Лекция #9. Сервлеты.

Краткий экскурс в Java EE.

Уютный мир Java SE не совпадает с суровой реальностью. Java-платформ много. Java SE = язык как спецификация + JVM как реализация спецификации. Идеология построения платформы Java базируется на спецификациях. Java 11 == использование какой-то JVM, которая реализует спецификакции, описаннные в стеке Java 11. JVM Hotspot, например, каноническая реализация спецификации.

Java SE = JVM + Java Core. Над ней настраиваются конструкции (почти приложения), на базе которых строятся большие enterprise приложения, например. Пример. На базе Java SE нужно построить какое-то приложение, называемое сервером приложения То есть Java SE здесь выступает средой, которая обеспечивает runtime для enterprise приложений.В нашем случае, для сервера приложения.

Java SE работает на уровне ОС. Java EE реализуется штуками, которые, как уже писал выше, называются серверами приложений (например, Apache Tomcat, Sun / Oracle GlassFish, BEA / Oracle WebLogic, IBM WebSphere, RedHat JBoss) и работает на базе Java SE. Runtime Java EE == Java SE.

По сути, Java EE - это раздутая лаба с первого курса, которая позволяет запускать внутри себя очень сложные сценарии. Java EE приложение развертывается на Application Server, а тот развернут на JVM, которая развернута на ОС. Java API там остается доступно. Расширение функционала такое своеобразное.

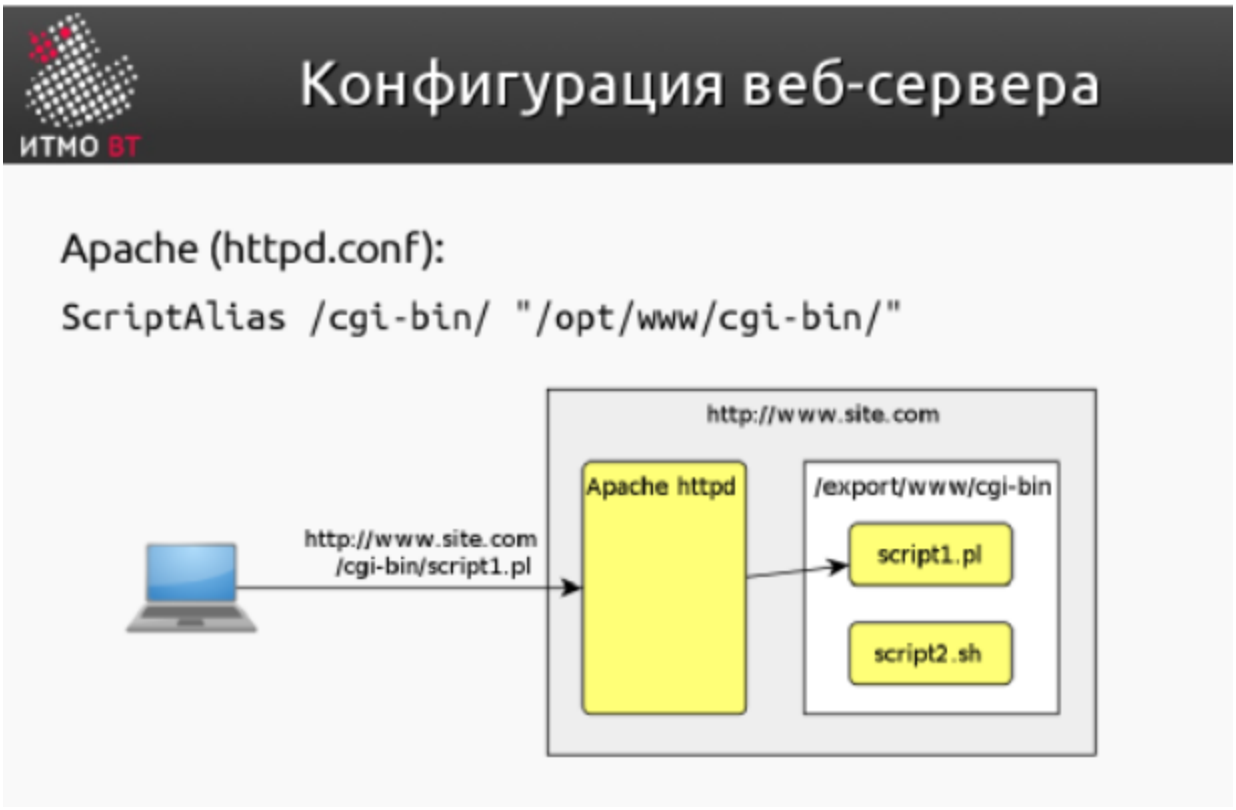
Лирические отступлени Еще есть Java ME - micro edition. Мертвая спецификация буквально, потому что раньше использовалась под написание игрушек на телефоны nokia. Еще есть вполне живая Java Embedded - для запуска непосредственно на железе: смарт-карты и пропуска например. И в этих картах часто живет Java Embedded - отдельная, слегка особенная, JVM и урезанный Java Core.

