
spray

Release *VERSION*

March 01, 2016

1	What is <i>spray</i>?	1
1.1	Principles	1
1.2	Modules	1
1.3	Philosophy	2
2	Getting Started	3
3	spray for Web Development	5
3.1	This Site	6
4	Other Resources	7
5	Documentation	9
5.1	spray-caching	9
5.2	spray-can	11
5.3	spray-client	33
5.4	spray-http	36
5.5	spray-httpx	37
5.6	spray-io	48
5.7	spray-routing	49
5.8	spray-servlet	157
5.9	spray-testkit	163
5.10	spray-util	167
6	Project Info	169
6.1	Current Versions	169
6.2	Migration from M8	170
6.3	Maven Repository	175
6.4	Contributing	175
6.5	Changelog	178
6.6	Credits	193
6.7	References	194
6.8	Sponsors	195
6.9	License	195
6.10	Community Projects	195
7	Blog	197
7.1	spray on the Raspberry Pi	197
7.2	spray 1.0-M8 / 1.1-M8 / 1.2-M8 released	199

7.3	Benchmarking spray	200
7.4	The Magnet Pattern	203
7.5	Welcome to the <i>spray</i> Blog	214
8	Contact	215
8.1	Mailing List	215
8.2	Twitter	215
8.3	Email	215
8.4	Commercial Support	215
	Bibliography	217

What is *spray*?

spray is a suite of lightweight [Scala](#) libraries providing client- and server-side **REST/HTTP** support on top [Akka](#).

We believe that, having chosen [Scala](#) (and possibly [Akka](#)) as primary tools for building software, you'll want to rely on their power not only in your application layer but throughout the full (JVM-level) network stack. *spray* provides just that: a set of integrated components for all your REST/HTTP needs that let you work with idiomatic [Scala](#) (and [Akka](#)) APIs at the stack level of your choice, all implemented without any wrapping layers around “legacy” Java libraries.

1.1 Principles

sprays development is guided by the following principles:

Fully asynchronous, non-blocking All APIs are fully asynchronous, blocking code is avoided wherever at all possible.

Actor- and Future-based *spray* fully embraces the programming model of the platform it is built upon. Akka Actors and Futures are key constructs of its APIs.

High-performance Especially *sprays* low-level components are carefully crafted for excellent performance in high-load environments.

Lightweight All dependencies are very carefully managed, *sprays* codebase itself is kept as lean as possible.

Modular Being structured into a set of integrated but loosely coupled components your application only needs to depend onto the parts that are actually used.

Testable All *spray* components are structured in a way that allows for easy and convenient testing.

1.2 Modules

Currently the *spray* suite consists of these modules:

spray-caching Fast and lightweight in-memory caching built upon [concurrentlinkedhashmap](#) and Akka Futures.

spray-can A low-level, low-overhead, high-performance HTTP server and client built on top of *spray-io*.

spray-client Provides client-side HTTP support at a higher level than the low-level *spray-can* HTTP Client APIs, which it builds on.

spray-http An immutable model of HTTP requests, responses and common headers. This module is completely stand-alone, it neither depends on [Akka](#) nor on any other part of *spray*.

spray-httpx Higher-level tools for working with HTTP messages (mainly marshalling, unmarshalling and (de)compression) that are used by both *spray-client* as well as *spray-routing*.

spray-io A low-level network IO layer for directly connecting Akka actors to asynchronous Java NIO sockets. We like to think of it a basic version of *Netty* for Scala. As of 1.0-M8/1.1-M8 it contains a backport of the new Akka IO layer coming with Akka 2.2. In 1.2-M8 it merely contains a few spray-specific “left-overs” that will likely go away completely in the future.

spray-servlet An adapter layer providing (a subset of) the *spray-can HTTP Server* interface on top of the Servlet API. Enables the use of *spray-routing* in a servlet container.

spray-routing A high-level routing DSL for elegantly defining RESTful web services.

spray-testkit A DSL for easily testing *spray-routing* services. Supports both *ScalaTest* as well as *Specs2*.

spray-util Small utility module used by all other modules except *spray-http*.

spray-json A lightweight, clean and simple JSON implementation in Scala. Because it neither depends on any other part of *spray* nor on Akka and is only an optional dependency of *spray-client* and *spray-httpx* it doesn't live in the main *spray* repository, but rather in its own [github repository](#) Note that you can easily use *spray* with whatever JSON library you like best, *spray-json* is just one of several alternatives.

1.3 Philosophy

Since its inception in early 2011 *sprays* development has been driven with a clear focus on providing tools for building integration layers rather than application cores. As such it regards itself as a suite of *libraries* rather than a framework.

A framework, as we'd like to think of the term, gives you a “frame”, in which you build your application. It comes with a lot of decisions already pre-made and provides a foundation including support structures that lets you get started and deliver results quickly. In a way a framework is like a skeleton onto which you put the “flesh” of your application in order to have it come alive. As such frameworks work best if you choose them *before* you start application development and try to stick to the frameworks “way of doing things” as you go along.

For example, if you are building a browser-facing web application it makes sense to choose a web framework and build your application on top of it because the “core” of the application is the interaction of a browser with your code on the web-server. The framework makers have chosen one “proven” way of designing such applications and let you “fill in the blanks” of a more or less flexible “application-template”. Being able to rely on best-practice architecture like this can be a great asset for getting things done quickly.

However, if your application is not primarily a web application because its core is not browser-interaction but some specialized maybe complex business service and you are merely trying to connect it to the world via a REST/HTTP interface a web-framework might not be what you need. In this case the application architecture should be dictated by what makes sense for the core not the interface layer. Also, you probably won't benefit from the possibly existing browser-specific framework components like view templating, asset management, JavaScript- and CSS generation/manipulation/minification, localization support, AJAX support, etc.

spray was designed specifically as “not-a-framework”, not because we don't like frameworks, but for use cases where a framework is not the right choice. *spray* is made for building integration layers based on HTTP and as such tries to “stay on the sidelines”. Therefore you normally don't build your application “on top of” *spray*, but you build your application on top of whatever makes sense and use *spray* merely for the HTTP integration needs.

Getting Started

To help you get going we created the [spray-template](#) project on GitHub. This provides everything you need to get a *spray* HTTP server application up-and-running in under 5 minutes and view the result in your browser.

Try running one of these examples:

- *spray* as a standalone service - this uses the *spray-routing* and *spray-can* modules
- *spray* inside a servlet container - uses *spray-routing* and *spray-servlet* inside a Jetty server

Note: each example lives in its own Git branch of the *spray-template* project.

spray for Web Development

Even though *sprays* development focus so far has not been web applications but HTTP-based integration layers, you can of course use it for powering browser-based GUIs as well. The recent trend of moving web application logic more and more away from the server and into the (JS-based) browser client as well as the increasing availability of good SBT-plugins for things *spray* itself does not provide (like view-templating or LESS- and CoffeeScript-Support) might even make such an approach gain attractiveness.

Currently a *spray*-based web development stack might consist of (a subset of) these components:

spray-can HTTP Server The web-server. Receives HTTP request and sends out responses. Optionally terminates SSL.

spray-routing The routing layer. Handles requests depending on URI, parameters, content, etc. and (un)marshals to and from the application-specific domain model. Forwards higher-level job requests to deeper application levels and converts the respective results into HTTP responses. Serves static content.

sbt-revolver SBT-plugin for hot reloading of changes to any type of sources (scala, twirl, CSS, LESS, JavaScript, CoffeeScript, images, etc.) without the need to restart the server. Can deliver an almost “dynamic-language” type of development experience.

twirl SBT-plugin providing for view-templating based on the [play 2.0](#) template engine.

less-sbt SBT-plugin for compilation of [LESS](#) sources to CSS.

coffeescripted-sbt SBT-plugin for compilation of [CoffeeScript](#) sources to JavaScript.

sbt-js SBT-plugin for Javascript and Coffeescript compilation, minification, and templating.

SLICK Library for elegant database query and access.

spray-json Library for clean and idiomatic JSON reading and writing.

sbt-assembly SBT-plugin for single-fat-JAR-deployment.

A client-side frontend framework One of the several established client-side JavaScript frameworks.

While a stack like this might not provide everything that a full-grown web framework can offer it could have all that’s required for your particular application. And, because you can pick the best tool for each individual job, the resulting application stack is a lot more flexible and possibly future-proof than any single framework. Of course the downside of this approach is that integrating the different components is now on your shoulders. Also, there is no single point of contact for support and upgrades.

Still, combining a client-side JavaScript framework with a *spray*-based application backend could prove itself as an interesting alternative to a “classic” server-side web framework. We’d love to hear about your experiences in this regard...

3.1 This Site

One example of a simple website running on a *spray*-based stack is this site (<http://spray.io>). Here is the stack we use for *spray.io*:

- *spray-can HTTP Server*
- *spray-routing*
- *sbt-revolver*
- *twirl*
- a custom SBT extension for compiling *Sphinx* sources to JSON
- *spray-json* (for reading *Sphinx* output)
- *sbt-assembly*
- *Mentor* (a non-free, responsive HTML5 template based on *Bootstrap*)

For more details check out the route definition of this site: <https://github.com/spray/spray/blob/master/site/src/main/scala/spray/site/SiteS>

Other Resources

Oct 2015: “Reactive Streams & Akka HTTP - Behold the Flow!” talk from Scala World 2015

Video: <https://www.youtube.com/watch?v=6VBn9V3S2aQ>

Code: <https://github.com/jrudolph/scala-world-2015>

Jun 2015: “Reactive Streams - The Now” talk from ScalaDays 2015

Video: <https://www.parleys.com/tutorial/the-reactive-streams-implementation-landscape>

Slides: </scaladays/2015/>

Feb 2015: “Reactive Streams & Akka HTTP” at Scala User Group - Berlin Brandenburg

Slides: </berlin/>

Feb 2015: “REST on Akka: Connect to the World” at Jfokus 2015

Slides: </jfokus/>

Jan 2015: “Akka HTTP — The What, Why and How” at NEScala 2015

Video: https://www.youtube.com/watch?v=y_slPbktLr0

Slides: </nescala2015/>

Jan 2015: “Reactive Streams and Akka HTTP” at the Vienna Scala User Group

Slides: </vienna/>

Dec 2014: “Reactive Streams and Akka HTTP” at the Dutch Scala Enthusiasts

Slides: </duse/>

Dec 2014: “Akka HTTP: Connect to the world” talk from Scala eXchange 2014

Video: <https://skillsmatter.com/skillscasts/5852-rest-on-akka-connect-to-the-world>

Slides: </scalax/2014/>

Oct 2014: “Akka HTTP: Connect to the world” talk from scala.io 2014

Slides: </scala.io/2014/>

Oct 2014: “Reactive Streams and Akka HTTP” at the Munich Scala User Group Meetup

Slides: </msug/>

June 2014: “akka-http: (un)REST your actors” talk from ScalaDays 2014

Video: <http://www.parleys.com/play/53a7d2c3e4b0543940d9e539/chapter0/about>

Slides: </scaladays/2014/>

May 2014: “Actors, HTTP and Reactive Streams” talk from JAX 2014

Slides: /jax14/

Apr 2014: “Introduction to Akka and spray” at the Zurich Scala Enthusiasts Meetup

Slides: /zse/

Dec 2013: Introductory live-coding session at Scala eXchange 2013

Video: <http://skillsmatter.com/podcast/scala/mathias-doenitz>

Nov 2013: “spray: REST on Akka” Typesafe webinar

Video: http://www.youtube.com/watch?v=7MqD7_YvZ8Q

Slides: /webinar/

Nov 2013: “Akka & spray: Actors, IO and HTTP” talk from wjax 2013

Slides: /wjax/

Oct 2013: “spray: REST on Akka” talk from scala.io 2013

Slides: /scala.io/2013/

Jan 2013: “The Scala Types” podcast: “Mathias Doenitz on spray”

Part 1: <http://bit.ly/scalatypes-podcast-on-spray-part-1>

Part 2: <http://bit.ly/scalatypes-podcast-on-spray-part-2>

Apr 2012: “spray: REST on Akka” talk from the Scala Days 2012

Video: <http://skillsmatter.com/podcast/scala/spray-rest-on-akka>

Slides: <http://www.slideshare.net/sirthias/spray-rest-on-akka-12616908>

Mar 2012: “spray: REST on Akka” talk from the Northeast Scala Symposium 2012

Video: http://www.youtube.com/watch?v=fUuLJmWB__E

Slides: <http://www.slideshare.net/sirthias/spray-rest-on-akka>

Documentation

This is the index of all documentation chapters for the different modules of the *spray* suite:

5.1 spray-caching

spray-caching provides a lightweight and fast in-memory caching functionality based on Akka Futures and `concurrentlinkedhashmap`. The primary use-case is the “wrapping” of an expensive operation with a caching layer that, based on a certain key of type `K`, runs the wrapped operation only once and returns the the cached value for all future accesses for the same key (as long as the respective entry has not expired).

The central idea of a *spray-caching* cache is to not store the actual values of type `T` themselves in the cache but rather corresponding Akka Futures, i.e. instances of type `Future[T]`. This approach has the advantage of nicely taking care of the thundering herds problem where many requests to a particular cache key (e.g. a resource URI) arrive before the first one could be completed. Normally (without special guarding techniques, like so-called “cowboy” entries) this can cause many requests to compete for system resources while trying to compute the same result thereby greatly reducing overall system performance. When you use a *spray-caching* cache the very first request that arrives for a certain cache key causes a future to be put into the cache which all later requests then “hook into”. As soon as the first request completes all other ones complete as well. This minimizes processing time and server load for all requests.

5.1.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-caching* depends on

- *spray-util*
- `concurrentlinkedhashmap`
- akka-actor 2.1.x (with ‘provided’ scope, i.e. you need to pull it in yourself)

5.1.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-caching* into your classpath.

Afterwards just `import spray.caching._` to bring all relevant identifiers into scope.

5.1.3 The *Cache* Interface

All *spray-caching* cache implementations implement the `Cache` trait, which allows you to interact with the cache through nine methods:

- `def apply(key: Any)(expr: => V): Future[V]` wraps an “expensive” expression with caching support. Note, that the generation expression is never run inside a `Future` with this overload. Instead, either, the cache already contains an entry for the key in which case the existing result is returned, or the generating expression is synchronously run to produce the value.
- `def apply(key: Any)(future: => Future[V]): Future[V]` is similar, but allows the expression to produce the future itself.
- `def apply(key: Any)(func: Promise[V] => Unit): Future[V]` provides a “push-style” alternative.
- `def get(key: Any): Option[Future[V]]` retrieves the future instance that is currently in the cache for the given key. Returns `None` if the key has no corresponding cache entry.
- `def remove(key: Any): Option[Future[V]]` removes the cache item for the given key. Returns the removed item if it was found (and removed).
- `def clear()` clears the cache by removing all entries.
- `def size(): Int` returns the number of entries.
- `def keys(): Set[Any]` returns the current keys as an unordered set.
- `def ascendingKeys(limit: Option[Int]): Iterator[Any]` allows one to iterate through the keys in order from the least recently used to the most recently used.

Note that the `apply` overloads require an implicit `ExecutionContext` to be in scope.

5.1.4 Example

```
import scala.concurrent.Future
import akka.actor.ActorSystem
import spray.caching.{LruCache, Cache}
import spray.util._

val system = ActorSystem()
import system.dispatcher

// if we have an "expensive" operation
def expensiveOp(): Double = new util.Random().nextDouble()

// and a Cache for its result type
val cache: Cache[Double] = LruCache()

// we can wrap the operation with caching support
// (providing a caching key)
def cachedOp[T](key: T): Future[Double] = cache(key) {
  expensiveOp()
}

// and profit
cachedOp("foo").await === cachedOp("foo").await
cachedOp("bar").await !== cachedOp("foo").await
```

5.1.5 Cache Implementations

`spray-caching` comes with two implementations of the `Cache` interface, `SimpleLruCache` and `ExpiringLruCache`, both featuring last-recently-used cache eviction semantics and both internally wrapping a `concurrentlinkedhashmap`. They

difference between the two only consists of whether they support time-based entry expiration or not.

The easiest way to construct a cache instance is via the `apply` method of the `LruCache` object, which has the following signature and creates a new `ExpiringLruCache` or `SimpleLruCache` depending on whether `timeToLive` and/or `timeToIdle` are finite (= expiring) or infinite:

```
/**
 * Creates a new [[spray.caching.ExpiringLruCache]] or
 * [[spray.caching.SimpleLruCache]] instance depending on whether
 * a non-zero and finite timeToLive and/or timeToIdle is set or not.
 */
def apply[V](maxCapacity: Int = 500,
             initialCapacity: Int = 16,
             timeToLive: Duration = Duration.Inf,
             timeToIdle: Duration = Duration.Inf): Cache[V] = {
```

SimpleLruCache

This cache implementation has a defined maximum number of entries it can store. After the maximum capacity is reached new entries cause old ones to be evicted in a last-recently-used manner, i.e. the entries that haven't been accessed for the longest time are evicted first.

ExpiringLruCache

This implementation has the same limited capacity behavior as the `SimpleLruCache` but in addition supports time-to-live as well as time-to-idle expiration. The former provides an upper limit to the time period an entry is allowed to remain in the cache while the latter limits the maximum time an entry is kept without having been accessed. If both values are finite the time-to-live has to be strictly greater than the time-to-idle.

Note: Expired entries are only evicted upon next access (or by being thrown out by the capacity constraint), so they might prevent garbage collection of their values for longer than expected.

5.2 spray-can

The *spray-can* module provides a low-level, low-overhead, high-performance HTTP server and client built on top of *spray-io*. Both are fully asynchronous, non-blocking and built 100% in Scala on top of Akka. Since their APIs are centered around Akka abstractions such as Actors and Futures they are very easy to integrate into your Akka-based applications.

5.2.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-can* depends on

- *spray-io*
- *spray-http*
- *spray-util*
- akka-actor 2.1.x (with 'provided' scope, i.e. you need to pull it in yourself)

5.2.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-can* into your classpath.

Once you have *spray-can* available you communicate with it mostly via the IO (Http) extension it provides. See the respective chapter for more information on this-

5.2.3 Configuration

Just like Akka *spray-can* relies on the *typesafe config* library for configuration. As such its JAR contains a `reference.conf` file holding the default values of all configuration settings. In your application you typically provide an `application.conf` in which you override Akka and/or *spray* settings according to your needs.

Note: Since *spray* uses the same configuration technique as Akka you might want to check out the [Akka Documentation on Configuration](#).

This is the `reference.conf` of the *spray-can* module:

```
#####
# spray-can Reference Config File #
#####

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

spray.can {

  server {
    # The value of the `Server` header to produce.
    # Set to the empty string to disable rendering of the server header.
    server-header = spray-can/${spray.version}

    # Enables/disables SSL encryption.
    # If enabled the server uses the implicit `ServerSSLEngineProvider` member
    # of the `Bind` command to create `SSLEngine` instances for the underlying
    # IO connection.
    ssl-encryption = off

    # The maximum number of requests that are accepted (and dispatched to
    # the application) on one single connection before the first request
    # has to be completed.
    # Incoming requests that would cause the pipelining limit to be exceeded
    # are not read from the connections socket so as to build up "back-pressure"
    # to the client via TCP flow control.
    # A setting of 1 disables HTTP pipelining, since only one request per
    # connection can be "open" (i.e. being processed by the application) at any
    # time. Set to higher values to enable HTTP pipelining.
    # Set to 'disabled' for completely disabling pipelining limits
    # (not recommended on public-facing servers due to risk of DoS attacks).
    # This value must be > 0 and <= 128.
    pipelining-limit = 1

    # The time after which an idle connection will be automatically closed.
    # Set to `infinite` to completely disable idle connection timeouts.
    idle-timeout = 60 s
  }
}
```



```
# If a request hasn't been responded to after the time period set here
# a `spray.http.Timedout` message will be sent to the timeout handler.
# Set to `infinite` to completely disable request timeouts.
request-timeout = 20 s

# After a `Timedout` message has been sent to the timeout handler and the
# request still hasn't been completed after the time period set here
# the server will complete the request itself with an error response.
# Set to `infinite` to disable timeout timeouts.
timeout-timeout = 2 s

# The period during which a service must respond to a `ChunkedRequestStart` message
# with a `RegisterChunkHandler` message. During the registration period reading from
# the network is suspended. It is still possible that some chunks have already been
# received which will be buffered until the registration is received or the timeout is
# triggered. If the timeout is triggered the connection is immediately aborted.
chunkhandler-registration-timeout = 500 ms

# The path of the actor to send `spray.http.Timedout` messages to.
# If empty all `Timedout` messages will go to the "regular" request
# handling actor.
timeout-handler = ""

# The "granularity" of timeout checking for both idle connections timeouts
# as well as request timeouts, should rarely be needed to modify.
# If set to `infinite` request and connection timeout checking is disabled.
reaping-cycle = 250 ms

# Enables/disables support for statistics collection and querying.
# Even though stats keeping overhead is small,
# for maximum performance switch off when not needed.
stats-support = on

# Enables/disables the addition of a `Remote-Address` header
# holding the clients (remote) IP address.
remote-address-header = off

# Enables/disables the addition of a `Raw-Request-URI` header holding the
# original raw request URI as the client has sent it.
raw-request-uri-header = off

# Enables/disables automatic handling of HEAD requests.
# If this setting is enabled the server dispatches HEAD requests as GET
# requests to the application and automatically strips off all message
# bodies from outgoing responses.
# Note that, even when this setting is off the server will never send
# out message bodies on responses to HEAD requests.
transparent-head-requests = on

# Enables/disables an alternative response streaming mode that doesn't
# use `Transfer-Encoding: chunked` but rather renders the individual
# MessageChunks coming in from the application as parts of the original
# response entity.
# Enabling this mode causes all connections to be closed after a streaming
# response has been finished since there is no other way to signal the
# response end to the client.
# Note that chunkless-streaming is implicitly enabled when streaming
# responses to HTTP/1.0 clients (since they don't support
```

```
# `Transfer-Encoding: chunked`)  
chunkless-streaming = off  
  
# Enables/disables the returning of more detailed error messages to  
# the client in the error response.  
# Should be disabled for browser-facing APIs due to the risk of XSS attacks  
# and (probably) enabled for internal or non-browser APIs.  
# Note that spray will always produce log messages containing the full  
# error details.  
verbose-error-messages = off  
  
# Enables/disables the logging of the full (potentially multiple line)  
# error message to the server logs.  
# If disabled only a single line will be logged.  
verbose-error-logging = off  
  
# If this setting is non-zero the HTTP server automatically aggregates  
# incoming request chunks into full HttpRequest objects before dispatching them to  
# the application. If the size of the aggregated requests surpasses the  
# specified limit the server responds with a `413 Request Entity Too Large`  
# error response before closing the connection.  
# Set to zero to disable automatic request chunk aggregation and have  
# ChunkedRequestStart, MessageChunk and ChunkedMessageEnd messages be  
# dispatched to the handler.  
request-chunk-aggregation-limit = 1m  
  
# The initial size of the buffer to render the response headers in.  
# Can be used for fine-tuning response rendering performance but probably  
# doesn't have to be fiddled with in most applications.  
response-header-size-hint = 512  
  
# For HTTPS connections this setting specifies the maximum number of  
# bytes that are encrypted in one go. Large responses are broken down in  
# chunks of this size so as to already begin sending before the response has  
# been encrypted entirely.  
max-encryption-chunk-size = 1m  
  
# The time period within which the TCP binding process must be completed.  
# Set to `infinite` to disable.  
bind-timeout = 1s  
  
# The time period within which the TCP unbinding process must be completed.  
# Set to `infinite` to disable.  
unbind-timeout = 1s  
  
# The time period within which a connection handler must have been  
# registered after the bind handler has received a `Connected` event.  
# Set to `infinite` to disable.  
registration-timeout = 1s  
  
# The time after which a connection is aborted (RST) after a parsing error  
# occurred. The timeout prevents a connection which is already known to be  
# erroneous from receiving evermore data even if all of the data will be ignored.  
# However, in case of a connection abortion the client usually doesn't properly  
# receive the error response. This timeout is a trade-off which allows the client  
# some time to finish its request and receive a proper error response before the  
# connection is forcibly closed to free resources.  
parsing-error-abort-timeout = 2s
```

```

# If this setting is empty the server only accepts requests that carry a
# non-empty `Host` header. Otherwise it responds with `400 Bad Request`.
# Set to a non-empty value to be used in lieu of a missing or empty `Host`
# header to make the server accept such requests.
# Note that the server will never accept HTTP/1.1 request without a `Host`
# header, i.e. this setting only affects HTTP/1.1 requests with an empty
# `Host` header as well as HTTP/1.0 requests.
# Examples: `www.spray.io` or `example.com:8080`
default-host-header = ""

# Enables/disables automatic back-pressure handling by write buffering and
# receive throttling
automatic-back-pressure-handling = on

back-pressure {
  # The reciprocal rate of requested Acks per NoAcks. E.g. the default value
  # '10' means that every 10th write request is acknowledged. This affects the
  # number of writes each connection has to buffer even in absence of back-pressure.
  noack-rate = 10

  # The lower limit the write queue size has to shrink to before reads are resumed.
  # Use 'infinite' to disable the low-watermark so that reading is resumed instantly
  # after the next successful write.
  reading-low-watermark = infinite
}

# Enables more verbose DEBUG logging for debugging SSL related issues.
ssl-tracing = off

# Modify to tweak parsing settings on the server-side only.
parsing = ${spray.can.parsing}
}

client {
  # The default value of the `User-Agent` header to produce if no
  # explicit `User-Agent`-header was included in a request.
  # If this value is the empty string and no header was included in
  # the request, no `User-Agent` header will be rendered at all.
  user-agent-header = spray-can/${spray.version}

  # The time after which an idle connection will be automatically closed.
  # Set to `infinite` to completely disable idle timeouts.
  idle-timeout = 60 s

  # The max time period that a client connection will be waiting for a response
  # before triggering a request timeout. The timer for this logic is not started
  # until the connection is actually in a state to receive the response, which
  # may be quite some time after the request has been received from the
  # application!
  # There are two main reasons to delay the start of the request timeout timer:
  # 1. On the host-level API with pipelining disabled:
  #   If the request cannot be sent immediately because all connections are
  #   currently busy with earlier requests it has to be queued until a
  #   connection becomes available.
  # 2. With pipelining enabled:
  #   The request timeout timer starts only once the response for the
  #   preceding request on the connection has arrived.
  # Set to `infinite` to completely disable request timeouts.

```

```
request-timeout = 20 s

# the "granularity" of timeout checking for both idle connections timeouts
# as well as request timeouts, should rarely be needed to modify.
# If set to `infinite` request and connection timeout checking is disabled.
reaping-cycle = 250 ms

# If this setting is non-zero the HTTP client connections automatically
# aggregate incoming response chunks into full HttpResponses before
# dispatching them to the application.
# If the size of the aggregated response surpasses the specified limit the
# HTTP client connection is closed and an error returned.
# Set to zero to disable automatic request chunk aggregation and have
# ChunkedResponseStart, MessageChunk and ChunkedMessageEnd messages be
# dispatched to the application.
response-chunk-aggregation-limit = 1m

# Enables/disables an alternative request streaming mode that doesn't
# use `Transfer-Encoding: chunked` but rather renders the individual
# MessageChunks coming in from the application as parts of the original
# request entity.
# Enabling this mode causes all requests to require an explicit `Content-Length`
# header for streaming requests.
# Note that chunkless-streaming is implicitly enabled when streaming
# HTTP/1.0 requests since they don't support `Transfer-Encoding: chunked`.
chunkless-streaming = off

# The initial size of the buffer to render the request headers in.
# Can be used for fine-tuning request rendering performance but probably
# doesn't have to be fiddled with in most applications.
request-header-size-hint = 512

# For HTTPS connections this setting specifies the maximum number of
# bytes that are encrypted in one go. Large requests are broken down in
# chunks of this size so as to already begin sending before the request has
# been encrypted entirely.
max-encryption-chunk-size = 1m

# The time period within which the TCP connecting process must be completed.
# Set to `infinite` to disable.
connecting-timeout = 10s

# The proxy configurations to be used for requests with the specified
# scheme.
proxy {
  # Proxy settings for unencrypted HTTP requests
  # Set to 'none' to always connect directly, 'default' to use the system
  # settings as described in http://docs.oracle.com/javase/6/docs/technotes/guides/net/proxies.html
  # or specify the proxy host, port and non proxy hosts as demonstrated
  # in the following example:
  # http {
  #   host = myproxy.com
  #   port = 8080
  #   non-proxy-hosts = ["*.direct-access.net"]
  # }
  http = default

  # Proxy settings for HTTPS requests (currently unsupported)
```

```

    https = default
  }

  # Enables more verbose DEBUG logging for debugging SSL related issues.
  ssl-tracing = off

  # Modify to tweak parsing settings on the client-side only.
  parsing = ${spray.can.parsing}
}

host-connector {
  # The maximum number of parallel connections that an `HttpHostConnector`
  # is allowed to establish to a host. Must be greater than zero.
  max-connections = 4

  # The maximum number of times an `HttpHostConnector` attempts to repeat
  # failed requests (if the request can be safely retried) before
  # giving up and returning an error.
  max-retries = 5

  # Configures redirection following.
  # If set to zero redirection responses will not be followed, i.e. they'll be returned to the user.
  # If set to a value > zero redirection responses will be followed up to the given number of times.
  # If the redirection chain is longer than the configured value the first redirection response that
  # is not followed anymore is returned to the user as is.
  max-redirects = 0

  # If this setting is enabled, the `HttpHostConnector` pipelines requests
  # across connections, otherwise only one single request can be "open"
  # on a particular HTTP connection.
  pipelining = off

  # The time after which an idle `HttpHostConnector` (without open
  # connections) will automatically terminate itself.
  # Set to `infinite` to completely disable idle timeouts.
  idle-timeout = 30 s

  # Modify to tweak client settings for this host-connector only.
  client = ${spray.can.client}
}

# The (default) configuration of the HTTP message parser for the server and
# the client.
# IMPORTANT: These settings (i.e. children of `spray.can.parsing`) can't be directly
# overridden in `application.conf` to change the parser settings for client and server
# altogether (see https://github.com/spray/spray/issues/346). Instead, override the
# concrete settings beneath `spray.can.server.parsing` and `spray.can.client.parsing`
# where these settings are copied to.
parsing {
  # The limits for the various parts of the HTTP message parser.
  max-uri-length = 2k
  max-response-reason-length = 64
  max-header-name-length = 64
  max-header-value-length = 8k
  max-header-count = 64
  max-content-length = 8m
  max-chunk-ext-length = 256
  max-chunk-size = 1m
}

```

```
# Sets the strictness mode for parsing request target URIs.
# The following values are defined:
#
# `strict`: RFC3986-compliant URIs are required,
#           a 400 response is triggered on violations
#
# `relaxed`: all visible 7-Bit ASCII chars are allowed
#
# `relaxed-with-raw-query`: like `relaxed` but additionally
#                           the URI query is not parsed, but delivered as one raw string
#                           as the `key` value of a single Query structure element.
#
uri-parsing-mode = strict

# Enables/disables the logging of warning messages in case an incoming
# message (request or response) contains an HTTP header which cannot be
# parsed into its high-level model class due to incompatible syntax.
# Note that, independently of this settings, spray will accept messages
# with such headers as long as the message as a whole would still be legal
# under the HTTP specification even without this header.
# If a header cannot be parsed into a high-level model instance it will be
# provided as a `RawHeader`.
illegal-header-warnings = on

# limits for the number of different values per header type that the
# header cache will hold
header-cache {
  default = 12
  Content-MD5 = 0
  Date = 0
  If-Match = 0
  If-Modified-Since = 0
  If-None-Match = 0
  If-Range = 0
  If-Unmodified-Since = 0
  User-Agent = 32
}

# Sets the size starting from which incoming http-messages will be delivered
# in chunks regardless of whether chunking is actually used on the wire.
# Set to infinite to disable auto chunking.
incoming-auto-chunking-threshold-size = infinite

# Enables/disables inclusion of an SSL-Session-Info header in parsed
# messages over SSL transports (i.e., HttpRequest on server side and
# HttpResponse on client side).
ssl-session-info-header = off
}

# Fully qualified config path which holds the dispatcher configuration
# to be used for the HttpManager.
manager-dispatcher = "akka.actor.default-dispatcher"

# Fully qualified config path which holds the dispatcher configuration
# to be used for the HttpClientSettingsGroup actors.
settings-group-dispatcher = "akka.actor.default-dispatcher"

# Fully qualified config path which holds the dispatcher configuration
```

```

# to be used for the HttpHostConnector actors.
host-connector-dispatcher = "akka.actor.default-dispatcher"

# Fully qualified config path which holds the dispatcher configuration
# to be used for HttpListener actors.
listener-dispatcher = "akka.actor.default-dispatcher"

# Fully qualified config path which holds the dispatcher configuration
# to be used for HttpServerConnection and HttpClientConnection actors.
connection-dispatcher = "akka.actor.default-dispatcher"
}

```

5.2.4 HTTP Server

The *spray-can* HTTP server is an embedded, actor-based, fully asynchronous, low-level, low-overhead and high-performance HTTP/1.1 server implemented on top of Akka IO / *spray-io*.

