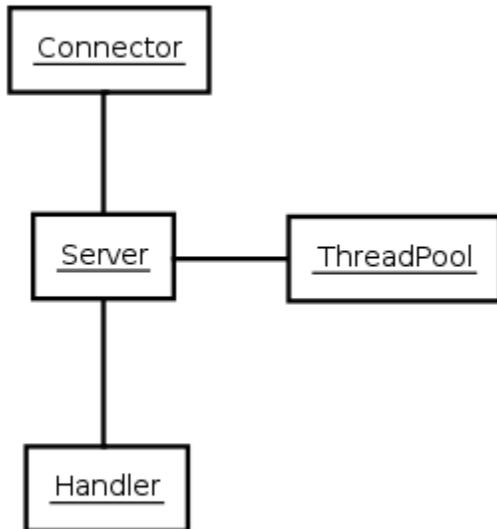


Architecture

Jetty 6 Architecture

View from 20,000 feet



The Jetty [Server](#) is the plumbing between a collection of [Connectors](#) that accept [HTTP connections](#), and a collection of [Handlers](#) that service [requests](#) from the connections and produce [responses](#), with the work being done by threads taken from a [thread pool](#).

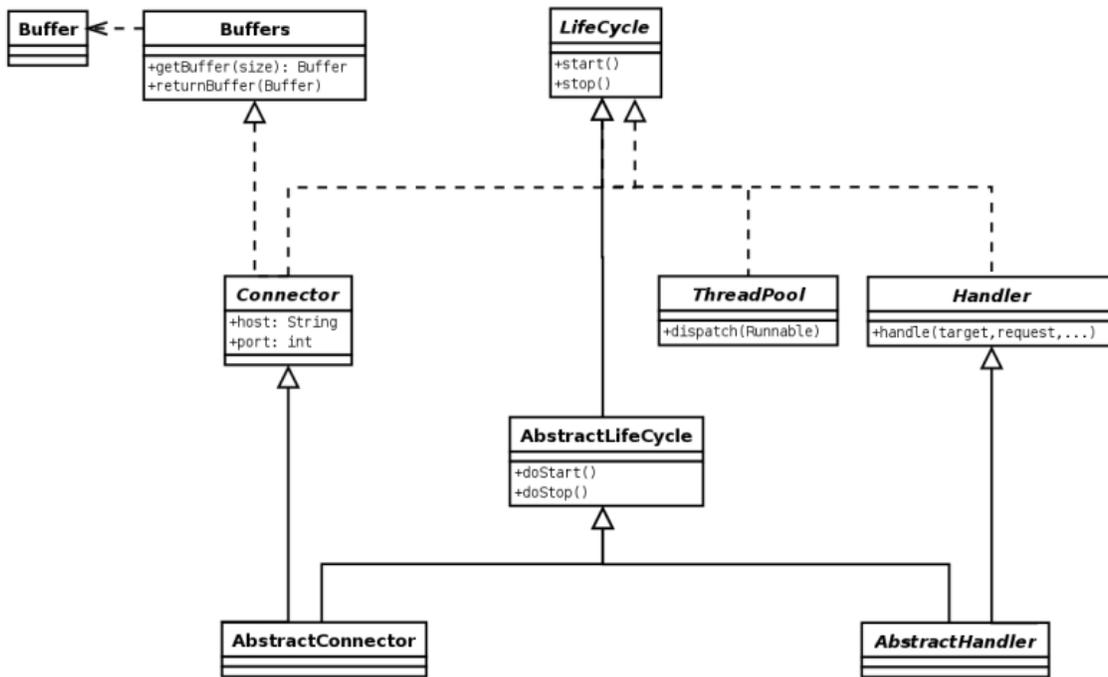
i While the jetty request/responses are derived from the [Servlet API](#), the full features of the servlet API are only available if the appropriate handlers are configured. For example, the session API on the request is inactive unless the request has been passed to a [Session Handler](#). The concept of a Servlet itself is implemented by a [Servlet Handler](#). If servlets are not required, there is very little overhead in the use of the servlet request/response APIs

Thus a Jetty server may be built using simply connectors and handlers, but without using Servlets.

