**Chapter 15  
Java Servlet Technology**

Shortly after the Web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for the same purpose. Initially, Common Gateway Interface (CGI) server-side scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology had many shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

Servlet是随着网络多动态内容而兴起的，开始是applet的一些早期的尝试，这些尝试使用client platform以达到目的。同时一些开发者投身到server playform中。

Server端的通用网管接口（CGI）脚本是提供动态内容的主要技术，尽管这项技术被很广泛的使用，但是他包含了很多的缺点。为了提供动态的、用户导向的内容，Java servlet 技术y应运而生。

The following topics are addressed here:

* [What Is a Servlet?](http://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html)
* [Servlet Lifecycle](http://docs.oracle.com/javaee/6/tutorial/doc/bnafi.html)
* [Sharing Information](http://docs.oracle.com/javaee/6/tutorial/doc/bnafo.html)
* [Creating and Initializing a Servlet](http://docs.oracle.com/javaee/6/tutorial/doc/bnafu.html)
* [Writing Service Methods](http://docs.oracle.com/javaee/6/tutorial/doc/bnafv.html)
* [Filtering Requests and Responses](http://docs.oracle.com/javaee/6/tutorial/doc/bnagb.html)
* [Invoking Other Web Resources](http://docs.oracle.com/javaee/6/tutorial/doc/bnagi.html)
* [Accessing the Web Context](http://docs.oracle.com/javaee/6/tutorial/doc/bnagl.html)
* [Maintaining Client State](http://docs.oracle.com/javaee/6/tutorial/doc/bnagm.html)
* [Finalizing a Servlet](http://docs.oracle.com/javaee/6/tutorial/doc/bnags.html)
* [The mood Example Application](http://docs.oracle.com/javaee/6/tutorial/doc/gkcpg.html)
* [Further Information about Java Servlet Technology](http://docs.oracle.com/javaee/6/tutorial/doc/bnagw.html)

What Is a Servlet?

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines lifecycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

Servlet是一个为了扩展servers的功能的java语言类，host application 通过请求-响应模型（request-response model）来接收访问。尽管servlets能够响应几乎任何的请求，但是他们被广泛的使用在扩展web服务器上的应用。为了这些web应用，java servlet技术定义了单独为http使用的类。

Javax.servlet和javax.servlet.http包为写servlet提供了接口和类。所有的servlet必须实现servlet接口，在接口中定义了存在于servlet的生命周期的方法。当要实现一个一般的service，可以使用或者扩展java servlet api中提供的一般的servlet类。例如，HttpServlet类提供了方法，如doGet和doPost来处理特定的http请求。

## Servlet Lifecycle

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
   1. Loads the servlet class.
   2. Creates an instance of the servlet class.
   3. Initializes the servlet instance by calling the init method. Initialization is covered in [Creating and Initializing a Servlet](http://docs.oracle.com/javaee/6/tutorial/doc/bnafu.html).
2. Invokes the service method, passing request and response objects. Service methods are discussed in [Writing Service Methods](http://docs.oracle.com/javaee/6/tutorial/doc/bnafv.html).

If it needs to remove the servlet, the container finalizes the servlet by calling the servlet’s destroy method. For more information, see [Finalizing a Servlet](http://docs.oracle.com/javaee/6/tutorial/doc/bnags.html).

一个servlet的生命周期是由该servlet被部署的容器控制的。当一个请求被映射到一个servlet处理后，这个容器就会执行以下步骤：

1. 如果该servlet实例不存在，那么web容器将会：
   1. 载入此servlet的类文件
   2. 创建这个servlet的实例
   3. 调用servlet中的init方法初始化这个servlet实例，初始化包含串讲和初始化一个servlet
2. 调用service方法，传递参数requet和response对象。

如果需要移除某个servlet，容器会通过调用servlet的destroy方法来终止该servlet。

### Handling Servlet Lifecycle Events

You can monitor and react to events in a servlet’s lifecycle by defining listener objects whose methods get invoked when lifecycle events occur. To use these listener objects, you must define and specify the listener class.

出来servlet生命周期中的事件：

通过定义listener对象，你可以监控和响应servlet生命周期中的事件，当生命周期中的事件发生时会激活这些方法。为了使用这些listener，你必须定义并指定这些监听类。

#### Defining the Listener Class

You define a listener class as an implementation of a listener interface. [Table 15-1](http://docs.oracle.com/javaee/6/tutorial/doc/bnafi.html#bnafl) lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the HttpSessionListener interface are passed an HttpSessionEvent, which contains an HttpSession.

定义监听类-listener class

通过实现一个监听接口-listener interface来定义一个监听类。下表列出了哪些事件可以被监听和相应的需要实现的接口。当一个监听方法被调用了之后，会传递一个包含了该事件信息的参数。

**Table 15-1 Servlet Lifecycle Events**

|  |  |  |
| --- | --- | --- |
| **Object** | **Event** | **Listener Interface and Event Class** |
| Web context | Initialization and destruction | javax.servlet.ServletContextListener and ServletContextEvent |
| Web context | Attribute added, removed, or replaced  添加、移除和替换属性事件 | javax.servlet.ServletContextAttributeListener and ServletContextAttributeEvent |
| Session | Creation, invalidation, activation, passivation, and timeout  创建、验证、激活和timeout事件 | javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener, and HttpSessionEvent |
| Session | Attribute added, removed, or replaced | javax.servlet.http.HttpSessionAttributeListener and HttpSessionBindingEvent |
| Request | A servlet request has started being processed by web components  被web组件开始处理一个servlet请求 | javax.servlet.ServletRequestListener and ServletRequestEvent |
| Request | Attribute added, removed, or replaced | javax.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent |

Use the @WebListener annotation to define a listener to get events for various operations on the particular web application context. Classes annotated with @WebListener must implement one of the following interfaces:

使用 @WebListener注解来定义一个监听器以监控web应用的各种事件，被此注解修饰的类必须实现以下接口中的至少一个。

javax.servlet.ServletContextListener

javax.servlet.ServletContextAttributeListener

javax.servlet.ServletRequestListener

javax.servlet.ServletRequestAttributeListener

javax.servlet..http.HttpSessionListener

javax.servlet..http.HttpSessionAttributeListener

For example, the following code snippet defines a listener that implements two of these interfaces:

import javax.servlet.ServletContextAttributeListener;

import javax.servlet.ServletContextListener;

import javax.servlet.annotation.WebListener;

@WebListener()

public class SimpleServletListener implements ServletContextListener,

ServletContextAttributeListener {

...

### Handling Servlet Errors

Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the following message:

A Servlet Exception Has Occurred

But you can also specify that the container should return a specific error page for a given exception.

处理servlet错误

当异常发生时，web容器会产生一个包含以下内容的默认页面：

但是你可以针对特定的异常指定容器返回特定的错误页面。

# 17.3 Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. Web components can do so by doing the following.

* Using private helper objects (for example, JavaBeans components).
* Sharing objects that are attributes of a public scope.
* Using a database.
* Invoking other web resources. The Java Servlet technology mechanisms that allow a web component to invoke other web resources are described in [Invoking Other Web Resources](https://docs.oracle.com/javaee/7/tutorial/servlets007.htm#BNAGI).

Web组件之间也会共同工作已完成他们的任务，他们可能通过以下操作：

使用私有的帮助对象（例如JavaBean组件）

共享public修饰的对象

使用数据库

使用其他的web资源，具体信息参考 [Invoking Other Web Resources](https://docs.oracle.com/javaee/7/tutorial/servlets007.htm#BNAGI)

## 17.3.1 Using Scope Objects

Collaborating web components share information by means of objects that are maintained as attributes of four scope objects. You access these attributes by using the getAttribute and setAttribute methods of the class representing the scope. [Table 17-2](https://docs.oracle.com/javaee/7/tutorial/servlets003.htm#BNAFQ) lists the scope objects.

***Table 17-2 Scope Objects***

| **Scope Object对象空间** | **Class** | **Accessible From** |
| --- | --- | --- |
| Web context | javax.servlet.ServletContext | Web components within a web context. See [Accessing the Web Context](https://docs.oracle.com/javaee/7/tutorial/servlets008.htm#BNAGL). |
| Session | javax.servlet.http.HttpSession | Web components handling a request that belongs to the session. See [Maintaining Client State](https://docs.oracle.com/javaee/7/tutorial/servlets009.htm#BNAGM). |
| Request | Subtype of javax.servlet.ServletRequest | Web components handling the request. |
| Page | javax.servlet.jsp.JspContext | The JSP page that creates the object. |

使用空间对象：

协作的web组件通过对象（这些对象作为四个空间对象的属性）来共享信息（或者可以传递信息）。你可以使用这写对象提供的getAttribute和setAttribute方法操作这些属性。

这些对象如下表所示，包括：ServletContext、httpSession、ServletRequest和JspContext

## 17.3.2 Controlling Concurrent Access to Shared Resources

In a multithreaded server, shared resources can be accessed concurrently. In addition to scope object attributes, shared resources include in-memory data, such as instance or class variables, and external objects, such as files, database connections, and network connections.

对并发访问共享资源的控制：

在多线程的服务器中，共享资源可能被并发访问。这些共享资源除了空间对象属性之外，还包括内存数据例如实例和类的变量，外部对象如文件、数据库连接和网络连接等。

Concurrent access can arise in several situations.

* Multiple web components accessing objects stored in the web context.
* Multiple web components accessing objects stored in a session.
* Multiple threads within a web component accessing instance variables. A web container will typically create a thread to handle each request. To ensure that a servlet instance handles only one request at a time, a servlet can implement the SingleThreadModel interface. If a servlet implements this interface, no two threads will execute concurrently in the servlet's service method. A web container can implement this guarantee by synchronizing access to a single instance of the servlet or by maintaining a pool of web component instances and dispatching each new request to a free instance. This interface does not prevent synchronization problems that result from web components' accessing shared resources, such as static class variables or external objects.

When resources can be accessed concurrently, they can be used in an inconsistent fashion. You prevent this by controlling the access using the synchronization techniques described in the Threads lesson at <http://docs.oracle.com/javase/tutorial/essential/concurrency/>.

