Процесс аутентификации

Аутентификация основана протоколе OAuth 2.0. Данный протокол предусматривает наличие четырёх ролей:

1. Владелец ресурса (Resource Owner) – сущность или пользователь получающий доступ к защищённым ресурсам.
2. Ресурсный сервер (Resource Server) – сервер, который предоставляет доступ к защищённым ресурсам. Данный сервер должен иметь возможность проверять токен выданный сервером авторизации и если токен корректный – отвечать на запрос клиентского приложения.
3. Клиентское приложение (англ. Client Applications) – приложение получающее данные от защищённого ресурсного сервера, данное приложение также запрашивает токен у сервера авторизации, который идентифицирует владельца ресурса.
4. Сервер авторизации (Authorization Server) – сервер, который управляет авторизацией выдавая токены доступа, позволяющие идентифицировать владельца ресурса при запросе данных с ресурсного сервера.



В данном приложении в качестве сервера авторизации выступают

Уровень доступа к данным.

Для большинства современны web-приложений основной функцией является предоставления доступа к данным. Интерфейс типичного приложения водится к предоставлению пользователю следующих типичных процедур для манипулирования данными: добавление (create), чтение (read), обновление (update), удаление (delete). Данный набор операций известен как CRUD-операции. Данная концепция используется не только серверами управления баз данных (СУБД), но и основным протоколом передачи данных HTTP. В таблице 1 представлены соответствия между CRUD-операциями, операторами языка SQL, типами HTTP запросов (HTTP-verbs).

Таблица 1.

|  |  |  |
| --- | --- | --- |
| Операция | Оператор языка SQL | Операция в протоколе HTTP |
| Создание (create) | INSERT | POST |
| Чтение (read) | SELECT | GET |
| Редактирование (update) | UPDATE | PUT или PATCH |
| Удаление (delete) | DELETE | DELETE |

Схожий подход используется также в нерялиционных базах (документо-ориентированные СУБД, графовых СУБД и т.д.), при этом в серверах баз данных использующихся в высоконагруженных системах широкое распространение получил подход к увеличению производительности и отказоустойчивости основанный на отказе от предоставления операции изменения. Для имитации данной операции используется комбинация трёх оставшихся команд: чтение, удаление, вставка. При в отличии от традиционной операции изменения такой подход изменяет первичный ключ (индекс в таблице) и временную метку. Такой подход используется наиболее распространённом NoSQL (not only SQL) решении в СУБД Cassandra. Данная СУБД используется во множестве высоко нагруженных систем, наиболее известной из которых является сервис Twitter, который отображает информацию опубликованную пользователем в хронологическом порядке. В силу вышеупомянутого подхода к оптимизации производительности можно с уверенностью ответить на вопрос, который беспокоит сотни миллионов пользователей Twitter-а: когда появиться функция редактирования сообщений? Ответ: она не появится до тех пор, пока Twitter не откажется от подхода оптимизации производительности, основанном на замене операции редактирования операциями чтения, удаления записи.

Абстракция доступа к данным. Текущая абстракция. Мост.

Трёхуровневая архитектура предполагает наличие независимого слоя доступа к данным. При этом его детали реализации должны быть скрыты от уровня сервисов. К деталям реализации относятся то, каким образом осуществляется подключение к СУБД (тип ORM фреймворка), какой язык запросов используется (SQL, Transact-SQL, CQL, Pig), какие промежуточные модели данных используются и как реализовано кеширование. Архитектура слоя данных должна выполнять ещё одну немаловажную задачу – сокрытие текущих абстракций (leaky abstraction).

Текущая абстракция – термин в разработке программного обеспечения, описывающий абстракцию, которая предоставляет подробную информацию о деталях внутренней реализации и её ограничениях, которые в идеале должны быть спрятаны. Наличие утечек в абстракциях является серьёзной проблемой, поскольку цель абстракций заключается в управлении сложностью, скрывая ненужные детали от пользователя. Иногда для определения этого термина используют понятия дырявая абстракция, излучающая абстракция.

Закон текущих абстракций гласит: все нетривиальные абстракции являются текущими. Из этого следует, что в попытках построить большую систему с хорошими абстракциями, мы всё равно не сможем скрыть все детали реализации вытекающие из абстракции. Лучшее, что мы можем сделать в данной ситуации это локализовать место, в котором нам приходится вдаваться в детали реализации, которые не скрыты абстракцией.

Язык SQL был создан, чтобы абстрагироваться от процедурных шагов, нужных для запросов к базе данных. Вместо этого он позволяет описать, что именно запрашивается, и пусть база данных сама догадается, какие процедурные шаги для этого нужны. Но в иных случаях некоторые запросы SQL на порядки медленнее, чем другие, логически им эквивалентные. Известный пример: некоторые сервера SQL значительно быстрее отрабатывают запрос where a=b and b=c and a=c, чем where a=b and b=c, хотя результат, конечно, тот же самый. Программисту на SQL вроде бы и не следует заботиться о процедуре, только о спецификациях. Но иногда абстракция протекает, что приводит к значительным, а иногда и критическим потерям в производительности, так что приходится лезть во внутренности планировщика запросов и смотреть, что там не так, и как заставить его работать эффективнее. Эта проблема не решается и ORM фреймворками, поэтому задачей слоя данных является инкапсуляция это и ей подобных утечек в абстракции.