Java EE 8 ≠ Java SE 8, они развиваются независимо. Java EE более долгоподдерживаемая. Она развивается медленне SE из-за этого. И идеологии у них о runtime принципиально разные. Но об этом позже.

Java Servlets. Маленький кусочек Java EE.

Это серверные сценарии, предназначенные для обработки HTTP запросов. Исходя из хода повествования, логично, что они написаны на Java. По своему назначению это такой аналог PHP (точнее, аналог CGI и FastCGI*) из первой лабы по вебу.

Есть веб-контейнер. Остановимся на термине подробнее. Это кусок application server-a. То есть, обощив и забежав вперед: “сервер” приложения - это набор контейнеров для разных компонентов. Веб-контейнер - это контейнер, кусочек application server-a, который реализует функциональность, обеспечивает runtime для компонентов веба.



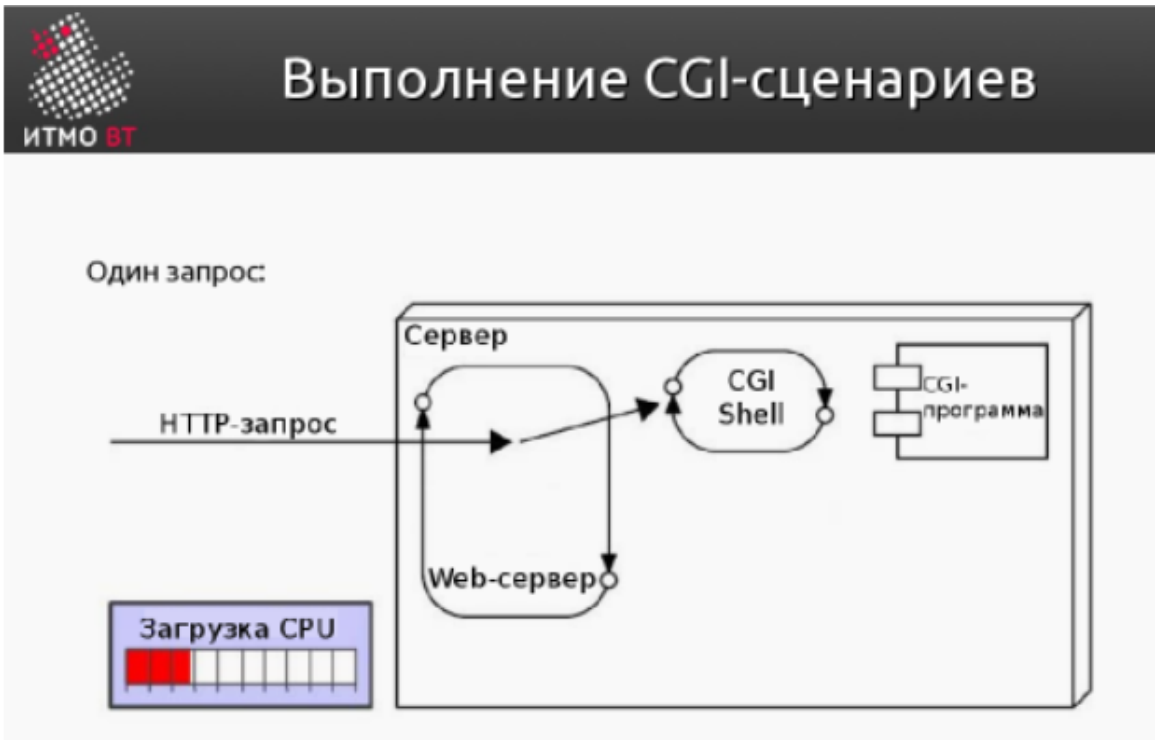
Существуют отдельные веб-контейнеры . Такие, какие реализуют только то что нужно. Это полезно, чтобы не тащить за собой весь большой и страшный application сервер, тогда, когда он не нужен целиком. Веб-контейнер == контейнер сервлетов. Здесь, в отличие от CGI и FastCGI* запросы обрабатываются в отдельных потоках гарантированно. Логика обработки запросов реализована сама. Это заложено в логике веб-контейнера: каждый запрос обрабатывать в отдельном