一些情况会导致并发访问：

多web组件访问存储在web context中的对象

多web组件访问存储在session中的对象

在一个web组件内的多线程访问实例变量：

通常，一个web容器会为每个请求创建一个线程。一个servlet实例每次只处理一个request，servlet可以实现SingleThreadModel接口。如果一个servlet实现了这个接口，不会有两个线程同时执行servlet中的方法。Web容器通过同步访问一个servlet的单实例，或者维护一个web组件的线程池然后将每个请求分发给空闲的实例来请求。实现了这个接口也不会阻止web组件对共享资源的同步访问，web组件仍然可以访问共享资源如类变量或者外部的对象(这个也间接说明了web的容器可以实现单例而php的cgi模式不能实现单例)。

# 17.4 Creating and Initializing a Servlet

Use the @WebServlet annotation to define a servlet component in a web application. This annotation is specified on a class and contains metadata about the servlet being declared. The annotated servlet must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation. All other attributes are optional, with default settings. Use the value attribute when the only attribute on the annotation is the URL pattern; otherwise, use the urlPatterns attribute when other attributes are also used.

通过使用@WebServlet注解可以定义一个web应用的servlet组件。这个注解定义在一个类上，包含了定义一个servlet的metadata.这个被注解修饰的servlet必须包含一个指定的url模式。可以使用url格式的字符串或者值属性在这个注解上。除此之外，所有的其他的属性是可选的，如果没有指定则使用默认的配置。

Classes annotated with @WebServlet must extend the javax.servlet.http.HttpServlet class. For example, the following code snippet defines a servlet with the URL pattern /report:

被@WebServlet注解修饰的类必须继承自HttpServlet类

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

@WebServlet("/report")

public class MoodServlet extends HttpServlet {

...

The web container initializes a servlet after loading and instantiating the servlet class and before delivering requests from clients. To customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities, you can either override the init method of the Servlet interface or specify the initParams attribute of the @WebServlet annotation. The initParams attribute contains a @WebInitParam annotation. If it cannot complete its initialization process, a servlet throws an UnavailableException.

Use an initialization parameter to provide data needed by a particular servlet. By contrast, a context parameter provides data that is available to all components of a web application.

当web容器加载和初始化一个servlet类后，会调用init方法初始化一个servlet，然后才会分发处理客户端的请求。为了自定义处理，如允许servlet读取持久性的配置数据，初始化资源和任何其他的一次性的活动，开发者可以重写Servlet接口中的init方法，或者在@WebServlet注解中指定initParms属性。InitParams属性包含一个 @WebInitParam属性。如果在初始化阶段没有完成此初始化过程，servlet会抛出一个UnavailableException异常。

# 17.5 Writing Service Methods

The service provided by a servlet is implemented in the service method of a GenericServlet, in the do*Method* methods (where *Method* can take the value Get, Delete, Options, Post, Put, or Trace) of an HttpServlet object, or in any other protocol-specific methods defined by a class that implements the Servlet interface. The term **service method** is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information. For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Response headers must always be set before the response has been committed. The web container will ignore any attempt to set or add headers after the response has been committed. The next two sections describe how to get information from requests and generate responses.

一个servlet提供服务实现了GenericServlet类中的服务方法（HttpServlet类继承了GenericServlet类）。在HttpServlet对象中doMethod方法中（method可以是Get,Delete,Options,Post,Put或者Trace），或者在其他协议(这里指的是http，tcp等协议，以为servlet可以处理几乎任何的请求)中定义的方法，这些方法从实现Servlet接口继承而来。术语**service method**指的是Servlet类中的任意方法，由这些方法来为client提供服务。

服务的一般编程模式是从requesth中获取信息，然后基于这些请求信息访问外部的资源然后做出响应。对于HTTP的Servlet来说，正确的响应请求是遵循以下流程：

1. 打开一个此响应的输出流
2. 填充响应头部
3. 先输出流中写入任何数据

注意：在响应提交之前必须设置好响应头。当响应提交之后，web容器会忽略一切响应头的添加和修改。

下面的两部分描述了如何从request获取信息以及如何产生response。

## 17.5.1 Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the ServletRequest interface. This interface defines methods for accessing the following information:

* Parameters, which are typically used to convey information between clients and servlets
* Object-valued attributes, which are typically used to pass information between the web container and a servlet or between collaborating servlets
* Information about the protocol used to communicate the request and about the client and server involved in the request
* Information relevant to localization

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the BufferedReader object returned by the request's getReader method. To read binary data, use the ServletInputStream returned by getInputStream.

从request获取请求信息：

一个request携带者client和servlet之前通信的数据，所有的request实现了ServletRequest接口，通过此接口中的方法可以访问以下信息：

1. 参数-典型的用于client和servlet之间转换数据
2. 对象值属性-典型的用于在web容器和servlet之间或者协同的servlets之间传递数据。
3. 协议信息-request的client和servlet之间交互使用的协议信息
4. 本地的某些信息

除了使用接口提供的方法外，你也可以通过解析request的输入流来手动获取这些相关信息。可以使用BufferedReader对象来读取字节流，或者通过ServletInputStream来获取二进制流。

HTTP servlets are passed an HTTP request object, HttpServletRequest, which contains the request URL, HTTP headers, query string, and so on. An HTTP request URL contains the following parts:

http://[*host*]:[*port*][*request-path*]?[*query-string*]

http servlet传递一个HTTP request对象给方法，HttpServletRequest包含了请求的url、请求头、请求参数等等。一个Http的请求的url包含以下部分：

[http://[*host*]:[*port*][*request-path*]?[*query-string*](http://[host]:[port][request-path]?[query-string)]、

The request path is further composed of the following elements.

* **Context path**: A concatenation of a forward slash (/) with the context root of the servlet's web application.
* **Servlet path**: The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (/).
* **Path info**: The part of the request path that is not part of the context path or the servlet path.

You can use the getContextPath, getServletPath, and getPathInfo methods of the HttpServletRequest interface to access this information. Except for URL encoding differences between the request URI and the path parts, the request URI is always comprised of the context path plus the servlet path plus the path info.

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the getParameter method. There are two ways to generate query strings.

* A query string can explicitly appear in a web page.
* A query string is appended to a URL when a form with a GET HTTP method is submitted.

一个请求的path包含以下部分：

1. Context path：前向斜线分隔的context，带有web应用的servlet
2. Servlet path：对以组件别名的部分的路径激活请求，这个路径一个前向斜线/开始
3. Path info：请求路径的path，这个路径不是context路径，也不是servlet path

可以使用HttpServletRequest接口的getContextPath, getServletPath, and getPathInfo方法来获取这些信息，除了Url的编码部分致使的URI和path部分的不同外，请求的URI总是由context path 和servlet path和path info组成。

请求的字符串是一些参数和值的集合。每个参数可以通过调用的request的getParameter方法来获得。这有两个方法来产生query strings:

* 1. 请求的字符串，显示的暴露在web页面中
  2. 当一个HTTPget请求被提交之后，请求的字符串是附加在url之后的

## 17.5.2 Constructing Responses

A response contains data passed between a server and the client. All responses implement the ServletResponse interface. This interface defines methods that allow you to do the following.

* Retrieve an output stream to use to send data to the client. To send character data, use the PrintWriter returned by the response's getWriter method. To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the ServletOutputStream returned by getOutputStream. To mix binary and text data, as in a multipart response, use a ServletOutputStream and manage the character sections manually.
* Indicate the content type (for example, text/html) being returned by the response with the setContentType(String) method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at <http://www.iana.org/assignments/media-types/>.
* Indicate whether to buffer output with the setBufferSize(int) method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.
* Set localization information, such as locale and character encoding. See [Chapter 20, "Internationalizing and Localizing Web Applications"](https://docs.oracle.com/javaee/7/tutorial/webi18n.htm#BNAXU) for details.

HTTP response objects, javax.servlet.http.HttpServletResponse, have fields representing HTTP headers, such as the following.

* Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.
* Cookies, which are used to store application-specific information at the client. Sometimes, cookies are used to maintain an identifier for tracking a user's session (see [Session Tracking](https://docs.oracle.com/javaee/7/tutorial/servlets009.htm#BNAGR)).

构造响应：

一个response包含了在server和client之间传递的数据。所有的响应实现了ServletResponse接口。这个接口定义了允许你调用以下的方法：

1. 使用一个输出流以向client发送数据
   1. 为了发送字节流数据，调用response的getWriter方法返回的PrintWriter对象；在一个MIME体类型中返回二进制数据可以使用getOutputStream方法返回的ServletOutputStream 对象。在多部分响应中，为了返回二进制数据和文本数据，可以使用ServletOutputStream 对象并手动控制字节部分的返回。
2. 通过setContentType方法告诉client响应的类型
   1. 这个方法必须在response被提交（commit）之前调用
   2. 可以在网站<http://www.iana.org/assignments/media-types/> 查看可用的MIME类型
3. 使用setBufferSize表明是否是buffer输出
   1. 默认的，任何输出到输出流的数据会被立即发送给client。缓冲的设置可以控制是的输出在发送到client之前先写入缓冲，因此为servlet提供更多的时间以设置合适的状态码和headers或者转发到另一个web资源。
   2. 这个方法必须在任何输出或者response提交之前调用才能生效
4. 设置初始化信息，例如本地和字符编码信息，详细信息查看 [Chapter 20, "Internationalizing and Localizing Web Applications"](https://docs.oracle.com/javaee/7/tutorial/webi18n.htm#BNAXU)

HTTP响应对象，javax.servlet.http.HttpServletResponse,包含了一下字段代表了响应的HTTP状态码，如下：

1. 状态码
   1. 用来指明request是否被很好的处理或者一个request被重定向了
2. Cookies
   1. 用来在client方存储特定的应用信息
   2. 某些时候，cookie可以被用来追踪用户的session

# 7.6 Filtering Requests and Responses

A **filter** is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.

过滤请求和响应

一个filter对象是用来转换一个request的header和content。Filters与web组件不同因为它们自己通常不产生响应。相反，一个filter提供了功能，能够被附加（attach）到任何的web资源上。而且，一个filter不应该依赖于任何的web资源因为让是一个filter，这样他才能够组成更多类型的web资源。

The main tasks that a filter can perform are as follows.