The job of configuring jetty is the job of building a network of connectors and handlers and providing their individual configurations. As Jetty components are simply Plain Old Java Objects (POJOs) this assembly and configuration of components can be done by a variety of techniques:

- In code. See the examples in the [org.mortbay.jetty.example](#) package.
- With [jetty.xml](#) - dependency injection style XML format.
- With your dependency injection framework of choice: [Spring](#) or [XBean](#)
- Deployers: [WebAppDeployer](#), [ContextDeployer](#)

Patterns

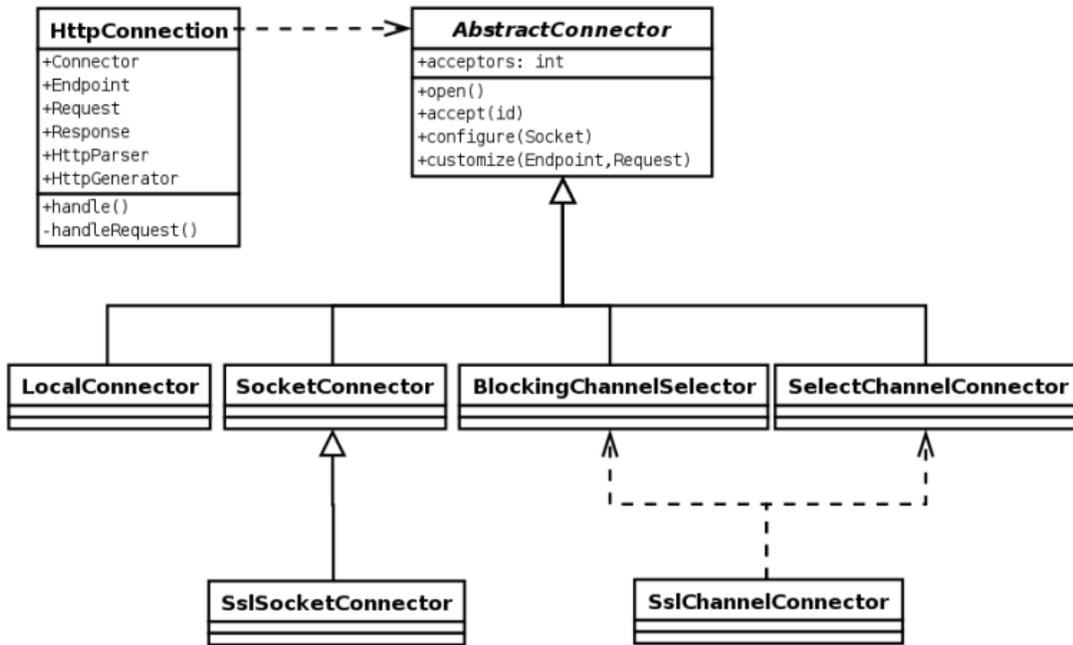


The implementation of Jetty follows some fairly standard patterns. Most abstract concepts such as Connector, Handler and Buffer are captured by interfaces. Generic handling for those interfaces is then provided in an Abstract implementation such as [AbstractConnector](#), [AbstractHandler](#) and [AbstractBuffer](#).

The JSR77 inspired life cycle of most jetty components is represented by [LifeCycle](#) interface and the [AbstractLifeCycle](#) implementation used as the base of many Jetty components.

Jetty provides its own [IO Buffering](#) abstract over String, byte arrays and NIO buffers. This allows for greater portability of Jetty as well as hiding some of the complexity of the NIO layer and its advanced features.