It sports the following features:

- Low per-connection overhead for supporting many thousand concurrent connections
- Efficient message parsing and processing logic for high throughput applications
- Full support for HTTP persistent connections
- Full support for HTTP pipelining
- Full support for asynchronous HTTP streaming (i.e. “chunked” transfer encoding)
- Optional SSL/TLS encryption
- Actor-based architecture and API for easy integration into your Akka applications

Design Philosophy

The *spray-can* `HttpServer` is scoped with a clear focus on the essential functionality of an HTTP/1.1 server:

- Connection management
- Message parsing and header separation
- Timeout management (for requests and connections)
- Response ordering (for transparent pipelining support)

All non-core features of typical HTTP servers (like request routing, file serving, compression, etc.) are left to the next-higher layer in the application stack, they are not implemented by *spray-can* itself. Apart from general focus this design keeps the server small and light-weight as well as easy to understand and maintain. It also makes a *spray-can* HTTP server a perfect “container” for a *spray-routing* application, since *spray-can* and *spray-routing* nicely complement and interface into each other.

Basic Architecture

The *spray-can* HTTP server is implemented by two types of Akka actors, which sit on top of Akka IO. When you tell *spray-can* to start a new server instance on a given port an `HttpListener` actor is started, which accepts incoming connections and for each one spawns a new `HttpServerConnection` actor, which then manages the connection for the rest of its lifetime. These connection actors process the requests coming in across their connection and dispatch

them as immutable *spray-http* `HttpRequest` instances to a “handler” actor provided by your application. The handler can complete a request by simply replying with an `HttpResponse` instance:

```
def receive = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
    sender ! HttpResponse(entity = "PONG")
}
```

Your code never deals with the `HttpListener` and `HttpServerConnection` actor classes directly, in fact they are marked `private` to the *spray-can* package. All communication with these actors happens purely via actor messages, the majority of which are defined in the `spray.can.Http` object.

Starting

A *spray-can* HTTP server is started by sending an `Http.Bind` command to the `Http` extension:

```
import akka.io.IO
import spray.can.Http

implicit val system = ActorSystem()

val myListener: ActorRef = // ...

IO(Http) ! Http.Bind(myListener, interface = "localhost", port = 8080)
```

With the `Http.Bind` command you register an application-level “listener” actor and specify the interface and port to bind to. Additionally the `Http.Bind` command also allows you to define socket options as well as a larger number of settings for configuring the server according to your needs.

The sender of the `Http.Bind` command (e.g. an actor you have written) will receive an `Http.Bound` reply after the HTTP layer has successfully started the server at the respective endpoint. In case the bind fails (e.g. because the port is already busy) an `Http.CommandFailed` message is dispatched instead.

The sender of the `Http.Bound` confirmation event is *spray-can*’s `HttpListener` instance. You will need this `ActorRef` if you want to stop the server later.

Stopping

To explicitly stop the server, send an `Http.Unbind` command to the `HttpListener` instance (the `ActorRef` for this instance is available as the sender of the `Http.Bound` confirmation event from when the server was started).

The listener will reply with an `Http.Unbound` event after successfully unbinding from the port (or with an `Http.CommandFailed` in the case of error). At that point no further requests will be accepted by the server.

Any requests which were in progress at the time will proceed to completion. When the last request has terminated, the `HttpListener` instance will exit. You can monitor for this (e.g. so that you can shutdown the `ActorSystem`) by watching the listener actor and awaiting a `Terminated` message.

Message Protocol

After having successfully bound an `HttpListener` your application communicates with the *spray-can*-level connection actors via a number of actor messages that are explained in this section.

Request-Response Cycle

When a new connection has been accepted the application-level listener, which was registered with the `Http.Bind` command, receives an `Http.Connected` event message from the connection actor. The application must reply to it with an `Http.Register` command within the configured `registration-timeout` period, otherwise the connection will be closed.

With the `Http.Register` command the application tells the connection actor which actor should handle incoming requests. The application is free to register the same actor for all connections (a “singleton handler”), a new one for every connection (“per-connection handlers”) or anything in between. After the connection actor has received the `Http.Register` command it starts reading requests from the connection and dispatches them as `spray.http.HttpRequestPart` messages to the handler. The handler actor should then process the request according to the application logic and respond by sending an `HttpResponsePart` instance to the sender of the request.

The `ActorRef` used as the sender of an `HttpRequestPart` received by the handler is unique to the request, i.e. several requests, even when coming in across the same connection, will appear to be sent from different senders. *spray-can* uses this sender `ActorRef` to coalesce the response with the request, so you cannot send several responses to the same sender. However, the different request parts of chunked requests arrive from the same sender, and the different response parts of a chunked response need to be sent to the same sender as well.

Caution: Since the `ActorRef` used as the sender of a request is an `UnregisteredActorRef` it is not reachable remotely. This means that the actor designated as handler by the application needs to live in the same JVM as the HTTP extension.

Chunked Requests

If the `request-chunk-aggregation-limit` config setting is set to zero the connection actor also dispatches the individual request parts of chunked requests to the handler actor. In these cases a full request consists of the following messages:

- One `ChunkedRequestStart`
- Zero or more `MessageChunks`
- One `ChunkedMessageEnd`

The timer for checking request handling timeouts (if not configured to `infinite`) only starts running when the final `ChunkedMessageEnd` message was dispatched to the handler.

Chunked Responses

Alternatively to a single `HttpResponse` instance the handler can choose to respond to the request sender with the following sequence of individual messages:

- One `ChunkedResponseStart`
- Zero or more `MessageChunks`
- One `ChunkedMessageEnd`

The timer for checking request handling timeouts (if not configured to `infinite`) will stop running as soon as the initial `ChunkedResponseStart` message has been received from the handler, i.e. there is currently no timeout checking for and in between individual response chunks.

Request Timeouts

If the handler does not respond to a request within the configured `request-timeout` period a `spray.http.Timedout` message is sent to the timeout handler, which can be the “regular” handler itself or another actor (depending on the `timeout-handler` config setting). The timeout handler then has the chance to complete the request within the time period configured as `timeout-timeout`. Only if the timeout handler also misses its deadline for completing the request will the connection actor complete the request itself with a “hard-coded” error response.

In order to change the respective config setting *for that connection only* the application can send the following messages to the sender of a request (part) or the connection actor:

- `spray.io.ConnectionTimeouts.SetIdleTimeout`
- `spray.http.SetRequestTimeout`
- `spray.http.SetTimeoutTimeout`

Closed Notifications

When a connection is closed, for whatever reason, the connection actor dispatches one of five defined `Http.ConnectionClosed` event message to the application (see the *Common Behavior* chapter for more info).

Exactly which actor receives it depends on the current state of request processing. The connection actor sends `Http.ConnectionClosed` events coming in from the underlying IO layer

- to the handler actor
- to the *request* chunk handler if one is defined and no response part was yet received
- to the sender of the last received response part
 - if the ACK for an ACKed response part has not yet been dispatched
 - if a *response* chunk stream has not yet been finished (with a `ChunkedMessageEnd`)

Note: The application can always choose to actively close a connection by sending one of the three defined `Http.CloseCommand` messages to the sender of a request or the connection actor (see *Common Behavior*). However, during normal operation it is encouraged to make use of the `Connection` header to signal to the connection actor whether or not the connection is to be closed after the response has been sent.

Server Statistics

If the `stats-support` config setting is enabled the server will continuously count connections, requests, timeouts and other basic statistics. You can ask the `HttpListener` actor (i.e. the sender `ActorRef` of the `Http.Bound` event message!) to reply with an instance of the `spray.can.server.Stats` class by sending it an `Http.GetStats` command. This is what you will get back:

```
case class Stats(  
  uptime: FiniteDuration,  
  totalRequests: Long,  
  openRequests: Long,  
  maxOpenRequests: Long,  
  totalConnections: Long,  
  openConnections: Long,
```

```
maxOpenConnections: Long,
requestTimeouts: Long)
```

By sending the listener an `Http.ClearStats` command message you can trigger a reset of the stats.

HTTP Headers

When a *spray-can* connection actor receives an HTTP request it tries to parse all its headers into their respective *spray-http* model classes. No matter whether this succeeds or not, the connection actor will always pass on all received headers to the application. Unknown headers as well as ones with invalid syntax (according to *spray*'s header parser) will be made available as `RawHeader` instances. For the ones exhibiting parsing errors a warning message is logged depending on the value of the `illegal-header-warnings` config setting.

When sending out responses the connection actor watches for a `Connection` header set by the application and acts accordingly, i.e. you can force the connection actor to close the connection after having sent the response by including a `Connection("close")` header. To unconditionally force a connection keep-alive you can explicitly set a `Connection("Keep-Alive")` header. If you don't set an explicit `Connection` header the connection actor will keep the connection alive if the client supports this (i.e. it either sent a `Connection: Keep-Alive` header or advertised HTTP/1.1 capabilities without sending a `Connection: close` header).

The following response headers are managed by the *spray-can* layer itself and as such are **ignored** if you “manually” add them to the response (you'll see a warning in your logs):

- `Content-Type`
- `Content-Length`
- `Transfer-Encoding`
- `Date`
- `Server`

There are three exceptions:

1. Responses to `HEAD` requests that have an empty entity are allowed to contain a user-specified `Content-Type` header.
2. Responses in `ChunkedResponseStart` messages that have an empty entity are allowed to contain a user-specified `Content-Type` header.
3. Responses in `ChunkedResponseStart` messages are allowed to contain a user-specified `Content-Length` header if `spray.can.server.chunkless-streaming` is enabled.

Note: The `Content-Type` header has special status in *spray* since its value is part of the `HttpEntity` model class. Even though the header also remains in the headers list of the `HttpRequest` *sprays* higher layers (like *spray-routing*) only work with the `ContentType` value contained in the `HttpEntity`.

HTTP Pipelining

spray-can fully supports HTTP pipelining. If the configured `pipelining-limit` is greater than one a connection actor will accept several requests in a row (coming in across a single connection) and dispatch them to the application even before the first one has been responded to. This means that several requests will potentially be handled by the application at the same time.

Since in many asynchronous applications request handling times can be somewhat undeterministic *spray-can* takes care of properly ordering all responses coming in from your application before sending them out to “the wire”. I.e.

your application will “see” requests in the order they are coming in but is *not* required to itself uphold this order when generating responses.

SSL Support

If enabled via the `ssl-encryption` config setting the *spray-can* connection actors pipe all IO traffic through an `SslTlsSupport` module, which can perform transparent SSL/TLS encryption. This module is configured via the implicit `ServerSSLAuthProvider` member on the `Http.Bind` command message. An `ServerSSLAuthProvider` is essentially a function `PipelineContext Option[SSLContext]`, which determines whether encryption is to be performed and, if so, which `javax.net.ssl.SSLContext` instance is to be used.

If you’d like to apply some custom configuration to your `SSLContext` instances an easy way would be to bring a custom engine provider into scope, e.g. like this:

```
import spray.io.ServerSSLAuthProvider

implicit val myEngineProvider = ServerSSLAuthProvider { engine =>
  engine.setEnabledCipherSuites(Array("TLS_RSA_WITH_AES_256_CBC_SHA"))
  engine.setEnabledProtocols(Array("SSLv3", "TLSv1"))
  engine
}
```

`EngineProvider` creation also relies on an implicitly available `SSLContextProvider`, which is defined like this:

```
trait SSLContextProvider extends (PipelineContext Option[SSLContext])
```

The default `SSLContextProvider` simply provides an implicitly available “constant” `SSLContext`, by default the `SSLContext.getDefault` is used. This means that the easiest way to have the server use a custom `SSLContext` is to simply bring one into scope implicitly:

```
import javax.net.ssl.SSLContext

implicit val mySSLContext: SSLContext = {
  val context = SSLContext.getInstance("TLS")
  // context.init(...)
  context
}
```

5.2.5 HTTP Client APIs

Apart from the server-side HTTP abstractions *spray-can* also contains a client-side HTTP implementation that enables your application to interact with other HTTP servers. And just like on the server side it is actor-based, fully asynchronous, low-overhead and built on top of *Akka IO / spray-io*.

As the counterpart of the *HTTP Server* it shares all core features as well as the basic “low-level” philosophy with the server-side constructs.

The *spray-can* client API offers three different levels of abstraction that you can work with (from lowest to highest level):

Connection-level API

The connection-level API is the lowest-level client-side API *spray-can* provides. It gives you full control over when HTTP connections are opened and closed and when requests are to be send across which connection. As such it offers the highest flexibility at the cost of providing the least convenience.

Opening HTTP Connections

With the connection-level API you open a new HTTP connection to a given host by sending an `Http.Connect` command message to the `Http` extensions as such:

```
IO(Http) ! Http.Connect("www.spray.io", port = 8080)
```

Apart from the host name and port the `Http.Connect` message also allows you to specify socket options and a larger number of configuration settings for the connection.

Upon receipt of an `Http.Connect` message *spray-can* internally spawns a new `HttpClientConnection` actor that manages a single HTTP connection across all of its lifetime. Your code never deals with the `HttpClientConnection` actor class directly, in fact it is marked `private` to the *spray-can* package. All communication with a connection actor happens purely via actor messages, the majority of which are defined in the `spray.can.Http` object.

After a new connection actor has been started it tries to open a new TCP connection to the given endpoint and responds with an `Http.Connected` event message to the sender of the `Http.Connect` command as soon as the connection has been successfully established. If the connection could not be opened for whatever reason an `Http.CommandFailed` event is being dispatched instead and the connection actor is stopped.

Request-Response Cycle

Once the connection actor has responded with an `Http.Connected` event you can send it one or more *spray-http* `HttpRequestPart` messages. The connection actor will serialize them across the connection and wait for responses. As soon as a response for a request has been received it is dispatched as a `HttpResponsePart` instance to the sender of the respective request.

After having received a response for a request the application can decide to send another request across the same connection (i.e. to the same connection actor) or close the connection and (potentially) open a new one.

Closing Connections

Unless some kind of error (or timeout) occurs the connection actor will never actively close an established connection, even if the response contains a `Connection: close` header. The application can decide to actively close a connection by sending the connection actor one of the `Http.CloseCommand` messages described in the chapter about *Common Behavior*.

Close notification events are dispatched to the senders of all requests that still have unfinished responses pending as well as all actors that might have already sent `Http.CloseCommand` messages.

Timeouts

If no response to a request is received within the configured `request-timeout` period the connection actor closes the connection and dispatches an `Http.Closed` event message to the senders of all requests that are currently open.

If the connection is closed after the configured `idle-timeout` has expired the connection actor simply closes the connection and stops itself. If the application would like to be notified of such events it should “watch” the connection actor and react to the respective `Terminated` events (which is a good idea in any case).

In order to change the respective config setting *for this connection only* the application can send the following messages to the connection actor:

- `spray.io.ConnectionTimeouts.SetIdleTimeout`

- `spray.http.SetRequestTimeout`

Host-level API

As opposed to the *Connection-level API* the host-level API relieves you from manually opening and closing each individual HTTP connection. It autonomously manages a configurable pool of connections to *one particular server*.

Starting an `HttpHostConnector`

The core of this API is the `HttpHostConnector` actor, whose class, as with all other *spray-can* actors, you don't get in direct contact with from your application. All communication happens purely via actor messages, the majority of which are defined in the `spray.can.Http` object.

You ask *spray-can* to start a new `HttpHostConnector` for a given host by sending an `Http.HostConnectorSetup` message to the `Http` extension as such:

```
IO(Http) ! Http.HostConnectorSetup("www.spray.io", port = 80)
```

Apart from the host name and port the `Http.HostConnectorSetup` message also allows you to specify socket options and a larger number of configuration settings for the connector and the connections it is to manage.

If there is no connector actor running for the given combination of hostname, port and settings *spray-can* will start a new one, otherwise the existing one is going to be re-used. The connector will then respond with an `Http.HostConnectorInfo` event message, which repeats the connectors `ActorRef` and setup command (for easy matching against the result of an “ask”).

Using an `HttpHostConnector`

Once you've got a hold of the connectors `ActorRef` you can send it one or more *spray-http* `HttpRequestPart` messages. The connector will send the request across one of the connections it manages according to the following logic:

- if **HTTP pipelining** is not enabled (the default) the request is
 - dispatched to the first idle connection in the pool if there is one
 - dispatched to a newly opened connection if there is no idle one and less than the configured `max-connections` have been opened so far
 - queued and sent across the first connection that becomes available (i.e. either idle or unconnected) if all available connections are currently busy with open requests
- if **HTTP pipelining** is enabled the request is dispatched to
 - the first idle connection in the pool if there is one
 - a newly opened connection if there is no idle one and less than the configured `max-connections` have been opened so far
 - the connection with the least open requests if all connections already have requests open

As soon as a response for a request has been received it is dispatched as a `HttpResponsePart` instance to the sender of the respective request. If the server indicated that it doesn't want to reuse the connection for other requests (either via a `Connection: close` header on an HTTP/1.1 response or a missing `Connection: Keep-Alive` header on an HTTP/1.0 response) the connector actor closes the connection after receipt of the response thereby freeing up the “slot” for a new connection.

Retrying a Request

If the `max-retries` connector config setting is greater than zero the connector retries idempotent requests for which a response could not be successfully retrieved. Idempotent requests are those whose HTTP method is defined to be idempotent by the HTTP spec, which are all the ones currently modelled by *spray-http* except for the `PATCH` and `POST` methods.

When a response could not be received for a certain request there are essentially three possible error scenarios:

1. The request got lost on the way to the server.
2. The server experiences a problem while processing the request.
3. The response from the server got lost on the way back.

Since the host connector cannot know which one of these possible reasons caused the problem and therefore `PATCH` and `POST` requests could have already triggered a non-idempotent action on the server these requests cannot be retried.

In these cases, as well as when all retries have not yielded a proper response, the connector dispatches a `Status.Failure` message with a `RuntimeException` holding a respective error message to the sender of the request.

Connector Shutdown

The connector config contains an `idle-timeout` setting which specifies the time period after which an idle connector, i.e. one without any open connections, will automatically shut itself down. Since, by default, the connections in the connectors connection pool also have an `idle-timeout` active an unused connector will eventually be cleaned up completely if left unused.

However, in order to speed up the shutdown a host connector can be sent an `Http.CloseAll` command, which triggers an explicit closing of all connections. After all connections have been properly closed the connector will dispatch an `Http.ClosedAll` event message to all senders of `Http.CloseAll` messages before stopping itself.

A subsequent sending of an identical `Http.HostConnectorSetup` command to the `Http` extension will then trigger the creation of a fresh connector instance.

Request-level API

The request-level API is the most convenient way of using *spray-can*'s client-side. It internally builds upon the *Host-level API* to provide you with a simple and easy-to-use way of retrieving HTTP responses from remote servers.

Just send an `HttpRequest` instance to the `Http` extensions like this:

```
import scala.concurrent.Future
import scala.concurrent.duration._

import akka.actor.ActorSystem
import akka.util.Timeout
import akka.pattern.ask
import akka.io.IO

import spray.can.Http
import spray.http._
import HttpMethods._

implicit val system: ActorSystem = ActorSystem()
implicit val timeout: Timeout = Timeout(15.seconds)
import system.dispatcher // implicit execution context
```

```
val response: Future[HttpResponse] =
  (IO(Http) ? HttpRequest(GET, Uri("http://spray.io"))).mapTo[HttpResponse]

// or, with making use of spray-httpx
import spray.httpx.RequestBuilding._

val response2: Future[HttpResponse] =
  (IO(Http) ? Get("http://spray.io")).mapTo[HttpResponse]
```

The request you send to `IO(Http)` must have an absolute URI or contain a `Host` header. *spray-can* will forward it to the host connector (see *Host-level API*) for the target host (and start it up if it is not yet running).

If you want to specify config settings for either the host connector or the underlying connections that differ from what you have configured in your `application.conf` you can either “prime” a host connector by sending an explicit `Http.HostConnectorSetup` command before issuing the first request to this host or send a tuple `(Request, Http.HostConnectorSetup)` combining the request with the `Http.HostConnectorSetup` command. The latter also allows the request to have a relative URI and no host header since the target host is already specified with the connector setup command.

All other aspects of the request-level API are identical to the host-level counterpart.

Basic API Structure

Depending on the specific needs of your use case you should pick the

Connection-level API for full-control over when HTTP connections are opened/closed and how requests are scheduled across them.

Host-level API for letting *spray-can* manage a connection-pool for *one specific* host.

Request-level API for letting *spray-can* take over all connection management.

You can interact with *spray-can* on different levels at the same time and, independently of which API level you choose, *spray-can* will happily handle many thousand concurrent connections to a single or many different hosts.

Chunked Requests

While the host- and request-level APIs do not currently support chunked (streaming) HTTP requests the connection-level API does. Alternatively to a single `HttpRequest` the application can choose to send this sequence of individual messages:

- One `ChunkedRequestStart`
- Zero or more `MessageChunks`
- One `ChunkedMessageEnd`

The connection actor will render these as one logical HTTP request with `Transfer-Encoding: chunked`. The timer for checking request timeouts (if configured to non-zero) only starts running when the final `ChunkedMessageEnd` message was sent out.

Chunked Responses

Chunked (streaming) responses are supported by all three API levels. If the `response-chunk-aggregation-limit` connection config setting is set to zero the individual response parts of chunked requests are dispatched to the application as they come in. In these cases a full response consists of the following messages:

- One `ChunkedResponseStart`
- Zero or more `MessageChunks`
- One `ChunkedMessageEnd`

The timer for checking request timeouts (if configured to non-zero) will stop running as soon as the initial `ChunkedResponseStart` message has been received, i.e. there is currently no timeout checking for and in between individual response chunks.

HTTP Headers

When a *spray-can* connection actor receives an HTTP response it tries to parse all its headers into their respective *spray-http* model classes. No matter whether this succeeds or not, the connection actor will always pass on all received headers to the application. Unknown headers as well as ones with invalid syntax (according to *spray*'s header parser) will be made available as `RawHeader` instances. For the ones exhibiting parsing errors a warning message is logged depending on the value of the `illegal-header-warnings` config setting.

The following message headers are managed by the *spray-can* layer itself and as such are **ignored** if you “manually” add them to an outgoing request:

- `Content-Type`
- `Content-Length`
- `Transfer-Encoding`

There are two exceptions for requests in `ChunkedRequestStart` messages:

1. They are allowed to contain a user-specified `Content-Type` header if their entity is empty.
2. They *must* contain a user-specified `Content-Length` header if `spray.can.client.chunkless-streaming` is enabled. This `Content-Length` header *must* fit the total length of all requests chunks.

Additionally *spray-can* will render a

- Host request header if none is explicitly added.
- User-Agent default request header if none is explicitly defined. The default value can be configured with the `spray.can.client.user-agent-header` configuration setting.

Note: The `Content-Type` header has special status in *spray* since its value is part of the `HttpEntity` model class. Even though the header also remains in the `headers` list of the `HttpResponse` *sprays* higher layers (like *spray-client*) only work with the `ContentType` value contained in the `HttpEntity`.

SSL Support

SSL support is enabled

- for the connection-level API by setting `Http.Connect(sslEncryption = true)` when connecting to a server
- for the host-level API by setting `Http.HostConnectorSetup(sslEncryption = true)` when creating a host connector
- for the request-level API by using an `https` URL in the request

Particular SSL settings can be configured via the implicit `ClientSSLAuthProvider` member on the `Http.Connect` and `Http.HostConnectorSetup` command messages. An `ClientSSLAuthProvider` is essentially a function `PipelineContext Option[SSLContext]` which determines whether encryption is to be performed and, if so, which `javax.net.ssl.SSLContext` instance is to be used. By returning `None` the `ClientSSLAuthProvider` can decide to disable SSL support even if SSL support was requested by the means described above.

If you'd like to apply some custom configuration to your `SSLContext` instances an easy way would be to bring a custom engine provider into scope, e.g. like this:

```
import spray.io.ClientSSLAuthProvider

implicit val myEngineProvider = ClientSSLAuthProvider { engine =>
  engine.setEnabledCipherSuites(Array("TLS_RSA_WITH_AES_256_CBC_SHA"))
  engine.setEnabledProtocols(Array("SSLv3", "TLSv1"))
  engine
}
```

EngineProvider creation also relies on an implicitly available `SSLContextProvider`, which is defined like this:

```
trait SSLContextProvider extends (PipelineContext Option[SSLContext])
```

The default `SSLContextProvider` simply provides an implicitly available “constant” `SSLContext`, by default the `SSLContext.getDefault` is used. This means that the easiest way to have the server use a custom `SSLContext` is to simply bring one into scope implicitly:

```
import javax.net.ssl.SSLContext

implicit val mySSLContext: SSLContext = {
  val context = SSLContext.getInstance("TLS")
  // context.init(...)
  context
}
```

Redirection Following

Automatic redirection following for 3xx responses is supported by setting configuring the `spray.can.host-connector.max-redirects` setting. This is the logic that is then applied:

- If set to zero redirection responses will not be followed, i.e. they'll be returned to the user as is.
- If set to a value > zero redirection responses will be followed up to the given number of times.
- If the redirection chain is longer than the configured value the first redirection response that is not followed anymore is returned to the user as is.

By default `max-redirects` is set to 0.

Since this setting is at the host level, it is possible to configure a different number of `max-redirects` for different hosts (see *Request-level API*). In this situation the `max-redirects` configured for the host of the initial request is respected for the entire redirection chain. This is true even if redirection means changing to another host.

Which redirects are followed?

This table shows which http method is used to follow redirects for given request methods and response status codes. Any request method and response status code combination not in the table will not result in redirection following and the response will be returned as is.

Request Method	Response Status Code	Redirection Method	Specification
GET / HEAD	301 / 302 / 303	Original request method	RFC 2616
Any (except GET / HEAD)	302 / 303	GET	RFC 2616
Any	307	Original request method	HttpBis Draft
Any	308	Original request method	308 Draft

5.2.6 Common Behavior

The *spray-can HTTP Server* and *HTTP Client APIs* share a number of command and event messages that are explained in this chapter.

Closing Connections

Server- and client-side connection actors can be sent one of three defined `Http.CloseCommand` messages in order to trigger the closing of an HTTP connection. They mirror the [TCP-level commands and events from Akka IO](#) and have the following semantics:

Http.Close A “regular” close. Potentially pending unsent data are flushed to the connection before a TCP FIN is sent. The peers FIN ACK is *not* awaited. If the close is successful the sender will be notified with an `Http.Closed` event message.

Http.ConfirmedClose The closing of the connection is initially started by flushing pending writes and sending a TCP FIN to the peer. Data will continue to be received until the peer closes the connection too with its own FIN. If the close is successful the sender will be notified with an `Http.ConfirmedClosed` event message.

Http.Abort Immediately terminates the connection by sending a RST message to the peer. Pending writes are **not** flushed. If the close is successful the sender will be notified with an `Http.Aborted` event message.

In addition to the confirmation events mentioned above the connection actor will dispatch two other events derived from the `Http.ConnectionClosed` trait in certain cases:

Http.PeerClosed Dispatched when the remote peer has closed the connection without “our” side having initiated the close first.

Http.ErrorClosed Dispatched whenever an error occurred that forced the connection to be closed.

ACKed Sends

If required the server- and client-side connection actors can confirm the successful delivery of an HTTP message (part) to the OS network layer by replying with a “send ACK” message. The application can request a send ACK by modifying a message part with the `withAck` method. For example, the following handler logic receives the String “ok” as an actor message after the response has been successfully written to the connections socket:

```
def receive = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
    sender ! HttpResponse(entity = "PONG").withAck("ok")

  case "ok" => println("Response was sent successfully")
}
```

Such ACK messages are especially helpful for triggering the sending of the next message part in a request- or response streaming scenario since with such a design the application will never produce more data than the network can handle.

Send ACKs are always dispatched to the actor which sent the respective message (part). They are only supported on the server-side as well as on the client-side connection-level API (i.e. not currently on the client-side host- and request-level APIs).

5.2.7 Examples

The `/examples/spray-can/` directory of the *spray* repository contains a number of example projects for *spray-can*, which are described here.

simple-http-client

This example demonstrates how you can use the three different client-side API levels for performing a simple request/response cycle.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project simple-http-client" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project simple-http-client` and `run` sequentially "inside" of SBT.)

simple-http-server

This examples implements a very simple web-site built with the *spray-can HTTP Server*. It shows off various features like streaming, stats support and timeout handling.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project simple-http-server" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project simple-http-server` and `run` sequentially "inside" of SBT.)

4. Browse to <http://127.0.0.1:8080/>

5. Alternatively you can access the service with `curl`:

```
curl -v 127.0.0.1:8080/ping
```

6. Stop the service with:

```
curl -v 127.0.0.1:8080/stop
```

server-benchmark

This example implements a very simple “ping/pong” server for benchmarking purposes, that mirrors the “JSON serialization” test setup from the techempower benchmark.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project server-benchmark" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project server-benchmark` and `run` sequentially “inside” of SBT.)

4. Use a load-generation tool like `ab`, `weighttp`, `wrk` or the like to fire test requests, e.g.:

```
wrk -t4 -c100 -d10 http://127.0.0.1:8080/ping
```

If you start the server with `re-start` rather than `run` it will run in a forked JVM that has `-verbose:gc` and `-XX:+PrintCompilation` flags set, so you can see how often GC is performed and whether the JIT compiler is “done” with compiling all the hot spots.

5.3 spray-client

spray-client provides high-level HTTP client functionality by adding another logic layer on top of the relatively basic *spray-can HTTP Client APIs*. It doesn't yet provide all the features that we'd like to include eventually, but it should already be of some utility for many applications.

Currently it allows you to wrap any one of the three *spray-can* client-side API levels with a pipelining logic, which provides for:

- Convenient request building
- Authentication
- Compression / Decompression
- Marshalling / Unmarshalling from and to your custom types

Currently, HTTP streaming (i.e. chunked transfer encoding) is not yet supported on the *spray-client* level (even though the underlying *spray-can HTTP Client APIs* do support it (the host- and request-level APIs only for responses)), i.e. you cannot send chunked requests and the `response-chunk-aggregation-limit` config setting for the underlying transport must be non-zero).

5.3.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-client* depends on

- *spray-can*
- *spray-http*

- *spray-httpx*
- *spray-util*
- akka-actor 2.1.x (with 'provided' scope, i.e. you need to pull it in yourself)

5.3.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-client* into your classpath.

5.3.3 Usage

The simplest of all use cases is this:

```
import spray.http._
import spray.client.pipelining._

implicit val system = ActorSystem()
import system.dispatcher // execution context for futures

val pipeline: HttpRequest => Future[HttpResponse] = sendReceive

val response: Future[HttpResponse] = pipeline(Get("http://spray.io/"))
```

The central element of a *spray-client* pipeline is `sendReceive`, which produces a function `HttpRequest => Future[HttpResponse]` (this function type is also aliased to `SendReceive`). When called without parameters `sendReceive` will automatically use the `IO(Http)` extension of an implicitly available `ActorSystem` to access the *spray-can Request-level API*. All requests must therefore either carry an absolute URI or an explicit `Host` header.

In order to wrap pipelining around *spray-can*'s *Host-level API* you need to tell `sendReceive` which host connector to use:

```
import akka.io.IO
import akka.pattern.ask
import spray.can.Http
import spray.http._
import spray.client.pipelining._

implicit val system = ActorSystem()
import system.dispatcher // execution context for futures

val pipeline: Future[SendReceive] =
  for (
    Http.HostConnectorInfo(connector, _) <-
      IO(Http) ? Http.HostConnectorSetup("www.spray.io", port = 80)
  ) yield sendReceive(connector)

val request = Get("/")
val response: Future[HttpResponse] = pipeline.flatMap(_(request))
```

You can then fire requests with relative URIs and without `Host` header into the pipeline.