Для упрощения реализации слоя доступа данных удобно использовать ORM фреймворки. Любой ORM фреймворк спроектирован так, чтобы быть наиболее универсальным. Наибольшей проблемой при интеграции стороннего решения в разрабатываемую систему является различие их интерфейсов. Некоторые фреймворки оперируют такими понятиями как таблицы, некоторые – множествами. Очень часто одинаковые по смыслу операции имеют разные названия: get и load, select и map, delete и remove. В общем случае данная проблема может быть описана следующим образом: необходимо преобразовать интерфейс класса таким образом, чтобы он был совместим с интерфейсом, который ожидает класс-клиент.

Для решения данной задачи подходит паттерн проектирования «Адаптер», который структурирует классы и объекты, обеспечивая совместимость интерфейсов. Существуют две возможные реализации паттерна «Адаптер». Первая реализация основана на использовании множественного наследования (рис. 1) и представляет собой адаптер класса.



Рисунок 1.

Вторая реализация применяет композицию объектов и является адаптером объектов.

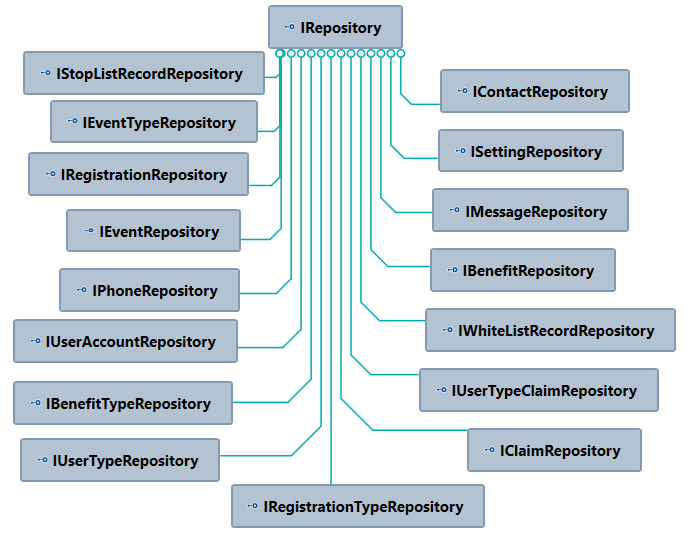


Рисунок 2.

Участники:

1. Target – определяет зависящий от предметной области интерфейс, который использует Client;
2. Client – вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
3. Adaptee – определяет существующий интерфейс, который нуждается в адаптации;
4. Adapter – адаптирует интерфейс Adaptee к интерфейсу Target.

Основной задачей адаптеров слоя доступа к данным является сокрытия ORM-фреймворка. С точки зрения пользователя наиболее является такой объект доступа к данным, чей интерфейс будет подобен коллекции. Данным критериям соответвует такая разновиднасть паттерна адаптер как «Репозиторий». Интерфейс репозитория включает в себя следующие методы: Get, Insert, Delete, Update. Для каждой таблицы в базе данных создан свой интерфейс репозитория, наследующий методы базового интерфейса и при необходимости расширяет допустимый набор методов. В терминологии паттерна «Адаптер» интерфейс IRepository соответствует Target, ORM-фреймворк – Adaptee, класс, реализующий интерфейс IRepository – Adapter. Иерархия репозиториев изображена на рисунке.



Наиболее сложным с точки зрения реализации является метод Get. Существует ряд проблем:

1. Как реализовать получения одного или нескольких объектов.
2. Как передавать критерий для выборки объектов.
3. Как каком уровне формировать запрос на выборку объектов.
4. Как избежать дублирования кода, если одинаковый запрос нужен в разных местах не делая из интерфейса перечисления списка всевозможных выборок.
5. Как организовать выборку только нужных полей из таблицы.

Решением первой проблемы является введение двух методов Get и GetList. Предполагать, что список из одного элемента это тоже список является нецелесообразным, так как ожидается наличие большого числа запросов, ответом на которые будет строго один элемент и каждый раз выбирать первый элемент из списка нецелесообразно.

Получать объекты из базы можно несколькими путями: использовать хранимые процедуры, передавать SQL код серверу напрямую, использовать LINQ-выражения. Первый подход нарушает инкапсуляцию и порождает размытие бизнес-логики. Задача сервера баз данных – хранить данные и предоставлять к ним доступ, но никак не реализовывать бизнес-логику. Второй подход крайне неэффективный, так как придётся передавать простые строки. Это значит, что не будет ни подсветки синтаксиса, ни подсказок при написании кода, ни ошибок компиляции при неправильно написанном запросе. LINQ – гибкий механизм без недостатков первых двух подходов: бизнес-логика не уходит на сервер баз данных, есть подсветка кода, ошибки синтаксиса будут отловлены при компиляции. Недостатком являются затраты процессорного времени на трансляцию LINQ-выражения в SQL код, однако данный недостаток компенсируется кешированием.