Connectors

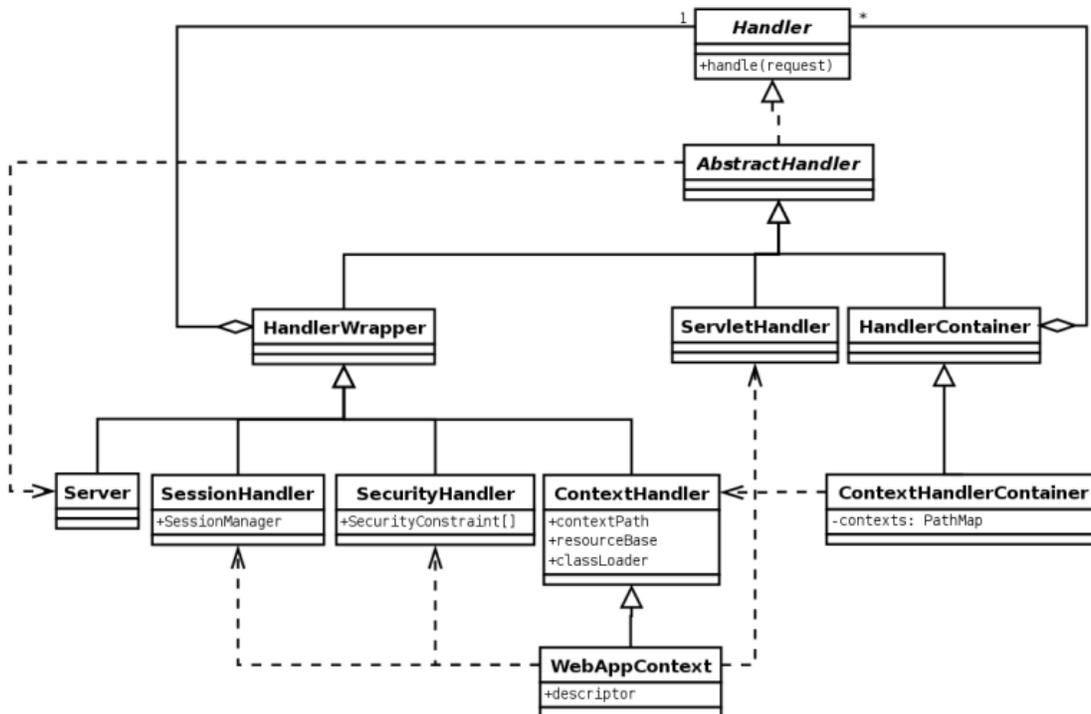


i This diagram is a little out of date, as a [Connection](#) interface has been extracted out of [HttpConnector](#) to allow support for the [AJP protocol](#)

The connectors represent the protocol handlers that accept connections, parse requests and generate responses. The different types of connectors available are based on the protocols, scheduling model and IO APIs used:

- [SocketConnector](#) - for few busy connections or when NIO is not available.
- [BlockingChannelConnector](#) - for few busy connections when NIO is available
- [SelectChannelConnector](#) - for many mostly idle connections or asynchronous handling of Ajax requests.
- [SslSocketConnector](#) - SSL without NIO
- [SslSelectChannelConnector](#) - SSL with non blocking NIO support.
- [AJPConnector](#) AJP protocol support for connections from apache mod_jk or mod_proxy_ajp

Handlers



The [Handler](#) is the component that deals with received requests. The core API of a handler is the handle method:

```

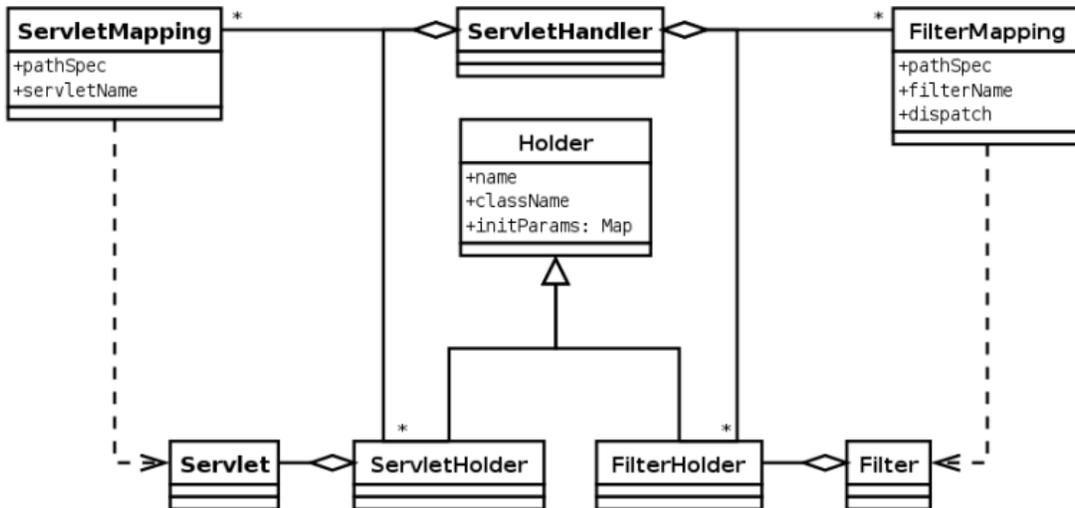
public void handle(String target,
    HttpServletRequest request,
    HttpServletResponse response, int dispatch)
    throws IOException, ServletException;
  