* Query the request and act accordingly.
* Block the request-and-response pair from passing any further.
* Modify the request headers and data. You do this by providing a customized version of the request.
* Modify the response headers and data. You do this by providing a customized version of the response.
* Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

一个filter的主要职责是：

* 1. 依照请求和act查询
  2. 阻断request和响应以便未来传递和处理
  3. 改变request的头和数据，通过为request自定义一个version信息
  4. 改变response的头和数据，通过为response自定义一个version信息
  5. 与外部的资源进行交互

应用的过滤功能包含：权限验证、日志、图片转换、数据压缩、编码、tokenzing流和xml数据转换等等。

## 17.6.1 Programming Filters

The filtering API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package. You define a filter by implementing the Filter interface.

Use the @WebFilter annotation to define a filter in a web application. This annotation is specified on a class and contains metadata about the filter being declared. The annotated filter must specify at least one URL pattern. This is done by using the urlPatterns or value attribute on the annotation. All other attributes are optional, with default settings. Use the value attribute when the only attribute on the annotation is the URL pattern; use the urlPatterns attribute when other attributes are also used.

Classes annotated with the @WebFilter annotation must implement the javax.servlet.Filter interface.

To add configuration data to the filter, specify the initParams attribute of the @WebFilter annotation. The initParams attribute contains a @WebInitParam annotation. The following code snippet defines a filter, specifying an initialization parameter:

Filters的编程

Filter的api由javax.servlet包中的Filter接口、FilterChain和FilterConfig接口来定义。需要实现Filter接口来定义一个filter

在web应用中使用@ WebFilter 注解来定义一个filter，这个注解需要在类上并且包含了声明一个filter的metadata。这个被注解修饰的类必须指定一个url的格式来表明它可以拦截的请求。这可以通过在注解的valu属性或者直接在注解参数里面使用url格式的字符串来定义。其他的属性都是可选的，如果没有指定则使用的是默认值。当只在注解中声明url格式的时候需要使用value的来声明。当还指定了其他的属性的时候使用url格式的字符串。

import javax.servlet.Filter;

import javax.servlet.annotation.WebFilter;

import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter",

urlPatterns = {"/\*"},

initParams = {

@WebInitParam(name = "mood", value = "awake")})

public class TimeOfDayFilter implements Filter {

...

The most important method in the Filter interface is doFilter, which is passed request, response, and filter chain objects. This method can perform the following actions.

* Examine the request headers.
* Customize the request object if the filter wishes to modify request headers or data.
* Customize the response object if the filter wishes to modify response headers or data.
* Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created. Alternatively, the filter can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
* Examine response headers after invoking the next filter in the chain.
* Throw an exception to indicate an error in processing.

In addition to doFilter, you must implement the init and destroy methods. The init method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the FilterConfig object passed to init.

Filter接口中最重要方法是doFilter，这个方法传递了request、response和filter链对象。这个方法能够处理以下的行为：

1. 检查request的头
2. 如果filter希望修改request的头或者数据的时候修改request对象（从而自定义了request的对象）
3. 如果filter希望修改response的头或者数据的时候修改request对象（从而自定义了response的对象）
4. 激活在filter链条里面的下一个filter实体：

如果当前的filter是chain中的最后一个filter,.然后就是目标web组件或者静态资源，那么下一个实体就是这个chain中的资源。否则下一个filter就是在war中配置的filter。

一个filter通过调用chain对象的doFilter方法来调用下一个实体。传递request和response作为参数。这些request和response可能是已经处理过的（即被此filter处理）。可选择的是，这个filter可以通过不调用下一个实体的方法来选择阻塞请求。在后面的例子中，这个filter负责填充这个response。

1. 在激活filter chain中的下一个filter实体后检查response的头（该filter chain利用了某种设计模式，在处理完请求后会自动回来调用之后的对响应的请求）
2. 如果在处理过程中出错则可以抛出异常

为了实现doFilter方法，必须实现Filter接口的init和destroy方法。当filter初始化的时候被容器调用。如果你希望传递init参数给filter则可以通过FilterConfig对象来传递给init方法

## 17.6.2 Programming Customized Requests and Responses

There are many ways for a filter to modify a request or a response. For example, a filter can add an attribute to the request or can insert data in the response.

A filter that modifies a response must usually capture the response before it is returned to the client. To do this, you pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the getWriter or getOutputStream method to return this stand-in stream. The wrapper is passed to the doFilter method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object.

To override request methods, you wrap the request in an object that extends either ServletRequestWrapper or HttpServletRequestWrapper. To override response methods, you wrap the response in an object that extends either ServletResponseWrapper or HttpServletResponseWrapper.

自定义request和response的编程：

Filter可以利用许多方式来修改一个request和response。例如，filter可以为request添加一个属性或者向response中插入一条数据。

修改了一个response的filter必须在这个reponse被发送到client之前捕获这个response。为了做到这一点，你可以传递一个标准输入流到servlet产生response。这个标准输入流会在原始的response stream完成并允许filter修改的时候阻止原始response steam的关闭。

为了传递一个标准的输入流到servlet，filter可以通过创建一个response wrapper，这个wrapper重写getWriter或者getOutputStream方法来返回一个标准输入流。这个wrapper被传递给filter chain对象的doFiltrer方法。Wrapper方法默认通过被封装的request和response对象来调用。

为了重写request方法，你的wrapper需要继承ServletRequestWrapper or HttpServletRequestWrapper；为了重写response方法，你的wrapper需要继承自ServletResponseWrapper or HttpServletResponseWrapper。

## 17.6.3 Specifying Filter Mappings

A web container uses filter mappings to decide how to apply filters to web resources. A filter mapping matches a filter to a web component by name or to web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a WAR in its deployment descriptor by either using NetBeans IDE or coding the list by hand with XML.

指定Filter的拦截映射

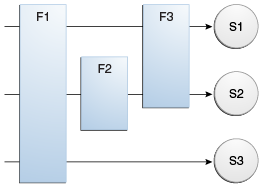
一个web容器使用filter映射来决定如何应用filter到web资源上。一个filter映射通过filter名字或者url格式定义的web资源匹配一个filter到一个web组件上。War包中的filter映射列表中的filter会按照顺序被调用。可以通过ide或者手动编写xml文件来为一个war包在部署目录指定一个filter 映射列表。

If you want to log every request to a web application, you map the hit counter filter to the URL pattern /\*.

You can map a filter to one or more web resources, and you can map more than one filter to a web resource. This is illustrated in [Figure 17-1](https://docs.oracle.com/javaee/7/tutorial/servlets006.htm#BNAGH), in which filter F1 is mapped to servlets S1, S2, and S3; filter F2 is mapped to servlet S2; and filter F3 is mapped to servlets S1 and S2.

如果你希望记录一个web应用的所有请求，你可以指明url模式为/\*。

***Figure 17-1 Filter-to-Servlet Mapping***

  
[Description of "Figure 17-1 Filter-to-Servlet Mapping"](https://docs.oracle.com/javaee/7/tutorial/img_text/jeett_dt_018.htm)

Recall that a filter chain is one of the objects passed to the doFilter method of a filter. This chain is formed indirectly by means of filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the web application deployment descriptor.

When a filter is mapped to servlet S1, the web container invokes the doFilter method of F1. The doFilter method of each filter in S1's filter chain is invoked by the preceding filter in the chain by means of the chain.doFilter method. Because S1's filter chain contains filters F1 and F3, F1's call to chain.doFilter invokes the doFilter method of filter F3. When F3's doFilter method completes, control returns to F1's doFilter method.

Filter到servlet的映射图：

回想一下filter的chain，这个对象的方法doFilter需要传递一个filter对象作为参数。这个chain被filter映射规范化了。在chain中的filter的顺序与web应用部署描述文件中定义的filter映射的顺序数一致的。

正如上面图表展示的，但一个filter被映射到servlet-S1，web容器 会调用F1的doFilter方法。在S1的filter chain中的每一个filter的doFilter方法都会被它之前的filter通过chain.doFilter方法来调用。因为servlet 1的filter chain上包含filter F1和F3，所以在F1中调用doFilter方法将会激活F3中的doFilter方法。当F3的doFilter方法处理完成时会返回到F1的doFilter方法。

### 17.6.3.1 To Specify Filter Mappings Using NetBeans IDE

1. Expand the application's project node in the **Project** tab.
2. Expand the **Web Pages** and **WEB-INF** nodes under the project node.
3. Double-click web.xml.
4. Click **Filters** at the top of the editor window.
5. Expand the **Servlet Filters** node in the editor window.
6. Click **Add Filter Element** to map the filter to a web resource by name or by URL pattern.
7. In the Add Servlet Filter dialog box, enter the name of the filter in the **Filter Name** field.
8. Click **Browse** to locate the servlet class to which the filter applies.

You can include wildcard characters so that you can apply the filter to more than one servlet.

1. Click **OK**.
2. To constrain how the filter is applied to requests, follow these steps.
   1. Expand the **Filter Mappings** node.
   2. Select the filter from the list of filters.
   3. Click **Add**.
   4. In the Add Filter Mapping dialog box, select one of the following dispatcher types:
      * **REQUEST**: Only when the request comes directly from the client
      * **ASYNC**: Only when the asynchronous request comes from the client
      * **FORWARD**: Only when the request has been forwarded to a component (see [Transferring Control to Another Web Component](https://docs.oracle.com/javaee/7/tutorial/servlets007.htm#BNAGK))
      * **INCLUDE**: Only when the request is being processed by a component that has been included (see [Including Other Resources in the Response](https://docs.oracle.com/javaee/7/tutorial/servlets007.htm#BNAGJ))
      * **ERROR**: Only when the request is being processed with the error page mechanism (see [Handling Servlet Errors](https://docs.oracle.com/javaee/7/tutorial/servlets002.htm#BNAFN))

You can direct the filter to be applied to any combination of the preceding situations by selecting multiple dispatcher types. If no types are specified, the default option is **REQUEST**.

# 7.7 Invoking Other Web Resources

Web components can invoke other web resources both indirectly and directly. A web component indirectly invokes another web resource by embedding a URL that points to another web component in content returned to a client. While it is executing, a web component directly invokes another resource by either including the content of another resource or forwarding a request to another resource.