A pipeline of type `HttpRequest => Future[HttpResponse]` is nice start but leaves the creation of requests and interpretation of responses completely to you. Many times you actually want to send and/or receive custom objects that need to be serialized to HTTP requests or deserialized from HTTP responses. Check out this snippet for an example of what *spray-client* pipelining can do for you in that regard:

```

import spray.http._
import spray.json.DefaultJsonProtocol
import spray.httpx.encoding.{Gzip, Deflate}
import spray.httpx.SprayJsonSupport._
import spray.client.pipelining._

case class Order(id: Int)
case class OrderConfirmation(id: Int)

object MyJsonProtocol extends DefaultJsonProtocol {
  implicit val orderFormat = jsonFormat1(Order)
  implicit val orderConfirmationFormat = jsonFormat1(OrderConfirmation)
}
import MyJsonProtocol._

implicit val system = ActorSystem()
import system.dispatcher // execution context for futures

val pipeline: HttpRequest => Future[OrderConfirmation] = (
  addHeader("X-My-Special-Header", "fancy-value")
  ~> addCredentials(BasicHttpCredentials("bob", "secret"))
  ~> encode(Gzip)
  ~> sendReceive
  ~> decode(Deflate)
  ~> unmarshal[OrderConfirmation]
)

val response: Future[OrderConfirmation] =
  pipeline(Post("http://example.com/orders", Order(42)))

```

This defines a more complex pipeline that takes an `HttpRequest`, adds headers and compresses its entity before dispatching it to the target server (the `sendReceive` element of the pipeline). The response coming back is then decompressed and its entity unmarshalled.

When you import `spray.client.pipelining._` you not only get easy access to `sendReceive` but also all elements of the *spray-httpx Request Building* and *Response Transformation* traits. Therefore you can easily create requests via something like `Post("/orders", Order(42))`, which is not only shorter but also provides for automatic marshalling of custom types.

5.3.4 Example

The `/examples/spray-client/` directory of the *spray* repository contains an example project for *spray-client*.

simple-spray-client

This example shows off how to use *spray-client* by querying Google's Elevation API to retrieve the elevation of Mt. Everest.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project simple-spray-client" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project simple-spray-client` and `run` sequentially "inside" of SBT.)

5.4 spray-http

The *spray-http* module contains a fully immutable, case-class based model of the major HTTP data structures, like HTTP requests, responses and common headers. It also includes a parser for the latter, which is able to construct the more structured header models from raw unstructured header name/value pairs.

5.4.1 Dependencies

spray-http depends on *akka-actor* (with 'provided' scope, i.e. you need to pull it in yourself). It also depends on *parboiled*, a lightweight PEG parsing library providing the basis for the header parser. Since *parboiled* is also written and maintained by the members of the *spray* team it's not an "outside" dependency that we have no control over.

5.4.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-http* into your classpath.

Afterwards just `import spray.http._` to bring all relevant identifiers into scope.

5.4.3 Overview

Since *spray-http* provides the central HTTP data structures for *spray* you will find the following import in quite a few places around the *spray* code base (and probably your own code as well):

```
import spray.http._
```

This brings in scope all of the relevant things that are defined [here](#) and that you'll want to work with, mainly:

- `HttpRequest` and `HttpResponse`, the central message models
- `ChunkedRequestStart`, `ChunkedResponseStart`, `MessageChunk` and `ChunkedMessageEnd` modeling the different message parts of request/response streams
- `HttpHeaders`, an object containing all the defined HTTP header models
- Supporting types like `Uri`, `HttpMethods`, `MediaTypes`, `StatusCodes`, etc.

A common pattern is that the model of a certain entity is represented by an immutable type (class or trait), while the actual instances of the entity defined by the HTTP spec live in an accompanying object carrying the name of the type plus a trailing 's'.

For example:

- The defined `HttpMethod` instances live in the `HttpMethods` object.
- The defined `HttpCharset` instances live in the `HttpCharsets` object.
- The defined `HttpEncoding` instances live in the `HttpEncodings` object.
- The defined `HttpProtocol` instances live in the `HttpProtocols` object.

- The defined `MediaType` instances live in the `MediaTypes` object.
- The defined `StatusCode` instances live in the `StatusCodes` object.

You get the point.

In order to develop a better understanding for how *spray* models HTTP you probably should take some time to browse around the [spray-http sources](#) (ideally with an IDE that supports proper code navigation).

5.4.4 Content-Type Header

One thing worth highlighting is the special treatment of the HTTP `Content-Type` header. Since the binary content of HTTP message entities can only be properly interpreted when the corresponding content-type is known *spray-http* puts the content-type value very close to the entity data. The `HttpEntity.NonEmpty` type (the non-empty variant of the `HttpEntity`) is essentially little more than a tuple of the `ContentType` and the entity's bytes. All logic in *spray* that needs to access the content-type of an HTTP message always works with the `ContentType` value in the `HttpEntity`. Potentially existing instances of the `Content-Type` header in the `HttpMessage`'s header list are ignored!

5.4.5 Custom Media-Types

spray-http defines the most important media types from the [IANA MIME media type registry](#) in the `MediaTypes` object, which also acts as a registry that you can register your own `CustomMediaType` instances with:

```
import spray.http.MediaTypes._

val MarkdownType = register(
  MediaType.custom(
    mainType = "text",
    subType = "x-markdown",
    compressible = true,
    binary = false,
    fileExtensions = Seq("markdown", "mdown", "md")))
```

Once registered the custom type will be properly resolved, e.g. for incoming requests by *spray-routing* or incoming responses by *spray-client*. File extension resolution (as used for example by the *FileAndResourceDirectives*) will work as expected.

5.5 spray-httpx

The *spray-httpx* module contains all higher-level logic for working with HTTP messages, which is not specific to either the server-side (*spray-routing*) or client-side (*spray-client*) modules on top and therefore (potentially) used by both of them.

5.5.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-httpx* depends on

- *spray-http*
- *spray-util*
- *spray-io* (only required until the upgrade to Akka 2.2, will go away afterwards)

- MIME pull
- akka-actor 2.1.x (with ‘provided’ scope, i.e. you need to pull it in yourself)
- Optionally (you need to provide these if you’d like to use the respective *spray-httpx* feature):
 - `spray-json` (for `SprayJsonSupport`)
 - `lift-json` (for `LiftJsonSupport`)
 - `twirl-api` (for `TwirlSupport`)
 - `json4s-native` (for `Json4sSupport`)
 - `json4s-jackson` (for `Json4sJacksonSupport`)

5.5.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-httpx* into your classpath.

Afterwards you can use the following imports to bring all relevant identifiers into scope:

- `import spray.httpx.encoding._` for everything related to (de)compression
- `import spray.httpx.marshalling._` for everything related to marshalling
- `import spray.httpx.unmarshalling._` for everything related to unmarshalling
- `import spray.httpx.RequestBuilding` for `RequestBuilding`
- `import spray.httpx.ResponseTransformation` for `ResponseTransformation`
- `import spray.httpx.Json4sJacksonSupport` for `Json4sJacksonSupport`
- `import spray.httpx.Json4sSupport` for `Json4sSupport`
- `import spray.httpx.LiftJsonSupport` for `LiftJsonSupport`
- `import spray.httpx.SprayJsonSupport` for `SprayJsonSupport`
- `import spray.httpx.TwirlSupport` for `TwirlSupport`

5.5.3 Marshalling

“Marshalling” is the process of converting a higher-level (object) structure into some kind of lower-level representation, often a “wire format”. Other popular names for it are “Serialization” or “Pickling”.

In *spray* “Marshalling” means the conversion of an object of type `T` into an `HttpEntity`, which forms the “entity body” of an HTTP request or response (depending on whether used on the client or server side).

Marshalling for instances of type `T` is performed by a `Marshaller[T]`, which is defined like this:

```
trait Marshaller[-T] {  
  def apply(value: T, ctx: MarshallingContext)  
}
```

So, a `Marshaller` is not a plain function `T => HttpEntity`, as might be initially expected. Rather it uses the given `MarshallingContext` to drive the marshalling process from its own side. There are three reasons why *spray* `Marshaller`s are designed in this way:

- Marshalling on the server-side must support `content negotiation`, which is easier to implement if the marshaller drives the process.

- Marshallers can delay their actions and complete the marshalling process from another thread at another time (e.g. when the result of a `Future` arrives), which is not something that ordinary functions can do. (We could have the `Marshaller` return a `Future`, but this would add overhead to the majority of cases that do not require delayed execution.)
- Marshallers can produce more than one response part, whereby the sequence of response chunks is available as a pull-style stream or from a push-style producer. Both these approaches need to be supported.

Default Marshallers

spray-httpx comes with pre-defined Marshallers for the following types:

- **BasicMarshallers**
 - `Array[Byte]`
 - `Array[Char]`
 - `String`
 - `NodeSeq`
 - `Throwable`
 - `spray.http.FormData`
 - `spray.http.HttpEntity`
- **MetaMarshallers**
 - `Option[T]`
 - `Either[A, B]`
 - `Try[T]`
 - `Future[T]`
 - `Stream[T]`
- **MultipartMarshallers**
 - `spray.http.MultipartContent`
 - `spray.http.MultipartFormData`

Implicit Resolution

Since the marshalling infrastructure uses a `type class` based approach `Marshaller` instances for a type `T` have to be available implicitly. The implicits for all the default Marshallers defined by *spray-httpx* are provided through the companion object of the `Marshaller` trait. This means that they are always available and never need to be explicitly imported. Additionally, you can simply “override” them by bringing your own custom version into local scope.

Custom Marshallers

spray-httpx gives you a few convenience tools for constructing Marshallers for your own types. One is the `Marshaller.of helper`, which is defined as such:

```
def of[T](marshalTo: ContentType*)
  (f: (T, ContentType, MarshallingContext) => Unit): Marshaller[T]
```

The default StringMarshaller for example is defined with it:

```
// prefer UTF-8 encoding, but also render with other encodings if the client requests them
implicit val StringMarshaller = stringMarshaller(ContentType.`text/plain(UTF-8)`, ContentType.`text/plain`)

def stringMarshaller(contentType: ContentType, more: ContentType*): Marshaller[String] =
  Marshaller.of[String](contentType +: more: _*) { (value, contentType, ctx) =>
    ctx.marshallTo(HttpEntity(contentType, value))
  }
```

As another example, here is a Marshaller definition for a custom type Person:

```
import spray.http._
import spray.httpx.marshalling._

val `application/vnd.acme.person` =
  MediaType.register(MediaType.custom("application/vnd.acme.person"))

case class Person(name: String, firstName: String, age: Int)

object Person {
  implicit val PersonMarshaller =
    Marshaller.of[Person](`application/vnd.acme.person`) { (value, contentType, ctx) =>
      val Person(name, first, age) = value
      val string = "Person: %s, %s, %s".format(name, first, age)
      ctx.marshallTo(HttpEntity(contentType, string))
    }
}

marshal(Person("Bob", "Parr", 32)) ===
  Right(HttpEntity(`application/vnd.acme.person`, "Person: Bob, Parr, 32"))
```

As can be seen in this example you best define the Marshaller for T in the companion object of T. This way your marshaller is always in-scope, without any import tax.

Deriving Marshallers

Sometimes you can save yourself some work by reusing existing Marshallers for your custom ones. The idea is to “wrap” an existing Marshaller with some logic to “re-target” it to your type.

In this regard wrapping a Marshaller can mean one or both of the following two things:

- Transform the input before it reaches the wrapped Marshaller
- Transform the output of the wrapped Marshaller

You can do both, but the existing support infrastructure favors the first over the second. The `Marshaller.delegate` helper allows you to turn a `Marshaller[B]` into a `Marshaller[A]` by providing a function `A => B`:

```
def delegate[A, B](marshalTo: ContentType*)
  (f: A => B)
  (implicit mb: Marshaller[B]): Marshaller[A]
```

This is used, for example, by the `NodeSeqMarshaller`, which delegates to the `StringMarshaller` like this:

```
implicit val NodeSeqMarshaller =
  Marshaller.delegate[NodeSeq, String](`text/xml`, `application/xml`,
    `text/html`, `application/xhtml+xml`)(_.toString)
```

There is also a second overload of the `delegate` helper that takes a function `(A, ContentType) => B` rather than a function `A => B`. It's helpful if your input conversion requires access to the `ContentType` that is marshalled to.

If you want the second wrapping type, transformation of the output, things are a bit harder (and less efficient), since `Marshallers` produce `HttpEntities` rather than `Strings`. An `HttpEntity` contains the *serialized* result, which is essentially an `Array[Byte]` and a `ContentType`. So, for example, prepending a string to the output of the underlying `Marshaller` would entail deserializing the bytes into a string, prepending your prefix and reserializing into a byte array... not pretty and quite inefficient. Nevertheless, you can do it. Just produce a custom `MarshallingContext`, which wraps the original one with custom logic, and pass it to the inner `Marshaller`. However, a general solution would also require you to think about the handling of chunked responses, errors, etc.

Because the second form of wrapping is less attractive there is no real helper infrastructure for it. We generally do not want to encourage such type of design. (With one exception: Simply overriding the `Content-Type` of another `Marshaller` can be done efficiently. This is why the `MarshallingContext` already comes with a `withContentTypeOverriding` copy helper.)

ToResponseMarshaller

The plain `Marshaller[T]` is agnostic to whether it is used on the server- or on the client-side. This means that it can be used to produce the entities (and additional headers) for responses as well as requests.

Sometimes, however, this is not enough. If you know that you need to only marshal to `HttpResponse` instances (e.g. because you only use *spray* on the server-side) you can also write a `ToResponseMarshaller[T]` for your type. This more specialized marshaller allows you to produce the complete `HttpResponse` instance rather than only its entity. As such the marshaller can also set the status code of the response (which doesn't exist on the request side).

When looking for a way to marshal a custom type `T` *spray* (or rather the Scala compiler) first looks for a `ToResponseMarshaller[T]` for the type. Only if none is found will an in-scope `Marshaller[T]` be used.

5.5.4 Unmarshalling

“Unmarshalling” is the process of converting some kind of a lower-level representation, often a “wire format”, into a higher-level (object) structure. Other popular names for it are “Deserialization” or “Unpickling”.

In *spray* “Unmarshalling” means the conversion of an `HttpEntity`, the model class for the entity body of an HTTP request or response (depending on whether used on the client or server side), into an object of type `T`.

Unmarshalling for instances of type `T` is performed by an `Unmarshaller[T]`, which is defined like this:

```
type Unmarshaller[T] = Deserializer[HttpEntity, T]
trait Deserializer[A, B] extends (A => Deserialized[B])
type Deserialized[T] = Either[DeserializationError, T]
```

So, an `Unmarshaller` is basically a function `HttpEntity => Either[DeserializationError, T]`. When compared to their counterpart, *Marshallers*, `Unmarshallers` are somewhat simpler, since they are straight functions and do not have to deal with chunk streams (which are currently not supported in unmarshalling) or delayed execution.)

Default Unmarshallers

spray-httpx comes with pre-defined `Unmarshallers` for the following types:

- `Array[Byte]`

- `Array[Char]`
- `String`
- `NodeSeq`
- `Option[T]`
- `spray.http.FormData`
- `spray.http.HttpForm`
- `spray.http.MultipartContent`
- `spray.http.MultipartFormData`

The relevant sources are:

- `Deserializer`
- `BasicUnmarshallers`
- `MetaUnmarshallers`
- `FormDataUnmarshallers`

Implicit Resolution

Since the unmarshalling infrastructure uses a `type class` based approach `Unmarshaller` instances for a type `T` have to be available implicitly. The implicits for all the default `Unmarshallers` defined by `spray-httpx` are provided through the companion object of the `Deserializer` trait (since `Unmarshaller[T]` is just an alias for a `Deserializer[HttpEntity, T]`). This means that they are always available and never need to be explicitly imported. Additionally, you can simply “override” them by bringing your own custom version into local scope.

Custom Unmarshallers

`spray-httpx` gives you a few convenience tools for constructing `Unmarshallers` for your own types. One is the `Unmarshaller.apply` helper, which is defined as such:

```
def apply[T](unmarshalFrom: ContentTypeRange*)(
  f: PartialFunction[HttpEntity, T]): Unmarshaller[T]
```

The default `NodeSeqUnmarshaller` for example is defined with it:

```
implicit val NodeSeqUnmarshaller =
  Unmarshaller[NodeSeq](`text/xml`, `application/xml`, `text/html`, `application/xhtml+xml`) {
    case HttpEntity.NonEmpty(contentType, data)
      XML.withSAXParser(createSAXParser())
        .load(new InputStreamReader(new ByteArrayInputStream(data.toByteArray), contentType.charset))
    case HttpEntity.Empty NodeSeq.Empty
  }
```

As another example, here is an `Unmarshaller` definition for a custom type `Person`:

```
import spray.httpx.unmarshalling._
import spray.util._
import spray.http._

val `application/vnd.acme.person` =
  MediaTypes.register(MediaType.custom("application/vnd.acme.person"))
```

```

case class Person(name: String, firstName: String, age: Int)

object Person {
  implicit val PersonUnmarshaller =
    Unmarshaller[Person](`application/vnd.acme.person`) {
      case HttpEntity.NonEmpty(contentType, data) =>
        // unmarshal from the string format used in the marshaller example
        val Array(_, name, first, age) =
          data.asString.split(":".toCharArray).map(_.trim)
        Person(name, first, age.toInt)

        // if we had meaningful semantics for the HttpEntity.Empty
        // we could add a case for the HttpEntity.Empty:
        // case HttpEntity.Empty => ...
    }
}

val body = HttpEntity(`application/vnd.acme.person`, "Person: Bob, Parr, 32")
body.as[Person] === Right(Person("Bob", "Parr", 32))

```

As can be seen in this example you best define the Unmarshaller for T in the companion object of T. This way your unmarshaller is always in-scope, without any import tax.

Deriving Unmarshallers

Unmarshaller.delegate

Sometimes you can save yourself some work by reusing existing Unmarshallers for your custom ones. The idea is to “wrap” an existing Unmarshaller with some logic to “re-target” it to your type.

In this regard “wrapping” a Unmarshaller can mean one or both of the following two things:

- Transform the input `HttpEntity` before it reaches the wrapped Unmarshaller
- Transform the output of the wrapped Unmarshaller

You can do both, but the existing support infrastructure favors the latter over the former. The `Unmarshaller.delegate` helper allows you to turn an `Unmarshaller[A]` into an `Unmarshaller[B]` by providing a function `A => B`:

```

def delegate[A, B](unmarshalFrom: ContentTypeRange*)(
  f: A => B)
  (implicit mb: Unmarshaller[A]): Unmarshaller[B]

```

For example, by using `Unmarshaller.delegate` the `Unmarshaller[Person]` from the example above could be simplified to this:

```

implicit val SimplerPersonUnmarshaller =
  Unmarshaller.delegate[String, Person](`application/vnd.acme.person`) { string =>
    val Array(_, name, first, age) = string.split(":".toCharArray).map(_.trim)
    Person(name, first, age.toInt)
  }

```

Unmarshaller.forNonEmpty

In addition to `Unmarshaller.delegate` there is also another “deriving Unmarshaller builder” called `Unmarshaller.forNonEmpty`. It “modifies” an existing Unmarshaller to not accept empty entities.

For example, the default `NodeSeqMarshaller` (see above) accepts empty entities as a valid representation of `NodeSeq.Empty`. It might be, however, that in your application context empty entities are not allowed. In order to achieve this, instead of “overriding” the existing `NodeSeqMarshaller` with an all-custom re-implementation you could be doing this:

```
implicit val myNodeSeqUnmarshaller = Unmarshaller.forNonEmpty [NodeSeq]

HttpEntity(MediaTypes.`text/xml`, "<xml>yeah</xml>").as[NodeSeq] === Right (<xml>yeah</xml>)
HttpEntity.Empty.as[NodeSeq] === Left (ContentExpected)
```

More specific Unmarshallers

The plain `Unmarshaller[T]` is agnostic to whether it is used on the server- or on the client-side. This means that it can be used to deserialize the entities from requests as well as responses. Also, the only information that an `Unmarshaller[T]` has access to for its job is the message entity. Sometimes this is not enough.

FromMessageUnmarshaller

If you need access to the message headers during unmarshalling you can write an `FromMessageUnmarshaller[T]` for your type. It is defined as such:

```
type FromMessageUnmarshaller[T] = Deserializer[HttpMessage, T]
```

and allows access to all members of the `HttpMessage` superclass of the `HttpRequest` and `HttpResponse` types, most importantly: the message headers. Since, like the plain `Unmarshaller[T]`, it can deserialize requests as well as responses it can be used on the server- as well as the client-side.

An in-scope `FromMessageUnmarshaller[T]` takes precedence before any potentially available plain `Unmarshaller[T]`.

FromRequestUnmarshaller

The `FromRequestUnmarshaller[T]` is the most “powerful” unmarshaller that can be used on the server-side (and only there). It is defined like this:

```
type FromRequestUnmarshaller[T] = Deserializer[HttpRequest, T]
```

and allows access to all members of the incoming `HttpRequest` instance.

An in-scope `FromRequestUnmarshaller[T]` takes precedence before any potentially available `FromMessageUnmarshaller[T]` or plain `Unmarshaller[T]`.

FromResponseUnmarshaller

The `FromResponseUnmarshaller[T]` is the most “powerful” unmarshaller that can be used on the client-side (and only there). It is defined like this:

```
type FromResponseUnmarshaller[T] = Deserializer[HttpResponse, T]
```

and allows access to all members of the incoming `HttpResponse` instance.

An in-scope `FromResponseUnmarshaller[T]` takes precedence before any potentially available `FromMessageUnmarshaller[T]` or plain `Unmarshaller[T]`.

5.5.5 (De)compression

The `HTTP spec` defines a `Content-Encoding` header, which signifies whether the entity body of an HTTP message is “encoded” and, if so, by which algorithm. The only commonly used content encodings, apart from `identity` (i.e. plain text), are compression algorithms.

Currently `spray` supports the compression and decompression of HTTP requests and responses with the `gzip` or `deflate` encodings. The core logic for this, which is shared by the `spray-client` and `spray-routing` modules for the client- and server-side (respectively), lives in the `spray.httpx.encoding` package.

The support is not enabled by default, but must be explicitly requested. For server configuration, see *When to use which decompression directive?*. For client configuration, see `spray.client.pipelining.decode` and `spray.httpx.ResponseTransformation`.

Compression of Chunk Streams

Properly combining HTTP compression with the chunked HTTP/1.1 Transfer-Encoding can be a little tricky. For optimal results the peer sending the message (i.e. the client or the server) should use a single compression context across all chunks, so that common patterns shared by several chunks contribute to a high compression ratio. At the same time the decompressor at the other end must be able to properly decompress each chunk as it arrives.

In order to achieve this the compressor must properly flush its compression stream after each chunk, something that the GZIP- and `DeflaterOutputStream` implementations of the Java 6 JDK unfortunately do not support correctly (see [this JDK bug](#), fixed only in Java 7). `sprays` compression implementation jumps through a few hoops to achieve the desired behavior also under Java 6, with no cost to you as the user.

5.5.6 Request Building

When you work with `spray` you’ll occasionally want to construct HTTP requests, e.g. when talking to an HTTP server with `spray-client` or when writing tests for your server-side API with `spray-testkit`.

For making request construction more convenient `spray-httpx` provides the `RequestBuilding` trait, that defines a simple DSL for assembling HTTP requests in a concise and readable manner.

Take a look at these examples:

```
import spray.httpx.RequestBuilding._
import spray.http._
import HttpMethods._
import HttpHeaders._
import ContentType._

// simple GET requests
Get() === HttpRequest(method = GET)
Get("/abc") === HttpRequest(method = GET, uri = "/abc")

// as second argument you can specify an object that is
// to be marshalled using the in-scope marshaller for the type
Put("/abc", "foobar") === HttpRequest(method = PUT, uri = "/abc", entity = "foobar")

implicit val intMarshaller = Marshaller.of[Int](`application/json`) {
  (value, ct, ctx) => ctx.marshallTo(HttpEntity(ct, s"{ value: $value }"))
}
Post("/int", 42) === HttpRequest(method = POST, uri = "/int",
  entity = HttpEntity(`application/json`, "{ value: 42 }"))

// add one or more headers by chaining in the `addHeader` modifier
```

```
Patch("/abc", "content") ~> addHeader("X-Yeah", "Naah") === HttpRequest (
  method = PATCH,
  uri = "/abc",
  entity = "content",
  headers = List(RawHeader("X-Yeah", "Naah"))
)
```

5.5.7 Response Transformation

The counterpart to *Request Building* is the `ResponseTransformation` trait, which is especially useful on the client-side when you want to transform an incoming HTTP response in a number of loosely coupled steps into some kind of higher-level result type (see also *spray-client*).

Just like with `RequestBuilding` the `ResponseTransformation` trait gives you the `~>` operator, which allows you to “append” a transformation function onto an existing function producing an `HttpResponse`. Thereby it doesn’t matter whether the result is a plain response or a response wrapped in a `Future`.

For example, if you have a function:

```
import scala.concurrent.ExecutionContext.Implicits.global // for futures

val sendReceive: HttpRequest => Future[HttpResponse] = // ...
```

and a “response transformer”:

```
val removeCookieHeaders: HttpResponse => HttpResponse =
  r => r.withHeaders(r.headers.filter(_.isNot("set-cookie")))
```

you can use the `~>` operator to combine the two:

```
import spray.httpx.ResponseTransformation._

val pipeline: HttpRequest => Future[HttpResponse] =
  sendReceive ~> removeCookieHeaders
```

More generally the `~>` operator combines functions in the following ways:

X	Y	X ~> Y
A => B	B => C	A => C
A => Future[B]	B => C	A => Future[C]
A => Future[B]	B => Future[C]	A => Future[C]

Predefined Response Transformers

decode(decoder: Decoder): HttpResponse `HttpResponse` Decodes a response using the given `Decoder` (`Gzip` or `Deflate`).

unmarshal[T: Unmarshaller]: HttpResponse `T` Unmarshalls the response to a custom type using the in-scope `Unmarshaller[T]`.

logResponse(...): HttpResponse `HttpResponse` Doesn’t actually change the response but simply logs it.

5.5.8 spray-json Support

The `SprayJsonSupport` trait provides a `Marshaller` and `Unmarshaller` for every type `T` that an implicit `spray.json.RootJsonReader` and/or `spray.json.RootJsonWriter` (respectively) is available for.

Just mix in `spray.httpx.SprayJsonSupport` or import `spray.httpx.SprayJsonSupport._`.

For example:

```
import spray.json.DefaultJsonProtocol
import spray.httpx.unmarshalling._
import spray.httpx.marshalling._
import spray.http._
import HttpCharsets._
import MediaTypes._

case class Person(name: String, firstName: String, age: Int)

object MyJsonProtocol extends DefaultJsonProtocol {
  implicit val PersonFormat = jsonFormat3(Person)
}

import MyJsonProtocol._
import spray.httpx.SprayJsonSupport._
import spray.util._

val bob = Person("Bob", "Parr", 32)
val body = HttpEntity(
  contentType = ContentType(`application/json`, `UTF-8`),
  string =
    """|{
      |  "name": "Bob",
      |  "firstName": "Parr",
      |  "age": 32
      |}""".stripMarginWithNewline("\n")
)

marshal(bob) === Right(body)
body.as[Person] === Right(bob)
```

If you bring an implicit `spray.json.JsonPrinter` into scope the marshaller will use it. Otherwise it uses the default `spray.json.PrettyPrinter`.

Note: Since *spray-httpx* only comes with an optional dependency on *spray-json* you still have to add it to your project yourself. Check the *spray-json* documentation for information on how to do this.

5.5.9 lift-json Support

In analogy to the *spray-json Support* *spray-httpx* also provides the `LiftJsonSupport` trait, which automatically provides `implicitMarshaller` and `Unmarshaller` instances for *all* types if an implicit `net.liftweb.json.Formats` instance is in scope.

When mixing in `LiftJsonSupport` you have to implement the abstract member:

```
implicit def liftJsonFormats: Formats
```

with your custom logic.

Note: Since *spray-httpx* only comes with an optional dependency on *lift-json* you still have to add it to your project yourself. Check the *lift-json* documentation for information on how to do this.

5.5.10 json4s Support

In analogy to the *spray-json Support* *spray-httpx* also provides the `Json4sSupport` and `Json4sJacksonSupport` traits, which automatically provide implicit `Marshaller` and `Unmarshaller` instances for *all* types if an implicit `org.json4s.Formats` instance is in scope.

When mixing in either one of the two traits you have to implement the abstract member:

```
implicit def json4sFormats: Formats
```

with your custom logic. See the [json4s documentation](#) for more information on how to do this.

Note: Since *spray-httpx* only comes with an optional dependency on `json4s-native` and `json4s-jackson` you still have to add either one of them to your project yourself. Check the [json4s documentation](#) for information on how to do this.

5.5.11 Twirl Support

Twirl complements *spray* with templating support.

The `TwirlSupport` trait provides the tiny glue layer required for being able to use twirl templates directly in *spray-routing* routes and *request building*.

Simply mix in the `TwirlSupport` trait or import `spray.httpx.TwirlSupport._`.

Note: Since *spray-httpx* only comes with an optional dependency on `twirl` you still have to add it to your project yourself. Check the [twirl documentation](#) for information on how to do this.

Side Note

This site, for example, makes use of twirl-templates and `TwirlSupport` for serving all of its [pages](#).

5.6 spray-io

Up to release 1.0/1.1-M7 the *spray-io* module provided a low-level network I/O layer for directly connecting Akka actors to asynchronous Java NIO sockets. Since then the *spray* and Akka teams have [joined forces](#) to build upon the work in *spray* and come up with an extended and improved implementation, which lives directly in Akka as of Akka 2.2.

Over time more and more things that were previously provided by *spray-io* (e.g. the pipelining infrastructure and the SSL/TLS support) have found their way, in an improved form, from the *spray* codebase into the Akka codebase, so that *spray's* own IO module will cease to exist in the near future.

For *spray* 1.0 and 1.1 the *spray-io* module contains a direct **back-port** of the new Akka-I/O infrastructure from Akka 2.2 to Akka 2.0 and Akka 2.1 (respectively). This means that users of Akka 2.0 and Akka 2.1 can enjoy the benefits of the new I/O layer even without upgrading by simply including *spray-io* as a dependency.

All documentation for the new I/O layer can be found in the [docs to Akka 2.2](#), namely:

- [Introduction](#)
- [I/O Layer Design](#)

- TCP Support
- UDP Support
- Pipeline Infrastructure

5.6.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-io* only depends on *spray-util* and *akka-actor* (with ‘provided’ scope, i.e. you need to pull it in yourself).

5.6.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-io* into your classpath. Just as in Akka 2.2 all the (backported) components of the new I/O layer live in the `akka.actor.io` package.

5.7 spray-routing

The *spray-routing* module provides a high-level, very flexible routing DSL for elegantly defining RESTful web services. Normally you would use it either on top of a *spray-can HTTP Server* or inside of a servlet container together with *spray-servlet*.

5.7.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-routing* depends on

- *spray-http*
- *spray-httpx*
- *spray-util*
- *spray-io* (optionally, required for `SimpleRoutingApp`)
- *spray-can* (optionally, required for `SimpleRoutingApp`)
- *spray-caching* (optionally, required for `CachingDirectives` and `CachedUserPassAuthenticator`)
- `shapeless (1.2.x)`
- `akka-actor 2.1.x` (with ‘provided’ scope, i.e. you need to pull it in yourself)

5.7.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-can* into your classpath.

5.7.3 Configuration

Just like Akka *spray-routing* relies on the `typesafe config` library for configuration. As such its JAR contains a `reference.conf` file holding the default values of all configuration settings. In your application you typically provide an `application.conf`, in which you override Akka and/or *spray* settings according to your needs.

Note: Since *spray* uses the same configuration technique as Akka you might want to check out the [Akka Documentation on Configuration](#).

This is the reference .conf of the *spray-routing* module:

```
#####
# spray-routing Reference Config File #
#####

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

spray.routing {

  # Enables/disables the returning of more detailed error messages to the
  # client in the error response
  # Should be disabled for browser-facing APIs due to the risk of XSS attacks
  # and (probably) enabled for internal or non-browser APIs
  # (Note that spray will always produce log messages containing the full error details)
  verbose-error-messages = off

  # the minimal file size triggering file content streaming
  # set to zero to disable automatic file-chunking in the FileAndResourceDirectives
  file-chunking-threshold-size = 128k

  # the size of an individual chunk when streaming file content
  file-chunking-chunk-size = 128k

  # Enables/disables ETag and `If-Modified-Since` support for FileAndResourceDirectives
  file-get-conditional = on

  # Enables/disables the rendering of the "rendered by" footer in directory listings
  render-verity-footer = yes

  # a config section holding plain-text user/password entries
  # for the default FromConfigUserPassAuthenticator
  users {
    # bob = secret
  }

  # the maximum size between two requested ranges.
  # Ranges with less space in between will be coalesced.
  range-coalescing-threshold = 80

  # the maximum number of allowed ranges per request.
  # Requests with more ranges will be rejected due to DOS suspicion.
  range-count-limit = 16
}
```

5.7.4 Getting Started

Check out the *Introduction / Getting Started* chapter for information about the template project you can use to quickly bootstrap your own *spray-routing* application.