Строить запросы необходимо на уровне слоя доступа к серверу баз данных, это его основная функциональность. При этом должна быть возможность написать такой запрос на уровне сервисов (для избежания перекомпиляции уровня доступа к данным, при написании тестового когда и кода имитирующего работу не реализованного сервиса или прототипа).

Решение четвёртой проблемы осложняется тем, что при решение первой проблемы количество точек входа в систему было ограничено двумя, это значит, что все запросы можно разделить не более чем на две группы с различными интерфейсами. При этом запросы первой группы должны возвращать объект не являющийся коллекцией элементов, а запросы второй группы возвращают коллекцию элементов, при этом стоит учесть, что объект со значение null это всё равно корректный объект, пустая коллекция и коллекция с одним элементом – также корректные результаты выполнения запроса, а вот получение от сервера баз данных коллекции вместо одного объекта является ошибкой целостности данных и ведёт к возникновению исключительной ситуации во время выполнения программы.

В общем случае необходимо иметь механизм позволяющий определить семейство взаимозаменяемых алгоритмов с разным поведением. Исходя из данного определения составленного на основе требований к архитектуре можно применить паттерн проектирования «Стратегия».



Рис.3

Участники:

1. Strategy – стратегия, общий для всех поддерживаемых алгоритмов интерфейс. Экземпляр класса Context использует этот интерфейс для вызова конкретного метода определённого в классе ConcreteStratege.
2. ConcreteStrategy – конкретная стратегия, реализующая алгоритм и предоставляющая к нему доступ через интерфейс объявленный в классе Strategy.
3. Context – контекст, использует алгоритм, реализованный в классе ConcreteStrategy через интерфейс Strategy.

Архитектура клиентского приложения. Визуализация данных.

Архитектура клиентского приложения должна быть построена таким образом, чтобы обеспечить наиболее удобный подход к написанию бизнес-логики и при этом уменьшить количество технического кода не связанного с предметной областью, к такому коду относится функционал по манипуляции с DOM-деревом, отслеживание состояния приложение, низкоуровневый контроль за выполнением асинхронных запросов.

У любого приложения есть некоторое состояние, от которого зависит набор и вид отображаемых компонентов. Состояние может быть двух типов: изменяемое и неизменяемое. С первым типом всё понятно: состояние компонента изменяется и в ответ на это изменяется DOM-дерево. Второй же подход к работе с состоянием является неочевидным. Возникает вопрос: как изменяется вид компонентов если состояние неизменно? Вид компонента изменяется из-за того, что состояние становится новым, то есть изменяется не значение, которое хранится по ссылке, а сама ссылка. Этот подход более эффективен, так как проще отслеживать изменения. Если мы имеем дело с изменением значений переменных, то чтобы отследить изменения необходимо иметь в памяти копию всего состояния и сравнивать каждое значение. Если отслеживать только изменение ссылок, то в памяти достаточно хранить только набор ссылок в древовидном виде, что уменьшает количество объектов для сравнения и делает сложность поиска изменений логарифмической, поэтому в процессе разработки будет использован второй подход.

Центром всего приложения является пользовательский интерфейс (View), с которым взаимодействует пользователь. Пользовательский интерфейс формируется на основе данных, размещённых в контейнере состояния. Взаимодействие пользователя с интерфейсом приводит в действие Dispatcher, который формирует объект Action, содержащий информацию о действии пользователя. Обработчик событий (Reducer), генерирует новое состояние либо возвращает старое неизменённое состояние в зависимости от алгоритма обработки события и предыдущего состояния. Новое состояние попадает в хранилище, тем самым завершает циклическое движение данных.



Для того чтобы скрыть детали реализации слоя доступа к данным необходимо продумать архитектуру удовлетворяющую следующим критериям:

1. Отсутствует привязка абстракции к реализации, чтобы иметь возможность выбрать необходимую реализацию в момент выполнения программы.
2. И абстракции, и реализации должны расширяться новыми подклассами, при этом необходимо иметь возможность применять их независимо.
3. Изменения в реализации не должны абстракции не должны сказываться на клиенте (слое сервисов), то есть клиентский код не должен ни изменяться, ни перекомпилироваться.

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование, Абстрактный класс определяет интерфейс абстракции, а его конкретный подклассы по-разному реализуют его, но такой подход не всегда обладает достаточной гибкостью (в некоторых языках программирования существуют такие сущности как интерфейсы, в которых нет реализаций, но есть описание полей и методов). Наследование жёстко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение повторное использование абстракции и её реализации.

Этим критерия соответствует паттерн проектирования Мост (bridge), также известный как описатель.