激活其他的web资源：

Web组件可以直接或者间接的激活其他的web资源。一个web组件可以通过在返回client的内容中嵌入一个指向其他web资源的url来间接激活其他web组件。一个web组件可以包含进一个其他的web资源或者直接将request请求到其他的资源来间接激活另一个web的资源。

？？？？？？？？？？？？？？？？？？

To invoke a resource available on the server that is running a web component, you must first obtain a RequestDispatcher object by using the getRequestDispatcher("URL") method. You can get a RequestDispatcher object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a /), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a RequestDispatcher object for that type of resource, getRequestDispatcher will return null. Your servlet should be prepared to deal with this condition.

为了在一个运行web组件的服务器上激活一个可获得的资源，必须通过getRequestDispatcher("URL")方法来包含进一个RequestDispatcher 对象。可以从一个请求或者一个web的context中获取一个RequestDispatcher 对象。然而这个两个方法有在行为上有一些的不同。这个方法将path作为一个参数传递给被请求的资源。

一个request可以传递一个相对路径（也就是说不是以/开头的路径），但是web context需要绝对路径。如果不能获取到资源或者server不是一个RequestDispatcher 对象类型的的资源，getRequestDispatcher 将会返回null。你的servlet应该预先处理好这种情况（即做这方面的预案）。

## 17.7.1 Including Other Resources in the Response

It is often useful to include another web resource, such as banner content or copyright information, in the response returned from a web component. To include another resource, invoke the include method of a RequestDispatcher object:

include(request, response);

If the resource is static, the include method enables programmatic server-side includes. If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet. An included web component has access to the request object but is limited in what it can do with the response object.

* It can write to the body of the response and commit a response.
* It cannot set headers or call any method, such as setCookie, that affects the headers of the response.

在一个web组件的返回内容中包含进其他web资源经常是很有用的，例如一个banner内容或者版权信息。为了包含其他的资源。调用RequestDispatcher 对象的include方法。如：include(request, response);。

如果资源是静态的，那么include方法将会启用程序的服务端include。如果资源是web组件，这个方法的效果是发送请求到这个被包含进入的web组件，执行这个web组件。并且将这个web组件的执行结果包含进servlet的响应中。一个包含进的web组件可以访问request的对象但是只能对response对象做有限的操作，如下：

1. 它能够向响应体中写入数据
2. 他能够设置头信息并且调用任何的方法。例如设置cookie，这个将会影响到响应的头信息。

## 17.7.2 Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component, depending on the nature of the request.

To transfer control to another web component, you invoke the forward method of a RequestDispatcher. When a request is forwarded, the request URL is set to the path of the forwarded page. The original URI and its constituent parts are saved as the following request attributes:

javax.servlet.forward.request\_uri

javax.servlet.forward.context\_path

javax.servlet.forward.servlet\_path

javax.servlet.forward.path\_info

javax.servlet.forward.query\_string

The forward method should be used to give another resource responsibility for replying to the user. If you have already accessed a ServletOutputStream or PrintWriter object within the servlet, you cannot use this method; doing so throws an IllegalStateException.

将控制权转到另一个web组件：

在一些应用中，你可能希望然一个web组件对request做基本的处理，然后用另外的web组件来产生响应。例如，更具自然段请求，你可能希望处理部分请求然后将请求转发到另外的web组件。

你可以调用RequestDispatcher对象的forward 方法来将request转发都另一个web组件。当请求被forwarded，这个请求的url会被设置为forwarded页面。源URI和它的其他成分将会被保存在请求的属性，这些部分包括：

Request\_uri

Context\_path

Servlet\_path

Path\_info

Query\_string

这个forward方法应该常常给出另一资源来响应用户。如果你已经在servlet中访问了ServletOutputStream 或者PrintWriter 对象，那么你不能使用这个方法，否则会抛出IllegalStateException异常。

## Accessing the Web Context

The context in which web components execute is an object that implements the ServletContext interface. You retrieve the web context by using the getServletContext method. The web context provides methods for accessing

* Initialization parameters
* Resources associated with the web context
* Object-valued attributes
* Logging capabilities

Web组件执行的context是一个实现了ServletContext接口的对象，开发者可以使用getServletContext方法得到这个web的context。Web context提供了一下访问功能：

1. 初始化参数
2. 关联web context和资源
3. 对象的值属性
4. 日志记录功能

The counter’s access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object by using the context’s getAttribute method. The incremented value of the counter is recorded in the log.

访问方法的数量是同步的改变的，因此会阻止当前servlet正在执行的不适当的操作。一个过滤器filter返回通过使用context的getAttribute方法返回/得到一个counter对象。Couter的增加被记录到日志。

## Maintaining Client State

Many applications require that a series of requests from a client be associated with one another. For example, a web application can save the state of a user’s shopping cart across requests. Web-based applications are responsible for maintaining such state, called a session, because HTTP is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

保持client的状态

许多的应用要求管理器从一个client获取的请求（me：即保存client的状态）。例如，一个web应用可以在多个请求之间保存用户的购物车数据。因为HTTP是无状态的，基于web的应用负责保存这些状态，被称作session（me:也就说session是服务端保存client的多次请求的状态的一种技术）。为了支持应用保存状态信息，java的servlet技术提供了一个API来管理session,并且允许很多的机制来实现session的功能。

### Accessing a Session

Sessions are represented by an HttpSession object. You access a session by calling the getSession method of a request object. This method returns the current session associated with this request; or, if the request does not have a session, this method creates one.

访问/获取session

Session以HttpSession对象存在。你可以通过调用request对象的getSession方法来获取一个session。这个方法返回与当前request联系的session。或者如果request没有session，则这个方法内部会创建一个并返回。

### Associating Objects with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any web component that belongs to the same web context and is handling a request that is part of the same session.

Recall that your application can notify web context and session listener objects of servlet lifecycle events ([Handling Servlet Lifecycle Events](https://javaee.github.io/tutorial/servlets002.html#BNAFJ)). You can also notify objects of certain events related to their association with a session, such as the following.

* When the object is added to or removed from a session. To receive this notification, your object must implement the javax.servlet.http.HttpSessionBindingListener interface.
* When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement thejavax.servlet.http.HttpSessionActivationListener interface.

通过session关联对象

可以通过session并提供一个名字来访问对象的值属性。这些属性可以被属于相同的web context的任何的web组件（即servlet filter listener等）访问。并且绑定了一个request

回想一下，应用可以通知web context和servlet声明周期中事件的监听器。你也可以通过session来通知一个事件，像下面这样：

1. 当对象从一个session中被添加或者删除时获取此事件的通知，你的对象必须实现javax.servlet.http.HttpSessionBindingListener 接口。
2. 当对象的session被激活或者是停止激活时，一个session将会同样的被激活或者停止激活即移动到虚拟机或者重新保存到持久存储介质。为了接受这个通知，你的对象必须实现javax.servlet.http.HttpSessionActivationListener接口。

### Session Management

Because an HTTP client has no way to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session’s getMaxInactiveInterval and setMaxInactiveInterval methods.

* To ensure that an active session is not timed out, you should periodically access the session by using service methods because this resets the session’s time-to-live counter.
* When a particular client interaction is finished, you use the session’s invalidate method to invalidate a session on the server side and remove any session data.

Session管理

因为http不能发出信号表明他不在需要session了，因此每个session都有一个关联的timeout时间以便于session所占用的资源被回收。这个timeout时间可以通过使用getMaxInactiveInterval和setMaxInactiveInterval方法来获取和设置。

1. 为了确保一个激活状态的session没有过期，我们需要不间断的通过服务方法来访问session，因为这样可以重置session的timeout计时起始时间。
2. 当一个特定的client交互完成后，你可以使用sesssion的invalidate方法来是的session在server端无效并且移除任何session的数据。

#### To Set the Timeout Period Using NetBeans IDE

To set the timeout period in the deployment descriptor using NetBeans IDE, follow these steps.

1. Open the project if you haven’t already.
2. Expand the node of your project in the Projects tab.
3. Expand the Web Pages and WEB-INF nodes that are under the project node.
4. Double-click web.xml.
5. Click General at the top of the editor.
6. In the Session Timeout field, enter an integer value.

The integer value represents the number of minutes of inactivity that must pass before the session times out.

### Session Tracking

To associate a session with a user, a web container can use several methods, all of which involve passing an identifier between the client and the server. The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the response’s encodeURL(URL) method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, the method returns the URL unchanged.

Session 追踪

为了将一个session与一个user关联起来，web容器能够使用许多的方法，所有的这些方法的核心都是在server和client之间传递一个标识符（此标识符能够唯一识别这个user）（**session工作方法的核心**）。这个标识符可以保存在client的cookie上，或者web组件通过将此标识符包含进每次返回给client的url中（**me:这也是在client端禁用cookie时服务端仍然想要记录client的状态信息所采取的策略**）。

如果你的应用使用了session的对象，你必须确保session追踪被被启动了，从而通过应用的重写url来在server和client之间实现sessio机制，而不管客户端是否关闭了cookie的功能。

你可以通过调用response的encodeURL(URL)方法处理所有返回到client的url上。通过这个方法在client的cookie关闭的时候包含一个sessionId到url中。如果client的cookie处启动状态，该方法不会对url做任何的处理直接返回。

## Finalizing a Servlet

The web container may determine that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down). In such a case, the container calls the destroy method of the Servlet interface. In this method, you release any resources the servlet is using and save any persistent state. The destroy method releases the database object created in the init method.

终止一个servlet

一个web容器也许会将一个servlet应该被从service上移除。例如，当一个web容器希望回收内存资源或者当web容器关闭的时候就会将servlet从service中移除。在servlet即将被关闭的时候，容器将会调用servlet接口并且被具体的Servlet类实现的destroy方法。在这个方法内，你释放了所有这个servlet使用的资源并且保存了它的任何持久性状态。方法destroy会释放在init方法中创建的数据库对象。

A servlet’s service methods should all be complete when a servlet is removed. The server tries to ensure this by calling the destroy method only after all service requests have returned or after a server-specific grace period, whichever comes first. If your servlet has operations that may run longer than the server’s grace period, the operations could still be running when destroy is called. You must make sure that any threads still handling client requests complete.

