# Лабораторная работа 4

# Тема: Создание приложения J2EE с помощью EJB 3.1

## **Асинхронная обработка запроса**

В Servlet 3.0 добавлена возможность приостановки и возобновления обработки запроса, которая позволяет обслуживать запросы в асинхронном, неблокирующем режиме (паттерн [Comet](http://ru.wikipedia.org/wiki/Comet_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Когда обработка запроса приостанавливается, поток возвращается в контейнер, и готов для выполнения других задач. Когда запрашиваемый ресурс становится доступным, поток обрабатывает это событие, возобновляет приостановленный запрос и приступает к генерации ответа. Ниже перечислены некоторые из возможностей асинхронных сервлетов:

* Возможность получать данные от клиента, без блокировки потока, даже если данные поступают медленно (неблокирующий ввод).
* Возможность отправки данных клиенту без блокировки, даже если клиент или сеть работает медленно (не блокирующий выход).
* Возможность откладывать обработку запроса. Задержка обработки бывает полезной, если до обслуживания должен быть получен удаленный/медленный ресурс или, если ограничено одновременное использование определенного ресурса.
* Возможность обрабатывать задержки ответа, то есть в ответ можно добавить информацию об асинхронных событиях.
* Возможность уведомить о блокировке или неблокирующих событиях.

Что бы различать код предназначенный для синхронной и асинхронной обработки, аннотации сервлетов @WebServlet и фильтров @WebFilter имеют атрибут asyncSupported. Он может принимать значения true или false. Естественно этот атрибут можно установить и в дескрипторе развертывания. Асинхронными могут быть как сервлеты, так и фильтры. Если сервлет или фильтр, пытается запустить цепочку асинхронной обработки, и при этом он не помечен как асинхронный, будет выброшено исключение IllegalStateException.

**Асинхронный принцип работы**

Один из вариантов решения вышеописанных проблем заключается в том, чтобы проектировать приложения с учетом латентности, чтобы они были способны работать асинхронно, на основе событий. При использовании асинхронных API приложение подписывается на внешние события и выполняет необходимые действия при получении уведомлений об их наступлении. Как правило, подобные приложения организуются в виде нескольких циклов обработки событий, каждый из которых размещается в собственном потоке.

Очевидным преимуществом асинхронного, событийно-ориентированного дизайна является то, что операции, ожидающие откликов от внешних сервисов, могут выполняться параллельно при отсутствии между ними зависимостей по данным. Кроме этого, подобные архитектуры обладают гораздо лучшей масштабируемостью по сравнению с традиционным синхронным подходом, даже если никакие операции не выполняются параллельно.

**Наглядная демонстрация преимуществ асинхронных API**

Улучшение масштабируемости при использовании асинхронных API можно проиллюстрировать на примере простого сервлета. Если вы согласны, что асинхронные архитектуры – это достойный ответ на проблемы масштабируемости приложений Web 2.0, то можете смело переходить к [разделу, описывающему возможные варианты применения Java EE в мире Web 2.0](https://www.ibm.com/developerworks/ru/library/wa-aj-web2jee/#solutions).

В нашем примере сервлет сначала выполняет определенные действия по обработке входящего запроса, затем производит запрос к базе данных, после чего использует информацию, полученную из базы данных, для вызова Web-сервиса. Конечный ответ, отправляемый клиенту, генерируется на основе ответа Web-сервиса.

Сервлет использует два типа ресурсов, обладающих относительно высокой латентностью и различающихся по своим характеристикам и поведению при увеличении нагрузки:

* **Соединения с базой данных.** Этот ресурс обычно доступен Web-приложениям в виде объекта класса DataSource, содержащего ограниченное число соединений, которые могут использоваться параллельно.
* **Сетевые соединения.** Этот ресурс используется для отправки ответа клиенту и для вызова Web-сервиса. До недавнего времени этот ресурс в большинстве серверов приложений был ограниченным. Однако серверы последнего поколения используют [неблокирующий ввод-вывод](https://www.ibm.com/developerworks/ru/library/wa-aj-web2jee/#nio) (NIO), поэтому можно считать, что сервлету доступно столько сетевых соединений, сколько необходимо. Они используются в следующих ситуациях:
  1. **Вызов Web-сервиса.** Несмотря на то, что внешний сервис может принимать ограниченное число запросов, это количество как правило достаточно велико, поэтому продолжительность вызова определяется временем передачи данных по сети.
  2. **Получение входящего запроса от клиента.** В нашем примере используются запросы типа GET, поэтому данные затраты можно проигнорировать.
  3. **Отправка ответа клиенту.** Эти затраты также игнорируются, потому что в случае коротких ответов они могут буферизоваться в памяти сервера и отправляться по мере необходимости через NIO. Кроме того, если ответы короткие (а они будут именно такими в нашем примере), время их формирования также несущественно по сравнению со временем обработки запросов.

API асинхронных сервлетов включает так же новые методы ServletRequest. Что бы начать асинхронную обработку используется один из методов startAsync(ServletRequest, ServletResponse) или startAsync ().

[?](http://jenya-p.ru/home/1-blog/6-servlets30.html)

|  |  |
| --- | --- |
| 1 | AsyncContext aCtx = req.startAsync(req, res); |

Разница между ними в том, что startAsync() использует оригинальные реквест и респонс. В то время как передаваемые в параметрах, могут быть, к примеру, обернуты врапперами, ранее в цепочке обработки, и содержать пользовательские данные. Методы возвращают объект класса AsyncContext, инициализированный соответствующими реквестом и респонсом.

AsyncListener – новый класс слушателя для асинхронной обработки. Его можно использовать в приложении, для обработки таких событий как, завершение асинхронная обработки, тайм-аут, ошибка, или последовательный вызов startAsync. Следующий фрагмент кода создает новый объект AsyncListener для завершения асинхронной обработки.

[?](http://jenya-p.ru/home/1-blog/6-servlets30.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | AsyncContext ас = req.startAsync ();  req.addAsyncListener (новые AsyncListener () {  **public** **void** onComplete (AsyncEvent event) **throws** IOException {          ...      }      ...  } |

Параметр, передаваемый в метод onComplete, экземпляр AsyncEvent. Это еще один новый класс, в рамках Servlet 3.0. AsyncEvent представляет события начала, завершения, тайм-аута, или ошибки в асинхронной обработке запросов.

**Cкриншот:**

