УДК 004.65:004.056.2

ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ ДАННЫХ В МНОГОПОТОЧНОМ TCP-СЕРВЕРЕ НА JAVA ПРИ ИСПОЛЬЗОВАНИИ HIBERNATE

Толстик А.С., Герасимович А.Ю., Скворец И.С., студенты

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Сидоров С.В. – старший преподаватель

Аннотация. В статье исследуется проблема обеспечения целостности данных в многопоточных серверных системах при конкурентном доступе. Предложен многоуровневый подход к защите данных для Java-сервера, принимающего TCP-запросы и использующего Hibernate. Рассмотрена послойная архитектура, включающая инфраструктурную маршрутизацию, сервисную валидацию, механизм логического удаления, транзакционное управление сессиями и оптимистическую блокировку. Результаты нагрузочного стресс-тестирования подтверждают высокую эффективность предложенной модели: система успешно предотвращает потерю обновлений и нарушение согласованности объектного графа, надежно преобразуя конкурентные транзакционные конфликты в контролируемые диагностические отказы без скрытого повреждения базы данных.

Ключевые слова. Многопоточный сервер, целостность данных, конкурентный доступ, ORM, Hibernate, управление транзакциями, оптимистическая блокировка, логическое удаление, потеря обновлений, послойная архитектура, стресс-тестирование, Java.

Многопользовательские серверные системы функционируют в условиях параллельного доступа к общим данным. При одновременной обработке запросов к одним и тем же записям или к связанным сущностям возникают потеря обновлений, конфликтные изменения и нарушение согласованности объектного графа. Для backend-разработки эта проблема носит прикладной характер, поскольку ошибка проявляется не только в таблицах базы данных, но и в поведении маршрутизации, проверке входных данных, правилах предметной области и в работе слоя сохранения. Поэтому обеспечение целостности целесообразно рассматривать как согласованную работу нескольких контуров защиты: от приема и нормализации запроса до транзакционной фиксации изменений в объектно-реляционном отображении (далее – ORM) и базе данных [1]. Практической основой исследования выступает многопоточный Java-сервер, принимающий клиентские запросы по Transmission Control Protocol (далее – TCP) и использующий Hibernate для работы с разработанными сущностями отделов, сотрудников, проектов и связей участия [2].

Рассматриваемый сервер построен по послойной схеме обработки запроса [3]. Клиентская команда принимается TCP-сервером, далее проходит инфраструктурный слой маршрутизации, контроллерный и сервисный контуры, после чего изменения фиксируются через Hibernate в реляционной базе данных. На рисунке 1 представлена макро-схема прохождения клиентского запроса.

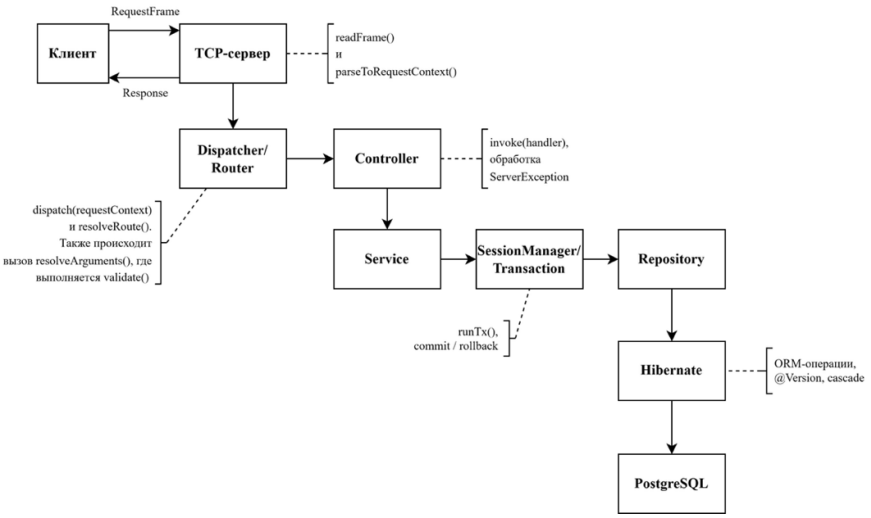

Рисунок 1 – Макро-схема обработки клиентского запроса

Представленная схема отражает не только порядок прохождения запроса, но и распределение ответственности между слоями.

Первая линия защиты формируется еще до обращения к бизнес-логике. Запрос поступает в сервер в унифицированном формате, после чего нормализуется, сопоставляется с зарегистрированным маршрутом и подготавливается к вызову конкретного обработчика. На этом этапе

исключаются обращения к несуществующим маршрутам, а аргументы контроллерного метода извлекаются централизованно из пути и тела запроса. Такой подход важен с позиции целостности, поскольку сервисный слой получает уже структурированные и типизированные данные, а не произвольный набор пользовательских значений.