当一个servlet被移除的时候，这个servlet的服务应该都被处理完成了。Server通过仅仅在所有的服务请求都被返回或者等待一段合理的事件，然后调用destroy方法来确保该servlet的服务都已经处理完成了。如果你的servlet的操作运行的时间比服务的优雅关闭时间长，调用destroy方法的时候这些操作仍然在运行。你需要确保任意线程处理完成client的请求。

The remainder of this section explains how to do the following.

* Keep track of how many threads are currently running the service method.
* Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete.
* Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

下面的部分说明了这些操作：

1. 追踪当前有多少个线程正在执行service方法(通过追踪thread方法可以看到tomcat启动了多个线程组成一个线程池，然后轮询这个线程池里面的线程)
2. 提供一个干净的shutdown，这可以通过调用destroy方法，在方法内部通知正在运行的长线程，并等待这些线程执行完成。
3. 使得长时间运行的线程每隔一段时间检查一次是否关闭，如果确实需要关闭的话，那就直接停止线程的工作并清理资源然后返回。

### Tracking Service Requests

To track service requests:

1. Include a field in your servlet class that counts the number of service methods that are running.

The field should have synchronized access methods to increment, decrement, and return its value:

追踪服务的请求：

1. 在你的servlet类中包含一个字段用来记录多少个服务方法正在运行

这个字段应该处理好同步访问以增加、减少和返回它的值。代码如下：

public class ShutdownExample extends HttpServlet {

private int serviceCounter = 0;

...

// Access methods for serviceCounter

protected synchronized void enteringServiceMethod() {

serviceCounter++;

}

protected synchronized void leavingServiceMethod() {

serviceCounter--;

}

protected synchronized int numServices() {

return serviceCounter;

}

}

The service method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your HttpServlet subclass should override the service method. The new method should callsuper.service to preserve the functionality of the original service method:

这个service方法需要在每次进入方法的时候增加service的counter（me：每次进入到servlet中都会优先进入到servlet的service方法中，这个方法在根据http的方法来转发请求到不同的servlet中），并且在每次方法返回的时候减少这个counter。这个需要你的继承自HttpServlet的子类重写服务方法。新的service方法中需要调用父类的service方法以提供原始的sevice方法的功能，然后增加自己的处理逻辑。

protected void service(HttpServletRequest req,

HttpServletResponse resp)

throws ServletException,IOException {

enteringServiceMethod();

try {

super.service(req, resp);

} finally {

leavingServiceMethod();

}

}

### Notifying Methods to Shut Down

To ensure a clean shutdown, your destroy method should not release any shared resources until all the service requests have completed:

1. Check the service counter.
2. Notify long-running methods that it is time to shut down.

For this notification, another field is required. The field should have the usual access methods:

通知方法以便关闭servlet:

为了确保干净的关闭servlet，你的destory方法，应该等到所有的服务请求都被完全的处理之后再释放共享的资源。可以遵照如下步骤：

1. 检查service的counter的值以确定多少个服务方法正在执行
2. 通知正在运行的线程，要关闭servlet了

为了解决这个问题，需要另一个字段，这个字段应该含有通用的访问方法

public class ShutdownExample extends HttpServlet {

private boolean shuttingDown;

...

//Access methods for shuttingDown

protected synchronized void setShuttingDown(boolean flag) {

shuttingDown = flag;

}

protected synchronized boolean isShuttingDown() {

return shuttingDown;

}

}

Here is an example of the destroy method using these fields to provide a clean shutdown:

public void destroy() {

/\* Check to see whether there are still service methods /\*

/\* running, and if there are, tell them to stop. \*/

if (numServices()> 0) {

setShuttingDown(true);

}

/\* Wait for the service methods to stop. \*/

while (numServices()> 0) {

try {

Thread.sleep(interval);

} catch (InterruptedException e) {

}

}

}

### Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary:

创建优雅的长运行方法：

提供一个干净的关闭servlet策略的最后一步是使得每个正在运行的方法行为优雅。运行了很长时间的方法应该检查是否关闭字段即isShutdown。如果需要的话直接结束掉当前正在运行的任务。

public void doPost(...) {

...

for(i = 0; ((i < lotsOfStuffToDo) && //循环表明了这是一个长运行的方法，然后定期的请求，看是否要关闭servlet以确定下一步的操作

!isShuttingDown()); i++) {

try {

partOfLongRunningOperation(i);

} catch (InterruptedException e) {

...

}

}

}

## Uploading Files with Java Servlet Technology

Supporting file uploads is a very basic and common requirement for many web applications. In prior versions of the Servlet specification, implementing file upload required the use of external libraries or complex input processing. The Java Servlet specification now helps to provide a viable solution to the problem in a generic and portable way. Java Servlet technology now supports file upload out of the box, so any web container that implements the specification can parse multipart requests and make mime attachments available through the HttpServletRequest object.

A new annotation, javax.servlet.annotation.MultipartConfig, is used to indicate that the servlet on which it is declared expects requests to be made using the multipart/form-data MIME type. Servlets that are annotated with @MultipartConfig can retrieve the Part components of a given multipart/form-data request by calling the request.getPart(String name) or request.getParts() method.

### The @MultipartConfig Annotation

The @MultipartConfig annotation supports the following optional attributes.

* location: An absolute path to a directory on the file system. The location attribute does not support a path relative to the application context. This location is used to store files temporarily while the parts are processed or when the size of the file exceeds the specified fileSizeThresholdsetting. The default location is "".
* fileSizeThreshold: The file size in bytes after which the file will be temporarily stored on disk. The default size is 0 bytes.
* MaxFileSize: The maximum size allowed for uploaded files, in bytes. If the size of any uploaded file is greater than this size, the web container will throw an exception (IllegalStateException). The default size is unlimited.
* maxRequestSize: The maximum size allowed for a multipart/form-data request, in bytes. The web container will throw an exception if the overall size of all uploaded files exceeds this threshold. The default size is unlimited.

For, example, the @MultipartConfig annotation could be constructed as follows:

@MultipartConfig(location="/tmp", fileSizeThreshold=1024\*1024,

maxFileSize=1024\*1024\*5, maxRequestSize=1024\*1024\*5\*5)

Instead of using the @MultipartConfig annotation to hard-code these attributes in your file upload servlet, you could add the following as a child element of the servlet configuration element in the web.xml file:

<multipart-config>

<location>/tmp</location>

<max-file-size>20848820</max-file-size>

<max-request-size>418018841</max-request-size>

<file-size-threshold>1048576</file-size-threshold>

</multipart-config>

### The getParts and getPart Methods

The Servlet specification supports two additional HttpServletRequest methods:

* Collection<Part> getParts()
* Part getPart(String name)

The request.getParts() method returns collections of all Part objects. If you have more than one input of type file, multiple Part objects are returned. Because Part objects are named, the getPart(String name) method can be used to access a particular Part. Alternatively, the getParts() method, which returns an Iterable<Part>, can be used to get an Iterator over all the Part objects.

The javax.servlet.http.Part interface is a simple one, providing methods that allow introspection of each Part. The methods do the following:

* Retrieve the name, size, and content-type of the Part
* Query the headers submitted with a Part
* Delete a Part
* Write a Part out to disk

For example, the Part interface provides the write(String filename) method to write the file with the specified name. The file can then be saved in the directory that is specified with the location attribute of the @MultipartConfig annotation or, in the case of the fileupload example, in the location specified by the Destination field in the form.

## Asynchronous Processing

Web containers in application servers normally use a server thread per client request. Under heavy load conditions, containers need a large amount of threads to serve all the client requests. Scalability limitations include running out of memory or exhausting the pool of container threads. To create scalable web applications, you must ensure that no threads associated with a request are sitting idle, so the container can use them to process new requests.

There are two common scenarios in which a thread associated with a request can be sitting idle.

* The thread needs to wait for a resource to become available or process data before building the response. For example, an application may need to query a database or access data from a remote web service before generating the response.
* The thread needs to wait for an event before generating the response. For example, an application may have to wait for a JMS message, new information from another client, or new data available in a queue before generating the response.

These scenarios represent blocking operations that limit the scalability of web applications. Asynchronous processing refers to assigning these blocking operations to a new thread and retuning the thread associated with the request immediately to the container.

### Asynchronous Processing in Servlets

Java EE provides asynchronous processing support for servlets and filters. If a servlet or a filter reaches a potentially blocking operation when processing a request, it can assign the operation to an asynchronous execution context and return the thread associated with the request immediately to the container without generating a response. The blocking operation completes in the asynchronous execution context in a different thread, which can generate a response or dispatch the request to another servlet.

To enable asynchronous processing on a servlet, set the parameter asyncSupported to true on the @WebServlet annotation as follows:

@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet { ... }

The javax.servlet.AsyncContext class provides the functionality that you need to perform asynchronous processing inside service methods. To obtain an instance of AsyncContext, call the startAsync() method on the request object of your service method; for example:

public void doGet(HttpServletRequest req, HttpServletResponse resp) {

...

AsyncContext acontext = req.startAsync();

...

}

This call puts the request into asynchronous mode and ensures that the response is not committed after exiting the service method. You have to generate the response in the asynchronous context after the blocking operation completes or dispatch the request to another servlet.

[Table 18-3](https://javaee.github.io/tutorial/servlets012.html#BEICFIEC) describes the basic functionality provided by the AsyncContext class.

**Table 18-3 Functionality Provided by the AsyncContext Class**

|  |  |
| --- | --- |
| **Method Signature** | **Description** |
| void start(Runnable run) | The container provides a different thread in which the blocking operation can be processed.  You provide code for the blocking operation as a class that implements the Runnable interface. You can provide this class as an inner class when calling the start method or use another mechanism to pass the AsyncContext instance to your class. |
| ServletRequest getRequest() | Returns the request used to initialize this asynchronous context. In the example above, the request is the same as in the service method.  You can use this method inside the asynchronous context to obtain parameters from the request. |
| ServletResponse getResponse() | Returns the response used to initialize this asynchronous context. In the example above, the response is the same as in the service method.  You can use this method inside the asynchronous context to write to the response with the results of the blocking operation. |
| void complete() | Completes the asynchronous operation and closes the response associated with this asynchronous context.  You call this method after writing to the response object inside the asynchronous context. |
| void dispatch(String path) | Dispatches the request and response objects to the given path.  You use this method to have another servlet write to the response after the blocking operation completes. |

### Waiting for a Resource

This section demonstrates how to use the functionality provided by the AsyncContext class for the following use case:

1. A servlet receives a parameter from a GET request.
2. The servlet uses a resource, such as a database or a web service, to retrieve information based on the value of the parameter. The resource can be slow at times, so this may be a blocking operation.
3. The servlet generates a response using the result from the resource.

