Клиент-серверная архитектура

JSP & Servlets
14 уровень, 0 лекция

ОТКРЫТА

1.1 Архитектура приложения

Этот курс рассчитан на новичков, потому что проектировать архитектуру серьезного приложения ты не будешь еще долго. Но не надо расстраиваться, хорошая архитектура – это скорее исключение, чем правило. Очень сложно выбрать правильную архитектуру приложения **до** создания приложения.

Примеры популярных архитектур больших серверных приложений:

- Многослойная архитектура (Layered Architecture).
- Многоуровневая архитектура (Tiered Architecture).
- Сервис-ориентированная архитектура (Service Oriented Architecture – SOA).
- Микросервисная архитектура (Microservice Architecture).

Каждая из них имеет свои плюсы и свои минусы. Но изучение их тебе ничего не даст. Архитектура – это ответ на вопрос **«как организовать взаимодействие тысяч объектов внутри системы»**. И пока ты на своем опыте не почувствуешь всю сложность проблемы, ты не сможешь понять и всю многогранность решения.

Все приложения используют какую-нибудь архитектуру или или хотя бы делают вид. Поэтому знание популярных подходов к проектированию приложений позволит тебе быстрее и лучше понимать, как приложение устроено. А значит вносить изменения именно в те места, куда нужно.

А что значит «вносить изменения куда нужно?». Есть места, куда не нужно вносить изменения? Именно.

Чтобы добавить конкретики, давай предположим, что ты работаешь над **средним backend-проектом**. Он пишется вот уже 5 лет командой из 20 человек. На проект потрачено 100 человеко-лет, в нем около 100 тысяч строк кода. Суммарно он состоит из двух тысяч классов, которые разбиты на 10 модулей разного размера.

И добавим суровой реальности. Логика некоторых задач размазана по нескольким модулям. Также бизнес-логика может быть во фронтенде (написанном на JavaScript) и/или записана в виде stored procedure прямо в базе данных. Ты все еще уверен, что сразу сможешь определить место, **куда именно вносить изменения**?

Это не какой-то кошмар, который я придумал, чтобы вас напугать. Это типичнейший проект. Бывает еще и похуже. Почему же так происходит? Причин может быть сколько угодно, но почти всегда присутствуют такие:

- На проекте работает куча людей – каждый из них видит его немного по-своему.
- За 5 лет в проекте сменилось 10 человек, новички не стали сильно в нем разбираться.
- Создание софта – это постоянное внесение изменений, которые постоянно все меняют.
- Пять лет назад, когда определялись с архитектурой, задумка проекта была несколько иной.

Но главное, что независимо от архитектуры проекта, все работающие над ним программисты, придерживались одного и того же понимания, как этот проект устроен. Начнем с самого простого понятия – клиент-серверной архитектуры.

1.2 Концепция взаимодействия клиент-сервер

Сейчас мы разберемся с концепцией, которая лежит в основе архитектуры **клиент-сервер** и позволит вам лучше понять как организовано взаимодействие миллионов программ в сети интернет.

Как понятно из названия, в данной концепции участвуют две стороны: **клиент** и **сервер**. Здесь все как в жизни: клиент – это заказчик той или иной услуги, а сервер – поставщик услуг. Клиент и сервер физически представляют собой **программы**, например, типичным клиентом является браузер.

В качестве сервера можно привести следующие примеры:

- Веб-сервера, например Tomcat.
- Сервера баз данных, например, MySQL.
- Платежные шлюзы, например Stripe.

Клиент с сервером обычно общаются через интернет (хотя могут работать и в одной локальной сети и вообще в любых других типах сетей). Общение происходит по стандартным протоколам, таким как HTTP, FTP или более низкоуровневым, таким как TCP или UDP.

Протокол обычно выбирается под тип услуги, которую оказывают сервера. Например, если это видеосвязь, то обычно используется UDP.

Помните, как работает Tomcat и его сервлеты? Сервер получает HTTP-сообщение, распаковывает его, достает оттуда всю нужную информацию и передает сервлету на обработку. Затем результат обработки упаковывает обратно в HTTP-response и отсылает клиенту.

Это и есть типичное взаимодействие клиент-сервер. Браузер – это веб-клиент, а Tomcat – это веб-сервер. Tomcat даже так и называется – веб-сервер.

Но если подумать, то важно не название, а суть – распределение ролей между программами. Твой JS-скрипт, работающий в HTML-странице, вполне можно назвать **клиентом**, а твой сервлет – **сервером**. Ведь они работают в паре в рамках **концепции клиент-сервер**.