Практический смысл этого уровня заключается в раннем отсечении некорректных состояний. Если путь не соответствует шаблону, тело запроса отсутствует либо идентификатор не приводится к требуемому типу, выполнение прекращается до начала предметной операции. В результате ошибка не маскируется под сбой ORM или базы данных, а локализуется в точке фактического возникновения. Для многопоточного сервера это принципиально, поскольку чем раньше отклонен некорректный запрос, тем меньше вероятность вовлечения в работу уже управляемых сущностей и транзакционных ресурсов.

После прохождения маршрутизации управление передается контроллерам и сервисам. Здесь обеспечивается прикладная согласованность данных. Сервисные методы проверяют существование и активность связанных сущностей, не допускают создания объектов по неполному или противоречивому набору идентификаторов и применяют правила предметной области до фиксации изменений. По этой причине целостность на сервисном уровне выражается не только в формальной валидности полей, но и в корректности самого сценария изменения данных.

В текущем сервере это проявляется в нескольких типовых механизмах. При создании или изменении сущностей выполняется проверка ссылок на связанные объекты, а логическое удаление реализуется через soft delete, при котором запись исключается из активного набора без физического удаления строки. Для связи сотрудника и проекта предусмотрена отдельная обработка: активная дублирующая связь запрещается, а ранее удаленная может быть восстановлена вместо повторного создания новой записи. Аналогично, при удалении сотрудника или проекта предварительно деактивируются связанные записи участия. Тем самым сервер предотвращает появление активных связей, ведущих к логически удаленным объектам. С прикладной точки зрения это означает, что система поддерживает не просто корректность отдельных строк, а согласованность всего связанного набора данных.

Следующий контур защиты связан с транзакционным выполнением операций. Все изменяющие действия должны выполняться атомарно, то есть либо завершаться полностью, либо не оставлять после себя частично зафиксированного результата. Именно для этого используется централизованный менеджер сессий, который открывает Hibernate Session, начинает транзакцию, выполняет прикладную операцию и в зависимости от результата либо подтверждает изменения, либо выполняет rollback. С точки зрения надежности многопоточного сервера это исключает ситуацию, когда одна часть связанного изменения уже записана, а другая не завершилась из-за исключения.

Hibernate рассматривает Session как контекст персистентности, внутри которого сущности находятся в состояниях transient, managed, detached или removed, а flush синхронизирует накопленные изменения с базой данных [4]. Такая модель важна для понимания того, почему целостность нельзя сводить только к изоляции на стороне СУБД. Между чтением данных, построением объектного графа и фиксацией изменений существует промежуток, внутри которого и возникает значительная часть конкурентных конфликтов. Именно в этом промежутке становятся критичными оптимистическая блокировка и корректное управление состояниями сущностей.

Ключевым механизмом предотвращения потерянных обновлений в рассматриваемом сервере выступает поле версии, помеченное аннотацией @Version. Такое поле используется поставщиком персистентности для оптимистической блокировки, а при записи сущности в базу данных версия в памяти должна сравниваться с версией в хранилище [5]. @Version позволяет обнаруживать конфликтующие обновления и предотвращать потерю изменений, которая возникала бы при стратегии last commit wins [6]. Механизм таков, что version-столбец включается в условие обновления, благодаря чему устаревший снимок строки не может незаметно перезаписать более новую версию.

Роль базы данных в этой архитектуре остается страхующей и одновременно завершающей. На уровне схемы поддерживаются обязательность критичных полей, обязательность ссылок в сущности связи и уникальность пары сотрудник-проект. Это означает, что даже при ошибке на верхних слоях система хранения не позволит зафиксировать ряд структурно противоречивых состояний. Дополнительно значим режим изоляции транзакций PostgreSQL. Уровень Read Committed используется по умолчанию, а Repeatable Read обеспечивает более сильные гарантии видимости данных в рамках транзакции [7].

Однако для прикладного сервера одного уровня изоляции недостаточно, когда чтение и последующая модификация разделены во времени и сопровождаются логикой ORM. По этой причине надежная защита целостности достигается только сочетанием транзакций, оптимистической блокировки, сервисных правил и ограничений схемы.

Работоспособность предложенного подхода проверялась стресс-тестом многопоточного TCP-сервера при 15 одновременных клиентах, каждый из которых последовательно отправлял по 250 запросов. Нагрузочный сценарий включал чтение коллекций и отдельных записей, создание, частичное обновление и логическое удаление сущностей, а также получение связанных данных по маршрутам сотрудник-проекты и проект-сотрудники. Результаты стресс-теста представлены на рисунке 2.