The following code shows a basic servlet that does not use asynchronous processing:

@WebServlet(urlPatterns={"/syncservlet"})

public class SyncServlet extends HttpServlet {

private MyRemoteResource resource;

@Override

public void init(ServletConfig config) {

resource = MyRemoteResource.create("config1=x,config2=y");

}

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

response.setContentType("text/html;charset=UTF-8");

String param = request.getParameter("param");

String result = resource.process(param);

/\* ... print to the response ... \*/

}

}

The following code shows the same servlet using asynchronous processing:

@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet {

/\* ... Same variables and init method as in SyncServlet ... \*/

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

response.setContentType("text/html;charset=UTF-8");

final AsyncContext acontext = request.startAsync();

acontext.start(new Runnable() {

public void run() {

String param = acontext.getRequest().getParameter("param");

String result = resource.process(param);

HttpServletResponse response = acontext.getResponse();

/\* ... print to the response ... \*/

acontext.complete();

}

}

AsyncServlet adds asyncSupported=true to the @WebServlet annotation. The rest of the differences are inside the service method.

* request.startAsync() causes the request to be processed asynchronously; the response is not sent to the client at the end of the service method.
* acontext.start(new Runnable() {…​}) gets a new thread from the container.
* The code inside the run() method of the inner class executes in the new thread. The inner class has access to the asynchronous context to read parameters from the request and write to the response. Calling the complete() method of the asynchronous context commits the response and sends it to the client.

The service method of AsyncServlet returns immediately, and the request is processed in the asynchronous context.

## Nonblocking I/O

Web containers in application servers normally use a server thread per client request. To develop scalable web applications, you must ensure that threads associated with client requests are never sitting idle waiting for a blocking operation to complete. [Asynchronous Processing](https://javaee.github.io/tutorial/servlets012.html#BEIGCFDF) provides a mechanism to execute application-specific blocking operations in a new thread, returning the thread associated with the request immediately to the container. Even if you use asynchronous processing for all the application-specific blocking operations inside your service methods, threads associated with client requests can be momentarily sitting idle because of input/output considerations.

For example, if a client is submitting a large HTTP POST request over a slow network connection, the server can read the request faster than the client can provide it. Using traditional I/O, the container thread associated with this request would be sometimes sitting idle waiting for the rest of the request.

Java EE provides nonblocking I/O support for servlets and filters when processing requests in asynchronous mode. The following steps summarize how to use nonblocking I/O to process requests and write responses inside service methods.

1. Put the request in asynchronous mode as described in [Asynchronous Processing](https://javaee.github.io/tutorial/servlets012.html#BEIGCFDF).
2. Obtain an input stream and/or an output stream from the request and response objects in the service method.
3. Assign a read listener to the input stream and/or a write listener to the output stream.
4. Process the request and the response inside the listener’s callback methods.

[Table 18-4](https://javaee.github.io/tutorial/servlets013.html#BEIFDICJ) and [Table 18-5](https://javaee.github.io/tutorial/servlets013.html#BEIFIIIH) describe the methods available in the servlet input and output streams for nonblocking I/O support. [Table 18-6](https://javaee.github.io/tutorial/servlets013.html#BEIFGJCG) describes the interfaces for read listeners and write listeners.

**Table 18-4 Nonblocking I/O Support in javax.servlet.ServletInputStream**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void setReadListener(ReadListener rl) | Associates this input stream with a listener object that contains callback methods to read data asynchronously. You provide the listener object as an anonymous class or use another mechanism to pass the input stream to the read listener object. |
| boolean isReady() | Returns true if data can be read without blocking. |
| boolean isFinished() | Returns true when all the data has been read. |

**Table 18-5 Nonblocking I/O Support in javax.servlet.ServletOutputStream**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void setWriteListener(WriteListener wl) | Associates this output stream with a listener object that contains callback methods to write data asynchronously. You provide the write listener object as an anonymous class or use another mechanism to pass the output stream to the write listener object. |
| boolean isReady() | Returns true if data can be written without blocking. |

**Table 18-6 Listener Interfaces for Nonblocking I/O Support**

|  |  |  |
| --- | --- | --- |
| **Interface** | **Methods** | **Description** |
| ReadListener | void onDataAvailable()  void onAllDataRead()  void onError(Throwable t) | A ServletInputStream instance calls these methods on its listener when there is data available to read, when all the data has been read, or when there is an error. |
| WriteListener | void onWritePossible()  void onError(Throwable t) | A ServletOutputStream instance calls these methods on its listener when it is possible to write data without blocking or when there is an error. |

### Reading a Large HTTP POST Request Using Nonblocking I/O

The code in this section shows how to read a large HTTP POST request inside a servlet by putting the request in asynchronous mode (as described in [Asynchronous Processing](https://javaee.github.io/tutorial/servlets012.html#BEIGCFDF)) and using the nonblocking I/O functionality from [Table 18-4](https://javaee.github.io/tutorial/servlets013.html#BEIFDICJ) and [Table 18-6](https://javaee.github.io/tutorial/servlets013.html#BEIFGJCG).

@WebServlet(urlPatterns={"/asyncioservlet"}, asyncSupported=true)

public class AsyncIOServlet extends HttpServlet {

@Override

public void doPost(HttpServletRequest request,

HttpServletResponse response)

throws IOException {

final AsyncContext acontext = request.startAsync();

final ServletInputStream input = request.getInputStream();

input.setReadListener(new ReadListener() {

byte buffer[] = new byte[4\*1024];

StringBuilder sbuilder = new StringBuilder();

@Override

public void onDataAvailable() {

try {

do {

int length = input.read(buffer);

sbuilder.append(new String(buffer, 0, length));

} while(input.isReady());

} catch (IOException ex) { ... }

}

@Override

public void onAllDataRead() {

try {

acontext.getResponse().getWriter()

.write("...the response...");

} catch (IOException ex) { ... }

acontext.complete();

}

@Override

public void onError(Throwable t) { ... }

});

}

}

This example declares the web servlet with asynchronous support using the @WebServlet annotation parameter asyncSupported=true. The service method first puts the request in asynchronous mode by calling the startAsync() method of the request object, which is required in order to use nonblocking I/O. Then, the service method obtains an input stream associated with the request and assigns a read listener defined as an inner class. The listener reads parts of the request as they become available and then writes some response to the client when it finishes reading the request.

## Protocol Upgrade Processing

In HTTP/1.1, clients can request to switch to a different protocol on the current connection by using the Upgrade header field. If the server accepts the request to switch to the protocol indicated by the client, it generates an HTTP response with status 101 (switching protocols). After this exchange, the client and the server communicate using the new protocol.

For example, a client can make an HTTP request to switch to the XYZP protocol as follows:

GET /xyzpresource HTTP/1.1

Host: localhost:8080

Accept: text/html

Upgrade: XYZP

Connection: Upgrade

OtherHeaderA: Value

The client can specify parameters for the new protocol using HTTP headers. The server can accept the request and generate a response as follows:

HTTP/1.1 101 Switching Protocols

Upgrade: XYZP

Connection: Upgrade

OtherHeaderB: Value

(XYZP data)

Java EE supports the HTTP protocol upgrade functionality in servlets, as described in [Table 18-7](https://javaee.github.io/tutorial/servlets014.html#BEIBDHAG).

**Table 18-7 Protocol Upgrade Support**

|  |  |
| --- | --- |
| **Class or Interface** | **Method** |
| HttpServletRequest | HttpUpgradeHandler upgrade(Class handler)  The upgrade method starts the protocol upgrade processing. This method instantiates a class that implements the HttpUpgradeHandler interface and delegates the connection to it.  You call the upgrade method inside a service method when accepting a request from a client to switch protocols. |
| HttpUpgradeHandler | void init(WebConnection wc)  The init method is called when the servlet accepts the request to switch protocols. You implement this method and obtain input and output streams from the WebConnectionobject to implement the new protocol. |
| HttpUpgradeHandler | void destroy()  The destroy method is called when the client disconnects. You implement this method and free any resources associated with processing the new protocol. |
| WebConnection | ServletInputStream getInputStream()  The getInputStream method provides access to the input stream of the connection. You can use [Nonblocking I/O](https://javaee.github.io/tutorial/servlets013.html#BEIHICDH) with the returned stream to implement the new protocol. |
| WebConnection | ServletOutputStream getOutputStream()  The getOutputStream method provides access to the output stream of the connection. You can use [Nonblocking I/O](https://javaee.github.io/tutorial/servlets013.html#BEIHICDH) with the returned stream to implement the new protocol. |

The following code demonstrates how to accept an HTTP protocol upgrade request from a client:

@WebServlet(urlPatterns={"/xyzpresource"})

public class XYZPUpgradeServlet extends HttpServlet {

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

if ("XYZP".equals(request.getHeader("Upgrade"))) {

/\* Accept upgrade request \*/

response.setStatus(101);

response.setHeader("Upgrade", "XYZP");

response.setHeader("Connection", "Upgrade");

response.setHeader("OtherHeaderB", "Value");

/\* Delegate the connection to the upgrade handler \*/

XYZPUpgradeHandler = request.upgrade(XYZPUpgradeHandler.class);

/\* (the service method returns immedately) \*/

} else {

/\* ... write error response ... \*/

}

}

}

The XYZPUpgradeHandler class handles the connection:

public class XYZPUpgradeHandler implements HttpUpgradeHandler {

@Override

public void init(WebConnection wc) {

ServletInputStream input = wc.getInputStream();

ServletOutputStream output = wc.getOutputStream();

/\* ... implement XYZP using these streams (protocol-specific) ... \*/

}

@Override

public void destroy() { ... }

}

The class that implements HttpUpgradeHandler uses the streams from the current connection to communicate with the client using the new protocol. See the Servlet 4.0 specification at <http://jcp.org/en/jsr/detail?id=369> for details on HTTP protocol upgrade support.