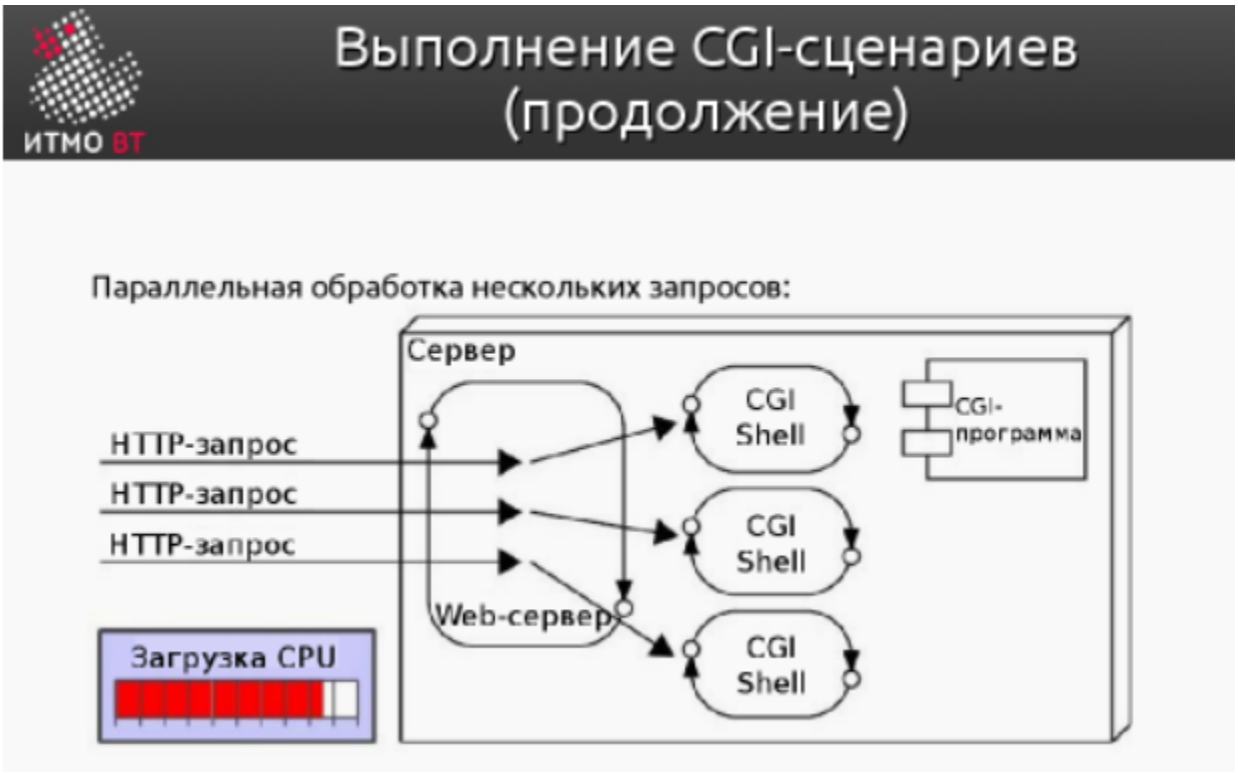
потоке. Application server включает в себя web server. В первой лабе есть настроенный веб-сервер и интегрированный с ним PHP интерпретатор. Во второй лабе ставится application server и он в себя включает web-server и контейнер, обеспечивающий рантайм для сервлетов. В силу вложенности, городить механизмы интеграции типа CGI не нужно, все и так нормально работает при помощи внутреннего API java.

