# Лабораторная работа 4: Создание приложения J2EE с помощью EJB 3.1

## **Асинхронная обработка запроса**

Каждый из нас, вероятно, имел дело с медленными сервлетами, например, когда нужно ожидать ответ веб-сервиса, подключения к JDBC или JMS-сообщения, и так далее. На сегодня ситуация такая, что сервлет должен ждать окончания всех длительных процессов перед генерацией ответа. Это приводит к неэффективному блокированию потока и трате других ресурсов контейнера. Другим нежелательным эффектом является, например, случай JDBC соединения, когда база данных блокирует множество ожидающих потоков. Этот сценарий в конечном итоге приводит к нехватке потоков и низкому качеству работы всего WEB-контейнера. Еще один пример, когда уместна асинхронная обработка запросов, это чат. В этом случае клиент устанавливает “долгоживущий” запрос с сервером, и получает ответ в после поступления сообщения от другого пользователя. В данном случае, “медленными” ресурсами являются другие пользователи.

Для преодоления этих трудностей, 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. Например:

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

|  |  |
| --- | --- |
| 1 | @WebServlet(url="/foo", asyncSupported=**true**) |

Объявление асинхронного фильтра в файле web.xml:

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <**filter**>      <**filter-name**> FooFilter </**filter-name**>      <**filter-class**>samples.FooFilter</**filter-class**>      <**async-supported**>true</**async-supported**>  </**filter**>  <**filter-mapping**>      <**filter-name**>FooFilter</**filter-name**>      <**url-pattern**>/foo</**url-pattern**>  </**filter-mapping**> |

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.

Есть еще несколько удобных методов: isAsyncSupported (), isAsyncStarted () и getAsyncContext (), смысл которых понятен.

AsyncContext, новый класс в Servlet 3.0, представляет контекст выполнения асинхронной операции. Класс имеет методы, используемые для доступа к инкапсулированным объектам реквеста и респонса. Например, AsyncContext.dispatch(), AsyncContext.dispatch (path), или AsyncContext.dispatch (ServletContext, path) используются для отправки запроса в контейнер. Любой из этих методов, возвращает обработку запроса в контейнер после завершения асинхронной операции, начатой в ServletRequest, например, после ответа от веб-сервис. Эти методы просто повторно передают запрос, поэтому допустимо использовать такие структуры, как JavaServer Pages (JSP) для генерации ответа.

Метод dispatch() без аргументов направляет запрос на оригинальный URL. Кроме того учитываются вызовы AsyncContext.dispatch () и RequestDispatcher.forward() после инициализации асинхронного контекста. Метод dispatch(path) перенаправляет запрос на новый URL. А AsyncContext.dispatch(ServletContext, path) может так же быть использован для задания контектса.

Другой метод AsyncContext.complete() используется для завершения асинхронной операции. Он закрывает респонс, который был использован для инициализации этого объекта AsyncContext. Вы можете вызывать его когда завершится асинхронная операция. Контейнер так же может неявно вызвать этот метод, если приложение отправляет запрос обратно в контейнер методом forward() , без последующего вызова startAsync().

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.

После обзора ключевых особенностей асинхронной обработки в Servlet 3.0, давайте взглянем на более полный пример. Ниже, приведен код в котором сервлет делает асинхронный запрос веб-службы, ожидается ответ, и затем, запрос отправляет обратно в контейнер для формирования ответа с помощью JSP. Код упрощен для наглядности.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | @WebServlet("/foo", asyncSupported=**true**)  **public** **class** MyServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest req, HttpServletResponse res) {          // Обрадатываем запрос          // Формируем контекст асинхронной обработки          AsyncContext aCtx = request.startAsync(req, res);          // запускаем обращение к медленным ресурсам в отдельном потоке.          ScheduledThreadPoolExecutor executor = **new** ThreadPoolExecutor(10);          executor.execute(**new** AsyncWebService(aCtx));      }  }    **public** **class** AsyncWebService **implements** Runnable {      AsyncContext ctx;  **public** AsyncWebService(AsyncContext ctx) {  **this**.ctx = ctx;      }  **public** **void** run() {          // Вызываем WEB-сервис и сохраняем результат в атрибутах запроса          // Перенаправляем запрос для формирования ответа в JSP          ctx.dispatch("/render.jsp");     }  } |

Cкриншот работы программы:

