-seller on the listing. Performance therefore suffers as the number of co-sellers increases.

3.7. Single Table Inheritance

The single table inheritance pattern maps all attributes of all classes in an inheritance hierarchy into a single table. The type of each object is recorded in the table and any attributes that do not exist in each object are left as null. With regard to performance, there are both pros and cons for this pattern. When loading rows to in-memory objects, the type field needs to be read to determine which type of domain object to create and which fields are needed. While this adds logic to the datasource layer, it is unlikely to have a noticeable effect on performance. Another consideration is the number of null fields in the database - if the objects in the inheritance hierarchy do not share many attributes, then this pattern could lead to lots of null fields, which may have an effect on performance. Single table inheritance was implemented for the *StandardUser* and *Admin* classes, which inherit from the *User* class, and *FixedPriceListing* and *AuctionListing* classes, which inherit from the *Listing* class. Since the child classes in each hierarchy inherit most or all of their attributes from the parent classes, having lots of null fields in the database is not an issue in this domain. On pages that load all of the classes within a single hierarchy,

the single table inheritance pattern is likely to improve performance, as only a single table needs to be queried. The endpoints that benefit from this are:

- GET /all-accounts
- GET /index
- POST /index
- GET /all-listings

As seen in the performance testing (section 2), the /all-accounts (GET) endpoint has one of the lowest average response times of 58 ms, which demonstrates how fast it is to fetch a collection of objects from a single table containing multiple types of domain classes.

Class table inheritance and concrete table inheritance are two alternative inheritance patterns. Class table inheritance involves mapping each domain class directly to a single table in the database. Whilst this pattern does not suffer from wasted space in the database, loading an object into memory requires table joins or multiple queries, which are less efficient than a single query. Furthermore, the superclass tables can easily become a bottleneck since they are used in each query. The concrete table inheritance pattern involves creating one table per concrete class in the hierarchy. Each table contains columns for all of the attributes in the mapped classes, including those inherited from parent classes. Similar to class table inheritance, this pattern does not suffer from wasted space, and it also eliminates bottlenecks caused by superclass tables, however, there are a couple of downsides. Table joins or multiple calls are still required when loading a collection of objects of different types (e.g., *Users* or *Listings*), and all tables must be checked when querying an instance whose type is not known. Since there are lots of pages in the marketplace that require loading collections of objects of different types, and all of the objects in each inheritance hierarchy share most of their attributes, the single table inheritance pattern is likely to have the best performance in this domain.

3.8. Authentication and Authorisation

When accessing many endpoints, the Agora Marketplace system will use an Authentication Enforcer perform a check against the database to ensure that the currently-logged in user is authorised to access the page, and that their account has not been deactivated. Additionally, user information is not stored in the client session; rather, only a session ID is maintained, which is used to authenticate the user with the server. As such, a read of the database to validate the user object is required everywhere authentication is required. While not a complex operation, only requiring a single database call, it does add additional overhead to most system functions.

As seen in the results table in section 2, endpoints which do not require authentication, such as /login (GET and POST), /listing, /register (GET and POST), and /index (GET and POST), do not have significantly lower average times than other operations requiring authentication, relative to the other complexity of the operations themselves. This indicates that the performance overhead of the Authentication Enforcer is low and therefore tolerable in the current system.

An alternative to the current approach would be to store the user object so that the additional database read is not required. The user object would either be stored with the server or the client. The latter is not preferred due to security issues. In either case, it would mean however that the user is not validated against the database when performing various actions, and hence could perform forbidden tasks after being deactivated by the Admin. Currently, only the user ID is stored with the server so that the full user object can be fetched.

3.9. Concurrency Control - Unique Column Constraints

The UNIQUE constraint ensures that the data in the corresponding column (or columns) is unique amongst all rows in the table. Whenever a new row is inserted, PostgreSQL checks if the value is already in the table and if it is, throws an exception. When declaring a column as unique, PostgreSQL automatically creates a unique index for that column, which is represented as a B-Tree. B-Trees have a time complexity of *O(log n)* for searches, inserts and deletes (where 'n' is the number of elements in the tree), which is extremely efficient. For the Agora marketplace, a unique constraint was enforced for the *username* and *email* fields in the *user* table, to ensure that they are unique amongst all users. This means the constraint is handled by the database when calling the following endpoints:

- POST /edit-account
- POST /register

As seen in the performance testing (section 2) both of these endpoints have relatively high average response times of 11508 ms and 4046 ms, respectively. After investigation it was discovered that the controllers are also checking whether the *username* and *email* fields are unique. This was how the constraint was implemented initially and the team forgot to remove this logic after delegating responsibility to the database. Removing the checks in the controllers would help to improve performance. Furthermore, hashing the provided password is deliberately slow to prevent brute force attacks. The time it takes to hash the password cannot be reduced without compromising the security of the system.

An alternative to declaring the *username* and *email* fields as unique in the database is to manually check that the supplied *username* and *email* do not already exist, either within the relevant controllers or mapper functions. This would require locking the entire *user* table to ensure that no new rows are created while performing the check and iterating through all objects in the table. Clearly this approach would be much less efficient than delegating responsibility to the database. It would also have a significant impact on the liveness of the system as users would be blocked from logging in or performing other actions that require authentication while the table is locked.