Резюме. Application Server - java программа, реализующая спецификацию Java EE. А Java EE описывает набор каких-то (информация будет в будущих лекциях) компонентов, и вот сервлеты - один из таких компонентов. Под каждый из компонентов application server дает контейнер - runtime для компонента. И веб-сервер включен, чтобы слушать http-запросы. Вместе с application server-ом часто поднимается какой-то независимый веб-сервер типа nginx. Веб-сервер внутри application server медленный, потому что написан на java. И какую-то логику, например, защиту от DDOS полезно выносить на отдельные веб-сервера, которые побыстрее. До веб-сервера на джаве лучше когда будут доходить только запросы именно о бизнес-логике.

* Отступление про CGI и FastCGI

CGI. Веб-сервер получил запрос на специальный (для редиректа на shell программы и предназначенный) URI, вызвал в shell программу, та отработала и умерла. В CGI запросы обратываются в новых процессах по определению (типа каждый в своем).



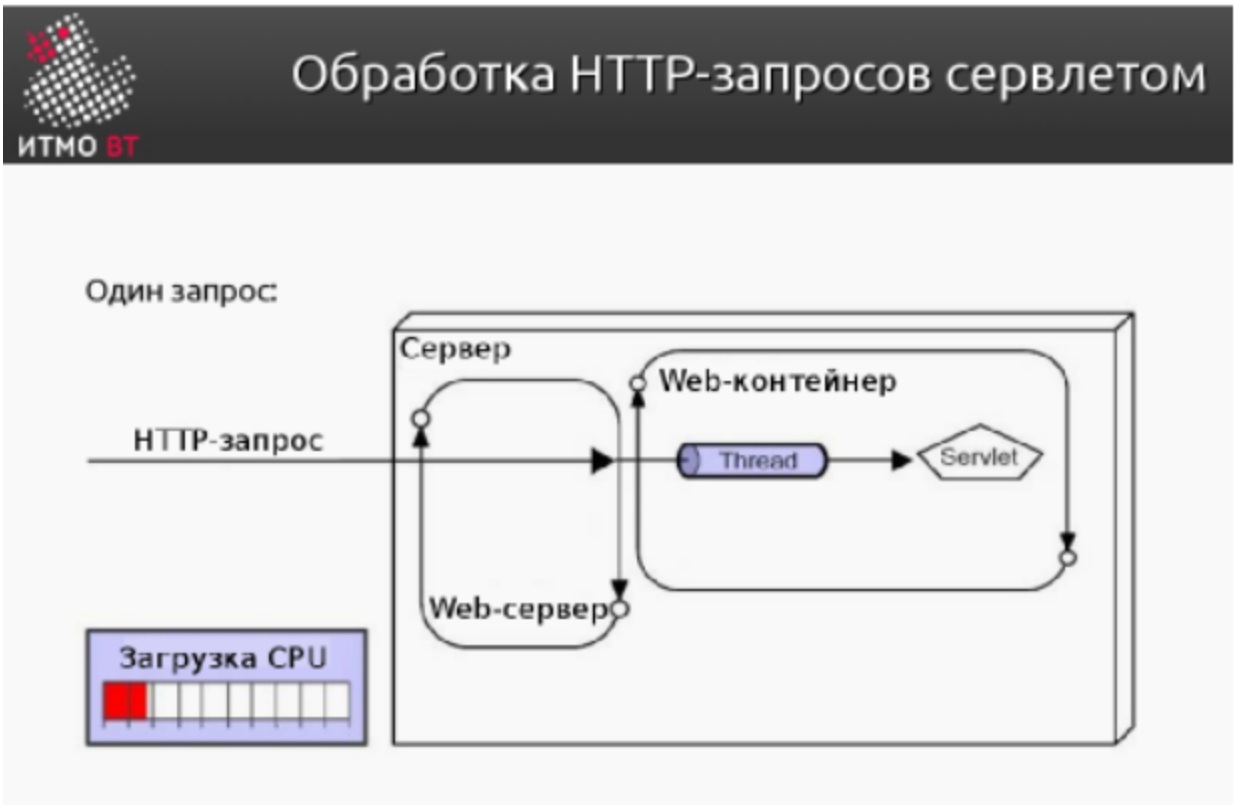


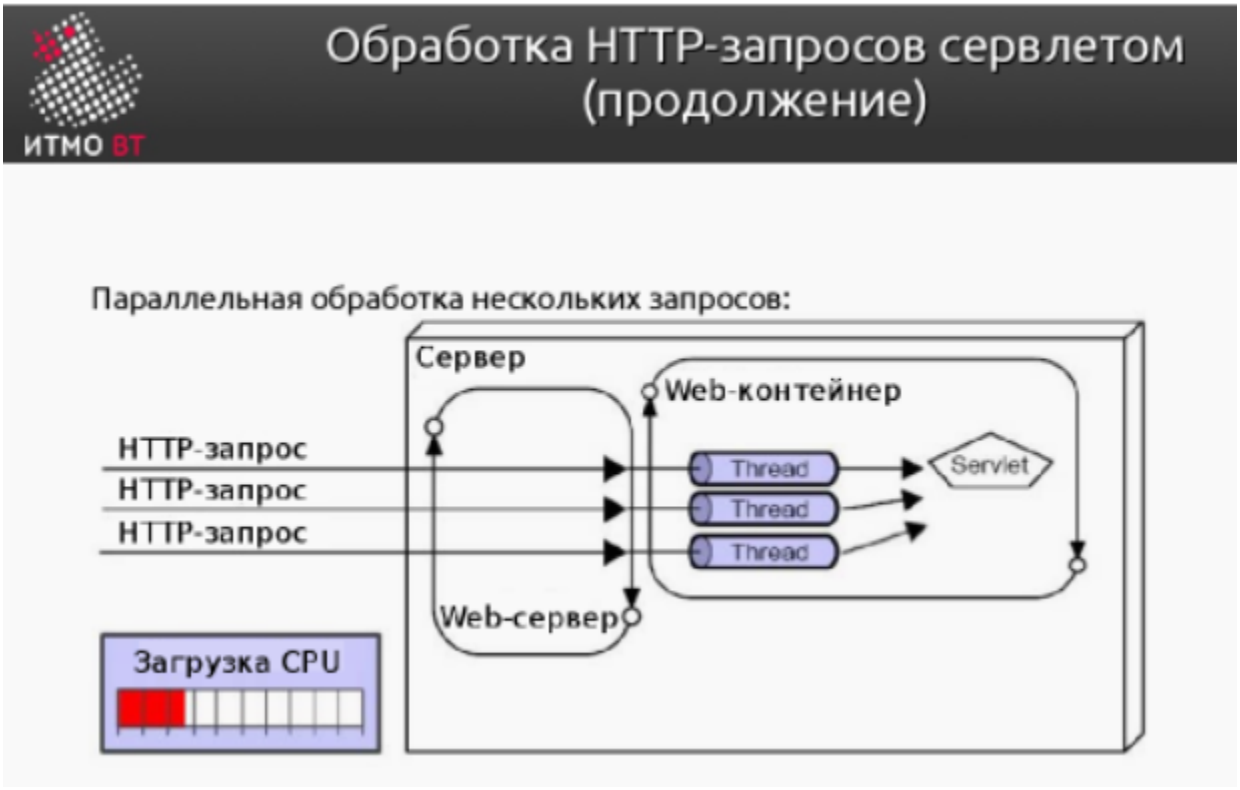
FastCGI. Пришел HTTP запрос. На уровне ОС крутится демон с HTTP листенером, ему отдается запрос, запрос обработался и отдал ответ. В FastCGI запросы

обратываются по той логике, по которой написано, как TCP листенер поступит с пришелшим запросом.