## Server Push

Server push is the ability of the server to anticipate what will be needed by the client in advance of the client’s request. It lets the server pre-populate the browser’s cache in advance of the browser asking for the resource to put in the cache.

Server push is the most visible of the improvements in HTTP/2 to appear in the servlet API. All of the new features in HTTP/2, including server push, are aimed at improving the performance of the web browsing experience.

Server push derives its contribution to improved browser performance from the fact that servers know what additional assets (such as images, stylesheets, and scripts) go along with initial requests. For example, servers might know that whenever a browser requests index.html, it will shortly thereafter request header.gif, footer.gif, and style.css. Servers can preemptively start sending the bytes of these assets along with the bytes of the index.html.

To use server push, obtain a reference to a PushBuilder from an HttpServletRequest, edit the builder as desired, then call push(). See the [javadoc](https://javaee.github.io/javaee-spec/) for the class javax.servlet.http.PushBuilder and the method javax.servlet.http.HttpServletRequest.newPushBuilder().

To view the GlassFish samples code for this feature, see <https://github.com/javaee/glassfish-samples/tree/master/ws/javaee8>.

## HTTP Trailer

HTTP trailer is a collection of a special type of HTTP headers that comes after the response body. The trailer response header allows the sender to include additional fields at the end of chunked messages in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status.

If trailer headers are ready for reading, isTrailerFieldsReady() will return true. Then a servlet can read trailer headers of the HTTP request using the getTrailerFields method of the HttpServletRequest interface. If trailer headers are not ready for reading, isTrailerFieldsReady()returns false and will cause an IllegalStateException.

A servlet can write trailer headers to the response by providing a supplier to the setTrailerFields() method of the HttpServletResponse interface. The following headers and types of headers must not be included in the set of keys in the map passed to setTrailerFields(): Transfer-Encoding, Content-Length, Host, controls and conditional headers, authentication headers, Content-Encoding, Content-Type, Content-Range, and Trailer. When sending response trailers, you must include a regular header, called Trailer, whose value is a comma-separated list of all the keys in the map that is supplied to the setTrailerFields() method. The value of the Trailer header lets the client know what trailers to expect.

The supplier of the trailer headers can be obtained by accessing the getTrailerFields() method of the HttpServletResponse interface.

See the [javadoc](https://javaee.github.io/javaee-spec/) for getTrailerFields() and isTrailerFieldsReady() in HttpServletRequest, and getTrailerFields() and setTrailerFields() in HttpServletResponse.

## The mood Example Application

The mood example application, located in the tut-install`/examples/web/servlet/mood/` directory, is a simple example that displays Duke’s moods at different times during the day. The example shows how to develop a simple application by using the @WebServlet, @WebFilter, and @WebListenerannotations to create a servlet, a listener, and a filter.

The following topics are addressed here:

* [Components of the mood Example Application](https://javaee.github.io/tutorial/servlets015.html#CHDEBFCB)
* [Running the mood Example](https://javaee.github.io/tutorial/servlets015.html#GKCOJ)

### Components of the mood Example Application

The mood example application is comprised of three components: mood.web.MoodServlet, mood.web.TimeOfDayFilter, and mood.web.SimpleServletListener.

MoodServlet, the presentation layer of the application, displays Duke’s mood in a graphic, based on the time of day. The @WebServlet annotation specifies the URL pattern:

@WebServlet("/report")

public class MoodServlet extends HttpServlet {

...

TimeOfDayFilter sets an initialization parameter indicating that Duke is awake:

@WebFilter(filterName = "TimeOfDayFilter",

urlPatterns = {"/\*"},

initParams = {

@WebInitParam(name = "mood", value = "awake")})

public class TimeOfDayFilter implements Filter {

...

The filter calls the doFilter method, which contains a switch statement that sets Duke’s mood based on the current time.

SimpleServletListener logs changes in the servlet’s lifecycle. The log entries appear in the server log.

### Running the mood Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the mood example.

The following topics are addressed here:

* [To Run the mood Example Using NetBeans IDE](https://javaee.github.io/tutorial/servlets015.html#GKCOB)
* [To Run the mood Example Using Maven](https://javaee.github.io/tutorial/servlets015.html#GKCPJ)

#### To Run the mood Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

tut-install/examples/web/servlet

1. Select the mood folder.
2. Click Open Project.
3. In the Projects tab, right-click the mood project and select Build.
4. In a web browser, enter the following URL:

http://localhost:8080/mood/report

The URL specifies the context root, followed by the URL pattern.

A web page appears with the title "Servlet MoodServlet at /mood", a text string describing Duke’s mood, and an illustrative graphic.

#### To Run the mood Example Using Maven

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. In a terminal window, go to:

tut-install/examples/web/servlet/mood/

1. Enter the following command to deploy the application:

mvn install

1. In a web browser, enter the following URL:

http://localhost:8080/mood/report

The URL specifies the context root, followed by the URL pattern.

A web page appears with the title "Servlet MoodServlet at /mood", a text string describing Duke’s mood, and an illustrative graphic.

## The fileupload Example Application

The fileupload example, located in the tut-install`/examples/web/servlet/fileupload/` directory, illustrates how to implement and use the file upload feature.

The Duke’s Forest case study provides a more complex example that uploads an image file and stores its content in a database.

|  |
| --- |
| **Note**:  Except where expressly provided otherwise, the site, and all content provided on or through the site, are provided on an "as is" and "as available" basis. Oracle expressly disclaims all warranties of any kind, whether express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose and non-infringement with respect to the site and all content provided on or through the site. Oracle makes no warranty that: (a) the site or content will meet your requirements; (b) the site will be available on an uninterrupted, timely, secure, or error-free basis; (c) the results that may be obtained from the use of the site or any content provided on or through the site will be accurate or reliable; or (d) the quality of any content purchased or obtained by you on or through the site will meet your expectations.  Any content accessed, downloaded or otherwise obtained on or through the use of the site is used at your own discretion and risk. Oracle shall have no responsibility for any damage to your computer system or loss of data that results from the download or use of content. |

The following topics are addressed here:

* [Architecture of the fileupload Example Application](https://javaee.github.io/tutorial/servlets016.html#CHDFGBGI)
* [Running the fileupload Example](https://javaee.github.io/tutorial/servlets016.html#CHDIHJCI)

### Architecture of the fileupload Example Application

The fileupload example application consists of a single servlet and an HTML form that makes a file upload request to the servlet.

This example includes a very simple HTML form with two fields, File and Destination. The input type, file, enables a user to browse the local file system to select the file. When the file is selected, it is sent to the server as a part of a POST request. During this process, two mandatory restrictions are applied to the form with input type file.

* The enctype attribute must be set to a value of multipart/form-data.
* Its method must be POST.

When the form is specified in this manner, the entire request is sent to the server in encoded form. The servlet then uses its own means to handle the request to process the incoming file data and extract a file from the stream. The destination is the path to the location where the file will be saved on your computer. Pressing the Upload button at the bottom of the form posts the data to the servlet, which saves the file in the specified destination.

The HTML form in index.html is as follows:

<!DOCTYPE html>

<html lang="en">

<head>

<title>File Upload</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

</head>

<body>

<form method="POST" action="upload" enctype="multipart/form-data" >

File:

<input type="file" name="file" id="file" /> <br/>

Destination:

<input type="text" value="/tmp" name="destination"/>

</br>

<input type="submit" value="Upload" name="upload" id="upload" />

</form>

</body>

</html>

A POST request method is used when the client needs to send data to the server as part of the request, such as when uploading a file or submitting a completed form. In contrast, a GET request method sends a URL and headers only to the server, whereas POST requests also include a message body. This allows arbitrary length data of any type to be sent to the server. A header field in the POST request usually indicates the message body’s Internet media type.

When submitting a form, the browser streams the content in, combining all parts, with each part representing a field of a form. Parts are named after the input elements and are separated from each other with string delimiters named boundary.

This is what submitted data from the fileupload form looks like, after selecting sample.txt as the file that will be uploaded to the tmp directory on the local file system:

POST /fileupload/upload HTTP/1.1

Host: localhost:8080

Content-Type: multipart/form-data;

boundary=---------------------------263081694432439 Content-Length: 441

-----------------------------263081694432439

Content-Disposition: form-data; name="file"; filename="sample.txt"

Content-Type: text/plain

Data from sample file

-----------------------------263081694432439

Content-Disposition: form-data; name="destination"

/tmp

-----------------------------263081694432439

Content-Disposition: form-data; name="upload"

Upload

-----------------------------263081694432439--

The servlet FileUploadServlet.java begins as follows:

@WebServlet(name = "FileUploadServlet", urlPatterns = {"/upload"})

@MultipartConfig

public class FileUploadServlet extends HttpServlet {

private final static Logger LOGGER =

Logger.getLogger(FileUploadServlet.class.getCanonicalName());

The @WebServlet annotation uses the urlPatterns property to define servlet mappings.

The @MultipartConfig annotation indicates that the servlet expects requests to be made using the multipart/form-data MIME type.

The processRequest method retrieves the destination and file part from the request, then calls the getFileName method to retrieve the file name from the file part. The method then creates a FileOutputStream and copies the file to the specified destination. The error-handling section of the method catches and handles some of the most common reasons why a file would not be found. The processRequest and getFileName methods look like this:

protected void processRequest(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

// Create path components to save the file

final String path = request.getParameter("destination");

final Part filePart = request.getPart("file");

final String fileName = getFileName(filePart);

OutputStream out = null;

InputStream filecontent = null;

final PrintWriter writer = response.getWriter();

try {

out = new FileOutputStream(new File(path + File.separator

+ fileName));

filecontent = filePart.getInputStream();

int read = 0;

final byte[] bytes = new byte[1024];

while ((read = filecontent.read(bytes)) != -1) {

out.write(bytes, 0, read);

}

writer.println("New file " + fileName + " created at " + path);

LOGGER.log(Level.INFO, "File{0}being uploaded to {1}",

new Object[]{fileName, path});

} catch (FileNotFoundException fne) {

writer.println("You either did not specify a file to upload or are "

+ "trying to upload a file to a protected or nonexistent "

+ "location.");

writer.println("<br/> ERROR: " + fne.getMessage());

LOGGER.log(Level.SEVERE, "Problems during file upload. Error: {0}",

new Object[]{fne.getMessage()});

} finally {

if (out != null) {

out.close();

}

if (filecontent != null) {

filecontent.close();

}

if (writer != null) {

writer.close();

}

}

}

private String getFileName(final Part part) {

final String partHeader = part.getHeader("content-disposition");

LOGGER.log(Level.INFO, "Part Header = {0}", partHeader);

for (String content : part.getHeader("content-disposition").split(";")) {

if (content.trim().startsWith("filename")) {

return content.substring(

content.indexOf('=') + 1).trim().replace("\"", "");

}

}

return null;

}

### Running the fileupload Example

You can use either NetBeans IDE or Maven to build, package, deploy, and run the fileupload example.

