УДК 004.05

ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ АРХИТЕКТУРНОГО СТИЛЯ ВЗАИМОДЕЙСТВИЯ REPRESENTATIONAL STATE TRANSFER

Д.С. ТАТАРИН
(Представлено: канд. техн. наук, доц. И.Б. БУРАЧЁНОК)

Рассмотрены особенности архитектурного стиля взаимодействия компонентов распределённого приложения в сети – REST и примеры его использования. Показаны основные преимущества при использовании набора ограничений, учитываемых при проектировании (разработке) распределённой системы с использованием компонентов REST.

Representational State Transfer или же как его часто сокращают REST – это архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы [1]. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) применение распределённой системы с использованием компонентов REST приводит к повышению производительности и упрощению архитектуры. В широком смысле компоненты в REST взаимодействуют наподобие взаимодействия клиентов и серверов во Всемирной паутине. REST является альтернативой RPC (Remote Procedure Call – вызов удалённых процедур).

Так же очень часто на ряду с аббревиатурой REST используется понятие RESTful. Им называют веб-сервис, который следует всем шести ограничениям, накладываемым на взаимодействие между сервером и клиентом. Ниже рассмотрим их более подробно.

Клиент-серверная архитектура. Принцип, лежащий в основе ограничений клиент-сервер, заключается в разделении ответственности [1]. Отделение проблем пользовательского интерфейса от проблем хранения данных улучшает переносимость пользовательских интерфейсов на нескольких платформах. Это также улучшает масштабируемость за счет упрощения серверных компонентов. Возможно, наиболее важным для Интернета является то, что разделение позволяет компонентам развиваться независимо, таким образом поддерживая требования в масштабе Интернета для нескольких доменов организации. Пример клиент-серверной архитектуры можно увидеть на рисунке 1.

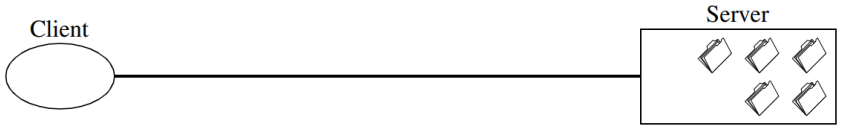

Рисунок 1. – Клиент-серверная архитектура

Отсутствие состояния. Связь клиент-сервер ограничена отсутствием клиентского контекста, хранящегося на сервере между запросами [1]. Каждый запрос от любого клиента содержит всю информацию, необходимую для обслуживания запроса, а состояние сеанса сохраняется в клиенте. Состояние сеанса может быть передано сервером другой службе, такой как база данных, для поддержания постоянного состояния в течение определенного периода времени и обеспечения аутентификации. Клиент начинает отправлять запросы, когда он готов к переходу в новое состояние. Пока один или несколько запросов не выполнены, считается, что клиент находится в процессе перехода. Представление каждого состояния приложения содержит ссылки, которые можно использовать в следующий раз, когда клиент решит инициировать новый переход между состояниями. Отсутствие состояние представлено на рисунке 2.

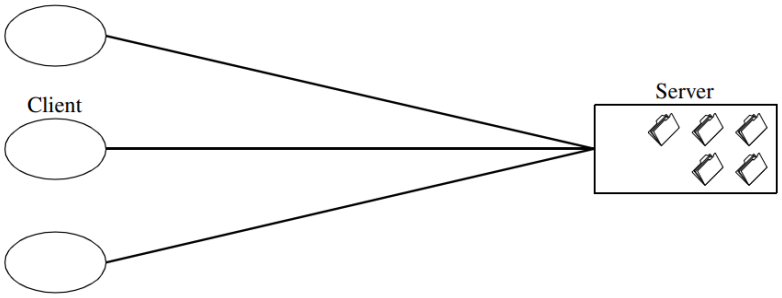

Рисунок 2. – Отсутствие состояния

Кешируемость. Как и во всемирной паутине, клиенты и посредники могут кэшировать ответы [1]. Ответы должны, неявно или явно, определять себя как кэшируемые или некэшируемые, чтобы клиенты не предоставляли устаревшие или несоответствующие данные в ответ на дальнейшие запросы. Хорошо управляемое кэширование частично или полностью исключает некоторые взаимодействия клиент-сервер, дополнительно улучшая масштабируемость и производительность. Пример кэшируемости можем представлен на рисунке 3.