1.3 Важный нюанс

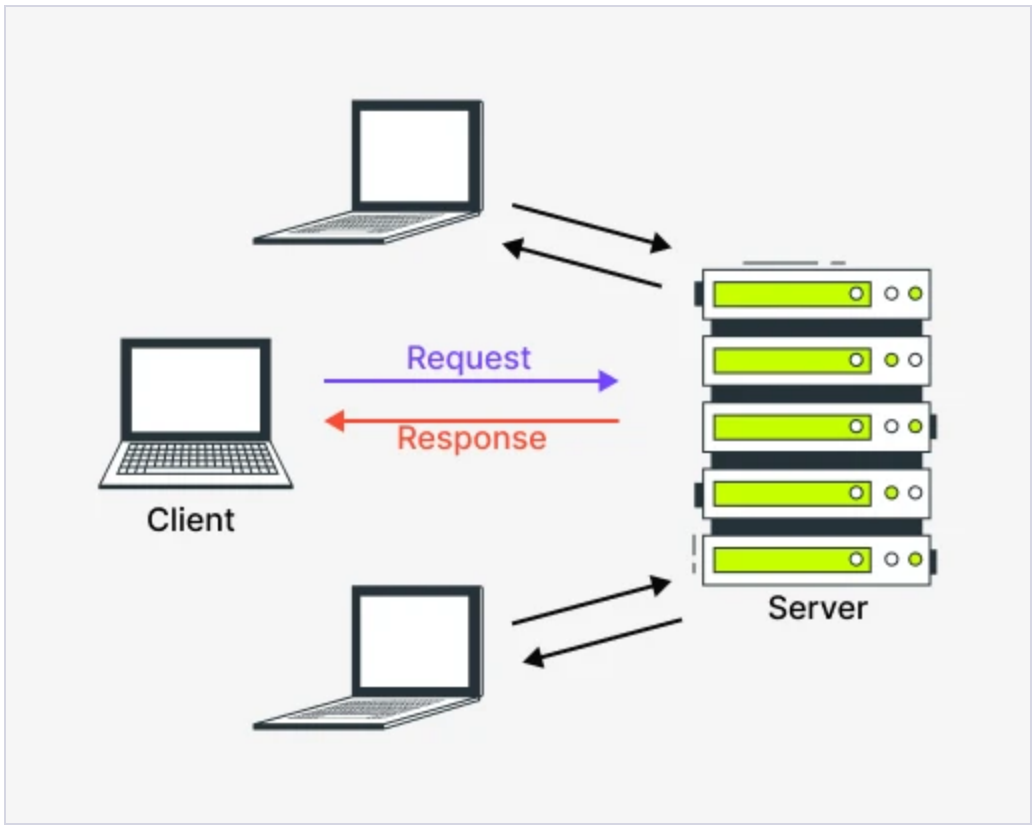
Еще стоит отметить, что в **основе взаимодействия клиент-сервер лежит принцип того, что такое взаимодействие начинает клиент**: сервер лишь отвечает клиенту и сообщает о том может ли он предоставить услугу клиенту и если может, то на каких условиях.

Не имеет значения, где физически находится клиент и где сервер. Клиентское программное обеспечение и серверное программное обеспечение обычно установлено на разных машинах, но также они могут работать и на одном компьютере.

Данную концепцию разработали как первый шаг в сторону упрощения сложной системы. У нее есть такие сильные стороны:

- **Упрощение логики**: сервер ничего не знает о клиенте и как он будет использовать его данные в дальнейшем.
- Могут быть **слабые клиенты**: все ресурсоемкие задачи можно перенести на сервер.
- Независимое развитие кода клиентов и кода сервера.
- Множество различных клиентов, пример – Tomcat и разные браузеры.

Самый базовый вариант взаимодействия клиента и сервера представлен на картинке:



Тут важно отметить две детали. Во-первых, из картинки видно, что к одному серверу могут обращаться множество клиентов. Во-вторых, они могут к нему обращаться одновременно. Это тоже важная часть работы сервера.

Один клиент обычно взаимодействует с одним пользователем, поэтому там часто даже авторизация не нужна. Однако сервер обрабатывает запросы тысяч клиентов и разрабатывая для него код нужно уметь отличать авторизацию от аутентификации.

Еще важно, что сервер обрабатывает тысячи запросов параллельно. А это значит, что, разрабатывая код бэкенда, тебе постоянно нужно будет думать над задачей одновременного доступа к ресурсам. Также у кода сервера очень высокая вероятность race condition (гонки потоков), deadlock (взаимная блокировка потоков).

Жизненный цикл важных объектов нужно обязательно контролировать:

Ты не можешь просто так запустить на сервере новый поток через `new Thread().start()`. Вместо этого тебе нужно иметь ThreadPool, который будет шариться между всеми сервисными потоками.