SimpleRoutingApp

spray-routing also comes with the `SimpleRoutingApp` trait, which you can use as a basis for your first *spray* endeavours. It reduces the boilerplate to a minimum and allows you to focus entirely on your route structure.

Just use this minimal example application as a starting point:

```
import spray.routing.SimpleRoutingApp

object Main extends App with SimpleRoutingApp {
  implicit val system = ActorSystem("my-system")

  startServer(interface = "localhost", port = 8080) {
    path("hello") {
      get {
        complete {
          <h1>Say hello to spray</h1>
        }
      }
    }
  }
}
```

This very concise way of bootstrapping a *spray-routing* application works nicely as long as you don't have any special requirements with regard to the actor which is running your route structure. Once you need more control over it, e.g. because you want to be able to use it as the receiver (or sender) of custom messages, you'll have to fall back to creating your service actor "manually". The *Complete Examples* demonstrate how to do that.

5.7.5 Key Concepts

We think that understanding the concepts presented in this chapter are crucial to being able to use *spray-routing* effectively:

Big Picture

The *spray-can HTTP Server* and the *spray-servlet* connector servlet both provide an actor-level interface that allows your application to respond to incoming HTTP requests by simply replying with an `HttpResponse`:

```
import spray.http._
import HttpMethods._

class MyHttpService extends Actor {
  def receive = {
    case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
      sender ! HttpResponse(entity = "PONG")
  }
}
```

While it'd be perfectly possible to define a complete REST API service purely by pattern-matching against the incoming `HttpRequest` (maybe with the help of a few extractors in the way of `Unfiltered`) this approach becomes somewhat unwieldy for larger services due to the amount of syntax "ceremony" required. Also, it doesn't help in keeping your service definition as *DRY* as you might like.

As an alternative *spray-routing* provides a flexible DSL for expressing your service behavior as a structure of composable elements (called *Directives*) in a concise and readable way. At the top-level, as the result of the `runRoute` wrapper, the "route structure" produces an `Actor.Receive` partial function that can be directly supplied to your service actor. The service definition from above for example, written using the routing DSL, would look like this:

```
import spray.routing._

class MyHttpService extends HttpServiceActor {
  def receive = runRoute {
    path("ping") {
      get {
        complete("PONG")
      }
    }
  }
}
```

This very short example is certainly not the best for illustrating the savings in “ceremony” and improvements in conciseness and readability that *spray-routing* promises. The *Longer Example* might do a better job in this regard.

For learning how to work with the *spray-routing* DSL you should first understand the concept of *Routes*.

The *HttpService*

spray-routing makes all relevant parts of the routing DSL available through the *HttpService* trait, which you can mix into your service actor or route test. The *HttpService* trait defines only one abstract member:

```
def actorRefFactory: ActorRefFactory
```

which connects the routing DSL to your actor hierarchy. In order to have access to all *HttpService* members in your service actor you can either mix in the *HttpService* trait and add this line to your actor class:

```
def actorRefFactory = context
```

or, alternatively, derive your service actor from *HttpServiceActor* class, which already defines the connecting `def actorRefFactory = context` for you.

The *runRoute* Wrapper

Apart from all the *predefined directives* the *HttpService* provides one important thing, the *runRoute* wrapper. This method connects your route structure to the enclosing actor by constructing an *Actor.Receive* partial function that you can directly use as the “behavior” function of your actor:

```
import spray.routing._

class MyHttpService extends HttpServiceActor {
  def receive = runRoute {
    path("ping") {
      get {
        complete("PONG")
      }
    }
  }
}
```

Routes

“Routes” are a central concept in *spray-routing* since all structures you build with the routing DSL are subtypes of type *Route*. In *spray-routing* a route is defined like this:


```
type Route = RequestContext => Unit
```

It's a simple alias for a function taking a `RequestContext` as parameter.

Contrary to what you might initially expect a route does not return anything. Rather, all response processing (i.e. everything that needs to be done after the route itself has handled a request) is performed in “continuation-style” via the `responder` of the `RequestContext`. If you don't know what this means, don't worry. It'll become clear soon. The key point is that this design has the advantage of being completely non-blocking as well as actor-friendly since, this way, it's possible to simply send off a `RequestContext` to another actor in a “fire-and-forget” manner, without having to worry about results handling.

Generally when a route receives a request (or rather a `RequestContext` for it) it can do one of three things:

- Complete the request by calling `requestContext.complete(...)`
- Reject the request by calling `requestContext.reject(...)`
- Ignore the request (i.e. neither complete nor reject it)

The first case is pretty clear, by calling `complete` a given response is sent to the client as reaction to the request. In the second case “reject” means that the route does not want to handle the request. You'll see further down in the section about route composition what this is good for. The third case is usually an error. If a route does not do anything with the request it will simply not be acted upon. This means that the client will not receive a response until the request times out, at which point a 500 `Internal Server Error` response will be generated. Therefore your routes should usually end up either completing or rejecting the request.

Constructing Routes

Since routes are ordinary functions `RequestContext => Unit`, the simplest route is:

```
ctx => ctx.complete("Response")
```

or shorter:

```
_.complete("Response")
```

or even shorter (using the *complete* directive):

```
complete("Response")
```

All these are different ways of defining the same thing, namely a `Route` that simply completes all requests with a static response.

Even though you could write all your application logic as one monolithic function that inspects the `RequestContext` and completes it depending on its properties this type of design would be hard to read, maintain and reuse. Therefore *spray-routing* allows you to construct more complex routes from simpler ones through composition.

Composing Routes

There are three basic operations we need for building more complex routes from simpler ones:

- Route transformation, which delegates processing to another, “inner” route but in the process changes some properties of either the incoming request, the outgoing response or both
- Route filtering, which only lets requests satisfying a given filter condition pass and rejects all others
- Route chaining, which tries a second route if a given first one was rejected

The last point is achieved with the simple operator `~`, which is available to all routes via a “pimp”, i.e. an implicit extension. The first two points are provided by so-called *Directives*, of which a large number is already predefined by *spray-routing* and which you can also easily create yourself. *Directives* deliver most of *spray-routing*s power and flexibility.

The Routing Tree

Essentially, when you combine directives and custom routes via nesting and the `~` operator, you build a routing structure that forms a tree. When a request comes in it is injected into this tree at the root and flows down through all the branches in a depth-first manner until either some node completes it or it is fully rejected.

Consider this schematic example:

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

Here five directives form a routing tree.

- Route 1 will only be reached if directives a, b and c all let the request pass through.
- Route 2 will run if a and b pass, c rejects and d passes.
- Route 3 will run if a and b pass, but c and d reject.

Route 3 can therefore be seen as a “catch-all” route that only kicks in, if routes chained into preceding positions reject. This mechanism can make complex filtering logic quite easy to implement: simply put the most specific cases up front and the most general cases in the back.

Directives

“Directives” are small building blocks of which you can construct arbitrarily complex route structures. Here is a simple example of a route built from directives:

```
import spray.routing._
import Directives._

val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
    put {
      complete {
```

```

    "Received PUT request for order " + id
  }
}
}

```

The general anatomy of a directive is as follows:

```

name(arguments) {
  extractions =>
  ... // inner Route
}

```

It has a name, zero or more arguments and optionally an inner Route. Additionally directives can “extract” a number of values and make them available to their inner routes as function arguments. When seen “from the outside” a directive with its inner Route form an expression of type `Route` (see the [Routes](#) chapter for more details).

What Directives do

A directive does one or more of the following:

- Transform the incoming `RequestContext` before passing it on to its inner Route
- Filter the `RequestContext` according to some logic, i.e. only pass on certain requests and reject all others
- Extract values from the `RequestContext` and make them available to its inner Route as “extractions”
- Complete the request

The first point deserves some more discussion. A `RequestContext` is the central object that is passed on through a route structure and, potentially, in between actors. It’s immutable but light-weight and can therefore be copied quickly. When a directive receives a `RequestContext` instance from the outside it can decide to pass this instance on unchanged to its inner Route or it can create a copy of the `RequestContext` instance, with one or more changes, and pass on this copy to its inner Route. Typically this is good for two things:

- Transforming the `HttpRequest` instance
- “Hooking in” another response transformation function into the responder chain.

The Responder Chain

For understanding the “responder chain” it is helpful to look at what happens when the `complete` method of a `RequestContext` instance is called in the inner-most route of a route structure.

Consider the following hypothetical route structure of three nested directives around a simple route:

```

foo {
  bar {
    baz {
      ctx => ctx.complete("Hello")
    }
  }
}
}

```

Assume that *foo* and *baz* “hook in” response transformation logic whereas *bar* leaves the responder of the `RequestContext` it receives unchanged before passing it on to its inner Route. This is what happens when the `complete("Hello")` is called:

1. The `complete` method creates an `HttpResponse` and sends it to responder of the `RequestContext`.

2. The response transformation logic supplied by the *baz* directive runs and sends its result to the responder of the `RequestContext` the *baz* directive received.
3. The response transformation logic supplied by the *foo* directive runs and sends its result to the responder of the `RequestContext` the *foo* directive received.
4. The responder of the original `RequestContext`, which is the *sender* `ActorRef` of the `HttpRequest`, receives the response and sends it out to the client.

As you can see all response handling logic forms a logic chain that directives can choose to “hook into”.

Composing Directives

As you have seen from the examples presented so far the “normal” way of composing directives is nesting. Let’s take another look at the example from above:

```
val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
    put {
      complete {
        "Received PUT request for order " + id
      }
    }
  }
}
```

Here the `get` and `put` directives are chained together with the `~` operator to form a higher-level route that serves as the inner `Route` of the `path` directive. To make this structure more explicit you could also write the whole thing like this:

```
def innerRoute(id: Int): Route =
  get {
    complete {
      "Received GET request for order " + id
    }
  } ~
  put {
    complete {
      "Received PUT request for order " + id
    }
  }

val route: Route = path("order" / IntNumber) { id => innerRoute(id) }
```

What you can’t see from this snippet is that directives are not implemented as simple methods but rather as stand-alone objects of type `Directive`. This gives you more flexibility when composing directives. For example you can also use the `|` operator on directives. Here is yet another way to write the example:

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) { ctx =>
      ctx.complete("Received " + ctx.request.method + " request for order " + id)
    }
  }
}
```

If you have a larger route structure where the (get | put) snippet appears several times you could also factor it out like this:

```
val getOrPut = get | put
val route =
  path("order" / IntNumber) { id =>
    getOrPut { ctx =>
      ctx.complete("Received " + ctx.request.method + " request for order " + id)
    }
  }
}
```

As an alternative to nesting you can also use the & operator:

```
val getOrPut = get | put
val route =
  (path("order" / IntNumber) & getOrPut) { id => ctx =>
    ctx.complete("Received " + ctx.request.method + " request for order " + id)
  }
}
```

And once again, you can factor things out if you want:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val route =
  orderGetOrPut { id => ctx =>
    ctx.complete("Received " + ctx.request.method + " request for order " + id)
  }
}
```

This type of combining directives with the | and & operators as well as “saving” more complex directive configurations as a val works across the board, with all directives taking inner routes.

There is one more “ugly” thing remaining in our snippet: we have to fall back to the lowest-level route definition, directly manipulating the RequestContext, in order to get to the request method. It’d be nicer if we could somehow “extract” the method name in a special directive, so that we can express our inner-most route with a simple complete. As it turns out this is easy with the extract directive:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val requestMethod = extract(_.request.method)
val route =
  orderGetOrPut { id =>
    requestMethod { m =>
      complete("Received " + m + " request for order " + id)
    }
  }
}
```

Or differently:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val requestMethod = extract(_.request.method)
val route =
  (orderGetOrPut & requestMethod) { (id, m) =>
    complete("Received " + m + " request for order " + id)
  }
}
```

Now, pushing the “factoring out” of directive configurations to its extreme, we end up with this:

```
val orderGetOrPutMethod =
  path("order" / IntNumber) & (get | put) & extract(_.request.method)
val route =
  orderGetOrPutMethod { (id, m) =>
```

```
complete("Received " + m + " request for order " + id)
}
```

Note that going this far with “compressing” several directives into a single one probably doesn’t result in the most readable and therefore maintainable routing code. It might even be that the very first of this series of examples is in fact the most readable one.

Still, the purpose of the exercise presented here is to show you how flexible directives can be and how you can use their power to define your web service behavior at the level of abstraction that is right for **your** application.

Type Safety

When you combine directives with the `|` and `&` operators *spray-routing* makes sure that all extractions work as expected and logical constraints are enforced at compile-time.

For example you cannot `|` a directive producing an extraction with one that doesn’t:

```
val route = path("order" / IntNumber) | get // doesn't compile
```

Also the number of extractions and their types have to match up:

```
val route = path("order" / IntNumber) | path("order" / DoubleNumber) // doesn't compile
val route = path("order" / IntNumber) | parameter('order.as[Int]) // ok
```

When you combine directives producing extractions with the `&` operator all extractions will be properly gathered up:

```
val order = path("order" / IntNumber) & parameters('oem, 'expired ?)
val route =
  order { (orderId, oem, expired) =>
    ...
  }
```

Directives offer a great way of constructing your web service logic from small building blocks in a plug and play fashion while maintaining DRYness and full type-safety. If the large range of *Predefined Directives (alphabetically)* does not fully satisfy your needs you can also very easily create *Custom Directives*.

Rejections

In the chapter about constructing *Routes* the `~` operator was introduced, which connects two routes in a way that allows a second route to get a go at a request if the first route “rejected” it. The concept of “rejections” is used by *spray-routing* for maintaining a more functional overall architecture and in order to be able to properly handle all kinds of error scenarios.

When a filtering directive, like the *get* directive, cannot let the request pass through to its inner Route because the filter condition is not satisfied (e.g. because the incoming request is not a GET request) the directive doesn’t immediately complete the request with an error response. Doing so would make it impossible for other routes chained in after the failing filter to get a chance to handle the request. Rather, failing filters “reject” the request in the same way as by explicitly calling `requestContext.reject(...)`.

After having been rejected by a route the request will continue to flow through the routing structure and possibly find another route that can complete it. If there are more rejections all of them will be picked up and collected.

If the request cannot be completed by (a branch of) the route structure an enclosing *handleRejections* directive can be used to convert a set of rejections into an `HttpResponse` (which, in most cases, will be an error response). *The runRoute Wrapper* defined by the *The HttpService* trait internally wraps its argument route with the *handleRejections* directive in order to “catch” and handle any rejection.

Predefined Rejections

A rejection encapsulates a specific reason why a Route was not able to handle a request. It is modeled as an object of type `Rejection`. *spray-routing* comes with a set of [predefined rejections](#), which are used by various *predefined directives*.

Rejections are gathered up over the course of a Route evaluation and finally converted to `HttpResponse` replies by the *handleRejections* directive if there was no way for the request to be completed.

RejectionHandler

The *handleRejections* directive delegates the actual job of converting a list of rejections to its argument, a `RejectionHandler`, which is defined like this:

```
trait RejectionHandler extends PartialFunction[List[Rejection], Route]
```

Since a `RejectionHandler` is a partial function it can choose, which rejections it would like to handle and which not. Unhandled rejections will simply continue to flow through the route structure. The top-most `RejectionHandler` applied by *The runRoute Wrapper* will handle *all* rejections that reach it.

So, if you'd like to customize the way certain rejections are handled simply bring a custom `RejectionHandler` into implicit scope of *The runRoute Wrapper* or pass it to an explicit *handleRejections* directive that you have put somewhere into your route structure.

Here is an example:

```
import spray.routing._
import spray.http._
import StatusCodes._
import Directives._

implicit val myRejectionHandler = RejectionHandler {
  case MissingCookieRejection(cookieName) :: _ =>
    complete(BadRequest, "No cookies, no service!!!")
}

class MyService extends HttpServiceActor {
  def receive = runRoute {
    `<my-route-definition>`
  }
}
```

Rejection Cancellation

As you can see from its definition above the `RejectionHandler` handles not single rejections but a whole list of them. This is because some route structure produce several “reasons” why a request could not be handled.

Take this route structure for example:

```
import spray.httpx.encoding._

val route =
  path("order") {
    get {
      complete("Received GET")
    } ~
  }
```

```
post {
  decodeRequest (Gzip) {
    complete ("Received POST")
  }
}
```

For uncompressed POST requests this route structure could yield two rejections:

- a `MethodRejection` produced by the `get` directive (which rejected because the request is not a GET request)
- an `UnsupportedRequestEncodingRejection` produced by the `decodeRequest` directive (which only accepts gzip-compressed requests)

In reality the route even generates one more rejection, a `TransformationRejection` produced by the `post` directive. It “cancels” all other potentially existing `MethodRejections`, since they are invalid after the `post` directive allowed the request to pass (after all, the route structure *can* deal with POST requests). These types of rejection cancellations are resolved *before* a `RejectionHandler` sees the rejection list. So, for the example above the `RejectionHandler` will be presented with only a single-element rejection list, containing nothing but the `UnsupportedRequestEncodingRejection`.

Empty Rejections

Since rejections are passed around in lists you might ask yourself what the semantics of an empty rejection list are. In fact, empty rejection lists have well defined semantics. They signal that a request was not handled because the respective resource could not be found. *spray-routing* reserves the special status of “empty rejection” to this most common failure a service is likely to produce.

So, for example, if the `path` directive rejects a request, it does so with an empty rejection list. The `host` directive behaves in the same way.

Exception Handling

Exceptions thrown during route execution bubble up through the route structure to the next enclosing `handleExceptions` directive, *The runRoute Wrapper* or the `onFailure` callback of a future created by `detach`.

Similarly to the way that `Rejections` are handled the `handleExceptions` directive delegates the actual job of converting a list of rejections to its argument, an `ExceptionHandler`, which is defined like this:

```
trait ExceptionHandler extends PartialFunction[Throwable, Route]
```

The runRoute Wrapper defined in *The HttpService* does the same but gets its `ExceptionHandler` instance implicitly.

Since an `ExceptionHandler` is a partial function it can choose, which exceptions it would like to handle and which not. Unhandled exceptions will simply continue to bubble up in the route structure. The top-most `ExceptionHandler` applied by *The runRoute Wrapper* will handle *all* exceptions that reach it.

So, if you’d like to customize the way certain exceptions are handled simply bring a custom `ExceptionHandler` into implicit scope of *The runRoute Wrapper* or pass it to an explicit `handleExceptions` directive that you have put somewhere into your route structure.

Here is an example:

```
import spray.util.LoggingContext
import spray.http.StatusCodes._
import spray.routing._
```



```
implicit def myExceptionHandler(implicit log: LoggingContext) =
  ExceptionHandler {
    case e: ArithmeticException =>
      requestUri { uri =>
        log.warning("Request to {} could not be handled normally", uri)
        complete(InternalServerError, "Bad numbers, bad result!!!")
      }
  }

class MyService extends HttpServiceActor {
  def receive = runRoute {
    `<my-route-definition>`
  }
}
```

Timeout Handling

spray-routing itself does not perform any timeout checking, it relies on the underlying *spray-can* or *spray-servlet* module to watch for request timeouts. Both, the *spray-can HTTP Server* and *spray-servlet*, define a `timeout-handler` config setting, which allows you to specify the path of the actor to send `spray.http.Timedout` messages to whenever a request timeout occurs. By default all `Timedout` messages go to same actor that also handles “regular” request, i.e. your service actor.

`Timedout` is a simple wrapper around `HttpRequest` or `ChunkedRequestStart` instances:

```
case class Timedout(request: HttpRequestPart with HttpMessageStart)
```

If a `Timedout` messages hits your service actor `runRoute` unpacks it and feeds the wrapped request, i.e. the one that timed out, to the `timeoutRoute` defined by the the *HttpService*. The default implementation looks like this:

```
def timeoutRoute: Route = complete(
  InternalServerError,
  "The server was not able to produce a timely response to your request.")
```

If you’d like to customize how your service reacts to request timeouts simply override the `timeoutRoute` method.

Alternatively you can also “catch” `Timedout` message before they are handled by `runRoute` and handle them in any way you want. Here is an example of what this might look like:

```
import spray.http._
import spray.routing._

class MyService extends HttpServiceActor {
  def receive = handleTimeouts orElse runRoute(myRoute)

  def myRoute: Route = `<my-route-definition>`

  def handleTimeouts: Receive = {
    case Timedout(x: HttpRequest) =>
      sender ! HttpResponse(StatusCodes.InternalServerError, "Too late")
  }
}
```

5.7.6 Advanced Topics

Event though the following topics are considered “advanced” usage of *spray-routing* they are not necessarily hard to understand. We simply assume that many users will be able to use *spray-routing* effectively without having to fully

understand the topics in this chapter.

Understanding the DSL Structure

spray-routing's rather compact route building DSL with its extensive use of function literals can initially appear tricky, especially for users without a lot of Scala experience, so in this chapter we are explaining the mechanics in some more detail.

Assume you have the following route:

```
val route: Route = complete("yeah")
```

This is equivalent to:

```
val route: Route = _.complete("yeah")
```

which is itself the same as:

```
val route: Route = { ctx => ctx.complete("yeah") }
```

which is a function literal. The function defined by the literal is created at the time the `val` statement is reached but the code inside of the function is not executed until an actual request is injected into the route structure. This is all probably quite clear.

Now let's look at this slightly more complex structure:

```
val route: Route =  
  get {  
    complete("yeah")  
  }
```

This is equivalent to:

```
val route: Route = {  
  val inner = { ctx => ctx.complete("yeah") }  
  get.apply(inner)  
}
```

All that the *complete* directive is doing is creating a function instance, which is then passed to the `apply` method of the object named “*get* directive”, which wraps its argument route (the inner route of the `get` directive) with some filter logic and produces the final route.

Now let's look at this code:

```
val route: Route = get {  
  println("MARK")  
  complete("yeah")  
}
```

This is equivalent to:

```
val route: Route = {  
  val inner = {  
    println("MARK")  
    { ctx => ctx.complete("yeah") }  
  }  
  get.apply(inner)  
}
```

As you can see from this different representation of the same code the `println` statement is executed when the route *val* is *created*, not when a request comes in and the route is *executed*! In order to execute the `println` at request processing time it must be *inside* of the leaf-level *complete* directive:

```
val route: Route = get {
  complete {
    println("MARK")
    "yeah"
  }
}
```

The mistake of putting custom logic inside of the route structure, but *outside* of a leaf-level route, and expecting it to be executed at request-handling time, is probably the most frequent error seen by new *spray* users.

Understanding Extractions

In the examples above there are essentially two “areas” of code that are executed at different times:

- code that runs at route construction time, so usually only once
- code that runs at request-handling time, so for every request anew

If a route structure contains extractions there is one more “area” coming into play. Let’s take a look at this example:

```
val route: Route = {
  println("MARK 1")
  get {
    println("MARK 2")
    path("abc" / Segment) { x =>
      println("MARK 3") //
      complete { // code "inside"
        println("MARK 4") // of the
          "yeah" // extraction
      } //
    }
  }
}
```

Here we have put logging statements at four different places in our route structure. Let’s see when exactly they will be executed.

MARK 1 and MARK 2 From the analysis in the section above you should be able to see that there is no real difference between the “MARK 1” and “MARK 2” statements. They are both executed exactly once, when the route is built.

MARK 3 This statement lies within a function literal of an extraction, but outside of the leaf-level route. It is executed when the request is handled, so essentially shortly before the “MARK 4” statement.

MARK 4 This statement lives inside of the leaf-level route. As such it is executed anew for every request hitting its route.

Why is the “MARK 3” statement executed for every request, even though it doesn’t live at the leaf level? Because it lives “underneath an extraction”. All branches of the route structure that lie inside of a function literal for an extraction can only be created when the extracted values have been determined. Since the value of the `Segment` in the example above can only be known after a request has come in and its path has been parsed the branch of the route structure “inside” of the extraction can only be built at request-handling time.

So essentially the sequence of events in the example above is as follows:

1. When the `val route = ...` declaration is executed the outer route structure is built. The “outer route structure” consists of the `get` directive and its direct children, in this case only the `path` directive.
2. When a GET request with a matching URI comes in it flows through the outer route structure up until the point the `path` directive has extracted the value of the `Segment` placeholder.
3. The extraction function literal is executed, with the extracted `Segment` value as argument. This function creates the underlying route structure inside of the extraction.
4. After the inner route structure has been created the request is injected into it. So the inner route structure underneath an extraction is being “executed” right after its creation.

Since the route structure inside of an extraction is fully dynamic it might look completely different depending on the value that has been extracted. In order to keep your route structure readable (and thus maintainable) you probably shouldn't go too crazy with regard to dynamically creating complex route structures depending on specific extraction values though. However, understanding why it'd be possible is helpful in getting the most out of the *spray-routing* DSL.

Performance Tuning

With the understanding of the above sections it should now be possible to discover optimization potential in your route structures for the (rare!) cases, where route execution performance really turns out to be a significant factor in your application.

Let's compare two route structures that are almost equivalent with regard to how they respond to requests:

```
val routeA =
  path("abc" / Segment) { x =>
    get {
      complete(responseFor(x))
    }
  }

val routeB =
  get {
    path("abc" / Segment) { x =>
      complete(responseFor(x))
    }
  }
```

The only difference between `routeA` and `routeB` is the order in which the `get` and the `path` directive are nested. `routeB` will be a tiny amount faster in responding to requests, because the dynamic part of the route structure, i.e. the one that is rebuilt anew for every request, is smaller.

A general recommendation could therefore be to “pull up” directives without extractions as far as possible and only start extracting values at the lower levels of your routing tree. However, in the grand majority of applications we'd expect the benefits of a cleanly and logically laid out structure to far outweigh potential performance improvements through a more complex solution that goes out of its way to push down or even avoid extractions for a tiny, non-perceivable bump in performance.

Case Class Extraction

The value extraction performed by *Directives* is a nice way of providing your route logic with interesting request properties, all with proper type-safety and error handling. However, in some case you might want even more. Consider this example:

```

case class Color(red: Int, green: Int, blue: Int)

val route =
  path("color") {
    parameters('red.as[Int], 'green.as[Int], 'blue.as[Int]) { (red, green, blue) =>
      val color = Color(red, green, blue)
      doSomethingWith(color) // route working with the Color instance
    }
  }
}

```

Here a *parameters* directive is employed to extract three `Int` values, which are then used to construct an instance of the `Color` case class. So far so good. However, if the model classes we'd like to work with have more than just a few parameters the overhead introduced by capturing the arguments as extractions only to feed them into the model class constructor directly afterwards can somewhat clutter up your route definitions.

If your model classes are case classes, as in our example, *spray-routing* supports an even shorter and more concise syntax. You can also write the example above like this:

```

case class Color(red: Int, green: Int, blue: Int)

val route =
  path("color") {
    parameters('red.as[Int], 'green.as[Int], 'blue.as[Int]).as(Color) { color =>
      doSomethingWith(color) // route working with the Color instance
    }
  }
}

```

You can postfix any directive with extractions with an `as(...)` call. By simply passing the companion object of your model case class to the `as` modifier method the underlying directive is transformed into an equivalent one, which extracts only one value of the type of your model class. Note that there is no reflection involved and your case class does not have to implement any special interfaces. The only requirement is that the directive you attach the `as` call to produces the right number of extractions, with the right types and in the right order.

If you'd like to construct a case class instance from extractions produced by *several* directives you can first join the directives with the `&` operator before using the `as` call:

```

case class Color(name: String, red: Int, green: Int, blue: Int)

val route =
  (path("color" / Segment) &
   parameters('r.as[Int], 'g.as[Int], 'b.as[Int])).as(Color) { color =>
    doSomethingWith(color) // route working with the Color instance
  }
}

```

Here the `Color` class has gotten another member, `name`, which is supplied not as a parameter but as a path element. By joining the `path` and `parameters` directives with `&` you create a directive extracting 4 values, which directly fit the member list of the `Color` case class. Therefore you can use the `as` modifier to convert the directive into one extracting only a single `Color` instance.

Generally, when you have routes that work with, say, more than 3 extractions it's a good idea to introduce a case class for these and resort to case class extraction. Especially since it supports another nice feature: validation.

Caution: There is one quirk to look out for when using case class extraction: If you create an explicit companion object for your case class, no matter whether you actually add any members to it or not, the syntax presented above will not (quite) work anymore. Instead of `as(Color)` you will then have to say `as(Color.apply)`. This behavior appears as if it's not really intended, we will try to work with the Typesafe team to fix this.

Case Class Validation

In many cases your web service needs to verify input parameters according to some logic before actually working with them. E.g. in the example above the restriction might be that all color component values must be between 0 and 255. You could get this done with a few *validate* directives but this would quickly become cumbersome and hard to read.

If you use case class extraction you can put the verification logic into the constructor of your case class, where it should be:

```
case class Color(name: String, red: Int, green: Int, blue: Int) {
  require(!name.isEmpty, "color name must not be empty")
  require(0 <= red && red <= 255, "red color component must be between 0 and 255")
  require(0 <= green && green <= 255, "green color component must be between 0 and 255")
  require(0 <= blue && blue <= 255, "blue color component must be between 0 and 255")
}
```

If you write your validations like this *spray-routings* case class extraction logic will properly pick up all error messages and generate a `ValidationRejection` if something goes wrong. By default, `ValidationRejections` are converted into 400 Bad Request error response by the default *RejectionHandler*, if no subsequent route successfully handles the request.

Custom Directives

Part of *spray-routings* power comes from the ease with which it's possible to define custom directives at differing levels of abstraction. There are essentially three ways of creating custom directives:

1. By introducing new “labels” for configurations of existing directives
2. By transforming existing directives
3. By writing a directive “from scratch”

Configuration Labelling

The easiest way to create a custom directive is to simply assign a new name for a certain configuration of one or more existing directives. In fact, most of *spray-routings* predefined directives can be considered named configurations of more low-level directives.

The basic technique is explained in the chapter about *Composing Directives*, where, for example, a new directive `getOrPut` is defined like this:

```
val getOrPut = get | put
```

Another example are the *MethodDirectives*, which are simply instances of a preconfigured *method* directive, such as:

```
val delete = method(DELETE)
val get = method(GET)
val head = method(HEAD)
val options = method(OPTIONS)
val patch = method(PATCH)
val post = method(POST)
val put = method(PUT)
```

The low-level directives that most often form the basis of higher-level “named configuration” directives are grouped together in the *BasicDirectives* trait.

Transforming Directives

The second option for creating new directives is to transform an existing one using one of the “transformation methods”, which are defined on the `Directive` class, the base class of all “regular” directives.

Apart from the combinator operators (`|` and `&`) and the case-class extractor (`as[T]`) there are these transformations defined on all `Directive[L <: HList]` instances:

- `map / hmap`
- `flatMap / hflatMap`
- `require / hrequire`
- `recover / recoverPF`

map / hmap The `hmap` modifier has this signature (somewhat simplified):

```
def hmap[R](f: L => R): Directive[R :: HNil]
```

It can be used to transform the `HList` of extractions into another `HList`. The number and/or types of the extractions can be changed arbitrarily. If `R <: HList` then the result is `Directive[R]`. Here is a somewhat contrived example:

```
import shapeless._
import spray.routing._
import Directives._

val twoIntParameters: Directive[Int :: Int :: HNil] =
  parameters('a.as[Int], 'b.as[Int])

val myDirective: Directive1[String] =
  twoIntParameters.hmap {
    case a :: b :: HNil => (a + b).toString
  }

// test `myDirective` using the testkit DSL
Get("/?a=2&b=5") ~> myDirective(x => complete(x)) ~> check {
  responseAs[String] === "7"
}
```

If the `Directive` is a single-value `Directive`, i.e. one that extracts exactly one value, you can also use the simple `map` modifier, which doesn’t take the directives `HList` as parameter but rather the single value itself.

One example of a predefined directive relying on `map` is the `optionalHeaderValue` directive.

flatMap / hflatMap With `hmap` or `map` you can transform the values a directive extracts, but you cannot change the “extracting” nature of the directive. For example, if you have a directive extracting an `Int` you can use `map` to turn it into a directive that extracts that `Int` and doubles it, but you cannot transform it into a directive, that doubles all positive `Int` values and rejects all others.

In order to do the latter you need `hflatMap` or `flatMap`. The `hflatMap` modifier has this signature:

```
def hflatMap[R <: HList](f: L => Directive[R]): Directive[R]
```

The given function produces a new directive depending on the `HList` of extractions of the underlying one. As in the case of `map / hmap` there is also a single-value variant called `flatMap`, which simplifies the operation for `Directives` only extracting one single value.