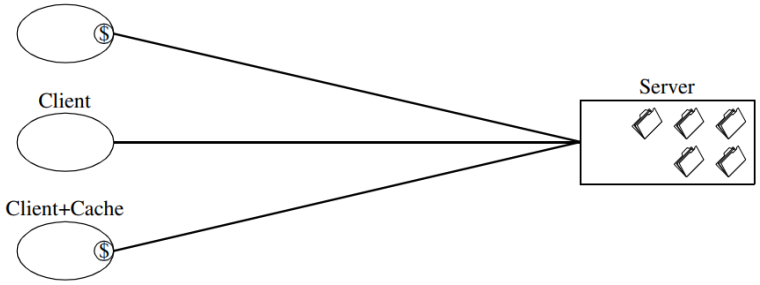

Рисунок 3. – Кешируемость

Многоуровневая система. Обычно клиент не может определить, подключен ли он напрямую к конечному серверу или к посреднику по пути [1]. Если между клиентом и сервером размещен прокси-сервер или балансировщик нагрузки, это не повлияет на их взаимодействие, и не потребуется обновлять код клиента или сервера. Промежуточные серверы могут улучшить масштабируемость системы за счет включения балансировки нагрузки и предоставления общих кешей. Кроме того, безопасность может быть добавлена как слой поверх веб-служб, а затем четко отделить бизнес-логику от логики безопасности. Добавление безопасности в качестве отдельного уровня обеспечивает соблюдение политик безопасности. Наконец, это также означает, что сервер может вызывать несколько других серверов для генерации ответа клиенту. Пример многоуровневой системы можем представлен на рисунке 4.

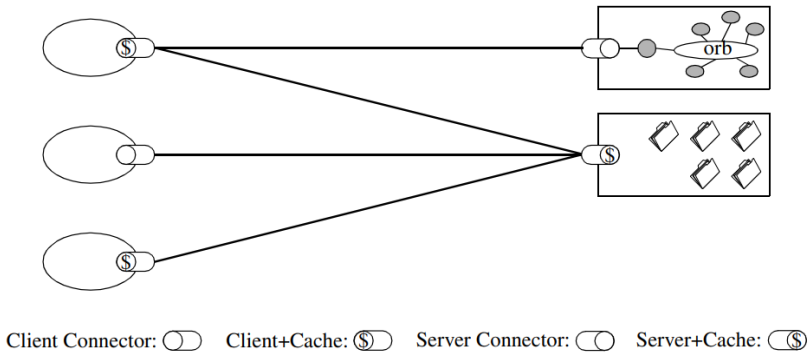

Рисунок 4. – Многоуровневая система

Код по запросу, это правило является не обязательным. Серверы могут временно расширять или настраивать функциональность клиента, передавая исполняемый код: например, скомпилированные компоненты, такие как апплеты Java, или сценарии на стороне клиента, такие как JavaScript.

Uniform interface. Ограничение унифицированного интерфейса является фундаментальным для проектирования любой системы RESTful [1]. Оно упрощает и разделяет архитектуру, что позволяет каждой части развиваться независимо. Далее рассмотрим четыре ограничения для этого унифицированного интерфейса.

– Идентификация ресурса в запросах. Отдельные ресурсы идентифицируются в запросах, например, с помощью URI (Uniform Resource Identifier) – унифицированный идентификатор ресурса в веб-службах RESTful. Сами ресурсы концептуально отделены от представлений, возвращаемых клиенту. Например, сервер может отправлять данные из своей базы данных в формате HTML, XML или JSON, ни один из которых не является внутренним представлением сервера.

– Управление ресурсами посредством представлений. Когда клиент содержит представление ресурса, включая любые прикрепленные метаданные, у него достаточно информации для изменения или удаления состояния ресурса.

– Самодостаточные сообщения. Каждое сообщение содержит достаточно информации, чтобы описать, как его обрабатывать. Например, анализатор для вызова может быть определен типом носителя.

– Гипермедиа как двигатель состояния приложения (HATEOAS). Получив доступ к начальному URI для приложения REST – аналогично доступу обычного пользователя Web к домашней странице вебсайта – клиент REST должен иметь возможность динамически использовать предоставляемые сервером ссылки для обнаружения всех необходимых ему доступных ресурсов. По мере доступа сервер отвечает текстом, который включает гиперссылки на другие доступные в настоящее время ресурсы. Клиенту не нужно жестко запрограммировать информацию о структуре или динамике приложения [2].

Таким образом, придерживаясь архитектурного принципа REST получаем легко масштабируемые приложения, простую архитектуру и снижаем нагрузку на вычислительные ресурсы. Принцип REST предлагается использовать при реализации проекта, а именно при построении бэкенда с использованием RESTful веб-сервиса для удобного взаимодействия с различными фронтэндными приложениями.

ЛИТЕРАТУРА

1. Architectural Styles and the Design. Диссертация Roy Thomas Fielding. [Электронный ресурс] / ics.uci.edu. – Режим доступа: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf – Дата доступа: 20.09.2020.
2. RESTful. Статья про рестфулл приложения. [Электронный ресурс] / habr.com. – Режим доступа: https://habr.com/ru/post/319984/ – Дата доступа: 20.09.2020.