

# Writing a Jetty Handler

## Writing a Jetty Handler

Many users of Jetty will not ever need to write a Jetty Handler, but instead will simply use the Servlet API. The existing jetty handlers for context, security, sessions and servlets can be reused without the need for extension.

However, some users may have special requirements or footprint concerns that prohibit the use of the full servlet API. For them implementing a Jetty handler is a straight forward way to provide dynamic web content with a minimum of fuss.

See also the [Architecture](#) page to understand more about Handlers vs Servlets.

## The Handler API

The [org.mortbay.jetty.Handler](#) interface provides Jetty's core of content generation or manipulation. Classes that implement this interface are used to coordinate requests, filter requests and generate content.

The core API of the Handler interface is

```
public void handle(String target,  
HttpServletRequest request,  
HttpServletResponse response, int dispatch)  
throws IOException, ServletException;
```

## The target

The target of a handler is an identifier for the resource that should handle the passed request. This is normally the URI that is parsed from a HTTP Request. However, in two key circumstances the target may differ from the URI of the passed request:

1. If the request has been dispatched to a named resource, such as a named servlet, then the target is the name of that resource.
2. If the request is being made by a call to [RequestDispatcher.include\(...\)](#) then the target is the URI of the included resource and will be different to the URI of the actual request.

## The Request and Response

The request and response objects used in the signature of the handle method are [HttpServletRequest](#) and [HttpServletResponse](#). These are the standard APIs and are moderately restricted in what can be done to the request and response. More often than not, access to the jetty implementations of these classes is required: [Request](#) and [Response](#). However, as the request and response may be wrapped by handlers, filters and servlets, it is not possible to pass the implementation directly. The following mantra will retrieve the core implementation objects from under any wrappers:

```
Request base_request = request instanceof
Request?(Request)request:HttpConnection.getC
urrentConnection().getRequest();
Response base_response = response instanceof
Response?(Response)request:HttpConnection.ge
tCurrentConnection().getResponse();
```

Note that if the handler passes the request on to another handler, it should use the request/response objects passed in and not the base objects. This is to preserve any wrapping done by upstream handlers.

## The dispatch

The dispatch argument indicates the state of the handling of the call and may be:

- REQUEST == 1 - An original request received from a connector.
- FORWARD == 2 - A request being forwarded by a RequestDispatcher
- INCLUDE == 4 - A request being included by a RequestDispatcher
- ERROR == 8 - A request being forwarded to a error handler by the container.

These mostly have significance for servlet and related handlers. For example, the security handler only applies authentication and authorization to REQUEST dispatches.

## Handling Requests

A Handler may handle a request by:

## Generating a Response

The [OneHandler](#) embedded example shows how a simple handler may generate a response.

The normal servlet response API may be used and will typically set some status, content headers and then write out the content:

```
response.setContentType("text/html");
response.setStatus(HttpServletResponse.SC_OK
);
response.getWriter().println("<h1>Hello
OneHandler</h1>");
```

It is also very important that a handler indicate that it has completed handling the request and that the request should not be passed to other handlers:

```
Request base_request = (request instanceof
Request) ?
(Request)request:HttpConnection.getCurrentCo
nnection().getRequest();
base_request.setHandled(true);
```

## Filtering the Request and/or Response

Once the base request or response object is obtained, it may be modified. Typically modifications are done to achieve:

- breaking the URI into contextPath, servletPath and pathInfo components
- to associate a resource base with a request for static content.
- to associate a session with a request
- to associate a security principal with a request
- to change the URI and paths during a request dispatch forward to another resource.

The context of the request may also be updated:

- Setting of the current threads context classloader
- Setting thread locals to identify the current ServletContext.

Typically a modified request is passed to another handler and then the modifications are undone in a finally block afterwards:

```
try
{
    base_request.setSession(a_session);

    next_handler.handle(target,request,response,
dispatch);
}
finally
{
    base_request.setSession(old_session);
}
```

The classes that implement the [HandlerWrapper](#) class are typically handler filters of this style.

## Passing the Request and Response to another handler

A handler may simply inspect the request and use the target, request URI or other information to select another handler to pass the request to. These handlers typically implement the [HandlerContainer](#) interface.

Examples include:

- [HandlerCollection](#) - A collection of handlers, where each handler is called regardless of the state of the request. This is typically used to pass a request to a [ContextHandlerCollection](#) and then the a [RequestLogHandler](#)
- [HandlerList](#) - A list of handlers which are called in turn until the request state is set as handled.
- [ContextHandlerCollection](#) - collection of Handlers of which one is selected by best match for the context path.

## Examples

See [org.mortbay.jetty.handler](#) package for some examples.