NAME

"IO::Async::Listener" – listen on network sockets for incoming connections

SYNOPSIS

```
use IO::Async::Listener;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $listener = IO::Async::Listener->new(
   on_stream => sub {
      my ( undef, $stream ) = @_;

      $stream->configure(
         on_read => sub {
            my ( $self, $buffref, $eof ) = @_;
            $self->write( $$buffref );
            $$buffref = "";
            return 0;
         },
      );

      $loop->add( $stream );
   },
);

$loop->add( $listener );

$listener->listen(
   service  => "echo",
   socktype => 'stream',
)->get;

$loop->run;
```

This object can also be used indirectly via an IO::Async::Loop:

```
use IO::Async::Stream;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

$loop->listen(
   service  => "echo",
   socktype => 'stream',

   on_stream => sub {
      ...
   },
)->get;

$loop->run;
```

DESCRIPTION

This subclass of IO::Async::Handle adds behaviour which watches a socket in listening mode, to accept
incoming connections on them.

A Listener can be constructed and given a existing socket in listening mode. Alternatively, the Listener can construct a socket by calling the `listen` method. Either a list of addresses can be provided, or a service name can be looked up using the underlying loop's `resolve` method.

EVENTS

The following events are invoked, either using subclass methods or CODE references in parameters:

on_accept `$clientsocket` | `$handle`

Invoked whenever a new client connects to the socket.

If neither `handle_constructor` nor `handle_class` parameters are set, this will be invoked with the new client socket directly. If a handle constructor or class are set, this will be invoked with the newly-constructed handle, having the new socket already configured onto it.

on_stream `$stream`

An alternative to `on_accept`, this is passed an instance of IO::Async::Stream when a new client connects. This is provided as a convenience for the common case that a Stream object is required as the transport for a Protocol object.

This is now vaguely deprecated in favour of using `on_accept` with a handle constructor or class.

on_socket `$socket`

Similar to `on_stream`, but constructs an instance of IO::Async::Socket. This is most useful for SOCK_DGRAM or SOCK_RAW sockets.

This is now vaguely deprecated in favour of using `on_accept` with a handle constructor or class.

on_accept_error `$socket`, `$errno`

Optional. Invoked if the `accept` syscall indicates an error (other than EAGAIN or EWOULDBLOCK). If not provided, failures of `accept` will be passed to the main `on_error` handler.

PARAMETERS

The following named parameters may be passed to `new` or `configure`:

on_accept => CODE
on_stream => CODE
on_socket => CODE

CODE reference for the event handlers. Because of the mutually-exclusive nature of their behaviour, only one of these may be set at a time. Setting one will remove the other two.

handle => IO

The IO handle containing an existing listen-mode socket.

handle_constructor => CODE

Optional. If defined, gives a CODE reference to be invoked every time a new client socket is accepted from the listening socket. It is passed the listener object itself, and is expected to return a new instance of IO::Async::Handle or a subclass, used to wrap the new client socket.

```
$handle = $handle_constructor->( $listener )
```

This can also be given as a subclass method

```
$handle = $listener->handle_constructor()
```

handle_class => STRING

Optional. If defined and `handle_constructor` isn't, then new wrapper handles are constructed by invoking the `new` method on the given class name, passing in no additional parameters.

```
$handle = $handle_class->new()
```

This can also be given as a subclass method

```
$handle = $listener->handle_class->new
```

acceptor => STRING|CODE
>   Optional. If defined, gives the name of a method or a CODE reference to use to implement the actual accept behaviour. This will be invoked as:

```
( $accepted ) = $listener->acceptor( $socket )->get

( $handle ) = $listener->acceptor( $socket, handle => $handle )->get
```

>   It is invoked with the listening socket as its its argument, and optionally an IO::Async::Handle instance as a named parameter, and is expected to return a `Future` that will eventually yield the newly-accepted socket or handle instance, if such was provided.

METHODS

>   The following methods documented with a trailing call to ->get return Future instances.

acceptor

```
$acceptor = $listener->acceptor
```

Returns the currently-set `acceptor` method name or code reference. This may be of interest to Loop `listen` extension methods that wish to extend or wrap it.

sockname

```
$name = $listener->sockname
```

Returns the `sockname` of the underlying listening socket

family

```
$family = $listener->family
```

Returns the socket address family of the underlying listening socket

socktype

```
$socktype = $listener->socktype
```

Returns the socket type of the underlying listening socket

listen

```
$listener->listen( %params )
```

This method sets up a listening socket and arranges for the acceptor callback to be invoked each time a new connection is accepted on the socket.

Most parameters given to this method are passed into the `listen` method of the IO::Async::Loop object. In addition, the following arguments are also recognised directly:

on_listen => CODE
>   Optional. A callback that is invoked when the listening socket is ready. Similar to that on the underlying loop method, except it is passed the listener object itself.

```
$on_listen->( $listener )
```

EXAMPLES

Listening on UNIX Sockets

>   The `handle` argument can be passed an existing socket already in listening mode, making it possible to listen on other types of socket such as UNIX sockets.

```
use IO::Async::Listener;
use IO::Socket::UNIX;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $listener = IO::Async::Listener->new(
   on_stream => sub {
      my ( undef, $stream ) = @_;
```

```
        $stream->configure(
           on_read => sub {
              my ( $self, $buffref, $eof ) = @_;
              $self->write( $$buffref );
              $$buffref = "";
              return 0;
           },
        );

        $loop->add( $stream );
     },
  );

  $loop->add( $listener );

  my $socket = IO::Socket::UNIX->new(
     Local => "echo.sock",
     Listen => 1,
  ) or die "Cannot make UNIX socket - $!\n";

  $listener->listen(
     handle => $socket,
  );

  $loop->run;
```

Passing Plain Socket Addresses

The `addr` or `addrs` parameters should contain a definition of a plain socket address in a form that the IO::Async::OS `extract_addrinfo` method can use.

This example shows how to listen on TCP port 8001 on address 10.0.0.1:

```
  $listener->listen(
     addr => {
        family   => "inet",
        socktype => "stream",
        port     => 8001,
        ip       => "10.0.0.1",
     },
     ...
  );
```

This example shows another way to listen on a UNIX socket, similar to the earlier example:

```
  $listener->listen(
     addr => {
        family   => "unix",
        socktype => "stream",
        path     => "echo.sock",
     },
     ...
  );
```

Using A Kernel-Assigned Port Number

Rather than picking a specific port number, is it possible to ask the kernel to assign one arbitrarily that is currently free. This can be done by requesting port number 0 (which is actually the default if no port number is otherwise specified). To determine which port number the kernel actually picked, inspect the

sockport accessor on the actual socket filehandle.

Either use the Future returned by the listen method:

```
$listener->listen(
   addr => { family => "inet" },
)->on_done( sub {
   my ( $listener ) = @_;
   my $socket = $listener->read_handle;

   say "Now listening on port ", $socket->sockport;
});
```

Or pass an on_listen continuation:

```
$listener->listen(
   addr => { family => "inet" },

   on_listen => sub {
      my ( $listener ) = @_;
      my $socket = $listener->read_handle;

      say "Now listening on port ", $socket->sockport;
   },
);
```

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>