The following topics are addressed here:

* [To Build, Package, and Deploy the fileupload Example Using NetBeans IDE](https://javaee.github.io/tutorial/servlets016.html#CHDGDJCI)
* [To Build, Package, and Deploy the fileupload Example Using Maven](https://javaee.github.io/tutorial/servlets016.html#CHDCFADG)
* [To Run the fileupload Example](https://javaee.github.io/tutorial/servlets016.html#CHDDDAAJ)

#### To Build, Package, and Deploy the fileupload Example Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

tut-install/examples/web/servlet

1. Select the fileupload folder.
2. Click Open Project.
3. In the Projects tab, right-click the fileupload project and select Build.

#### To Build, Package, and Deploy the fileupload Example Using Maven

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. In a terminal window, go to:

tut-install/examples/web/servlet/fileupload/

1. Enter the following command to deploy the application:

mvn install

#### To Run the fileupload Example

1. In a web browser, enter the following URL:

http://localhost:8080/fileupload/

1. On the File Upload page, click Choose File to display a file browser window.
2. Select a file to upload and click Open.

The name of the file you selected is displayed in the File field. If you do not select a file, an exception will be thrown.

1. In the Destination field, type a directory name.

The directory must have already been created and must also be writable. If you do not enter a directory name or if you enter the name of a nonexistent or protected directory, an exception will be thrown.

1. Click Upload to upload the file that you selected to the directory that you specified in the Destination field.

A message reports that the file was created in the directory that you specified.

1. Go to the directory that you specified in the Destination field and verify that the uploaded file is present.

## The dukeetf Example Application

The dukeetf example application, located in the tut-install`/examples/web/dukeetf/` directory, demonstrates how to use asynchronous processing in a servlet to provide data updates to web clients. The example resembles a service that provides periodic updates on the price and trading volume of an electronically traded fund (ETF).

The following topics are addressed here:

* [Architecture of the dukeetf Example Application](https://javaee.github.io/tutorial/servlets017.html#CHDBBEDA)
* [Running the dukeetf Example Application](https://javaee.github.io/tutorial/servlets017.html#CHDHBBBI)

### Architecture of the dukeetf Example Application

The dukeetf example application consists of a servlet, an enterprise bean, and an HTML page.

* The servlet puts requests in asynchronous mode, stores them in a queue, and writes the responses when new data for price and trading volume becomes available.
* The enterprise bean updates the price and volume information once every second.
* The HTML page uses JavaScript code to make requests to the servlet for new data, parse the response from the servlet, and update the price and volume information without reloading the page.

The dukeetf example application uses a programming model known as long polling. In the traditional HTTP request and response model, the user must make an explicit request (such as clicking a link or submitting a form) to get any new information from the server, and the page has to be reloaded. Long polling provides a mechanism for web applications to push updates to clients using HTTP without the user making an explicit request. The server handles connections asynchronously, and the client uses JavaScript to make new connections. In this model, clients make a new request immediately after receiving new data, and the server keeps the connection open until new data becomes available.

#### The Servlet

The DukeETFServlet class uses asynchronous processing:

@WebServlet(urlPatterns={"/dukeetf"}, asyncSupported=true)

public class DukeETFServlet extends HttpServlet {

...

}

In the following code, the init method initializes a queue to hold client requests and registers the servlet with the enterprise bean that provides the price and volume updates. The send method gets called once per second by the PriceVolumeBean to send updates and close the connection:

@Override

public void init(ServletConfig config) {

/\* Queue for requests \*/

requestQueue = new ConcurrentLinkedQueue<>();

/\* Register with the enterprise bean that provides price/volume updates \*/

pvbean.registerServlet(this);

}

/\* PriceVolumeBean calls this method every second to send updates \*/

public void send(double price, int volume) {

/\* Send update to all connected clients \*/

for (AsyncContext acontext : requestQueue) {

try {

String msg = String.format("%.2f / %d", price, volume);

PrintWriter writer = acontext.getResponse().getWriter();

writer.write(msg);

logger.log(Level.INFO, "Sent: {0}", msg);

/\* Close the connection

\* The client (JavaScript) makes a new one instantly \*/

acontext.complete();

} catch (IOException ex) {

logger.log(Level.INFO, ex.toString());

}

}

}

The service method puts client requests in asynchronous mode and adds a listener to each request. The listener is implemented as an anonymous class that removes the request from the queue when the servlet finishes writing a response or when there is an error. Finally, the service method adds the request to the request queue created in the init method. The service method is the following:

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

response.setContentType("text/html");

/\* Put request in async mode \*/

final AsyncContext acontext = request.startAsync();

/\* Remove from the queue when done \*/

acontext.addListener(new AsyncListener() {

public void onComplete(AsyncEvent ae) throws IOException {

requestQueue.remove(acontext);

}

public void onTimeout(AsyncEvent ae) throws IOException {

requestQueue.remove(acontext);

}

public void onError(AsyncEvent ae) throws IOException {

requestQueue.remove(acontext);

}

public void onStartAsync(AsyncEvent ae) throws IOException {}

});

/\* Add to the queue \*/

requestQueue.add(acontext);

}

#### The Enterprise Bean

The PriceVolumeBean class is an enterprise bean that uses the timer service from the container to update the price and volume information and call the servlet’s send method once every second:

@Startup

@Singleton

public class PriceVolumeBean {

/\* Use the container's timer service \*/

@Resource TimerService tservice;

private DukeETFServlet servlet;

...

@PostConstruct

public void init() {

/\* Initialize the EJB and create a timer \*/

random = new Random();

servlet = null;

tservice.createIntervalTimer(1000, 1000, new TimerConfig());

}

public void registerServlet(DukeETFServlet servlet) {

/\* Associate a servlet to send updates to \*/

this.servlet = servlet;

}

@Timeout

public void timeout() {

/\* Adjust price and volume and send updates \*/

price += 1.0\*(random.nextInt(100)-50)/100.0;

volume += random.nextInt(5000) - 2500;

if (servlet != null)

servlet.send(price, volume);

}

}

See [Using the Timer Service](https://javaee.github.io/tutorial/ejb-basicexamples005.html#BNBOY) in [Chapter 37, "Running the Enterprise Bean Examples"](https://javaee.github.io/tutorial/ejb-basicexamples.html#GIJRB) for more information on the timer service.

#### The HTML Page

The HTML page consists of a table and some JavaScript code. The table contains two fields referenced from JavaScript code:

<html xmlns="http://www.w3.org/1999/xhtml">

<head>...</head>

<body onload="makeAjaxRequest();">

...

<table>

...

<td id="price">--.--</td>

...

<td id="volume">--</td>

...

</table>

</body>

</html>

The JavaScript code uses the XMLHttpRequest API, which provides functionality for transferring data between a client and a server. The script makes an asynchronous request to the servlet and designates a callback method. When the server provides a response, the callback method updates the fields in the table and makes a new request. The JavaScript code is the following:

var ajaxRequest;

function updatePage() {

if (ajaxRequest.readyState === 4) {

var arraypv = ajaxRequest.responseText.split("/");

document.getElementById("price").innerHTML = arraypv[0];

document.getElementById("volume").innerHTML = arraypv[1];

makeAjaxRequest();

}

}

function makeAjaxRequest() {

ajaxRequest = new XMLHttpRequest();

ajaxRequest.onreadystatechange = updatePage;

ajaxRequest.open("GET", "http://localhost:8080/dukeetf/dukeetf",

true);

ajaxRequest.send(null);

}

The XMLHttpRequest API is supported by most modern browsers, and it is widely used in Ajax web client development (Asynchronous JavaScript and XML).

See [The dukeetf2 Example Application](https://javaee.github.io/tutorial/websocket011.html#BABGCEHE) in [Chapter 19, "Java API for WebSocket"](https://javaee.github.io/tutorial/websocket.html#GKJIQ5) for an equivalent version of this example implemented using a WebSocket endpoint.

### Running the dukeetf Example Application

This section describes how to run the dukeetf example application using NetBeans IDE and from the command line.

The following topics are addressed here:

* [To Run the dukeetf Example Application Using NetBeans IDE](https://javaee.github.io/tutorial/servlets017.html#CHDCGCJD)
* [To Run the dukeetf Example Application Using Maven](https://javaee.github.io/tutorial/servlets017.html#CHDHHAFG)

#### To Run the dukeetf Example Application Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

tut-install/examples/web/servlet

1. Select the dukeetf folder.
2. Click Open Project.
3. In the Projects tab, right-click the dukeetf project and select Run.

This command builds and packages the application into a WAR file (dukeetf.war) located in the target directory, deploys it to the server, and launches a web browser window with the following URL:

http://localhost:8080/dukeetf/

Open the same URL in a different web browser to see how both pages get price and volume updates simultaneously.

#### To Run the dukeetf Example Application Using Maven

1. Make sure that GlassFish Server has been started (see [Starting and Stopping GlassFish Server](https://javaee.github.io/tutorial/usingexamples002.html#BNADI)).
2. In a terminal window, go to:

tut-install/examples/web/servlet/dukeetf/

1. Enter the following command to deploy the application:

mvn install

1. Open a web browser window and type the following address:

http://localhost:8080/dukeetf/

Open the same URL in a different web browser to see how both pages get price and volume updates simultaneously.

## Further Information about Java Servlet Technology

For more information on Java Servlet technology, see the Java Servlet 4.0 specification at <http://jcp.org/en/jsr/detail?id=369>.

For additional samples, see the GlassFish samples at <https://github.com/javaee/glassfish-samples/tree/master/ws/javaee8>.