Обработка HTTP запроса сервлетом.

Полезно сравнить логику обработки запросов сервлетом и CGI-м. Наглядно картинками.





Веб-сервер перенаправляет (?) запрос на веб-контейнер, а тот помещает запрос в поток (который thread) и в нем будет исполняться класс сервлета. Когда запросов много, то же самое, но потоков много.

LWP - light-weight process. Java обеспечивает возможность многопоточной обработки независимо от того, обеспечивает ли это ее платформа (ОС, например), на которой java работает. Плодятся отдельные процессы, которые дублируется в потоки. LWP легковесный процесс на уровне операционки. Один thread работы java (в котором несколько своих thread-ов) транслируется на сущности уровня ОС, на ее LWP-шки. То есть все запросы обрабатывается в отдельных LWP. Это возвращаясь к сервлетам. По идее, FastCGI может быть быстрее всей этой радости с сервлетами, однако не факт, что действительно быстрее. Потому что там оптимизацией занимается сам программист, и не факт, что он соптмиизирует лучше, чем это сделано в сервлетах, хотя в принципе может. Но маловероятно.

Преимущества и недостатки сервлетов.

Преимущества:

1. Быстрее CGI. Логично: в CGI процессы, в сервлетах - потоки

2. Надежность и безопасность (из-за того что java надежна и безопасна). Это надежнее полностью открытого наружу CGI-shell-а.
3. Масштабируемость. Определяется многопоточностью, и тем что внизу java, которую можно как-то распределенно растащить. Это не свойства сервлетов. Это следсттвие работы application сервера, в который встроены какие-то механизмы расширения. Про масштабируемость - про возможность “адекватно” реагировать на возрастающую нагрузку. Она бывает вертикальная - наращивание мощности одного потока, а горизонтальная - через распараллеливание, например, добавление дополнительного сервера для передачи части нагрузки ему. Вертикальная масштабируемость лучше. Ну и то что сама java поддерживает масштабируемость, на уровне многопоточки например, это тоже играет роль, конечно.
4. Платоформенно-независимы. Следствия того, что это Java, что есть Java EE и application server со своими сервлетами.
5. Множество инструментов мониторинга и отладки. Это исторически так сложилось. Java изначально заточена под enterprise, там важно, чтобы возможности отладки и мониторинга были хорошие, поэтому они там и заложены. Богатый набор утилит для этого там есть. И этот набор сравнительно богаче, чем для каких-то других платформ, например, python-а (удобного и красивого, но со слабой отладкой)

Минусы - обеспечены не спецификацией сервлетов как таковых, а их сутью.

1. Слабое разделение уровня представления и бизнес логики. Написанием кода и компоновкой + версткой занимаются разные люди, чаще всего. И здорово, если бы они этим независимо друг от друга занимались. Однако сделать это сложно... Но это не проблема сервлетом. Вывод http разметки через System.out.println() - это ок со стороны сервлета, как серверного сценария, написанного на java, и это слезы верстальщика. Поэтому представление надо бы рендерить на уровне движка шаблонного. Верстальщик делает шаблон, а программист пишет код под этой шаблон. Все это нормально связывается и все счастливы.
2. Возможны конфликты при параллельной обработке запросов - тоже не проблема сервлетов как таковых. Но это проблемы просто параллельной разработки. С многопоточностью-то работать тяжко. Вот из-за этого и

проблема. О параллелизме надо думать, просто сервлеты писать недостаточно, параллелизм - не магия.