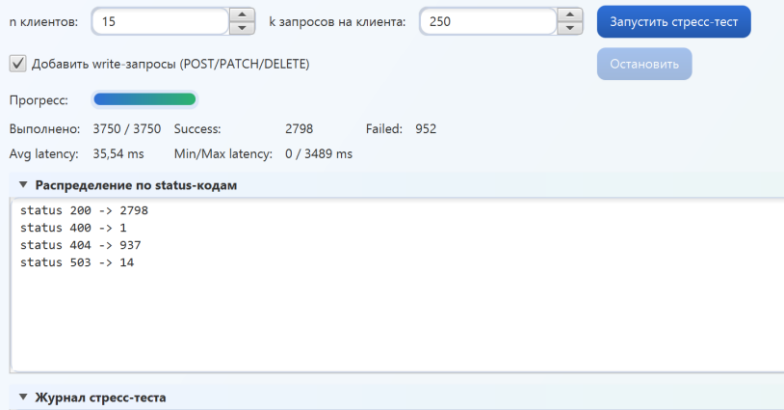

Рисунок 2 – Результаты стресс-теста

В ходе эксперимента выполнено 3750 тестовых запросов. Успешно завершились 2798 операций, неуспешно – 952. Средняя задержка составила 35,54 мс, максимальная – 3489 мс. Распределение ответов показало 2798 ответов с кодом 200, 937 ответов с кодом 404, 14 ответов с кодом 503 и 1 ответ с кодом 400. Принципиально важно, что в журнале не зафиксированы внутренние ошибки уровня 500. Следовательно, под конкурентной нагрузкой сервер не переходил в аварийное состояние, а возвращал диагностируемые ответы, отражающие конкретный тип конфликта.

Ответы 404 не рассматривались как признак нарушения целостности. Их природа связана не с повреждением данных в базе, а с обращением к сущностям, которые к моменту повторного чтения или изменения уже были логически удалены другим потоком. Следовательно, в данном случае сервер возвращал ожидаемый результат конкурентного доступа к уже неактуальному объекту.

Ключевое значение для оценки защиты целостности имеют ответы 400 и 503. Единственный ответ 400 возник при повторной попытке создать уже существующую связь между сотрудником и проектом. Это показывает, что сервер не допускает дублирования логически одинаковых связей и прерывает конфликтную операцию до фиксации некорректного состояния.

Ответы 503 отражают более сложные конфликты конкурентной записи. Такие ошибки возникали в тех случаях, когда один поток успевал изменить или удалить сущность раньше, чем другой поток завершал обновление той же записи. В результате устаревшая операция не применялась к данным, а отклонялась сервером как конфликт конкурентного доступа. Практически это означает, что при одновременной работе нескольких потоков более позднее состояние не перезаписывалось устаревшим значением.

Полученные результаты подтверждают, что под нагрузкой сервер не переводит конфликтные ситуации в скрытое повреждение данных. Напротив, попытки повторного создания уже существующей связи и попытки записи по устаревшему состоянию завершаются явным отказом. Следовательно, стресс-тест подтвердил работоспособность примененных средств защиты целостности данных при конкурентной обработке запросов.

Проведенное исследование показало, что целостность данных в многопоточном TCP-сервере на Java обеспечивается наиболее надежно при совместной работе нескольких уровней защиты. Сочетание маршрутизации запроса, сервисной валидации, логики soft delete, транзакционного выполнения операций, optimistic locking и ограничений базы данных снижает риск конфликтных изменений и повышает устойчивость серверной обработки. Результаты теста подтверждают, что при конкурентной нагрузке сервер переводит конфликтные ситуации в контролируемые ответы приложения, а не в скрытое повреждение данных. Это позволяет сделать вывод об эффективности предложенного подхода для защиты согласованности данных и предотвращения потери обновлений.

Список использованных источников:

1. Keith, M. Pro JPA 2 in Java EE 8 : an in-depth guide to Java Persistence APIs / M. Keith, M. Schincariol. – Berkeley : Apress, 2018. – 535 p.

2. Bauer, C. Java Persistence with Hibernate : a definitive guide / C. Bauer, G. King, G. Gregory. – Shelter Island : Manning Publications, 2015. – 608 p.

3. Fowler, M. Patterns of Enterprise Application Architecture : definitive guide / M. Fowler. – Boston : Addison-Wesley, 2002. – 533 p.

4. Mihalcea, V. High-Performance Java Persistence : a comprehensive guide / V. Mihalcea. – Bukarest : Vlad Mihalcea, 2016. – 486 p.

5. Гонсалвес, Э. Изучаем Java EE 7 : пер. с англ. / Э. Гонсалвес. – СПб. : Питер, 2014. – 640 с.

6. Дейт, К.Дж. Введение в системы баз данных : пер. с англ. / К.Дж. Дейт. – М. : Вильямс, 2005. – 1328 с.

7. Моргунов, Е.П. PostgreSQL. Основы языка SQL : учеб. пособие / Е.П. Моргунов. – СПб. : БХВ-Петербург, 2018. – 336 с.