```

An implementation of this method may handle the request, pass the request onto another handler (or servlet) or may modify and/or wrap the request and then pass it on. This gives three styles of Handler:

1. Coordinating Handlers - Handlers that route requests to other handlers (eg [HandlerCollection](#), [ContextHandlerCollection](#))
2. Filtering Handlers - Handlers that augment a request and pass it on to other handlers (eg. [HandlerWrapper](#), [ContextHandler](#), [SessionHandler](#))
3. Generating Handlers - Handlers that produce content (eg [ResourceHandler](#) and [ServletHandler](#))

See also [Writing a Jetty Handler](#).

Servlets



The [ServletHandler](#) is a Handler that generates content by passing the request to any configured [Filters](#) and then to a [Servlet](#) mapped by a URI pattern.

A ServletHandler is normally deployed within the scope of a servlet [Context](#), which is a ContextHandler that provides convenience methods for mapping URIs to servlets.

Filters and Servlets may also use a [RequestDispatcher](#) to reroute a request to another context or another servlet in the current context.

Context

Contexts are handlers that group other handlers below a particular URI context path or a virtual host. Typically a context may have :

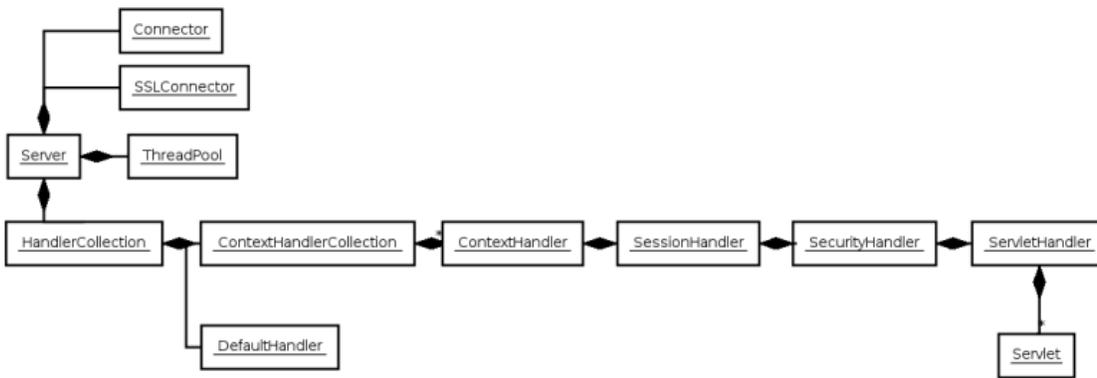
- A context path that defines which requests are handled by the context (eg /myapp)
- A resource base for static content (a docroot)
- A class loader to obtain classes specific to the context (typically docroot/WEB-INF/classes)
- Virtual host names

Contexts implementations include:

- [ContextHandler](#)
- Servlet [Context](#)
- or a [Web Application Context](#).

A web application context combines handlers for security, session and servlets in a single unit that can be configured with a web.xml descriptor.

Web Applications



A [WebApplicationContext](#) is a derivation of the servlet [Context](#) that supports the standardized layout of a web application and configuration of session, security, listeners, filter, servlets and JSP via a web.xml descriptor normally found in the WEB-INF directory of a webapplication.

Essentially the WebApplicationContext is a convenience class to assist the construction and configuration of other handlers to achieve a standard web application configuration.

Configuration is actually done by pluggable implementations of the [Configuration](#) class and the prime among these is [WebXmlConfiguration](#)