Также нельзя просто так запустить асинхронную задачу, они ведь тоже выполняются в отдельных потоках. Создавая такую задачу, ты всегда должен знать, какой пул-потоков ее выполняет и что произойдет, если такой пул переполнится.

Все работы с файлами и директориями нужно делать через try-with-resources. Если в обычном приложении ты забыл закрыть поток или файл – разве это проблема? Закроется сам при выходе из программы. А вот если ты где-то в коде забыл закрыть файл на сервере, и ваш сервер работает месяцами… Скоро таких незакрытых файлов накопятся тысячи и ОС перестанет открывать новые файлы на чтение (работу с файлами контролирует ОС). Тимлид вас по головке не погладит…

1.4 Архитектура «клиент-сервер»

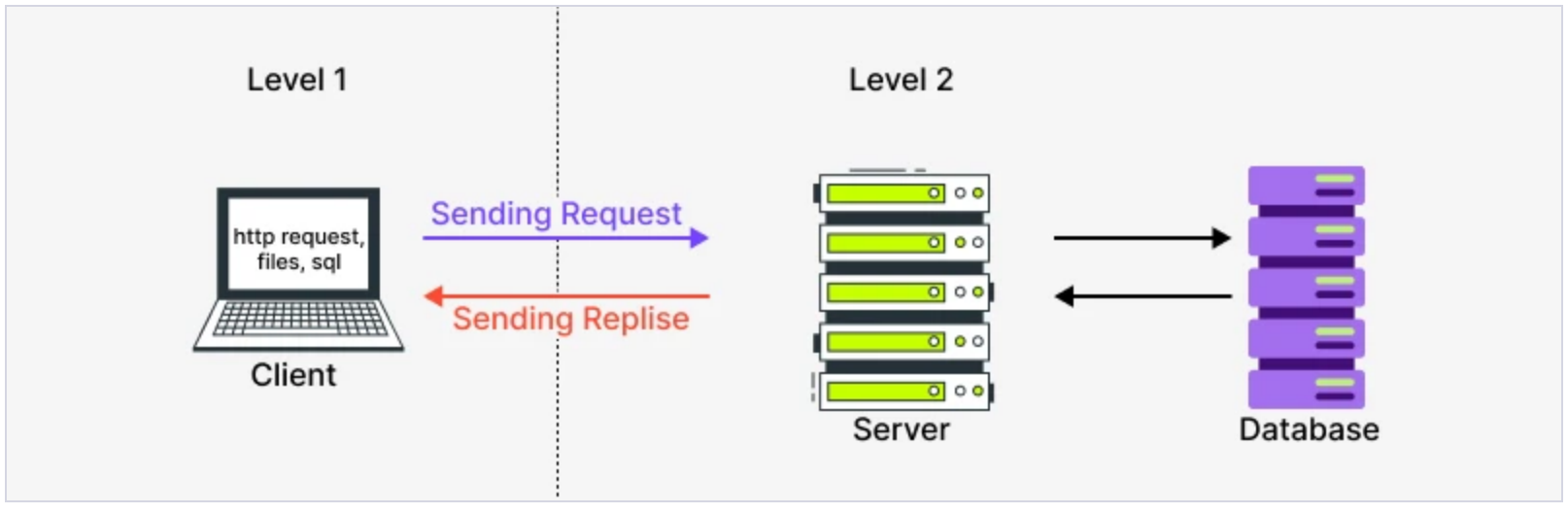
еще один важный момент. **Архитектура клиент-сервер определяет лишь общие принципы взаимодействия между компьютерами**, детали взаимодействия определяют различные протоколы.

Данная концепция (клиент-сервер) говорит нам, что нужно разделять машины в сети на клиентские, которым всегда что-то надо и на серверные, которые дают то, что надо. При этом взаимодействие всегда начинает клиент, а правила, по которым происходит взаимодействие описывает протокол.

Существует два вида архитектуры взаимодействия клиент-сервер: первый получил название **двухуровневая архитектура** клиент-серверного взаимодействия, второй – многоуровневая архитектура клиент-сервер (иногда его называют **трехуровневая архитектура** или трехзвенная архитектура, но это частный случай).

Принцип работы двухуровневой архитектуры взаимодействия клиент-сервер заключается в том, что обработка запроса происходит на одном сервере без обращения к другим серверам в процессе этой обработки.

Двухуровневую модель взаимодействия клиент-сервер можно нарисовать в виде простой схемы.



Тут видно, что первый уровень – это все, что касается клиента, а второй уровень – это все, что касается сервера.

‹ Предыдущая лекция

Следующая лекция ›