Here is the (contrived) example from above, which doubles positive `Int` values and rejects all others:

```
import shapeless._
import spray.routing._
import Directives._

val intParameter: Directive1[Int] = parameter('a.as[Int])

val myDirective: Directive1[Int] =
  intParameter.flatMap {
    case a if a > 0 => provide(2 * a)
    case _ => reject
  }

// test `myDirective` using the testkit DSL
Get("/?a=21") ~> myDirective(i => complete(i.toString)) ~> check {
  responseAs[String] === "42"
}
Get("/?a=-18") ~> myDirective(i => complete(i.toString)) ~> check {
  handled must beFalse
}
```

A common pattern that relies on `flatMap` is to first extract a value from the `RequestContext` with the *extract* directive and then `flatMap` with some kind of filtering logic. For example, this is the implementation of the *method* directive:

```
/**
 * Rejects all requests whose HTTP method does not match the given one.
 */
def method(httpMethod: HttpMethod): Directive0 =
  extract(_.request.method).flatMap[HNil] {
    case `httpMethod` pass
    case _ reject (MethodRejection(httpMethod))
  } & cancelAllRejections(ofType[MethodRejection])
```

The explicit type parameter `[HNil]` on the `flatMap` is needed in this case because the result of the `flatMap` is directly concatenated with the *cancelAllRejections* directive, thereby preventing “outside-in” inference of the type parameter value.

require / hrequire The *require* modifier transforms a single-extraction directive into a directive without extractions, which filters the requests according to a predicate function. All requests, for which the predicate is `false` are rejected, all others pass unchanged.

The signature of *require* is this (slightly simplified):

```
def require[T](predicate: T => Boolean): Directive[HNil]
```

One example of a predefined directive relying on *require* is the first overload of the *host* directive.

You can only call *require* on single-extraction directives. The *hrequire* modifier is the more general variant, which takes a predicate of type `HList => Boolean`. It can therefore also be used on directives with several extractions.

recover / recoverPF The *recover* modifier allows you “catch” rejections produced by the underlying directive and, instead of rejecting, produce an alternative directive with the same type(s) of extractions.

The signature of *recover* is this:


```
def recover(recovery: List[Rejection] => Directive[L]): Directive[L]
```

In many cases the very similar `recoverPF` modifier might be little bit easier to use since it doesn't require the handling of *all* rejections:

```
def recoverPF(recovery: PartialFunction[List[Rejection], Directive[L]]): Directive[L]
```

One example of a predefined directive relying `recoverPF` is the `optionalHeaderValue` directive.

Directives from Scratch

The third option for creating custom directives is to do it “from scratch”, by directly subclassing the `Directive` class. The `Directive` is defined like this (leaving away operators and modifiers):

```
abstract class Directive[L <: HList] {
  def apply(f: L => Route): Route
}
```

It only has one abstract member that you need to implement, the `apply` method, which creates the `Route` the directives presents to the outside from its inner `Route` building function (taking the extractions as parameter).

Extractions are kept as a `shapeless HList`. Here are a few examples:

- A `Directive[HNil]` extracts nothing (like the `get` directive). Because this type is used quite frequently `spray-routing` defines a type alias for it:

```
type Directive0 = Directive[HNil]
```

- A `Directive[String :: HNil]` extracts one `String` value (like the `hostName` directive). The type alias for it is:

```
type Directive1[T] = Directive[T :: HNil]
```

- A `Directive[Int :: String :: HNil]` extracts an `Int` value and a `String` value (like a `parameters('a.as[Int], 'b.as[String])` directive).

Keeping extractions as `HLists` has a lot of advantages, mainly great flexibility while upholding full type safety and “inferability”. However, the number of times where you'll really have to fall back to defining a directive from scratch should be very small. In fact, if you find yourself in a position where a “from scratch” directive is your only option, we'd like to hear about it, so we can provide a higher-level “something” for other users.

Response Streaming

Apart from completing requests with simple `HttpResponse` instances `spray-routing` also supports asynchronous response streaming. If you run `spray-routing` on top of the `spray-can HTTP Server` a response stream can be rendered as an HTTP/1.1 chunked response or, if `chunkless-streaming` is enabled, as a single response, whose entity body is sent in several parts, one by one, across the network.

When running `spray-routing` on top of `spray-servlet` the exact interpretation of the individual response chunks depends on the servlet container implementation (see the `spray-servlet` docs for more info on this).

A streaming response is started by sending a `ChunkedResponseStart` message to the responder of the `RequestContext`. Afterwards the responder is ready to receive a number of `MessageChunk` messages. A streaming response is terminated with a `ChunkedMessageEnd` message.

In order to not flood the network with chunks that it might not be able to currently digest it's always a good idea to not send out another chunk before having received a “ACK” confirmation message from the underlying layer (see `ACKed Sends` in the `spray-can` documentation).

The *Complete Examples* both contain sample code, which shows how to send a streaming response that is “pulled” by the network via send confirmation messages.

5.7.7 Predefined Directives (alphabetically)

Directive	Description
<i>alwaysCache</i>	Wraps its inner Route with caching support using a given cache instance, ignores request Cache-
<i>anyParam</i>	Extracts a parameter either from a form field or from query parameters (in that order), rejects if no
<i>anyParams</i>	Same as <i>anyParam</i> , except for several parameters at once
<i>authenticate</i>	Tries to authenticate the user with a given authenticator and either extract a an object representing
<i>authorize</i>	Applies a given authorization check to the request and rejects if it doesn't pass
<i>autoChunk</i>	Converts non-rejected responses from its inner Route to chunked responses using a given chunk s
<i>autoChunkFileBytes</i>	Converts non-rejected responses from its inner Route to chunked responses using a given chunk s
<i>cache</i>	Wraps its inner Route with caching support using a given cache instance
<i>cachingProhibited</i>	Rejects the request if it doesn't contain a Cache-Control header with no-cache or max-age
<i>cancelAllRejections</i>	Adds a TransformationRejection to rejections from its inner Route, which cancels other
<i>cancelRejection</i>	Adds a TransformationRejection cancelling all rejections equal to a given one
<i>clientIP</i>	Extracts the IP address of the client from either the X-Forwarded-For, Remote-Address o
<i>complete</i>	Completes the request with a given response, several overloads
<i>compressResponse</i>	Compresses responses coming back from its inner Route using either Gzip or Deflate unless t
<i>compressResponseIfRequested</i>	Compresses responses coming back from its inner Route using either Gzip or Deflate, but onl
<i>conditional</i>	Depending on the given ETag and Last-Modified values responds with 304 Not Modified if
<i>cookie</i>	Extracts an HttpCookie with a given name or rejects if no such cookie is present in the request
<i>decodeRequest</i>	Decompresses incoming requests using a given Decoder
<i>decompressRequest</i>	Decompresses incoming requests using either Gzip, Deflate, or NoEncoding
<i>delete</i>	Rejects all non-DELETE requests
<i>deleteCookie</i>	Adds a Set-Cookie header expiring the given cookie to all HttpResponse replies of its inn
<i>detach</i>	Executes its inner Route in a Future
<i>dynamic</i>	Rebuilds its inner Route for every request anew
<i>dynamicIf</i>	Conditionally rebuilds its inner Route for every request anew
<i>encodeResponse</i>	Compresses responses coming back from its inner Route using a given Encoder
<i>entity</i>	Unmarshalls the requests entity according to a given definition, rejects in case of problems
<i>extract</i>	Extracts a single value from the RequestContext using a function RequestContext =>
<i>failWith</i>	Bubbles the given error up the response chain, where it is dealt with by the closest <i>handleExceptio</i>
<i>formField</i>	Extracts the value of an HTTP form field, rejects if the request doesn't come with a field matching
<i>formFields</i>	Same as <i>formField</i> , except for several fields at once
<i>get</i>	Rejects all non-GET requests
<i>getFromBrowseableDirectories</i>	Same as <i>getFromBrowseableDirectory</i> , but allows for serving the “union” of several directories as
<i>getFromBrowseableDirectory</i>	Completes GET requests with the content of a file underneath a given directory, renders directory
<i>getFromDirectory</i>	Completes GET requests with the content of a file underneath a given directory
<i>getFromFile</i>	Completes GET requests with the content of a given file
<i>getFromResource</i>	Completes GET requests with the content of a given resource
<i>getFromResourceDirectory</i>	Same as <i>getFromDirectory</i> except that the file is not fetched from the file system but rather from a
<i>handleExceptions</i>	Converts exceptions thrown during evaluation of its inner Route into HttpResponse replies usi
<i>handleRejections</i>	Converts rejections produced by its inner Route into HttpResponse replies using a given Rejec
<i>handleWith</i>	Completes the request using a given function. Uses the in-scope Unmarshaller and Marshal
<i>head</i>	Rejects all non-HEAD requests
<i>headerValue</i>	Extracts an HTTP header value using a given function, rejects if no value can be extracted
<i>headerValueByName</i>	Extracts an HTTP header value by selecting a header by name
<i>headerValueByType</i>	Extracts an HTTP header value by selecting a header by type

Table 5.1 – continued from previous page

Directive	Description
<i>headerValuePF</i>	Same as <i>headerValue</i> , but with a <code>PartialFunction</code>
<i>hextract</i>	Extracts an <code>HList</code> of values from the <code>RequestContext</code> using a function
<i>host</i>	Rejects all requests with a hostname different from a given definition, can extract the hostname using <code>RequestContext</code>
<i>hostName</i>	Extracts the hostname part of the requests <code>Host</code> header value
<i>hprovide</i>	Injects an <code>HList</code> of values into a directive, which provides them as extractions
<i>jsonpWithParameter</i>	Wraps its inner <code>Route</code> with JSONP support
<i>listDirectoryContents</i>	Completes GET requests with a unified listing of the contents of one or more given directories
<i>logRequest</i>	Produces a log entry for every incoming request
<i>logRequestResponse</i>	Produces a log entry for every response or rejection coming back from its inner route, allowing for logging of the response
<i>logResponse</i>	Produces a log entry for every response or rejection coming back from its inner route
<i>mapHttpResponse</i>	Transforms the <code>HttpResponse</code> coming back from its inner <code>Route</code>
<i>mapHttpResponsePart</i>	More general than <i>mapHttpResponse</i> , transforms the <code>HttpResponsePart</code> coming back from its inner <code>Route</code>
<i>mapHttpResponseEntity</i>	Transforms the entity of the <code>HttpResponse</code> coming back from its inner <code>Route</code>
<i>mapHttpResponseHeaders</i>	Transforms the headers of the <code>HttpResponse</code> coming back from its inner <code>Route</code>
<i>mapInnerRoute</i>	Transforms its inner <code>Route</code> with a <code>Route => Route</code> function
<i>mapRejections</i>	Transforms all rejections coming back from its inner <code>Route</code>
<i>mapRequest</i>	Transforms the incoming <code>HttpRequest</code>
<i>mapRequestContext</i>	Transforms the <code>RequestContext</code>
<i>mapRouteResponse</i>	Transforms all responses coming back from its inner <code>Route</code> with a <code>Any => Any</code> function
<i>mapRouteResponsePF</i>	Same as <i>mapRouteResponse</i> , but with a <code>PartialFunction</code>
<i>method</i>	Rejects if the request method does not match a given one
<i>overrideMethodWithParameter</i>	Changes the HTTP method of the request to the value of the specified query string parameter
<i>noop</i>	Does nothing, i.e. passes the <code>RequestContext</code> unchanged to its inner <code>Route</code>
<i>onComplete</i>	“Unwraps” a <code>Future[T]</code> and runs its inner route after future completion with the future’s value
<i>onFailure</i>	“Unwraps” a <code>Future[T]</code> and runs its inner route when the future has failed with the future’s failure value
<i>onSuccess</i>	“Unwraps” a <code>Future[T]</code> and runs its inner route after future completion with the future’s value
<i>optionalAuthenticate</i>	Tries to authenticate the user with a given authenticator and either extract an object representing the user or reject
<i>optionalCookie</i>	Extracts an <code>HttpCookie</code> with a given name, if the cookie is not present in the request extracts <code>None</code>
<i>optionalHeaderValue</i>	Extracts an optional HTTP header value using a given function
<i>optionalHeaderValueByName</i>	Extracts an optional HTTP header value by selecting a header by name
<i>optionalHeaderValueByType</i>	Extracts an optional HTTP header value by selecting a header by type
<i>optionalHeaderValuePF</i>	Extracts an optional HTTP header value using a given partial function
<i>options</i>	Rejects all non-OPTIONS requests
<i>parameter</i>	Extracts the value of a request query parameter, rejects if the request doesn’t come with a parameter
<i>parameterMap</i>	Extracts the requests query parameters as a <code>Map[String, String]</code>
<i>parameterMultiMap</i>	Extracts the requests query parameters as a <code>Map[String, List[String]]</code>
<i>parameters</i>	Same as <i>parameter</i> , except for several parameters at once
<i>parameterSeq</i>	Extracts the requests query parameters as a <code>Seq[(String, String)]</code>
<i>pass</i>	Alias for <i>noop</i>
<i>patch</i>	Rejects all non-PATCH requests
<i>path</i>	Extracts zero+ values from the <code>unmatchedPath</code> of the <code>RequestContext</code> according to a given <code>PathMatcher</code>
<i>pathEnd</i>	Only passes on the request to its inner route if the request path has been matched completely, rejects otherwise
<i>pathEndOrSingleSlash</i>	Only passes on the request to its inner route if the request path has been matched completely or only a single slash
<i>pathPrefix</i>	Same as <i>path</i> , but also matches (and consumes) prefixes of the unmatched path (rather than only the exact path)
<i>pathPrefixTest</i>	Like <i>pathPrefix</i> but without “consumption” of the matched path (prefix).
<i>pathSingleSlash</i>	Only passes on the request to its inner route if the request path consists of exactly one remaining slash
<i>pathSuffix</i>	Like as <i>pathPrefix</i> , but for suffixes rather than prefixed of the unmatched path
<i>pathSuffixTest</i>	Like <i>pathSuffix</i> but without “consumption” of the matched path (suffix).
<i>post</i>	Rejects all non-POST requests

Table 5.1 – continued from previous page

Directive	Description
<i>produce</i>	Uses the in-scope marshaller to extract a function that can be used for completing the request with
<i>provide</i>	Injects a single value into a directive, which provides it as an extraction
<i>put</i>	Rejects all non-PUT requests
<i>rawPathPrefix</i>	Applies a given <code>PathMatcher</code> directly to the unmatched path of the <code>RequestContext</code> , i.e. v
<i>rawPathPrefixTest</i>	Checks whether the <code>unmatchedPath</code> of the <code>RequestContext</code> has a prefix matched by a <code>PathM</code>
<i>redirect</i>	Completes the request with redirection response of the given type to a given URI
<i>reject</i>	Rejects the request with a given set of rejections
<i>rejectEmptyResponse</i>	Converts responses with an empty entity into a rejection
<i>requestEncodedWith</i>	Rejects the request if its encoding doesn't match a given one
<i>requestEntityEmpty</i>	Rejects the request if its entity is not empty
<i>requestEntityPresent</i>	Rejects the request if its entity is empty
<i>requestInstance</i>	Extracts the complete request
<i>requestUri</i>	Extracts the complete request URI
<i>respondWithHeader</i>	Adds a given response header to all <code>HttpResponse</code> replies from its inner Route
<i>respondWithHeaders</i>	Same as <i>respondWithHeader</i> , but for several headers at once
<i>respondWithLastModifiedHeader</i>	Adds a <code>Last-Modified</code> header to all <code>HttpResponse</code> replies from its inner Route
<i>respondWithMediaType</i>	Overrides the media-type of all <code>HttpResponse</code> replies from its inner Route, rejects if the media
<i>respondWithSingletonHeader</i>	Adds a given response header to all <code>HttpResponse</code> replies from its inner Route, if a header wit
<i>respondWithSingletonHeaders</i>	Same as <i>respondWithSingletonHeader</i> , but for several headers at once
<i>respondWithStatus</i>	Overrides the response status of all <code>HttpResponse</code> replies coming back from its inner Route
<i>responseEncodingAccepted</i>	Rejects the request if the client doesn't accept a given encoding for the response
<i>rewriteUnmatchedPath</i>	Transforms the <code>unmatchedPath</code> of the <code>RequestContext</code> using a given function
<i>routeRouteResponse</i>	Chains a partial function into the response chain, which, for certain responses from its inner route
<i>scheme</i>	Rejects a request if its Uri scheme does not match a given one
<i>schemeName</i>	Extracts the request Uri scheme
<i>setCookie</i>	Adds a <code>Set-Cookie</code> header to all <code>HttpResponse</code> replies of its inner Route
<i>unmatchedPath</i>	Extracts the unmatched path from the <code>RequestContext</code>
<i>validate</i>	Passes or rejects the request depending on evaluation of a given conditional expression
<i>withRangeSupport</i>	Transforms the response from its inner route into a 206 <code>Partial Content</code> response if the cl

5.7.8 Predefined Directives (by trait)

All predefined directives are organized into traits that form one part of the overarching `Directives` trait, which is defined like this:

```

trait Directives extends RouteConcatenation
  with AnyParamDirectives
  with BasicDirectives
  with CacheConditionDirectives
  with ChunkingDirectives
  with CookieDirectives
  with DebuggingDirectives
  with EncodingDirectives
  with ExecutionDirectives
  with FileAndResourceDirectives
  with FormFieldDirectives
  with FutureDirectives
  with HeaderDirectives
  with HostDirectives
  with MarshallingDirectives
  with MethodDirectives
  with MiscDirectives

```

```

with ParameterDirectives
with PathDirectives
with RangeDirectives
with RespondWithDirectives
with RouteDirectives
with SchemeDirectives
with SecurityDirectives

object Directives extends Directives

```

Directives filtering or extracting from the request

MethodDirectives Filter and extract based on the request method.

HeaderDirectives Filter and extract based on request headers.

PathDirectives Filter and extract from the request URI path.

HostDirectives Filter and extract based on the target host.

ParameterDirectives, *FormFieldDirectives*, *AnyParamDirectives* Filter and extract based on query parameters, form fields, or both.

EncodingDirectives Filter and decode compressed request content.

Marshalling Directives Extract the request entity.

SchemeDirectives Filter and extract based on the request scheme.

SecurityDirectives Handle authentication data from the request.

CookieDirectives Filter and extract cookies.

BasicDirectives and *MiscDirectives* Directives handling request properties.

Directives creating or transforming the response

CacheConditionDirectives Support for conditional requests (304 Not Modified responses).

ChunkingDirectives Automatically break a response into chunks.

CookieDirectives Set, modify, or delete cookies.

EncodingDirectives Compress responses.

FileAndResourceDirectives Deliver responses from files and resources.

RangeDirectives Support for range requests (206 Partial Content responses).

RespondWithDirectives Change response properties.

RouteDirectives Complete or reject a request with a response.

BasicDirectives and *MiscDirectives* Directives handling or transforming response properties.

List of predefined directives by trait

AnyParamDirectives

anyParam Alias for *anyParams*.

Signature

```
def anyParam(apdm: AnyParamDefMagnet): apdm.Out
```

Description See *anyParams*.

anyParams The *anyParams* directive allows to extract values both from query parameters and form fields.

Signature

```
def anyParams(params: <ParamDef[T_i]>*) : Directive[T_0 :: ... T_i ... :: HNil]
def anyParams(params: <ParamDef[T_0]> :: ... <ParamDef[T_i]> ... :: HNil) : Directive[T_0 :: ... T_i
```

The signature shown is simplified and written in pseudo-syntax, the real signature uses magnets.¹ The type `<ParamDef>` doesn't really exist but consists of the syntactic variants as shown in the description and the examples of the *parameters* directive.

Description The directives combines the functionality from *parameters* and *formFields* in one directive. To be able to unmarshal a parameter to a value of a specific type (e.g. with `as[Int]`) you need to fulfill the requirements as explained both for *parameters* and *formFields*.

There's a singular version, *anyParam*.

Example

```
val route =
  anyParams('name, 'age.as[Int])((name, age) =>
    complete(s"$name is $age years old")
  )

// extracts query parameters
Get("/?name=Herman&age=168") ~> route ~> check {
  responseAs[String] === "Herman is 168 years old"
}

// extracts form fields
Post("/", FormData(Seq("name" -> "Herman", "age" -> "168"))) ~> route ~> check {
  responseAs[String] === "Herman is 168 years old"
}
```

BasicDirectives

Basic directives are building blocks for building *Custom Directives*. As such they usually aren't used in a route directly but rather in the definition of new directives.

Directives to provide values to inner routes These directives allow to provide the inner routes with extractions. They can be distinguished on two axes: a) provide a constant value or extract a value from the `RequestContext` b) provide a single value or an `HList` of values.

- *extract*
- *hextract*

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

- *provide*
- *hprovide*

Directives transforming the request

- *mapRequestContext*
- *mapRequest*

Directives transforming the response These directives allow to hook into the response path and transform the complete response or the parts of a response or the list of rejections:

- *mapHttpResponse*
- *mapHttpResponseEntity*
- *mapHttpResponseHeaders*
- *mapHttpResponsePart*
- *mapRejections*

Directives hooking into the responder chain These directives allow to hook into *The Responder Chain*. The first two allow transforming the response message to a new message. The latter one allows to completely replace the response message with the execution of a new route.

- *mapRouteResponse*
- *mapRouteResponsePF*
- *routeRouteResponse*

Directives changing the execution of the inner route

- *mapInnerRoute*

Directives alphabetically

extract Calculates a value from the request context and provides the value to the inner route.

Signature

```
def extract[T] (f: RequestContext T) : Directive1[T]
```

Description The `extract` directive is used as a building block for *Custom Directives* to extract data from the `RequestContext` and provide it to the inner route. It is a special case for extracting one value of the more general *hextract* directive that can be used to extract more than one value.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
val uriLength = extract(_.request.uri.toString.length)
val route =
  uriLength { len =>
    complete(s"The length of the request URI is $len")
  }

Get("/abcdef") ~> route ~> check {
  responseAs[String] === "The length of the request URI is 25"
}
```

hextract Calculates an HList of values from the request context and provides them to the inner route.

Signature

```
def hextract[L <: HList](f: RequestContext L): Directive[L]
```

Description The `hextract` directive is used as a building block for *Custom Directives* to extract data from the `RequestContext` and provide it to the inner route. To extract just one value use the *extract* directive. To provide a constant value independent of the `RequestContext` use the *hprovide* directive instead.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
import shapeless.HNil
val pathAndQuery = hextract { ctx =>
  val uri = ctx.request.uri
  uri.path :: uri.query :: HNil
}
val route =
  pathAndQuery { (p, query) =>
    complete(s"The path is $p and the query is $query")
  }

Get("/abcdef?ghi=12") ~> route ~> check {
  responseAs[String] === "The path is /abcdef and the query is ghi=12"
}
```

hprovide Provides an HList of values to the inner route.

Signature

```
def hprovide[L <: HList](values: L): Directive[L]
```

Description The `hprovide` directive is used as a building block for *Custom Directives* to provide data to the inner route. To provide just one value use the *provide* directive. If you want to provide values calculated from the `RequestContext` use the *hextract* directive instead.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example


```
import shapeless.HNil
def provideStringAndLength(value: String) = hprovide(value :: value.length :: HNil)
val route =
  provideStringAndLength("test") { (value, len) =>
    complete(s"Value is $value and its length is $len")
  }
Get("/") ~> route ~> check {
  responseAs[String] === "Value is test and its length is 4"
}
```

mapHttpResponse Changes the response that was generated by the inner route.

Signature

```
def mapHttpResponse(f: HttpResponse HttpResponse): Directive0
```

Description The `mapHttpResponse` directive is used as a building block for *Custom Directives* to transform a response that was generated by the inner route. This directive transforms only complete responses. Use *mapHttpResponsePart*, instead, to transform parts of chunked responses as well.

See *Directives transforming the response* for similar directives.

Example

```
def overwriteResultStatus(response: HttpResponse): HttpResponse =
  response.copy(status = StatusCodes.BadGateway)
val route = mapHttpResponse(overwriteResultStatus)(complete("abc"))

Get("/abcdef?ghi=12") ~> route ~> check {
  status === StatusCodes.BadGateway
}
```

mapHttpResponseEntity Changes the response entity that was generated by the inner route.

Signature

```
def mapHttpResponseEntity(f: HttpEntity HttpEntity): Directive0
```

Description The `mapHttpResponseEntity` directive is used as a building block for *Custom Directives* to transform a response entity that was generated by the inner route.

See *Directives transforming the response* for similar directives.

Example

```
def prefixEntity(entity: HttpEntity): HttpEntity =
  HttpEntity(HttpData("test") +: entity.data)
val prefixWithTest: Directive0 = mapHttpResponseEntity(prefixEntity)
val route = prefixWithTest(complete("abc"))

Get("/") ~> route ~> check {
  responseAs[String] === "testabc"
}
```

mapHttpResponseHeaders Changes the list of response headers that was generated by the inner route.

Signature

```
def mapHttpResponseHeaders (f: List[HttpHeader] List[HttpHeader]): Directive0
```

Description The `mapHttpResponseHeaders` directive is used as a building block for *Custom Directives* to transform the list of response headers that was generated by the inner route.

See *Directives transforming the response* for similar directives.

Example

```
// adds all request headers to the response
val echoRequestHeaders = extract(_.request.headers).flatMap(respondWithHeaders)

val removeIdHeader = mapHttpResponseHeaders(_.filterNot(_.lowercaseName == "id"))
val route =
  removeIdHeader {
    echoRequestHeaders {
      complete("test")
    }
  }

Get("/") ~> RawHeader("id", "12345") ~> RawHeader("id2", "67890") ~> route ~> check {
  header("id") === None
  header("id2").get.value === "67890"
}
```

mapHttpResponsePart Changes response parts generated by the inner route.

Signature

```
def mapHttpResponsePart (f: HttpResponsePart HttpResponsePart): Directive0
```

Description The `mapHttpResponsePart` directive is used as a building block for *Custom Directives* to transform a response part that was generated by the inner route. In contrast to *mapHttpResponse* this directive allows to transform parts of chunked responses.

See *Directives transforming the response* for similar directives.

Example

```
val prefixChunks = mapHttpResponsePart {
  case MessageChunk(data, _) => MessageChunk(HttpData("prefix"+data.asString))
  case x => x
}
val route =
  prefixChunks { ctx =>
    val resp = ctx.responder
    resp ! ChunkedResponseStart(HttpResponse())
    resp ! MessageChunk(HttpData("abc"))
    resp ! MessageChunk(HttpData("def"))
    resp ! ChunkedMessageEnd
  }
}
```

```
Get("/") ~> route ~> check {
  chunks ===
    List(MessageChunk(HttpData("prefixabc")),
          MessageChunk(HttpData("prefixdef")))
}
```

mapInnerRoute Changes the execution model of the inner route by wrapping it with arbitrary logic.

Signature

```
def mapInnerRoute(f: Route Route): Directive0
```

Description The `mapInnerRoute` directive is used as a building block for *Custom Directives* to replace the inner route with any other route. Usually, the returned route wraps the original one with custom execution logic.

Example

```
val completeWithInnerException =
  mapInnerRoute { route => ctx =>
    try {
      route(ctx)
    } catch {
      case NonFatal(e) => ctx.complete(s"Got ${e.getClass.getSimpleName} '${e.getMessage}'")
    }
  }

val route =
  completeWithInnerException {
    complete(throw new IllegalArgumentException("BLIP! BLOP! Everything broke"))
  }

Get("/") ~> route ~> check {
  responseAs[String] === "Got IllegalArgumentException 'BLIP! BLOP! Everything broke'"
}
```

mapRejections Transforms the list of rejections the inner route produced.

Signature

```
def mapRejections(f: List[Rejection] List[Rejection]): Directive0
```

Description The `mapRejections` directive is used as a building block for *Custom Directives* to transform a list of rejections from the inner route to a new list of rejections.

See *Directives transforming the response* for similar directives.

Example

```
// ignore any rejections and replace them by AuthorizationFailedRejection
val replaceByAuthorizationFailed = mapRejections(_ => List(AuthorizationFailedRejection))
val route =
  replaceByAuthorizationFailed {
    path("abc") (complete("abc"))
  }
```

```
}  
  
Get("/") ~> route ~> check {  
  rejection === AuthorizationFailedRejection  
}
```

mapRequest Transforms the request before it is handled by the inner route.

Signature

```
def mapRequest(f: HttpRequest HttpRequest): Directive0
```

Description The `mapRequest` directive is used as a building block for *Custom Directives* to transform a request before it is handled by the inner route. Changing the `request.uri` parameter has no effect on path matching in the inner route because the unmatched path is a separate field of the `RequestContext` value which is passed into routes. To change the unmatched path or other fields of the `RequestContext` use the *mapRequestContext* directive.

See *Directives transforming the request* for an overview of similar directives.

Example

```
def transformToPostRequest(req: HttpRequest): HttpRequest = req.copy(method = HttpMethod.POST)  
val route =  
  mapRequest(transformToPostRequest) {  
    requestInstance { req =>  
      complete(s"The request method was ${req.method}")  
    }  
  }  
  
Get("/") ~> route ~> check {  
  responseAs[String] === "The request method was POST"  
}
```

mapRequestContext Transforms the `RequestContext` before it is passed to the inner route.

Signature

```
def mapRequestContext(f: RequestContext RequestContext): Directive0
```

Description The `mapRequestContext` directive is used as a building block for *Custom Directives* to transform the request context before it is passed to the inner route. To change only the request value itself the *mapRequest* directive can be used instead.

See *Directives transforming the request* for an overview of similar directives.

Example

```
val probe = TestProbe()  
val replaceResponder = mapRequestContext(_.copy(responder = probe.ref))  
  
val route =  
  replaceResponder {  
    complete("abc")  
  }
```

```

}
Get("/abc/def/ghi") ~> route ~> check {
  handled === false
}
probe.expectMsgType[HttpMessagePartWrapper].messagePart === HttpResponse(entity = HttpEntity("abc"))

```

mapRouteResponse Changes the message the inner route sends to the responder.

Signature

```
def mapRouteResponse(f: Any Any): Directive0
```

Description The `mapRouteResponse` directive is used as a building block for *Custom Directives* to transform what the inner route sends to the responder (see *The Responder Chain*).

See *Directives hooking into the responder chain* for similar directives.

Example

```

val rejectAll = // not particularly useful directive
  mapRouteResponse {
    case _ => Rejected(List(AuthorizationFailedRejection))
  }
val route =
  rejectAll {
    complete("abc")
  }

Get("/") ~> route ~> check {
  rejections.nonEmpty === true
}

```

mapRouteResponsePF Changes the message the inner route sends to the responder.

Signature

```
def mapRouteResponsePF(f: PartialFunction[Any, Any]): Directive0
```

Description The `mapRouteResponsePF` directive is used as a building block for *Custom Directives* to transform what the inner route sends to the responder (see *The Responder Chain*). It's similar to the `mapRouteResponse` directive but allows to specify a partial function that doesn't have to handle all the incoming response messages.

See *Directives hooking into the responder chain* for similar directives.

Example

```

case object MyCustomRejection extends Rejection
val rejectRejections = // not particularly useful directive
  mapRouteResponsePF {
    case Rejected(_) => Rejected(List(AuthorizationFailedRejection))
  }
val route =

```

```
rejectRejections {
  reject(MyCustomRejection)
}

Get("/") ~> route ~> check {
  rejection === AuthorizationFailedRejection
}
```

noop A directive that passes the request unchanged to its inner route.

Signature

```
def noop: Directive0
```

Description The directive is usually used as a “neutral element” when combining directives generically.

Example

```
Get("/") ~> noop(complete("abc")) ~> check {
  responseAs[String] === "abc"
}
```

pass An alias for the *noop* directive.

provide Provides a constant value to the inner route.