3.10. Concurrency Control - Online Row-Level Locking

FOR UPDATE in PostgreSQL locks all of the rows retrieved by the SELECT statement. Any other transactions that attempt to lock, modify or delete the locked rows are blocked until the current transaction ends. If the other transactions are also retrieving the rows FOR UPDATE (which is also acquired for DELETEs and UPDATEs), they are queued and continue to execute once the locks are released. Row-level locking was implemented in the unit of work for deactivating/cancelling users, listings and orders to ensure that deactivation/cancellation always goes through. A row lock is also acquired on a listing when placing a bid so that multiple bid requests are queued, and multiple bids are created if they are all valid. The row-level locking method is therefore used by the following endpoints:

- POST /cancel-order
- POST /deactivate-account
- POST /deactivate-listing
- POST /place-bid

As seen in the performance testing (section 2), the majority of these endpoints have a fairly low throughput, however this is expected as the transactions obtain a row lock and will wait until the locks are available rather than returning an exception when they are not.

The alternatives to row-level locking are setting the transaction isolation level for the transaction, or using optimistic or pessimistic offline locking. Setting the transaction isolation level would result in a similar performance to row-level locking, however, the same type of locks are applied for all of the data accessed during the transaction, which may impact performance if objects that do not need to be locked are being locked. Unlike the FOR UPDATE statement, transactions accessing the same data are not queued, instead an exception is thrown, which would decrease the response time in some cases but may not be the desired behaviour of the system. Optimistic offline locking also throws an exception when conflicts occur, but is likely to exhibit poorer performance than setting the transaction isolation level because multiple database calls are required when conflicts occur. Pessimistic offline locking involves implementing some form of a lock manager to manage locks on various objects. Since this form of concurrency control does not require access to the database, which could be a bottleneck in some cases, it is likely to provide slightly better performance than the other options, however, it would be dependent on the specific lock manager implementation.

3.11. Concurrency Control - Optimistic Offline Locking

Optimistic offline locking involves saving a version number with each record in the database, and loading that version number into the respective domain objects. When committing any changes to the database, the version numbers are compared, and if they are different, the transaction is rolled back. As mentioned in section 3.10, when conflicts do occur, an additional call to the database is required to confirm the cause of the exception, which can lead to a slightly slower response time than alternatives. This pattern does

not, however, lock objects for the entire length of the transaction, which means if conflicts are not likely to occur very often, the pattern provides good liveness. This correlates to increased throughput for other transactions accessing the same data. Optimistic locking was implemented for modifying *Listings* and *Orders* (which are unlikely to have frequent conflicts), and therefore has an effect on the following endpoints:

- POST /edit-listing
- POST /edit-order

Compared to the endpoints using online row-level locking (section 3.10), these endpoints exhibit a slightly higher throughput, especially /edit-listing (POST) which has an average throughput of 22/s. This is because rather than waiting for the objects in the database to become available before carrying out each transaction, the transactions execute and conflicts are detected at commit time.

The alternatives to optimistic offline locking are pessimistic offline locking, or some form of online locking where the database handles conflicts. For both modifying *Listings* and *Orders*, pessimistic locking would have a significant impact on system liveness and other transactions attempting to access objects of these types would be blocked while changes are being made. The online locking methods discussed in section 3.10 would also suffer from the same problem.

4. Performance of Patterns & Principles Not Implemented

4.1. Identity Map

Identity map is a pattern that aims to improve the performance of each transaction by keeping a record of all objects that it has read from the database in memory. It ensures that each object loaded from the database is only loaded once into memory, and any changes made to that object are done in memory. Whenever an object is required by a transaction, it checks if the object is already stored in the identity map, whenever it is not, it is loaded from the database into that identity map. At the end of the transaction each object kept in the identity map is then saved back to the database.

The use of the Identity map pattern would have increased performance of the Agora system as currently whenever an object is required by a transaction, it is directly loaded from the database into memory. However it does not keep track of the object kept in memory and if the object is needed again in the same transaction then it has to be reloaded from the database back into memory. This causes memory inefficiencies by loading the same object to memory more than once and slows the system down by having to re-establish a database connection and fetch from the database whenever an object is required.

It is important to note, however, that the identity map pattern needs to be implemented per transaction rather than system wide in order to benefit from the increase in performance it provides. It should not be implemented as a singleton that is shared across all sessions and transactions, as that would essentially be replicating the database in memory without all of the added benefits the database provides (such as concurrency management).

4.2. Pipelining

Pipelining involves sending multiple requests asynchronously, rather than waiting for each individual request to complete before moving onto the next. In this way, pipelining can improve performance by allowing the application to continue while waiting for the result of a request (assuming that this result isn't yet needed) and reduce the waiting time to complete the function. To accomplish this, however, would make the system more complex - asynchronous message passing and handling would need to be implemented via multithreading.

Pipelining could have been used throughout the Agora Marketplace system in conjunction with the Authentication Enforcer. Because the Authentication Enforcer loads in the user object from the database, it serves as an extra overhead to functions which require loading other data. By pipelining these requests, other data could be loaded prior to their use in anticipation of successful authorisation, rather than waiting for authentication to pass before loading in this data. As the Authentication Enforcer is used in most server functions, this would no doubt have led to a system-wide performance increase.

4.3. Caching

Caching involves storing data in a local or remote storage medium outside of the database as a means of providing faster access to data which is requested more frequently or with higher probability. This is useful, particularly when dealing with remote databases, as it can sometimes greatly decrease the amount of time spent fetching a resource (caused by network latency). However, there are some drawbacks. If the cache is too large, then lookup of data within the cache may still take some time. Additionally, if the data in the database has been updated, then the cached data may be out-of-date and display incorrect information to the user.

Caching has not been implemented in the Agora Marketplace system, although there are situations where it could have been. On many pages, a resource is loaded in the GET request, and then loaded again when a POST update occurs. Instead, caching could have been used to retain the resource between the two requests, preventing a possibly unnecessary read from the database. This could have been implemented with an identity map (discussed in section 4.1). However, this implementation was avoided in the system's development due to a lack of guarantee that the resource had not been updated by another user after the initial GET request.