Signature

```
def provide[T] (value: T): Directive1[T]
```

Description The *provide* directive is used as a building block for *Custom Directives* to provide a single value to the inner route. To provide several values use the *hprovide* directive.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
def providePrefixedString(value: String): Directive1[String] = provide("prefix:"+value)
val route =
  providePrefixedString("test") { value =>
    complete(value)
  }
Get("/") ~> route ~> check {
  responseAs[String] === "prefix:test"
}
```

routeRouteResponse Replaces the message the inner route sends to the responder with the result of a new route.

Signature

```
def routeRouteResponse(f: PartialFunction[Any, Route]): Directive0
```

Description The `routeRouteResponse` directive is used as a building block for *Custom Directives* to replace what the inner route sends to the responder (see *The Responder Chain*) with the result of a completely new route.

See *Directives hooking into the responder chain* for similar directives.

Example

```
val completeWithRejectionNames =
  routeRouteResponse {
    case Rejected(rejs) => complete(s"Got these rejections: ${rejs.map(_.getClass.getSimpleName).mkString} ")
  }

val route = completeWithRejectionNames {
  reject (AuthorizationFailedRejection) ~
  post (complete ("post"))
}

Get("/") ~> route ~> check {
  responseAs[String] === "Got these rejections: AuthorizationFailedRejection$, MethodRejection$"
}
```

CacheConditionDirectives

conditional Wraps its inner route with support for Conditional Requests as defined by <http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-26>.

Signature

```
def conditional(eTag: EntityTag, lastModified: DateTime): Directive0
```

Description Depending on the given `eTag` and `lastModified` values this directive immediately responds with 304 Not Modified or 412 Precondition Failed (without calling its inner route) if the request comes with the respective conditional headers. Otherwise the requests is simply passed on to its inner route.

The algorithm implemented by this directive closely follows what is defined in [this section](#) of the *HTTPbis spec*.

All responses (the ones produces by this directive itself as well as the ones coming back from the inner route) are augmented with respective `ETag` and `Last-Modified` response headers.

Since this directive requires the `EntityTag` and `lastModified` time stamp for the resource as concrete arguments it is usually used quite deep down in the route structure (i.e. close to the leaf-level), where the exact resource targeted by the request has already been established and the respective `ETag/Last-Modified` values can be determined.

The *FileAndResourceDirectives* internally use the `conditional` directive for `ETag` and `Last-Modified` support (if the `spray.routing.file-get-conditional` setting is enabled).

CachingDirectives

alwaysCache Wraps its inner Route with caching support using the given `spray.caching.Cache` implementation and the in-scope keyer function.

Signature

```
def alwaysCache(cache: Cache[CachingDirectives.RouteResponse])
    (implicit keyer: CacheKeyer, factory: ActorRefFactory): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description Like *cache* but doesn't regard a Cache-Control request header for deciding if the cache should be circumvented.

Note: Caching directives are not automatically in scope, see *Usage* about how to enable them.

Example

```
var i = 0
val route =
  cache(routeCache()) {
    complete {
      i += 1
      i.toString
    }
  }

Get("/") ~> route ~> check {
  responseAs[String] === "1"
}
// now cached
Get("/") ~> route ~> check {
  responseAs[String] === "1"
}
// caching prevented
Get("/") ~> `Cache-Control`(CacheDirectives.`no-cache`) ~> route ~> check {
  responseAs[String] === "2"
}
```

cache Wraps its inner Route with caching support using the given `spray.caching.Cache` implementation and the in-scope keyer function.

Signature

```
def cache(cache: Cache[CachingDirectives.RouteResponse])
    (implicit keyer: CacheKeyer, factory: ActorRefFactory): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

The `routeCache` constructor for caches:

```
def routeCache(maxCapacity: Int = 500, initialCapacity: Int = 16, timeToLive: Duration = Duration.Inf,
    timeToIdle: Duration = Duration.Inf): Cache[RouteResponse] =
  LruCache(maxCapacity, initialCapacity, timeToLive, timeToIdle)
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Description The directive tries to serve the request from the given cache and only if not found runs the inner route to generate a new response. A simple cache can be constructed using `routeCache` constructor.

The directive is implemented in terms of `cacheProhibited` and `alwaysCache`. This means that clients can circumvent the cache using a `Cache-Control` request header. This behavior may not be adequate depending on your backend implementation (i.e how expensive a call circumventing the cache into the backend is). If you want to force all requests to be handled by the cache use the `alwaysCache` directive instead. In complexer cases, e.g. when the backend can validate that a cached request is still acceptable according to the request `Cache-Control` header the predefined caching directives may not be sufficient and a custom solution is necessary.

Note: Caching directives are not automatically in scope, see *Usage* about how to enable them.

Example

```
var i = 0
val route =
  cache(routeCache()) {
    complete {
      i += 1
      i.toString
    }
  }

Get("/") ~> route ~> check {
  responseAs[String] === "1"
}
// now cached
Get("/") ~> route ~> check {
  responseAs[String] === "1"
}
Get("/") ~> route ~> check {
  responseAs[String] === "1"
}
```

cacheProhibited Passes only requests that explicitly forbid caching with a `Cache-Control` header with either a `no-cache` or `max-age=0` setting.

Signature

```
def cachingProhibited: Directive0
```

Description This directive is used to filter out requests that forbid caching. It is used as a building block of the `cache` directive to prevent caching if the client requests so.

Note: Caching directives are not automatically in scope, see *Usage* about how to enable them.

Example

```
val route =
  cachingProhibited {
    complete("abc")
  }
```

```

Get("/") ~> route ~> check {
  handled === false
}
Get("/") ~> `Cache-Control` (CacheDirectives.`no-cache`) ~> route ~> check {
  responseAs[String] === "abc"
}

```

Usage To use the caching directives you need to add a dependency to the *spray-caching* module. Caching directives are not automatically in scope using the `HttpService` or `Directives` trait but must either be brought into scope by extending from `CachingDirectives` or by using `import CachingDirectives._`.

ChunkingDirectives

autoChunk Converts unchunked responses coming back from its inner route into chunked responses of which each chunk is smaller or equal to the given size if the response entity is at least as large as the given threshold.

Signature

```

def autoChunk(maxChunkSize: Long)(implicit factory: ActorRefFactory): Directive0
def autoChunk(threshold: Long, maxChunkSize: Long)(implicit factory: ActorRefFactory): Directive0

```

The signature shown is simplified, the real signature uses magnets. ¹

Description The parameter of type `ChunkSizeMagnet` decides for which values of `HttpData` the directive should apply and how to chunk the data. Predefined instances of `ChunkSizeMagnet` decide this on the basis of two parameters, the threshold size and the chunk size (if only one number is supplied it is used for both values). The threshold parameter decides from which size on an entity should be converted into a chunked request. The chunk size parameter decides how big each chunk should be at most.

See the *autoChunkFileBytes* directive for an alternative that adds another restriction to chunk a response only when it consists only of `FileBytes`, i.e. it is completely backed by a file.

Auto chunking is especially effective in combination with encoding. Encoding (gzip, deflate) always encodes the complete response part in one step. For big entities this can be a disadvantage especially when the data has to be read from a file into JVM heap buffers. Auto chunking helps here because it produces a lazy stream of response chunks that can be encoded one by one by an encoder so that only one chunk is loaded into the JVM heap at one time.

Example

```

val route =
  autoChunk(5) {
    path("long") (complete("This is a long text")) ~
    path("short") (complete("Short"))
  }

Get("/short") ~> route ~> check {
  responseAs[String] === "Short"
}

Get("/long") ~> route ~> check {
  val HttpResponse(_, c0, _, _) = response
  val List(c1, c2, c3) = chunks
  c0.data === HttpData("This ")
}

```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```
c1.data === HttpData("is a ")
c2.data === HttpData("long ")
c3.data === HttpData("text")
}
```

autoChunkFileBytes Converts unchunked responses coming back from its inner route into chunked responses of which each chunk is smaller or equal to the given size if the response entity is at least as large as the given threshold and contains only `HttpData.FileBytes`.

Signature

```
def autoChunkFileBytes(maxChunkSize: Long)(implicit factory: ActorRefFactory): Directive0
def autoChunkFileBytes(threshold: Long, maxChunkSize: Long)(implicit factory: ActorRefFactory): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description See *autoChunk* for a more detailed description of the parameters as this directive is basically the same with the added restriction to chunk only entities completely backed by files.

Example

```
val route =
  autoChunkFileBytes(5) {
    path("long") (complete("This is a long text"))
  }

Get("/long") ~> route ~> check {
  // don't chunk long request because it's not from a file
  responseAs[String] === "This is a long text"
}
```

CookieDirectives

cookie Extracts a cookie with a given name from a request or otherwise rejects the request with a `MissingCookieRejection` if the cookie is missing.

Signature

```
def cookie(name: String): Directive1[HttpCookie]
```

Description Use the *optionalCookie* directive instead if you want to support missing cookies in your inner route.

Example

```
val route =
  cookie("userName") { nameCookie =>
    complete(s"The logged in user is '${nameCookie.content}')}
  }

Get("/") ~> Cookie(HttpCookie("userName", "paul")) ~> route ~> check {
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```

    responseAs[String] === "The logged in user is 'paul'"
  }
  // missing cookie
  Get("/") ~> route ~> check {
    rejection === MissingCookieRejection("userName")
  }
  Get("/") ~> sealRoute(route) ~> check {
    responseAs[String] === "Request is missing required cookie 'userName'"
  }
}

```

deleteCookie Adds a header to the response to request the removal of the cookie with the given name on the client.

Signature

```

def deleteCookie(first: HttpCookie, more: HttpCookie*): Directive0
def deleteCookie(name: String, domain: String = "", path: String = ""): Directive0

```

Description Use the *setCookie* directive to update a cookie.

Example

```

val route =
  deleteCookie("userName") {
    complete("The user was logged out")
  }

Get("/") ~> route ~> check {
  responseAs[String] === "The user was logged out"
  header[`Set-Cookie`] === Some(`Set-Cookie` (HttpCookie("userName", content = "deleted", expires = S
}

```

optionalCookie Extracts an optional cookie with a given name from a request.

Signature

```

def optionalCookie(name: String): Directive1[Option[HttpCookie]]

```

Description Use the *cookie* directive instead if the inner route does not handle a missing cookie.

Example

```

val route =
  optionalCookie("userName") {
    case Some(nameCookie) => complete(s"The logged in user is '${nameCookie.content}')"
    case None => complete("No user logged in")
  }

Get("/") ~> Cookie(HttpCookie("userName", "paul")) ~> route ~> check {
  responseAs[String] === "The logged in user is 'paul'"
}
Get("/") ~> route ~> check {

```

```
responseAs[String] === "No user logged in"
}
```

setCookie Adds a header to the response to request the update of the cookie with the given name on the client.

Signature

```
def setCookie(first: HttpCookie, more: HttpCookie*): Directive0
```

Description Use the *deleteCookie* directive to delete a cookie.

Example

```
val route =
  setCookie(HttpCookie("userName", content = "paul")) {
    complete("The user was logged in")
  }

Get("/") ~> route ~> check {
  responseAs[String] === "The user was logged in"
  header[`Set-Cookie`] === Some(`Set-Cookie` (HttpCookie("userName", content = "paul")))
}
```

DebuggingDirectives

logRequest Logs the request.

Signature

```
def logRequest(marker: String) (implicit log: LoggingContext): Directive0
def logRequest(marker: String, level: LogLevel) (implicit log: LoggingContext): Directive0
def logRequest(show: HttpRequest => String) (implicit log: LoggingContext): Directive0
def logRequest(show: HttpRequest => LogEntry) (implicit log: LoggingContext): Directive0
def logRequest(magnet: LoggingMagnet[HttpRequest => Unit]) (implicit log: LoggingContext): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description Logs the request using the supplied `LoggingMagnet[HttpRequest => Unit]`. This `LoggingMagnet` is a wrapped function `HttpRequest => Unit` that can be implicitly created from the different constructors shown above. These constructors build a `LoggingMagnet` from these components:

- A marker to prefix each log message with.
- A log level.
- A show function that calculates a string representation for a request.
- An implicit `LoggingContext` that is used to emit the log message.
- A function that creates a `LogEntry` which is a combination of the elements above.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

It is also possible to use any other function `HttpRequest => Unit` for logging by wrapping it with `LoggingMagnet`. See the examples for ways to use the `logRequest` directive.

Use `logResponse` for logging the response, or `logRequestResponse` for logging both.

Example

```
// different possibilities of using logRequest

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpRequest.toString
DebuggingDirectives.logRequest("get-user")

// marks with "get-user", log with info level, HttpRequest.toString
DebuggingDirectives.logRequest("get-user", Logging.InfoLevel)

// logs just the request method at debug level
def requestMethod(req: HttpRequest): String = req.method.toString
DebuggingDirectives.logRequest(requestMethod _)

// logs just the request method at info level
def requestMethodAsInfo(req: HttpRequest): LogEntry = LogEntry(req.method.toString, Logging.InfoLevel)
DebuggingDirectives.logRequest(requestMethodAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printRequestMethod(req: HttpRequest): Unit = println(req.method)
val logRequestPrintln = DebuggingDirectives.logRequest(LoggingMagnet(printRequestMethod))

Get("/") ~> logRequestPrintln(complete("logged")) ~> check {
  responseAs[String] === "logged"
}
```

logRequestResponse Logs request and response.

Signature

```
def logRequestResponse(marker: String) (implicit log: LoggingContext): Directive0
def logRequestResponse(marker: String, level: LogLevel) (implicit log: LoggingContext): Directive0
def logRequestResponse(show: HttpRequest HttpResponsePart Option[LogEntry])
  (implicit log: LoggingContext): Directive0
def logRequestResponse(show: HttpRequest Any Option[LogEntry]) (implicit log: LoggingContext): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description This directive is a combination of `logRequest` and `logResponse`. See `logRequest` for the general description how these directives work.

Example

```
// different possibilities of using logRequestResponse

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpRequest.toString, HttpResponse.toString
DebuggingDirectives.logRequestResponse("get-user")
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```
// marks with "get-user", log with info level, HttpRequest.toString, HttpResponse.toString
DebuggingDirectives.logRequestResponse("get-user", Logging.InfoLevel)

// logs just the request method and response status at info level
def requestMethodAndResponseStatusAsInfo(req: HttpRequest): Any => Option[LogEntry] = {
  case res: HttpResponse => Some(LogEntry(req.method + ":" + res.message.status, Logging.InfoLevel))
  case _ => None // other kind of responses
}
DebuggingDirectives.logRequestResponse(requestMethodAndResponseStatusAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printRequestMethodAndResponseStatus(req: HttpRequest) (res: Any): Unit =
  println(requestMethodAndResponseStatusAsInfo(req) (res).map(_.obj.toString).getOrElse(""))
val logRequestResponsePrintln = DebuggingDirectives.logRequestResponse(LoggingMagnet(printRequestMethodAndResponseStatus))

Get("/") ~> logRequestResponsePrintln(complete("logged")) ~> check {
  responseAs[String] === "logged"
}
```

logResponse Logs the response.

Signature

```
def logResponse(marker: String) (implicit log: LoggingContext): Directive0
def logResponse(marker: String, level: LogLevel) (implicit log: LoggingContext): Directive0
def logResponse(show: Any => String) (implicit log: LoggingContext): Directive0
def logResponse(show: Any => LogEntry) (implicit log: LoggingContext): Directive0
def logResponse(magnet: LoggingMagnet[Any => Unit]) (implicit log: LoggingContext): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description See logRequest for the general description how these directives work. This directive is different as it requires a LoggingMagnet[Any => Unit]. Instead of just logging HttpResponses, logResponse is able to log anything passing through *The Responder Chain* (which can either be a HttpResponsePart or a Rejected message reporting rejections).

Use logRequest for logging the request, or logRequestResponse for logging both.

Example

```
// different possibilities of using logResponse

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpResponse.toString
DebuggingDirectives.logResponse("get-user")

// marks with "get-user", log with info level, HttpResponse.toString
DebuggingDirectives.logResponse("get-user", Logging.InfoLevel)

// logs just the response status at debug level
def responseStatus(res: Any): String = res match {
  case x: HttpResponse => x.status.toString
  case _ => "unknown response part"
}
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```

DebuggingDirectives.logResponse(responseStatus _)

// logs just the response status at info level
def responseStatusAsInfo(res: Any): LogEntry = LogEntry(responseStatus(res), Logging.InfoLevel)
DebuggingDirectives.logResponse(responseStatusAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printResponseStatus(res: Any): Unit = println(responseStatus(res))
val logResponsePrintln = DebuggingDirectives.logResponse(LoggingMagnet(printResponseStatus))

Get("/") ~> logResponsePrintln(complete("logged")) ~> check {
  responseAs[String] === "logged"
}

```

EncodingDirectives

compressResponse Uses the first of a given number of encodings that the client accepts. If none are accepted the request is rejected with an `UnacceptedResponseEncodingRejection`.

Signature

```

def compressResponse()(implicit refFactory: ActorRefFactory): Directive0
def compressResponse(e1: Encoder)(implicit refFactory: ActorRefFactory): Directive0
def compressResponse(e1: Encoder, e2: Encoder)(implicit refFactory: ActorRefFactory): Directive0
def compressResponse(e1: Encoder, e2: Encoder, e3: Encoder)(implicit refFactory: ActorRefFactory): D

```

The signature shown is simplified, the real signature uses magnets. ¹

Description The `compressResponse` directive allows to specify zero to three encoders to try in the specified order. If none are specified the tried list is `Gzip`, `Deflate`, and then `NoEncoding`.

The `compressResponse()` directive (without an explicit list of encoders given) will therefore behave as follows:

Accept-Encoding header	resulting response
Accept-Encoding: gzip	compressed with Gzip
Accept-Encoding: deflate	compressed with Deflate
Accept-Encoding: deflate, gzip	compressed with Gzip
Accept-Encoding: identity	uncompressed
no Accept-Encoding header present	compressed with Gzip

For an overview of the different `compressResponse` directives see [When to use which compression directive?](#).

Example This example shows the behavior of `compressResponse` without any encoders specified:

```

val route = compressResponse() { complete("content") }

Get("/") ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {

```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.


```

    response must haveContentEncoding(deflate)
  }
  Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
    status === StatusCodes.OK
    response must haveContentEncoding(identity)
    responseAs[String] === "content"
  }
}

```

This example shows the behaviour of `compressResponse(Gzip)`:

```

val route = compressResponse(Gzip) { complete("content") }

Get("/") ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  rejection === UnacceptedResponseEncodingRejection(gzip)
}
Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
  rejection === UnacceptedResponseEncodingRejection(gzip)
}
}

```

compressResponseIfRequested Only compresses the response when specifically requested by the `Accept-Encoding` request header (i.e. the default is “no compression”).

Signature

```
def compressResponseIfRequested()(implicit refFactory: ActorRefFactory): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description The `compressResponseIfRequested` directive is an alias for `compressResponse(NoEncoding, Gzip, Deflate)` and will behave as follows:

Accept-Encoding header	resulting response
Accept-Encoding: gzip	compressed with Gzip
Accept-Encoding: deflate	compressed with Deflate
Accept-Encoding: deflate, gzip	compressed with Gzip
Accept-Encoding: identity	uncompressed
no Accept-Encoding header present	uncompressed

For an overview of the different `compressResponse` directives see [When to use which compression directive?](#).

Example

```

val route = compressResponseIfRequested() { complete("content") }

Get("/") ~> route ~> check {
  response must haveContentEncoding(identity)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response must haveContentEncoding(gzip)
}
}

```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```

}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  response must haveContentEncoding(deflate)
}
Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
  response must haveContentEncoding(identity)
}

```

decodeRequest Tries to decode the request with the specified Decoder or rejects the request with an UnacceptedRequestEncodingRejection(supportedEncoding).

Signature

```
def decodeRequest(decoder: Decoder): Directive0
```

Description The decodeRequest directive is the building block for the decompressRequest directive.

decodeRequest and decompressRequest are related like this:

```

decompressRequest(Gzip)           = decodeRequest(Gzip)
decompressRequest(a, b, c)       = decodeRequest(a) | decodeRequest(b) | decodeRequest(c)
decompressRequest()              = decodeRequest(Gzip) | decodeRequest(Deflate) | decodeRequest(NoEnc

```

Example

```

val route =
  decodeRequest(Gzip) {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Get("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] === "Request content: 'Hello'"
}
Get("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejection === UnsupportedRequestEncodingRejection(gzip)
}
Get("/", "hello") ~> `Content-Encoding`(identity) ~> route ~> check {
  rejection === UnsupportedRequestEncodingRejection(gzip)
}

```

decompressRequest Decompresses the request if it is can be decoded with one of the given decoders. Otherwise, the request is rejected with an UnsupportedRequestEncodingRejection(supportedEncoding).

Signature

```

def decompressRequest(): Directive0
def decompressRequest(first: Decoder, more: Decoder*): Directive0

```

Description The decompressRequest directive allows either to specify a list of decoders or none at all. If no Decoder is specified Gzip, Deflate, or NoEncoding will be tried.

The decompressRequest directive will behave as follows:

Content-Encoding header	resulting request
Content-Encoding: gzip	decompressed
Content-Encoding: deflate	decompressed
Content-Encoding: identity	unchanged
no Content-Encoding header present	unchanged

For an overview of the different `decompressRequest` directives and which one to use when, see *When to use which decompression directive?*.

Example This example shows the behavior of `decompressRequest()` without any decoders specified:

```
val route =
  decompressRequest() {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Get("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] === "Request content: 'Hello'"
}
Get("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  responseAs[String] === "Request content: 'Hello'"
}
Get("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
  responseAs[String] === "Request content: 'hello uncompressed'"
}
```

This example shows the behaviour of `decompressRequest(Gzip, NoEncoding)`:

```
val route =
  decompressRequest(Gzip, NoEncoding) {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Get("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] === "Request content: 'Hello'"
}
Get("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejections === List(UnsupportedRequestEncodingRejection(gzip),
    UnsupportedRequestEncodingRejection(identity))
}
Get("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
  responseAs[String] === "Request content: 'hello uncompressed'"
}
```

encodeResponse Tries to encode the response with the specified `Encoder` or rejects the request with an `UnacceptedResponseEncodingRejection(supportedEncodings)`.

Signature

```
def encodeResponse(encoder: Encoder)(implicit refFactory: ActorRefFactory): Directive0
def encodeResponse(encoder: Encoder, threshold: Long, maxChunkSize: Long)
  (implicit refFactory: ActorRefFactory): Directive0
```

The signature shown is simplified, the real signature uses magnets. ¹

Description The directive automatically applies the `autoChunkFileBytes` directive as well to avoid having to load an entire file into JVM heap.

The parameter to the directive is either just an `Encoder` or all of an `Encoder`, a threshold, and a chunk size to configure the automatically applied `autoChunkFileBytes` directive.

The `encodeResponse` directive is the building block for the `compressResponse` and `compressResponseIfRequested` directives.

`encodeResponse`, `compressResponse`, and `compressResponseIfRequested` are related like this:

```
compressResponse (Gzip)           = encodeResponse (Gzip)
compressResponse (a, b, c)       = encodeResponse (a) | encodeResponse (b) | encodeResponse (c)
compressResponse ()              = encodeResponse (Gzip) | encodeResponse (Deflate) | encodeResponse (NoEncoding)
compressResponseIfRequested ()   = encodeResponse (NoEncoding) | encodeResponse (Gzip) | encodeResponse (Deflate)
```

Example

```
val route = encodeResponse (Gzip) { complete("content") }

Get("/") ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding` (gzip, deflate) ~> route ~> check {
  response must haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding` (deflate) ~> route ~> check {
  rejection === UnacceptedResponseEncodingRejection (gzip)
}
Get("/") ~> `Accept-Encoding` (identity) ~> route ~> check {
  rejection === UnacceptedResponseEncodingRejection (gzip)
}
```

requestEncodedWith Passes the request to the inner route if the request is encoded with the argument encoding. Otherwise, rejects the request with an `UnacceptedRequestEncodingRejection (encoding)`.

Signature

```
def requestEncodedWith(encoding: HttpEncoding): Directive0
```

Description This directive is the building block for `decodeRequest` to reject unsupported encodings.

responseEncodingAccepted Passes the request to the inner route if the request accepts the argument encoding. Otherwise, rejects the request with an `UnacceptedResponseEncodingRejection (encoding)`.

Signature

```
def responseEncodingAccepted(encoding: HttpEncoding): Directive0
```

Description This directive is the building block for `encodeResponse` to reject unsupported encodings.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

When to use which compression directive? There are three different directives for performing response compressing with slightly different behavior:

encodeResponse Always compresses the response with the one given encoding, rejects the request with an `UnacceptedResponseEncodingRejection` if the client doesn't accept the given encoding. The other compression directives are built upon this one. See its description for an overview how they relate exactly.

compressResponse Uses the first of a given number of encodings that the client accepts. If none are accepted the request is rejected.

compressResponseIfRequested Only compresses the response when specifically requested by the `Accept-Encoding` request header (i.e. the default is "no compression").

See the individual directives for more detailed usage examples.

When to use which decompression directive? There are two different directives for performing request decompressing with slightly different behavior:

decodeRequest Attempts to decompress the request using **the one given decoder**, rejects the request with an `UnsupportedRequestEncodingRejection` if the request is not encoded with the given encoder.

decompressRequest Decompresses the request if it is encoded with **one of the given encoders**. If the request's encoding doesn't match one of the given encoders it is rejected.

Combining compression and decompression As with all Spray directives, the above single directives can be combined using `&` to produce compound directives that will decompress requests and compress responses in whatever combination required. Some examples:

```
"the (decompressRequest & compressResponse) compound directive" should {
  val decompressCompress = (decompressRequest() & compressResponse())
  "decompress a GZIP compressed request and produce a GZIP compressed response if the request has no
  Get("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> {
    decompressCompress { echoRequestContent }
  } ~> check {
    response must haveContentEncoding(gzip)
    body === HttpEntity(ContentType(`text/plain`, `UTF-8`), helloGzipped)
  }
}
"decompress a GZIP compressed request and produce a Deflate compressed response if the request has
  Get("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> `Accept-Encoding`(deflate) ~> {
    decompressCompress { echoRequestContent }
  } ~> check {
    response must haveContentEncoding(deflate)
    body === HttpEntity(ContentType(`text/plain`, `UTF-8`), helloDeflated)
  }
}
"decompress an uncompressed request and produce a GZIP compressed response if the request has an `A
  Get("/", "Hello") ~> `Accept-Encoding`(gzip) ~> {
    decompressCompress { echoRequestContent }
  } ~> check {
    response must haveContentEncoding(gzip)
    body === HttpEntity(ContentType(`text/plain`, `UTF-8`), helloGzipped)
  }
}
}
```

ExecutionDirectives

detach Executes the inner route inside a future.

Signature

```
def detach() (implicit ec: ExecutionContext): Directive0
def detach() (implicit refFactory: ActorRefFactory): Directive0
def detach(ec: ExecutionContext): Directive0
```

The signature shown is simplified, the real signature uses magnets.¹

Description This directive needs either an implicit `ExecutionContext` (`detach()`) or an explicit one (`detach(ec)`).

Caution: It is a common mistake to access actor state from code run inside a future that is created inside an actor by accidentally accessing instance methods or variables of the actor that are available in the scope. This also applies to the `detach` directive if a route is run inside an actor which is the usual case. Make sure not to access any actor state from inside the `detach` block directly or indirectly.

A lesser known fact is that the current semantics of executing *The Routing Tree* encompasses that every route that rejects a request also runs the alternative routes chained with `~`. This means that when a route is rejected out of a `detach` block, also all the alternatives tried afterwards are then run out of the future originally created for running the `detach` block and not any more from the original (actor) context starting the request processing. To avoid that use `detach` only at places inside the routing tree where no rejections are expected.

Example

```
val route =
  detach() {
    complete("Result") // route executed in future
  }
Get("/") ~> route ~> check {
  responseAs[String] === "Result"
}
```

This example demonstrates the effect of the note above:

```
/// / a custom directive to extract the id of the current thread
def currentThreadId: Directive1[Long] = extract(_ => Thread.currentThread().getId)
val route =
  currentThreadId { originThread =>
    path("rejectDetached") {
      detach() {
        reject()
      }
    } ~
    path("reject") {
      reject()
    } ~
    currentThreadId { alternativeThread =>
      complete(s"$originThread, $alternativeThread")
    }
  }
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```

Get("/reject") ~> route ~> check {
  val Array(original, alternative) = responseAs[String].split(",")
  original === alternative
}
Get("/rejectDetached") ~> route ~> check {
  val Array(original, alternative) = responseAs[String].split(",")
  original !== alternative
}

```

dynamic Enforces that the code **constructing the inner route** is run for every request.

Signature

```
def dynamic: ByNameDirective0
```

Description `dynamic` is a special directive because, in fact, it doesn't implement `Directive` at all. That means you cannot use it in combination with the usual directive operators.

Use `dynamicIf` to run the inner route constructor dynamically depending on a static condition.

Example

```

var value = 0
val route =
  dynamic {
    value += 1 /// executed for each request
    complete(s"Result is now $value") // route executed in future
  }
Get("/") ~> route ~> check {
  responseAs[String] === "Result is now 1"
}
Get("/") ~> route ~> check {
  responseAs[String] === "Result is now 2"
}

```

dynamicIf Enforces that the code **constructing the inner route** is run for every request if the condition is true.

Signature

```
def dynamicIf(enabled: Boolean): ByNameDirective0
```

Description The effect of `dynamicIf(true)` is the same as for `dynamic`. The effect of `dynamicIf(false)` is the same as just the nested block.

`dynamicIf` is a special directive because, in fact, it doesn't implement `Directive` at all. That means you cannot use it in combination with the usual directive operators.

Use `dynamic` to run the inner route constructor dynamically unconditionally.

Example

```
def countDynamically(dyn: Boolean) = {
  var value = 0
  dynamicIf(dyn) {
    value += 1 /// executed for each request
    complete(s"Result is now $value") // route executed in future
  }
}

val route =
  path("dynamic") (countDynamically(true)) ~
  path("static") (countDynamically(false))

Get("/dynamic") ~> route ~> check {
  responseAs[String] === "Result is now 1"
}
Get("/dynamic") ~> route ~> check {
  responseAs[String] === "Result is now 2"
}
Get("/dynamic") ~> route ~> check {
  responseAs[String] === "Result is now 3"
}

Get("/static") ~> route ~> check {
  responseAs[String] === "Result is now 1"
}
Get("/static") ~> route ~> check {
  responseAs[String] === "Result is now 1"
}
Get("/static") ~> route ~> check {
  responseAs[String] === "Result is now 1"
}
}
```

handleExceptions Catches exceptions thrown by the inner route and handles them using the specified `ExceptionHandler`.

Signature

```
def handleExceptions(handler: ExceptionHandler): Directive0
```

Description Using this directive is an alternative to using a global implicitly defined `ExceptionHandler` that applies to the complete route.

See *Exception Handling* for general information about options for handling exceptions.

Example

```
val divByZeroHandler = ExceptionHandler {
  case _: ArithmeticException => complete(StatusCode.BadRequest, "You've got your arithmetic wrong,")
}

val route =
  path("divide" / IntNumber / IntNumber) { (a, b) =>
    handleExceptions(divByZeroHandler) {
      complete(s"The result is ${a / b}")
    }
  }
}
```



```

Get("/divide/10/5") ~> route ~> check {
  responseAs[String] === "The result is 2"
}
Get("/divide/10/0") ~> route ~> check {
  status === StatusCodes.BadRequest
  responseAs[String] === "You've got your arithmetic wrong, fool!"
}

```

handleRejections Handles rejections produced by the inner route and handles them using the specified `RejectionHandler`.

Signature

```
def handleRejections(handler: RejectionHandler): Directive0
```

Description Using this directive is an alternative to using a global implicitly defined `RejectionHandler` that applies to the complete route.

See *Rejections* for general information about options for handling rejections.

Example

```

val totallyMissingHandler = RejectionHandler {
  case Nil /* secret code for path not found */ =>
    complete(StatusCodes.NotFound, "Oh man, what you are looking for is long gone.")
}
val route =
  pathPrefix("handled") {
    handleRejections(totallyMissingHandler) {
      path("existing") (complete("This path exists"))
    }
  }

Get("/handled/existing") ~> route ~> check {
  responseAs[String] === "This path exists"
}
Get("/missing") ~> sealRoute(route) /* applies default handler */ ~> check {
  status === StatusCodes.NotFound
  responseAs[String] === "The requested resource could not be found."
}
Get("/handled/missing") ~> route ~> check {
  status === StatusCodes.NotFound
  responseAs[String] === "Oh man, what you are looking for is long gone."
}

```

FileAndResourceDirectives

Like the *RouteDirectives* the *FileAndResourceDirectives* are somewhat special in spray's routing DSL. Contrary to all other directives they do not produce instances of type `Directive[L <: HList]` but rather "plain" routes of type `Route`. The reason is that they are not meant for wrapping an inner route (like most other directives, as intermediate-level elements of a route structure, do) but rather form the actual route structure **leaves**.

So in most cases the inner-most element of a route structure branch is one of the *RouteDirectives* or *FileAndResourceDirectives*.

getFromBrowseableDirectory The single-directory variant of `getFromBrowseableDirectories`.

Signature

```
def getFromBrowseableDirectory(directory: String)
    (implicit renderer: Marshaller[DirectoryListing], settings: RoutingSettings,
     resolver: ContentTypeResolver, refFactory: ActorRefFactory, log: Logger): Route
```

getFromBrowseableDirectories Serves the content of the given directories as a file system browser, i.e. files are sent and directories served as browsable listings.

Signature

```
def getFromBrowseableDirectories(directories: String*)
    (implicit renderer: Marshaller[DirectoryListing], settings: RoutingSettings,
     resolver: ContentTypeResolver, refFactory: ActorRefFactory, log: Logger): Route
```

Description The `getFromBrowseableDirectories` is a combination of serving files from the specified directories (like `getFromDirectory`) and listing a browsable directory with `listDirectoryContents`. Nesting this directive beneath `get` is not necessary as this directive will only respond to GET requests.

Use `getFromBrowseableDirectory` to serve only one directory. Use `getFromDirectory` if directory browsing isn't required.

getFromDirectory Completes GET requests with the content of a file underneath the given directory.

Signature

```
def getFromDirectory(directoryName: String)
    (implicit settings: RoutingSettings, resolver: ContentTypeResolver,
     refFactory: ActorRefFactory, log: LoggingContext): Route
```

Description The `unmatchedPath` of the `RequestContext` is first transformed by the given `pathRewriter` function before being appended to the given directory name to build the final file name.

The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread. If the file cannot be read the route rejects the request.

To serve a single file use `getFromFile`. To serve browsable directory listings use `getFromBrowseableDirectories`. To serve files from a classpath directory use `getFromResourceDirectory` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to GET requests.

getFromFile Completes GET requests with the content of the given file.

Signature

```
def getFromFile(fileName: String)
    (implicit settings: RoutingSettings, resolver: ContentTypeResolver, refFactory: ActorRefFactory): Route
def getFromFile(file: File)
    (implicit settings: RoutingSettings, resolver: ContentTypeResolver, refFactory: ActorRefFactory): Route
def getFromFile(file: File, contentType: ContentType)
    (implicit settings: RoutingSettings, refFactory: ActorRefFactory): Route
```

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread!). If the file cannot be found or read the request is rejected.

To serve files from a directory use `getFromDirectory`, instead. To serve a file from a classpath resource use `getFromResource` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to GET requests.

getFromResource Completes GET requests with the content of the given classpath resource.

Signature

```
def getFromResource(resourceName: String)
    (implicit resolver: ContentTypeResolver, refFactory: ActorRefFactory): Route
def getFromResource(resourceName: String, contentType: ContentType)
    (implicit refFactory: ActorRefFactory): Route
```

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread!). If the file cannot be found or read the request is rejected.

To serve files from a classpath directory use `getFromResourceDirectory` instead. To serve files from a filesystem directory use `getFromDirectory`, instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to GET requests.

getFromResourceDirectory Completes GET requests with the content of the given classpath resource directory.

Signature

```
def getFromResourceDirectory(directoryName: String)
    (implicit resolver: ContentTypeResolver, refFactory: ActorRefFactory, log: LoggingContext): Route
```

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread!). If the file cannot be found or read the request is rejected.

To serve a single resource use `getFromResource`, instead. To server files from a filesystem directory use `getFromDirectory` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to GET requests.

listDirectoryContents Completes GET requests with a unified listing of the contents of all given directories. The actual rendering of the directory contents is performed by the in-scope *Marshaller[DirectoryListing]*.

Signature

```
def listDirectoryContents(directories: String*)
    (implicit renderer: Marshaller[DirectoryListing], refFactory: ActorRefFactory, log: LoggingContext): Route
```

Description The `listDirectoryContents` directive renders a response only for directories. To just serve files use `getFromDirectory`. To serve files and provide a browsable directory listing use `getFromBrowsableDirectories` instead.

The rendering can be overridden by providing a custom `Marshaller[DirectoryListing]`.

Note that it's not required to wrap this directive with `get` as this directive will only respond to GET requests.

respondWithLastModifiedHeader Adds a `Last-Modified` header to all `HttpResponses` from its inner `Route`.

Signature

```
def respondWithLastModifiedHeader(timestamp: Long): Directive0
```

FormFieldDirectives

formField An alias for `formFields`.

Signature

```
def formField(fdm: FieldDefMagnet): fdm.Out
```

Description See `formFields`.

formFields Extracts fields from POST requests generated by HTML forms.

Signature

```
def formFields(field: <FieldDef[T]>): Directive1[T]
def formFields(fields: <FieldDef[T_i]>...): Directive[T_0 :: ... T_i ... :: HNil]
def formFields(fields: <FieldDef[T_0]> :: ... <FieldDef[T_i]> ... :: HNil): Directive[T_0 :: ... T_i ... :: HNil]
```

The signature shown is simplified and written in pseudo-syntax, the real signature uses magnets.¹ The type `<FieldDef>` doesn't really exist but consists of the syntactic variants as shown in the description and the examples.

Description Form fields can be either extracted as a `String` or can be converted to another type. The parameter name can be supplied either as a `String` or as a `Symbol`. Form field extraction can be modified to mark a field as required or optional or to filter requests where a form field has a certain value:

`"color"` extract value of field "color" as `String`

`"color".?` extract optional value of field "color" as `Option[String]`

`"color" ? "red"` extract optional value of field "color" as `String` with default value "red"

`"color" ! "blue"` require value of field "color" to be "blue" and extract nothing

`"amount".as[Int]` extract value of field "amount" as `Int`, you need a matching `Deserializer` in scope for that to work (see also *Unmarshalling*)

`"amount".as(deserializer)` extract value of field "amount" with an explicit `Deserializer`

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

You can use *Case Class Extraction* to group several extracted values together into a case-class instance.

Requests missing a required field or field value will be rejected with an appropriate rejection.

There's also a singular version, `formField`. Query parameters can be handled in a similar way, see `parameters`. If you want unified handling for both query parameters and form fields, see `anyParams`.

Unmarshalling Data POSTed from **HTML forms** is either of type `application/x-www-form-urlencoded` or of type `multipart/form-data`. The value of an url-encoded field is a `String` while the value of a `multipart/form-data-encoded` field is a “body part” containing an entity. This means that different kind of deserializers are needed depending on what the `Content-Type` of the request is:

- A `application/x-www-form-urlencoded` encoded field needs an implicit `Deserializer[Option[String], T]`
- A `multipart/form-data` encoded field needs an implicit `Deserializer[Option[BodyPart], T]`

For common data-types, these implicits are predefined so that you usually don't need to care. For custom data-types it should usually suffice to create a `Deserializer[String, T]` if the value will be encoded as a `String`. This should be valid for all values generated by HTML forms apart from file uploads.

Details It should only be necessary to read and understand this paragraph if you have very special needs and need to process arbitrary forms, especially ones not generated by HTML forms.

The `formFields` directive contains this logic to find and decide how to deserialize a POSTed form field:

- It tries to find implicits of both types at the definition site if possible or otherwise at least one of both. If none is available compilation will fail with an “implicit not found” error.
- Depending on the `Content-Type` of the incoming request it first tries the matching (see above) one if available.
- If only a `Deserializer[Option[String], T]` is available when a request of type `multipart/form-data` is received, this deserializer will be tried to deserialize the body part for a field if the entity is of type `text/plain` or unspecified.
- If only a `Deserializer[Option[BodyPart], T]` is available when a request of type `application/x-www-form-urlencoded` is received, this deserializer will be tried to deserialize the field value by packing the field value into a body part with an entity of type `text/plain`. Deserializing will only succeed if the deserializer accepts entities of type `text/plain`.

If you need to handle encoded fields of a `multipart/form-data-encoded` request for a custom type, you therefore need to provide a `Deserializer[Option[BodyPart], T]`.

Example

```
val route =
  formFields('color, 'age.as[Int]) { (color, age) =>
    complete(s"The color is '$color' and the age ten years ago was ${age - 10}")
  }

Post("/", FormData(Seq("color" -> "blue", "age" -> "68"))) ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the age ten years ago was 58"
}

Get("/") ~> sealRoute(route) ~> check {
  status === StatusCodes.BadRequest
}
```

```
responseAs[String] === "Request is missing required form field 'color'"
}
```

For more examples about the way how fields can specified see the examples for the `parameters` directive.

FuturesDirectives

Future directives can be used to run inner routes once the provided `Future[T]` has been completed.

onComplete Evaluates its parameter of type `Future[T]`, and once the `Future` has been completed, extracts its result as a value of type `Try[T]` and passes it to the inner route.

Signature

```
def onComplete[T](future: Future[T])(implicit ec: ExecutionContext): Directive1[Try[T]]
```

The signature shown is simplified, the real signature uses magnets.¹

Description The evaluation of the inner route passed to a `onComplete` directive is deferred until the given future has completed and provided with a extraction of type `Try[T]`.

It is necessary to bring a `ExecutionContext` into implicit scope for this directive to work.

To handle the `Failure` case automatically and only work with the result value, use `onSuccess`. To complete with a successful result automatically and just handle the failure result, use `onFailure`, instead.

Example

```
def divide(a: Int, b: Int): Future[Int] = Future {
  a / b
}

val route =
  path("divide" / IntNumber / IntNumber) { (a, b) =>
    onComplete(divide(a, b)) {
      case Success(value) => complete(s"The result was $value")
      case Failure(ex)    => complete(InternalServerError, s"An error occurred: ${ex.getMessage}")
    }
  }

Get("/divide/10/2") ~> route ~> check {
  responseAs[String] === "The result was 5"
}

Get("/divide/10/0") ~> sealRoute(route) ~> check {
  status === InternalServerError
  responseAs[String] === "An error occurred: / by zero"
}
```

onSuccess Evaluates its parameter of type `Future[T]`, and once the `Future` has been completed successfully, extracts its result as a value of type `T` and passes it to the inner route.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Signature

```
def onSuccess(future: Future[T])(ec: ExecutionContext): Directive1[T]
def onSuccess(future: Future[L <: HList])(ec: ExecutionContext): Directive[L]
```

The signature shown is simplified, the real signature uses magnets. ¹

Description The execution of the inner route passed to a `onSuccess` directive is deferred until the given future has completed successfully, exposing the future's value as an extraction of type `T`. If the future fails its failure throwable is bubbled up to the nearest `ExceptionHandler`.

It is necessary to bring a `ExecutionContext` into implicit scope for this directive to work.

To handle the `Failure` case manually as well, use `onComplete`, instead.

Example

```
val route =
  path("success") {
    onSuccess(Future { "Ok" }) { extraction =>
      complete(extraction)
    }
  } ~
  path("failure") {
    onSuccess(Future.failed[String](TestException)) { extraction =>
      complete(extraction)
    }
  }

Get("/success") ~> route ~> check {
  responseAs[String] === "Ok"
}

Get("/failure") ~> sealRoute(route) ~> check {
  status === InternalServerError
  responseAs[String] === "Unsuccessful future!"
}
```

onFailure Completes the request with the result of the computation given as argument of type `Future[T]` by marshalling it with the implicitly given `ToResponseMarshaller[T]`. Runs the inner route if the `Future` computation fails.

Signature

```
def onFailure(future: Future[T])(implicit m: ToResponseMarshaller[T], ec: ExecutionContext): Directive1[T]
```

The signature shown is simplified, the real signature uses magnets. ¹

Description If the future succeeds the request is completed using the values marshaller (this directive therefore requires a marshaller for the future's type to be implicitly available). The execution of the inner route passed to a `onFailure` directive is deferred until the given future has completed with a failure, exposing the reason of failure as an extraction of type `Throwable`.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

It is necessary to bring a `ExecutionContext` into implicit scope for this directive to work.

To handle the successful case manually as well, use the `onComplete` directive, instead.

Example

```
val route =
  path("success") {
    onFailure(Future { "Ok" }) { extraction =>
      failWith(extraction) // not executed.
    }
  } ~
  path("failure") {
    onFailure(Future.failed[String](TestException)) { extraction =>
      failWith(extraction)
    }
  }

Get("/success") ~> route ~> check {
  responseAs[String] === "Ok"
}

Get("/failure") ~> sealRoute(route) ~> check {
  status === InternalServerError
  responseAs[String] === "Unsuccessful future!"
}
```

All future directives take a by-name parameter so that the parameter is not evaluated at route building time but only when the request comes in.

HeaderDirectives

Header directives can be used to extract header values from the request. To change response headers use one of the *RespondWithDirectives*.

headerValue Traverses the list of request headers with the specified function and extracts the first value the function returns as `Some(value)`.

Signature

```
def headerValue[T](f: HttpHeadersOption[T]): Directive1[T]
```

Description The `headerValue` directive is a mixture of `map` and `find` on the list of request headers. The specified function is called once for each header until the function returns `Some(value)`. This value is extracted and presented to the inner route. If the function throws an exception the request is rejected with a `MalformedHeaderRejection`. If the function returns `None` for every header the request is rejected as “Not-Found”.

This directive is the basis for building other request header related directives. See `headerValuePF` for a nicer syntactic alternative.

Example

```
def extractHostPort: HttpHeader => Option[Int] = {
  case h: `Host` => Some(h.port)
  case x => None
}

val route =
  headerValue(extractHostPort) { port =>
    complete(s"The port was $port")
  }

Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] === "The port was 5043"
}

Get("/") ~> sealRoute(route) ~> check {
  status === NotFound
  responseAs[String] === "The requested resource could not be found."
}
```

headerValueByName Extracts the value of the HTTP request header with the given name.

Signature

```
def headerValueByName(headerName: Symbol): Directive1[String]
def headerValueByName(headerName: String): Directive1[String]
```

Description The name can be given as a `String` or as a `Symbol`. If no header with a matching name is found the request is rejected with a `MissingHeaderRejection`. If the header is expected to be missing in some cases or to customize handling when the header is missing use the *optionalHeaderValueByName* directive instead.

Example

```
val route =
  headerValueByName("X-User-Id") { userId =>
    complete(s"The user is $userId")
  }

Get("/") ~> RawHeader("X-User-Id", "Joe42") ~> route ~> check {
  responseAs[String] === "The user is Joe42"
}

Get("/") ~> sealRoute(route) ~> check {
  status === BadRequest
  responseAs[String] === "Request is missing required HTTP header 'X-User-Id'"
}
```

headerValueByType Traverses the list of request headers and extracts the first header of the given type.

Signature

```
def headerValueByType[T <: HttpHeader: ClassTag](): Directive1[T]
```

The signature shown is simplified, the real signature uses magnets. ¹

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Description The `headerValueByType` directive finds a header of the given type in the list of request header. If no header of the given type is found the request is rejected with a `MissingHeaderRejection`. If the header is expected to be missing in some cases or to customize handling when the header is missing use the `optionalHeaderValueByType` directive instead.

Example

```
val route =
  headerValueByType[Origin]() { origin
    complete(s"The first origin was ${origin.originList.head}")
  }

val originHeader = Origin(Seq(HttpOrigin("http://localhost:8080")))

// extract a header if the type is matching
Get("abc") ~> originHeader ~> route ~> check {
  responseAs[String] === "The first origin was http://localhost:8080"
}

// reject a request if no header of the given type is present
Get("abc") ~> route ~> check {
  rejection must beLike { case MissingHeaderRejection("Origin") ok }
}
```

headerValuePF Calls the specified partial function with the first request header the function is `isDefinedAt` and extracts the result of calling the function.

Signature

```
def headerValuePF[T](pf: PartialFunction[HttpHeader, T]): Directive1[T]
```

Description The `headerValuePF` directive is an alternative syntax version of `headerValue`. If the function throws an exception the request is rejected with a `MalformedHeaderRejection`. If the function is not defined for any header the request is rejected as “`NotFound`”.

Example

```
def extractHostPort: PartialFunction[HttpHeader, Int] = {
  case h: `Host` => h.port
}

val route =
  headerValuePF(extractHostPort) { port =>
    complete(s"The port was $port")
  }

Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] === "The port was 5043"
}

Get("/") ~> sealRoute(route) ~> check {
  status === NotFound
  responseAs[String] === "The requested resource could not be found."
}
```

optionalHeaderValue Traverses the list of request headers with the specified function and extracts the first value the function returns as `Some (value)`.

Signature

```
def optionalHeaderValue[T](f: HttpHeaders Option[T]) : Directive1[Option[T]]
```

Description The `optionalHeaderValue` directive is similar to the `headerValue` directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

optionalHeaderValueByName Optionally extracts the value of the HTTP request header with the given name.

Signature

```
def optionalHeaderValueByName(headerName: Symbol) : Directive1[Option[String]]
def optionalHeaderValueByName(headerName: String) : Directive1[Option[String]]
```

Description The `optionalHeaderValueByName` directive is similar to the `headerValueByName` directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

optionalHeaderValueByType Optionally extracts the value of the HTTP request header of the given type.

Signature

```
def optionalHeaderValueByType[T <: HttpHeaders: ClassTag]() : Directive1[Option[T]]
```

The signature shown is simplified, the real signature uses magnets. ¹

Description The `optionalHeaderValueByType` directive is similar to the `headerValueByType` directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

Example

```
val route =
  optionalHeaderValueByType[Origin]() {
    case Some(origin) complete(s"The first origin was ${origin.originList.head}")
    case None          complete("No Origin header found.")
  }

val originHeader = Origin(Seq(HttpOrigin("http://localhost:8080")))
// extract Some(header) if the type is matching
Get("abc") ~> originHeader ~> route ~> check {
  responseAs[String] === "The first origin was http://localhost:8080"
}

// extract None if no header of the given type is present
Get("abc") ~> route ~> check {
  responseAs[String] === "No Origin header found."
}
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

optionalHeaderValuePF Calls the specified partial function with the first request header the function is `isDefinedAt` and extracts the result of calling the function.

Signature

```
def optionalHeaderValuePF[T](pf: PartialFunction[HttpHeader, T]): Directive1[Option[T]]
```

Description The `optionalHeaderValuePF` directive is similar to the `headerValuePF` directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

HostDirectives

HostDirectives allow you to filter requests based on the hostname part of the `Host` header contained in incoming requests as well as extracting its value for usage in inner routes.

host Filter requests matching conditions against the hostname part of the `Host` header value in the request.

Signature

```
def host(hostNames: String*): Directive0
def host(predicate: String Boolean): Directive0
def host(regex: Regex): Directive1[String]
```

Description The `def host(hostNames: String*)` overload rejects all requests with a hostname different from the given ones.

The `def host(predicate: String Boolean)` overload rejects all requests for which the hostname does not satisfy the given predicate.

The `def host(regex: Regex)` overload works a little bit different: it rejects all requests with a hostname that doesn't have a prefix matching the given regular expression and also extracts a `String` to its inner route following this rules:

- For all matching requests the prefix string matching the regex is extracted and passed to the inner route.
- If the regex contains a capturing group only the string matched by this group is extracted.
- If the regex contains more than one capturing group an `IllegalArgumentException` is thrown.

Example Matching a list of hosts:

```
val route =
  host("api.company.com", "rest.company.com") {
    complete("Ok")
  }

Get() ~> Host("rest.company.com") ~> route ~> check {
  status === OK
  responseAs[String] === "Ok"
}

Get() ~> Host("notallowed.company.com") ~> route ~> check {
  handled must beFalse
}
```

Making sure the host satisfies the given predicate

```
val shortOnly: String => Boolean = (hostname) => hostname.length < 10

val route =
  host(shortOnly) {
    complete("Ok")
  }

Get() ~> Host("short.com") ~> route ~> check {
  status === OK
  responseAs[String] === "Ok"
}

Get() ~> Host("verylonghostname.com") ~> route ~> check {
  handled must beFalse
}
```

Using a regular expressions:

```
val route =
  host("api|rest".r) { prefix =>
    complete(s"Extracted prefix: $prefix")
  } ~
  host("public.(my|your)company.com".r) { captured =>
    complete(s"You came through $captured company")
  }

Get() ~> Host("api.company.com") ~> route ~> check {
  status === OK
  responseAs[String] === "Extracted prefix: api"
}

Get() ~> Host("public.mycompany.com") ~> route ~> check {
  status === OK
  responseAs[String] === "You came through my company"
}
```

Beware that in the case of introducing multiple capturing groups in the regex such as in the case bellow, the directive will fail at runtime, at the moment the route tree is evaluated for the first time. This might cause your http handler actor to enter in a fail/restart loop depending on your supervision strategy.

```
{
  host("server-([0-9]).company.(com|net|org)".r) { target =>
    complete("Will never complete :'(")
  }
} must throwAn[IllegalArgumentException]
```

hostName Extracts the hostname part of the Host header value in the request.

Signature

```
def hostName: Directive1[String]
```

Description Extract the hostname part of the Host request header and expose it as a String extraction to its inner route.

Example

```

val route =
  hostname { hn =>
    complete(s"Hostname: $hn")
  }

Get() ~> Host("company.com", 9090) ~> route ~> check {
  status === OK
  responseAs[String] === "Hostname: company.com"
}

```

Marshalling Directives

Marshalling directives work in conjunction with `spray.httpx.marshalling` and `spray.httpx.unmarshalling` to convert a request entity to a specific type or a type to a response. See *marshalling* and *unmarshalling* for specific serialization (also known as pickling) guidance.

Marshalling directives usually rely on an in-scope implicit marshaller to handle conversion.

entity Unmarshalls the request entity to the given type and passes it to its inner Route. An unmarshaller returns an `Either` with `Right` (value) if successful or `Left` (exception) for a failure. The `entity` method will either pass the value to the inner route or map the exception to a `spray.routing.Rejection`.

Signature

```
def entity[T](um: FromRequestUnmarshaller[T]): Directive1[T]
```

Description The `entity` directive works in conjunction with `as` and `spray.httpx.unmarshalling` to convert some serialized “wire format” value into a higher-level object structure. *The unmarshalling documentation* explains this process in detail. This directive simplifies extraction and error handling to the specified type from the request.

An unmarshaller will return a `Left(exception)` in the case of an error. This is converted to a `spray.routing.Rejection` within the `entity` directive. The following table lists how exceptions are mapped to rejections:

Left(exception)	Rejection
<code>ContentExpected</code>	<code>RequestEntityExpectedRejection</code>
<code>UnsupportedContentType</code>	<code>UnsupportedRequestContentTypeRejection</code> , which lists the supported types
<code>MalformedContent</code>	<code>MalformedRequestContentRejection</code> , with an error message and cause

Examples The following example uses `spray-json` to unmarshall a json request into a simple `Person` class. It utilizes `SprayJsonSupport` via the `PersonJsonSupport` object as the in-scope unmarshaller.

```
case class Person(name: String, favoriteNumber: Int)
```

```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortfolioFormats = jsonFormat2(Person)
}
```

```
import PersonJsonSupport._

val route = post {
  entity(as[Person]) { person =>
    complete(s"Person: ${person.name} - favorite number: ${person.favoriteNumber}")
  }
}

Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
  route ~> check {
    responseAs[String] === "Person: Jane - favorite number: 42"
  }
}
```

produce Uses the marshaller for a given type to produce a completion function that is passed to its inner route. You can use it to decouple marshaller resolution from request completion.

Signature

```
def produce[T](marshaller: ToResponseMarshaller[T]): Directive[(T Unit) :: HNil]
```

Description The `produce` directive works in conjunction with `instanceOf` and `spray.httpx.marshalling` to convert higher-level (object) structure into some lower-level serialized “wire format”. *The marshalling documentation* explains this process in detail. This directive simplifies exposing types to clients via a route while providing some form of access to the current context.

`produce` is similar to `handleWith`. The main difference is with `produce` you must eventually call the completion function generated by `produce`. `handleWith` will automatically call `complete` when the `handleWith` function returns.

Examples The following example uses `spray-json` to marshal a simple `Person` class to a json response. It utilizes `SprayJsonSupport` via the `PersonJsonSupport` object as the in-scope unmarshaller.

```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
```

```
case class Person(name: String, favoriteNumber: Int)
```

The `findPerson` takes an argument of type `Person => Unit` which is generated by the `produce` call. We can handle any logic we want in `findPerson` and call our completion function to complete the request.

```
import PersonJsonSupport._

val findPerson = (f: Person => Unit) => {
  //... some processing logic...

  //complete the request
  f(Person("Jane", 42))
}

val route = get {
  produce(instanceOf[Person]) { completionFunction => ctx => findPerson(completionFunction) }
}
```

```
Get("/") ~> route ~> check {
  mediaType === `application/json`
  responseAs[String] must contain("""name": "Jane""")
  responseAs[String] must contain("""favoriteNumber": 42""")
}
```

handleWith Completes the request using the given function. The input to the function is produced with the in-scope entity unmarshaller and the result value of the function is marshalled with the in-scope marshaller. `handleWith` can be a convenient method combining `entity` with `complete`.

Signature

```
def handleWith[A, B](f: A => B)(implicit um: FromRequestUnmarshaller[A], m: ToResponseMarshaller[B]): I
```

Description The `handleWith` directive is used when you want to handle a route with a given function of type `A` to `B`. `handleWith` will use both an in-scope unmarshaller to convert a request into type `A` and an in-scope marshaller to convert type `B` into a response. This is helpful when your core business logic resides in some other class or you want your business logic to be independent of *Spray*. You can use `handleWith` to “hand off” processing to a given function without requiring any spray-specific functionality.

`handleWith` is similar to `produce`. The main difference is `handleWith` automatically calls `complete` when the function passed to `handleWith` returns. Using `produce` you must explicitly call the completion function passed from the `produce` function.

See *marshalling* and *unmarshalling* for guidance on marshalling entities with *Spray*.

Examples The following example uses an `updatePerson` function with a `Person` case class as an input and output. We plug this function into our route using `handleWith`.

```
case class Person(name: String, favoriteNumber: Int)
```

```
import PersonJsonSupport._

val updatePerson = (person: Person) => {
  //... some processing logic...

  //return the person
  person
}

val route = post {
  handleWith(updatePerson)
}

Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
route ~> check {
  mediaType === `application/json`
  responseAs[String] must contain("""name": "Jane""")
  responseAs[String] must contain("""favoriteNumber": 42""")
}
```

The `PersonJsonSupport` object handles both marshalling and unmarshalling of the `Person` case class.


```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
```

Understanding Specific Marshalling Directives

directive	behavior
<i>entity</i>	Unmarshalls the request entity to the given type and passes it to its inner route to convert requests to objects.
<i>produce</i>	Uses a marshaller for a given type to produce a completion function for a given route in conjunction with <i>instanceOf</i> to format responses.
<i>handle- With</i>	Completes a request with a given function, using an in-scope unmarshaller for the output.

MethodDirectives

delete Matches requests with HTTP method DELETE.

Signature

```
def delete: Directive0
```

Description This directive filters an incoming request by its HTTP method. Only requests with method DELETE are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = Directives.delete { complete("This is a DELETE request.") }

Delete("/") ~> route ~> check {
  responseAs[String] === "This is a DELETE request."
}
```

get Matches requests with HTTP method GET.

Signature

```
def get: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method GET are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = get { complete("This is a GET request.") }

Get("/") ~> route ~> check {
  responseAs[String] === "This is a GET request."
}
```

head Matches requests with HTTP method HEAD.

Signature

```
def head: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method HEAD are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Note: By default, spray-can handles HEAD-requests transparently by dispatching a GET-request to the handler and stripping of the result body. See the `spray.can.server.transparent-head-requests` setting for how to disable this behavior.

Example

```
val route = head { complete("This is a HEAD request.") }

Head("/") ~> route ~> check {
  responseAs[String] === "This is a HEAD request."
}
```

method Matches HTTP requests based on their method.

Signature

```
/**
 * Rejects all requests whose HTTP method does not match the given one.
 */
def method(httpMethod: HttpMethod): Directive0 =
  extract(_.request.method).flatMap[HNil] {
    case `httpMethod` pass
    case _             reject(MethodRejection(httpMethod))
  } & cancelAllRejections(ofType[MethodRejection])
```

Description This directive filters the incoming request by its HTTP method. Only requests with the specified method are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = method(HttpMethods.PUT) { complete("This is a PUT request.") }

Put("/", "put content") ~> route ~> check {
  responseAs[String] === "This is a PUT request."
}

Get("/") ~> sealRoute(route) ~> check {
  status === StatusCodes.MethodNotAllowed
  responseAs[String] === "HTTP method not allowed, supported methods: PUT"
}
```

options Matches requests with HTTP method OPTIONS.

Signature

```
def options: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method OPTIONS are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = options { complete("This is an OPTIONS request.") }

Options("/") ~> route ~> check {
  responseAs[String] === "This is an OPTIONS request."
}
```

overrideMethodWithParameter Changes the HTTP method of the request to the value of the specified query string parameter. If the query string parameter is not specified this directive has no effect. If the query string is specified as something that is not a HTTP method, then this directive completes the request with a 501 Not Implemented response.

This directive is useful for:

- Use in combination with JSONP (JSONP only supports GET)
- Supporting older browsers that lack support for certain HTTP methods. E.g. IE8 does not support PATCH

patch Matches requests with HTTP method PATCH.

Signature

```
def patch: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method PATCH are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = patch { complete("This is a PATCH request.") }

Patch("/", "patch content") ~> route ~> check {
  responseAs[String] === "This is a PATCH request."
}
```

post Matches requests with HTTP method POST.

Signature

```
def post: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method `POST` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = post { complete("This is a POST request.") }

Post("/", "post content") ~> route ~> check {
  responseAs[String] === "This is a POST request."
}
```

put Matches requests with HTTP method `PUT`.

Signature

```
def put: Directive0
```

Description This directive filters the incoming request by its HTTP method. Only requests with method `PUT` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a 405 Method Not Allowed response by the default `RejectionHandler`.

Example

```
val route = put { complete("This is a PUT request.") }

Put("/", "put content") ~> route ~> check {
  responseAs[String] === "This is a PUT request."
}
```

MiscDirectives

cancelAllRejections Cancels all rejections created by the inner route for which the condition argument function returns `true`.

Signature

```
def cancelAllRejections(cancelFilter: Rejection Boolean): Directive0
```

Description Use the `cancelRejection` to cancel a specific rejection instance.

Example

```
def isMethodRejection: Rejection => Boolean = {
  case MethodRejection(_) => true
  case _ => false
}

val route =
  cancelAllRejections(isMethodRejection) {
    post {
      complete("Result")
    }
  }
```

```

    }
  }
  Get("/") ~> route ~> check {
    rejections === Nil
    handled === false
  }
}

```

cancelRejection Cancels a rejection that is equal to the given one.

Signature

```
def cancelRejection(rejection: Rejection): Directive0
```

Description Use `cancelAllRejections` to cancel rejections by predicate.

Example

```

val route =
  cancelRejection(MethodRejection(HttpMethods.POST)) {
    post {
      complete("Result")
    }
  }

Get("/") ~> route ~> check {
  rejections === Nil
  handled === false
}

```

clientIP Provides the value of X-Forwarded-For, Remote-Address, or X-Real-IP headers as an instance of `HttpIp`.

Signature

```
def clientIP: Directive1[RemoteAddress]
```

Description `spray-can` and `spray-servlet` adds the `Remote-Address` header to every request automatically if the respective setting `spray.can.server.remote-address-header` or `spray.servlet.remote-address-header` is set to `on`. Per default it is set to `off`.

Example

```

val route = clientIP { ip =>
  complete("Client's ip is " + ip.toOption.map(_.getHostAddress).getOrElse("unknown"))
}

Get("/").withHeaders(`Remote-Address`("192.168.3.12")) ~> route ~> check {
  responseAs[String] === "Client's ip is 192.168.3.12"
}

```

jsonpWithParameter Wraps a response of type `application/json` with an invocation to a callback function which name is given as an argument. The new type of the response is `application/javascript`.

Signature

```
def jsonpWithParameter(parameterName: String): Directive0
```

Description Find more information about JSONP in [Wikipedia](#). Note that JSONP is not considered the solution of choice for many reasons. Be sure to understand its drawbacks and security implications.

Example

```
case class Test(abc: Int)
object TestProtocol {
  import spray.json.DefaultJsonProtocol._
  implicit val testFormat = jsonFormat(Test, "abc")
}
val route =
  jsonpWithParameter("jsonp") {
    import TestProtocol._
    import spray.httpx.SprayJsonSupport._
    complete(Test(456))
  }

Get("/?jsonp=result") ~> route ~> check {
  responseAs[String] ===
  """result({
    | "abc": 456
  })""".stripMarginWithNewline("\n")
  contentType === MediaType.`application/javascript`.withCharset(HttpCharsets.`UTF-8`)
}
Get("/") ~> route ~> check {
  responseAs[String] ===
  """{
    | "abc": 456
  }""".stripMarginWithNewline("\n")
  contentType === ContentType.`application/json`
}
```

rejectEmptyResponse Replaces a response with no content with an empty rejection.

Signature

```
def rejectEmptyResponse: Directive0
```

Description The `rejectEmptyResponse` directive is mostly used with marshalling `Option[T]` instances. The value `None` is usually marshalled to an empty but successful result. In many cases `None` should instead be handled as `404 Not Found` which is the effect of using `rejectEmptyResponse`.

Example

```
val route = rejectEmptyResponse {
  path("even" / IntNumber) { i =>
    complete {
```

```

    // returns Some(evenNumberDescription) or None
    Option(i).filter(_ % 2 == 0).map { num =>
      s"Number $num is even."
    }
  }
}
}

Get("/even/23") ~> sealRoute(route) ~> check {
  status === StatusCodes.NotFound
}
Get("/even/28") ~> route ~> check {
  responseAs[String] === "Number 28 is even."
}

```

requestEntityEmpty A filter that checks if the request entity is empty and only then passes processing to the inner route. Otherwise, the request is rejected.

Signature

```
def requestEntityEmpty: Directive0
```

Description The opposite filter is available as `requestEntityPresent`.

Example

```

val route =
  requestEntityEmpty {
    complete("request entity empty")
  } ~
  requestEntityPresent {
    complete("request entity present")
  }

Post("/", "text") ~> sealRoute(route) ~> check {
  responseAs[String] === "request entity present"
}
Post("/") ~> route ~> check {
  responseAs[String] === "request entity empty"
}

```

requestEntityPresent A simple filter that checks if the request entity is present and only then passes processing to the inner route. Otherwise, the request is rejected.

Signature

```
def requestEntityPresent: Directive0
```

Description The opposite filter is available as `requestEntityEmpty`.

Example

```
val route =
  requestEntityEmpty {
    complete("request entity empty")
  } ~
  requestEntityPresent {
    complete("request entity present")
  }

Post("/", "text") ~> sealRoute(route) ~> check {
  responseAs[String] === "request entity present"
}
Post("/") ~> route ~> check {
  responseAs[String] === "request entity empty"
}
```

requestInstance Extracts the complete `HttpRequest` instance.

Signature

```
def requestInstance: Directive1[HttpRequest]
```

Description Use `requestUri` to extract just the complete URI of the request. Usually there's little use of extracting the complete request because extracting of most of the aspects of `HttpRequests` is handled by specialized directives. See *Directives filtering or extracting from the request*.

Example

```
val route =
  requestInstance { request =>
    complete(s"Request method is ${request.method} and length is ${request.entity.data.length}")
  }

Post("/", "text") ~> route ~> check {
  responseAs[String] === "Request method is POST and length is 4"
}
Get("/") ~> route ~> check {
  responseAs[String] === "Request method is GET and length is 0"
}
```

requestUri Access the full URI of the request.

Signature

```
def requestUri: Directive1[Uri]
```

Description Use *SchemeDirectives*, *HostDirectives*, *PathDirectives*, and *ParameterDirectives* for more targeted access to parts of the URI.

Example


```

val route =
  requestUri { uri =>
    complete(s"Full URI: $uri")
  }

Get("/") ~> route ~> check {
  // tests are executed with the host assumed to be "example.com"
  responseAs[String] === "Full URI: http://example.com/"
}
Get("/test") ~> route ~> check {
  responseAs[String] === "Full URI: http://example.com/test"
}

```

rewriteUnmatchedPath Transforms the `unmatchedPath` field of the request context for inner routes.

Signature

```
def rewriteUnmatchedPath(f: Uri.Path Uri.Path): Directive0
```

Description The `rewriteUnmatchedPath` directive is used as a building block for writing *Custom Directives*. You can use it for implementing custom path matching directives.

Use `unmatchedPath` for extracting the current value of the unmatched path.

Example

```

def ignore456(path: Path) = path match {
  case s@Path.Segment(head, tail) if head.startsWith("456") =>
    val newHead = head.drop(3)
    if (newHead.isEmpty) tail
    else s.copy(head = head.drop(3))
  case _ => path
}
val ignoring456 = rewriteUnmatchedPath(ignore456)

val route =
  pathPrefix("123") {
    ignoring456 {
      path("abc") {
        complete(s"Content")
      }
    }
  }

Get("/123/abc") ~> route ~> check {
  responseAs[String] === "Content"
}
Get("/123456/abc") ~> route ~> check {
  responseAs[String] === "Content"
}

```

unmatchedPath Extracts the unmatched path from the request context.

Signature

```
def unmatchedPath: Directive1[Uri.Path]
```

Description The `unmatchedPath` directive extracts the remaining path that was not yet matched by any of the *PathDirectives* (or any custom ones that change the unmatched path field of the request context). You can use it for building directives that handle complete suffixes of paths (like the `getFromDirectory` directives and similar ones).

Use `rewriteUnmatchedPath` to change the value of the unmatched path.

Example

```
val route =
  pathPrefix("abc") {
    unmatchedPath { remaining =>
      complete(s"Unmatched: '$remaining'")
    }
  }

Get("/abc") ~> route ~> check {
  responseAs[String] === "Unmatched: ''"
}
Get("/abc/456") ~> route ~> check {
  responseAs[String] === "Unmatched: '/456'"
}
```

validate Checks an arbitrary condition and passes control to the inner route if it returns `true`. Otherwise, rejects the request with a `ValidationRejection` containing the given error message.

Signature

```
def validate(check: Boolean, errorMsg: String): Directive0
```

Example

```
val route =
  requestUri { uri =>
    validate(uri.path.toString.size < 5, s"Path too long: '${uri.path.toString}'") {
      complete(s"Full URI: $uri")
    }
  }

Get("/234") ~> route ~> check {
  responseAs[String] === "Full URI: http://example.com/234"
}
Get("/abcdefghijkl") ~> route ~> check {
  rejection === ValidationRejection("Path too long: '/abcdefghijkl'", None)
}
```

ParameterDirectives

parameter An alias for *parameters*.

Signature

```
def parameter(pdm: ParamDefMagnet): pdm.Out
```

Description See *parameters*

Example

```
val route =
  parameter('color) { color =>
    complete(s"The color is '$color'")
  }

Get("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue'"
}

Get("/") ~> sealRoute(route) ~> check {
  status === StatusCodes.NotFound
  responseAs[String] === "Request is missing required query parameter 'color'"
}
```

parameterMap Extracts all parameters at once as a Map[String, String] mapping parameter names to parameter values.

Signature

```
def parameterMap: Directive[Map[String, String] :: HNil]
```

Description If a query contains a parameter value several times, the map will contain the last one.

See *When to use which parameter directive?* for other choices.

Example

```
val route =
  parameterMap { params =>
    def paramString(param: (String, String)): String = s"$param._1 = '$param._2'"
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }

Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] === "The parameters are color = 'blue', count = '42'"
}

Get("/?x=1&x=2") ~> route ~> check {
  responseAs[String] === "The parameters are x = '2'"
}
```

parameterMultiMap Extracts all parameters at once as a multi-map of type Map[String, List[String]] mapping a parameter name to a list of all its values.

Signature

```
def parameterMultiMap: Directive[Map[String, List[String]] :: HNil]
```

Description This directive can be used if parameters can occur several times. The order of values is not specified. See *When to use which parameter directive?* for other choices.

Example

```
val route =
  parameterMultiMap { params =>
    complete(s"There are parameters ${params.map(x => x._1+" -> "+x._2.size).mkString(", ")}")
  }

Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] === "There are parameters color -> 1, count -> 1"
}
Get("/?x=23&x=42") ~> route ~> check {
  responseAs[String] === "There are parameters x -> 2"
}
```

parameters The parameters directive filters on the existence of several query parameters and extract their values.

Signature

```
def parameters(param: <ParamDef[T]>): Directive1[T]
def parameters(params: <ParamDef[T_i]>...): Directive[T_0 :: ... T_i :: HNil]
def parameters(params: <ParamDef[T_0]> :: ... <ParamDef[T_i]> ... :: HNil): Directive[T_0 :: ... T_i
```

The signature shown is simplified and written in pseudo-syntax, the real signature uses magnets.¹ The type `<ParamDef>` doesn't really exist but consists of the syntactic variants as shown in the description and the examples.

Description Query parameters can be either extracted as a String or can be converted to another type. The parameter name can be supplied either as a String or as a Symbol. Parameter extraction can be modified to mark a query parameter as required or optional or to filter requests where a parameter has a certain value:

"color" extract value of parameter "color" as String

"color".? extract optional value of parameter "color" as Option[String]

"color" ? "red" extract optional value of parameter "color" as String with default value "red"

"color" ! "blue" require value of parameter "color" to be "blue" and extract nothing

"amount".as[Int] extract value of parameter "amount" as Int, you need a matching `Deserializer` in scope for that to work (see also *Unmarshalling*)

"amount".as(deserializer) extract value of parameter "amount" with an explicit `Deserializer`

You can use *Case Class Extraction* to group several extracted values together into a case-class instance.

Requests missing a required parameter or parameter value will be rejected with an appropriate rejection.

There's also a singular version, *parameter*. Form fields can be handled in a similar way, see `formFields`. If you want unified handling for both query parameters and form fields, see `anyParams`.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Examples

Required parameter

```

val route =
  parameters('color, 'backgroundColor) { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

Get ("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'red'"
}

Get ("/?color=blue") ~> sealRoute(route) ~> check {
  status === StatusCodes.NotFound
  responseAs[String] === "Request is missing required query parameter 'backgroundColor'"
}

```

Optional parameter

```

val route =
  parameters('color, 'backgroundColor.?) { (color, backgroundColor) =>
    val backgroundStr = backgroundColor.getOrElse("<undefined>")
    complete(s"The color is '$color' and the background is '$backgroundStr'")
  }

Get ("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'red'"
}

Get ("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is '<undefined>'"
}

val route =
  parameters('color, 'backgroundColor ? "white") { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

Get ("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'red'"
}

Get ("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'white'"
}

```

Optional parameter with default value

```

val route =
  parameters('color, 'backgroundColor ? "white") { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

Get ("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'red'"
}

Get ("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and the background is 'white'"
}

```

Parameter with required value

```

val route =
  parameters('color, 'action ! "true") { (color) =>
    complete(s"The color is '$color'.")
  }

Get("/?color=blue&action=true") ~> route ~> check {
  responseAs[String] === "The color is 'blue'."
}

Get("/?color=blue&action=false") ~> sealRoute(route) ~> check {
  status === StatusCodes.NotFound
  responseAs[String] === "The requested resource could not be found."
}

```

Deserialized parameter

```

val route =
  parameters('color, 'count.as[Int]) { (color, count) =>
    complete(s"The color is '$color' and you have $count of it.")
  }

Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and you have 42 of it."
}

Get("/?color=blue&count=blub") ~> sealRoute(route) ~> check {
  status === StatusCodes.BadRequest
  responseAs[String] === "The query parameter 'count' was malformed:\n'blub' is not a valid 32-bit in
}

```

parameterSeq Extracts all parameters at once in the original order as (name, value) tuples of type (String, String).

Signature

```
def parameterSeq: Directive[Seq[(String, String)] :: HNil]
```

Description This directive can be used if the exact order of parameters is important or if parameters can occur several times.

See *When to use which parameter directive?* for other choices.

Example

```

val route =
  parameterSeq { params =>
    def paramString(param: (String, String)): String = s""${param._1} = '${param._2}'""
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }

Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] === "The parameters are color = 'blue', count = '42'"
}

Get("/?x=1&x=2") ~> route ~> check {

```

```
responseAs[String] === "The parameters are x = '1', x = '2'"
}
```

When to use which parameter directive? Usually, you want to use the high-level *parameters* directive. When you need more low-level access you can use the table below to decide which directive to use which shows properties of different parameter directives.

directive	level	ordering	multi
<i>parameter</i>	high	no	no
<i>parameters</i>	high	no	no
<i>parameterMap</i>	low	no	no
<i>parameterMultiMap</i>	low	no	yes
<i>parameterSeq</i>	low	yes	yes

level high-level parameter directives extract subset of all parameters by name and allow conversions and automatically report errors if expectations are not met, low-level directives give you all parameters at once, leaving all further processing to you

ordering original ordering from request URL is preserved

multi multiple values per parameter name are possible

PathDirectives

path Matches the complete unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

```
def path[L <: HList] (pm: PathMatcher[L]): Directive[L]
```

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing *pathPrefix* directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to the *rawPathPrefix* or *rawPathPrefixTest* directives `path` automatically adds a leading slash to its `PathMatcher` argument, you therefore don't have to start your matching expression with an explicit slash.

The `path` directive attempts to match the **complete** remaining path, not just a prefix. If you only want to match a path prefix and then delegate further filtering to a lower level in your routing structure use the *pathPrefix* directive instead. As a consequence it doesn't make sense to nest a `path` or *pathPrefix* directive underneath another `path` directive, as there is no way that they will ever match (since the unmatched path underneath a `path` directive will always be empty).

Depending on the type of its `PathMatcher` argument the `path` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val route =
  path("foo") {
    complete("/foo")
  } ~
  path("foo" / "bar") {
    complete("/foo/bar")
  }
```

```
} ~
pathPrefix("ball") {
  pathEnd {
    complete("/ball")
  } ~
  path(IntNumber) { int =>
    complete(if (int % 2 == 0) "even ball" else "odd ball")
  }
}

Get("/") ~> route ~> check {
  handled === false
}

Get("/foo") ~> route ~> check {
  responseAs[String] === "/foo"
}

Get("/foo/bar") ~> route ~> check {
  responseAs[String] === "/foo/bar"
}

Get("/ball/1337") ~> route ~> check {
  responseAs[String] === "odd ball"
}
```

pathEnd Only passes the request to its inner route if the unmatched path of the RequestContext is empty, i.e. the request path has been fully matched by a higher-level *path* or *pathPrefix* directive.

Signature

```
def pathEnd: Directive0
```

Description This directive is a simple alias for `rawPathPrefix(PathEnd)` and is mostly used on an inner-level to discriminate “path already fully matched” from other alternatives (see the example below).

Example

```
val route =
  pathPrefix("foo") {
    pathEnd {
      complete("/foo")
    } ~
    path("bar") {
      complete("/foo/bar")
    }
  }

Get("/foo") ~> route ~> check {
  responseAs[String] === "/foo"
}

Get("/foo/") ~> route ~> check {
  handled === false
}
```



```
Get("/foo/bar") ~> route ~> check {
  responseAs[String] === "/foo/bar"
}
```

pathEndOrSingleSlash Only passes the request to its inner route if the unmatched path of the `RequestContext` is either empty or contains only one single slash.

Signature

```
def pathEndOrSingleSlash: Directive0
```

Description This directive is a simple alias for `rawPathPrefix(Slash.? ~ PathEnd)` and is mostly used on an inner-level to discriminate “path already fully matched” from other alternatives (see the example below).

It is equivalent to `pathEnd | pathSingleSlash` but slightly more efficient.

Example

```
val route =
  pathPrefix("foo") {
    pathEndOrSingleSlash {
      complete("/foo")
    } ~
    path("bar") {
      complete("/foo/bar")
    }
  }

Get("/foo") ~> route ~> check {
  responseAs[String] === "/foo"
}

Get("/foo/") ~> route ~> check {
  responseAs[String] === "/foo"
}

Get("/foo/bar") ~> route ~> check {
  responseAs[String] === "/foo/bar"
}
```

pathPrefix Matches and consumes a prefix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

```
def pathPrefix[L <: HList](pm: PathMatcher[L]): Directive[L]
```

Description This directive filters incoming requests based on the part of their URI that hasn’t been matched yet by other potentially existing `pathPrefix` or `rawPathPrefix` directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to its `rawPathPrefix` counterpart `pathPrefix` automatically adds a leading slash to its `PathMatcher` argument, you therefore don’t have to start your matching expression with an explicit slash.

Depending on the type of its `PathMatcher` argument the `pathPrefix` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val route =
  pathPrefix("ball") {
    pathEnd {
      complete("/ball")
    } ~
    path(IntNumber) { int =>
      complete(if (int % 2 == 0) "even ball" else "odd ball")
    }
  }

Get("/") ~> route ~> check {
  handled === false
}

Get("/ball") ~> route ~> check {
  responseAs[String] === "/ball"
}

Get("/ball/1337") ~> route ~> check {
  responseAs[String] === "odd ball"
}
```

pathPrefixTest Checks whether the unmatched path of the `RequestContext` has a prefix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

```
def pathPrefixTest[L <: HList](pm: PathMatcher[L]): Directive[L]
```

Description This directive is very similar to the `pathPrefix` directive with the one difference that the path prefix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

For more info on how to create a `PathMatcher` see *The PathMatcher DSL*.

As opposed to its `rawPathPrefixTest` counterpart `pathPrefixTest` automatically adds a leading slash to its `PathMatcher` argument, you therefore don't have to start your matching expression with an explicit slash.

Depending on the type of its `PathMatcher` argument the `pathPrefixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  unmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefixTest("foo" | "bar") {
```

```

    pathPrefix("foo") { completeWithUnmatchedPath } ~
    pathPrefix("bar") { completeWithUnmatchedPath }
  }

  Get("/foo/doo") ~> route ~> check {
    responseAs[String] === "/doo"
  }

  Get("/bar/yes") ~> route ~> check {
    responseAs[String] === "/yes"
  }
}

```

pathSingleSlash Only passes the request to its inner route if the unmatched path of the `RequestContext` contains exactly one single slash.

Signature

```
def pathSingleSlash: Directive0
```

Description This directive is a simple alias for `pathPrefix(PathEnd)` and is mostly used for matching requests to the root URI (`/`) on an inner-level to discriminate “all path segments matched” from other alternatives (see the example below).

Example

```

val route =
  pathSingleSlash {
    complete("root")
  } ~
  pathPrefix("ball") {
    pathSingleSlash {
      complete("/ball/")
    } ~
    path(IntNumber) { int =>
      complete(if (int % 2 == 0) "even ball" else "odd ball")
    }
  }

  Get("/") ~> route ~> check {
    responseAs[String] === "root"
  }

  Get("/ball") ~> route ~> check {
    handled === false
  }

  Get("/ball/") ~> route ~> check {
    responseAs[String] === "/ball/"
  }

  Get("/ball/1337") ~> route ~> check {
    responseAs[String] === "odd ball"
  }
}

```

pathSuffix Matches and consumes a suffix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

```
def pathSuffix[L <: HList] (pm: PathMatcher[L]): Directive[L]
```

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing path matching directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to *pathPrefix* this directive matches and consumes the unmatched path from the right, i.e. the end.

Caution: For efficiency reasons, the given `PathMatcher` must match the desired suffix in reversed-segment order, i.e. `pathSuffix("baz" / "bar")` would match `/foo/bar/baz!` The order within a segment match is not reversed.

Depending on the type of its `PathMatcher` argument the *pathPrefix* directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  unmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefix("start") {
    pathSuffix("end") {
      completeWithUnmatchedPath
    } ~
    pathSuffix("foo" / "bar" ~ "baz") {
      completeWithUnmatchedPath
    }
  }

Get("/start/middle/end") ~> route ~> check {
  responseAs[String] === "/middle/"
}

Get("/start/something/barbaz/foo") ~> route ~> check {
  responseAs[String] === "/something/"
}
```

pathSuffixTest Checks whether the unmatched path of the `RequestContext` has a suffix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

```
def pathSuffixTest[L <: HList] (pm: PathMatcher[L]): Directive[L]
```

Description This directive is very similar to the `pathSuffix` directive with the one difference that the path suffix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

As opposed to `pathPrefixTest` this directive matches and consumes the unmatched path from the right, i.e. the end.

Caution: For efficiency reasons, the given `PathMatcher` must match the desired suffix in reversed-segment order, i.e. `pathSuffixTest("baz" / "bar")` would match `/foo/bar/baz!` The order within a segment match is not reversed.

Depending on the type of its `PathMatcher` argument the `pathSuffixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  unmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathSuffixTest(Slash) {
    complete("slashed")
  } ~
  complete("unslashed")

Get("/foo/") ~> route ~> check {
  responseAs[String] === "slashed"
}
Get("/foo") ~> route ~> check {
  responseAs[String] === "unslashed"
}
```

rawPathPrefix Matches and consumes a prefix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

```
def rawPathPrefix[L <: HList](pm: PathMatcher[L]): Directive[L]
```

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing `rawPathPrefix` or `pathPrefix` directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to its `pathPrefix` counterpart `rawPathPrefix` does *not* automatically add a leading slash to its `PathMatcher` argument. Rather its `PathMatcher` argument is applied to the unmatched path as is.

Depending on the type of its `PathMatcher` argument the `rawPathPrefix` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  unmatchedPath { p =>
    complete(p.toString)
  }
```

```
val route =
  pathPrefix("foo") {
    rawPathPrefix("bar") { completeWithUnmatchedPath } ~
    rawPathPrefix("doo") { completeWithUnmatchedPath }
  }

Get("/foobar/baz") ~> route ~> check {
  responseAs[String] === "/baz"
}

Get("/foodoo/baz") ~> route ~> check {
  responseAs[String] === "/baz"
}
```

rawPathPrefixTest Checks whether the unmatched path of the `RequestContext` has a prefix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

```
def rawPathPrefixTest[L <: HList] (pm: PathMatcher[L]) : Directive[L]
```

Description This directive is very similar to the `pathPrefix` directive with the one difference that the path prefix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

For more info on how to create a `PathMatcher` see *The PathMatcher DSL*.

As opposed to its `pathPrefixTest` counterpart `rawPathPrefixTest` does *not* automatically add a leading slash to its `PathMatcher` argument. Rather its `PathMatcher` argument is applied to the unmatched path as is.

Depending on the type of its `PathMatcher` argument the `rawPathPrefixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  unmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefix("foo") {
    rawPathPrefixTest("bar") {
      completeWithUnmatchedPath
    }
  }

Get("/foobar") ~> route ~> check {
  responseAs[String] === "bar"
}

Get("/foobaz") ~> route ~> check {
  handled === false
}
```

The PathMatcher DSL For being able to work with the *PathDirectives* effectively you should have some understanding of the `PathMatcher` mini-DSL that *spray-routing* provides for elegantly defining URI matching behavior.

Overview When a request (or rather the respective `RequestContext` instance) enters the route structure it has an “unmatched path” that is identical to the `request.uri.path`. As it descends the routing tree and passes through one or more *pathPrefix/path* directives the “unmatched path” progressively gets “eaten into” from the left until, in most cases, it eventually has been consumed completely.

What exactly gets matched and consumed as well as extracted from the unmatched path in each directive is defined with the patch matching DSL, which is built around these types:

```
trait PathMatcher[L <: HList]
type PathMatcher0 = PathMatcher[HNil]
type PathMatcher1[T] = PathMatcher[T :: HNil]
```

The number and types of the values extracted by a `PathMatcher` instance is represented by the `L <: HList` type parameter. The convenience alias `PathMatcher0` can be used for all matchers which don't extract anything while `PathMatcher1[T]` defines a matcher which only extracts a single value of type `T`.

Here is an example of a more complex `PathMatcher` expression:

```
val matcher: PathMatcher1[Option[Int]] =
  "foo" / "bar" / "X" ~ IntNumber.? / ("edit" | "create")
```

This will match paths like `foo/bar/X42/edit` or `foo/bar/X/create`.

Note: The path matching DSL describes what paths to accept **after** URL decoding. This is why the path-separating slashes have special status and cannot simply be specified as part of a string! The string “foo/bar” would match the raw URI path “foo%2Fbar”, which is most likely not what you want!

Basic PathMatchers A complex `PathMatcher` can be constructed by combining or modifying more basic ones. Here are the basic matchers that *spray-routing* already provides for you:

String You can use a `String` instance as a `PathMatcher0`. Strings simply match themselves and extract no value. Note that strings are interpreted as the decoded representation of the path, so if they include a `'` character this character will match “%2F” in the encoded raw URI!

Regex You can use a `Regex` instance as a `PathMatcher1[String]`, which matches whatever the regex matches and extracts one `String` value. A `PathMatcher` created from a regular expression extracts either the complete match (if the regex doesn't contain a capture group) or the capture group (if the regex contains exactly one capture group). If the regex contains more than one capture group an `IllegalArgumentException` will be thrown.

Map[String, T] You can use a `Map[String, T]` instance as a `PathMatcher1[T]`, which matches any of the keys and extracts the respective map value for it.

Slash: `PathMatcher0` Matches exactly one path-separating slash (`/`) character and extracts nothing.

Segment: `PathMatcher1[String]` Matches if the unmatched path starts with a path segment (i.e. not a slash). If so the path segment is extracted as a `String` instance.

PathEnd: `PathMatcher0` Matches the very end of the path, similar to `$` in regular expressions and extracts nothing.

Rest: `PathMatcher1[String]` Matches and extracts the complete remaining unmatched part of the request's URI path as an (encoded!) `String`. If you need access to the remaining *decoded* elements of the path use `RestPath` instead.

RestPath: `PathMatcher1[Path]` Matches and extracts the complete remaining, unmatched part of the request's URI path.

IntNumber: `PathMatcher1[Int]` Efficiently matches a number of decimal digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

LongNumber: `PathMatcher1[Long]` Efficiently matches a number of decimal digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

HexIntNumber: `PathMatcher1[Int]` Efficiently matches a number of hex digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

HexLongNumber: `PathMatcher1[Long]` Efficiently matches a number of hex digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

DoubleNumber: `PathMatcher1[Double]` Matches and extracts a `Double` value. The matched string representation is the pure decimal, optionally signed form of a double value, i.e. without exponent.

JavaUUID: `PathMatcher1[UUID]` Matches and extracts a `java.util.UUID` instance.

Neutral: `PathMatcher0` A matcher that always matches, doesn't consume anything and extracts nothing. Serves mainly as a neutral element in `PathMatcher` composition.

Segments: `PathMatcher1[List[String]]` Matches all remaining segments as a list of strings. Note that this can also be "no segments" resulting in the empty list. If the path has a trailing slash this slash will *not* be matched, i.e. remain unmatched and to be consumed by potentially nested directives.

separateOnSlashes(string: String): PathMatcher0 Converts a path string containing slashes into a `PathMatcher0` that interprets slashes as path segment separators. This means that a matcher matching "%2F" cannot be constructed with this helper.

provide[L <: HList](extractions: L): PathMatcher[L] Always matches, consumes nothing and extracts the given `HList` of values.

PathMatcher[L <: HList](prefix: Path, extractions: L): PathMatcher[L] Matches and consumes the given path prefix and extracts the given list of extractions. If the given prefix is empty the returned matcher matches always and consumes nothing.

Combinators Path matchers can be combined with these combinators to form higher-level constructs:

Tilde Operator (~) The tilde is the most basic combinator. It simply concatenates two matchers into one, i.e. if the first one matched (and consumed) the second one is tried. The extractions of both matchers are combined type-safely. For example: `"foo" ~ "bar"` yields a matcher that is identical to `"foobar"`.

Slash Operator (/) This operator concatenates two matchers and inserts a `Slash` matcher in between them. For example: `"foo" / "bar"` is identical to `"foo" ~ Slash ~ "bar"`.

Pipe Operator (|) This operator combines two matcher alternatives in that the second one is only tried if the first one did *not* match. The two sub-matchers must have compatible types. For example: `"foo" | "bar"` will match either "foo" or "bar".

Modifiers Path matcher instances can be transformed with these modifier methods:

/ The slash operator cannot only be used as combinator for combining two matcher instances, it can also be used as a postfix call. `matcher /` is identical to `matcher ~ Slash` but shorter and easier to read.

? By postfixing a matcher with ? you can turn any `PathMatcher` into one that always matches, optionally consumes and potentially extracts an `Option` of the underlying matchers extraction. The result type depends on the type of the underlying matcher:

If a matcher is of type	then <code>matcher.?</code> is of type
<code>PathMatcher0</code>	<code>PathMatcher0</code>
<code>PathMatcher1[T]</code>	<code>PathMatcher1[Option[T]]</code>
<code>PathMatcher[L <: HList]</code>	<code>PathMatcher[Option[L]]</code>

repeat(separator: PathMatcher0 = PathMatchers.Neutral) By postfixing a matcher with `repeat(separator)` you can turn any `PathMatcher` into one that always matches, consumes zero or more times (with the given separator) and potentially extracts a `List` of the underlying matchers' extractions. The result type depends on the type of the underlying matcher:

If a matcher is of type	then <code>matcher.repeat(...)</code> is of type
<code>PathMatcher0</code>	<code>PathMatcher0</code>
<code>PathMatcher1[T]</code>	<code>PathMatcher1[List[T]]</code>
<code>PathMatcher[L <: HList]</code>	<code>PathMatcher[List[L]]</code>

unary_! By prefixing a matcher with ! it can be turned into a `PathMatcher0` that only matches if the underlying matcher does *not* match and vice versa.

transform / (h) flatMap / (h) map These modifiers allow you to append your own "post-application" logic to another matcher in order to form a custom one. You can map over the extraction(s), turn mismatches into matches or vice-versa or do anything else with the results of the underlying matcher. Take a look at the method signatures and implementations for more guidance as to how to use them.

Examples

```
// matches /foo/
path("foo" /)

// matches e.g. /foo/123 and extracts "123" as a String
path("foo" / ""\d+"" .r)

// matches e.g. /foo/bar123 and extracts "123" as a String
path("foo" / ""bar(\d+)"" .r)

// identical to `path(Segments)`
path(Segment.repeat(separator = Slash))

// matches e.g. /i42 or /hCAFE and extracts an Int
path("i" ~ IntNumber | "h" ~ HexIntNumber)

// identical to path("foo" ~ (PathEnd | Slash))
path("foo" ~ Slash.?)

// matches /red or /green or /blue and extracts 1, 2 or 3 respectively
path(Map("red" -> 1, "green" -> 2, "blue" -> 3))

// matches anything starting with "/foo" except for /foobar
pathPrefix("foo" ~ !"bar")
```

RangeDirectives

withRangeSupport Transforms the response from its inner route into a 206 Partial Content response if the client requested only part of the resource with a Range header.

Signature

```
def withRangeSupport(): Directive0
def withRangeSupport(rangeCountLimit: Int, rangeCoalescingThreshold: Long): Directive0
```

The signature shown is simplified, the real signature uses magnets.¹

Description Augments responses to GET requests with an `Accept-Range: bytes` header and converts them into partial responses if the request contains a valid `Range` request header. The requested byte-ranges are coalesced (merged) if they lie closer together than the specified `rangeCoalescingThreshold` argument.

In order to prevent the server from becoming overloaded with trying to prepare `multipart/byteranges` responses for high numbers of potentially very small ranges the directive rejects requests requesting more than `rangeCountLimit` ranges with a `TooManyRangesRejection`. Requests with unsatisfiable ranges are rejected with an `UnsatisfiableRangeRejection`.

The `withRangeSupport()` form (without parameters) uses the `range-coalescing-threshold` and `range-count-limit` settings from the `spray.routing` configuration.

This directive is transparent to non-GET requests.

See also: <https://tools.ietf.org/html/draft-ietf-httpbis-p5-range/>

Example

```
val route =
  withRangeSupport(4, 2L) {
    complete("ABCDEFGH")
  }

Get() ~> addHeader(Range(ByteRange(3, 4))) ~> route ~> check {
  headers must contain(`Content-Range` (ContentRange(3, 4, 8)))
  status === StatusCodes.PartialContent
  responseAs[String] === "DE"
}

Get() ~> addHeader(Range(ByteRange(0, 1), ByteRange(1, 2), ByteRange(6, 7))) ~> route ~> check {
  headers must not(contain(like[HttpHeader] { case `Content-Range`(_, _) ok }))
  responseAs[MultipartByteRanges] must beLike {
    case MultipartByteRanges (
      BodyPart(entity1, `Content-Range` (RangeUnit.Bytes, range1) +: _) +:
      BodyPart(entity2, `Content-Range` (RangeUnit.Bytes, range2) +: _) +: Seq()
    ) entity1.asString === "ABC" and range1 === ContentRange(0, 2, 8) and
      entity2.asString === "GH" and range2 === ContentRange(6, 7, 8)
  }
}
```

RespondWithDirectives

respondWithHeader Adds a given HTTP header to all responses coming back from its inner route.

Signature

```
def respondWithHeader(responseHeader: HttpHeader): Directive0
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by adding the given `HttpHeader` instance to the headers list. If you'd like to add more than one header you can use the *respondWithHeaders* directive instead.

Example

```
val route =
  path("foo") {
    respondWithHeader(RawHeader("Funky-Muppet", "gonzo")) {
      complete("beep")
    }
  }

Get("/foo") ~> route ~> check {
  header("Funky-Muppet") === Some(RawHeader("Funky-Muppet", "gonzo"))
  responseAs[String] === "beep"
}
```

respondWithHeaders Adds the given HTTP headers to all responses coming back from its inner route.

Signature

```
def respondWithHeaders(responseHeaders: HttpHeader*): Directive0
def respondWithHeaders(responseHeaders: List[HttpHeader]): Directive0
```

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by adding the given `HttpHeader` instances to the headers list. If you'd like to add just a single header you can use the *respondWithHeader* directive instead.

Example

```
val route =
  path("foo") {
    respondWithHeaders(RawHeader("Funky-Muppet", "gonzo"), Origin(Seq(HttpOrigin("http://spray.io"))))
    complete("beep")
  }
}

Get("/foo") ~> route ~> check {
  header("Funky-Muppet") === Some(RawHeader("Funky-Muppet", "gonzo"))
  header[Origin] === Some(Origin(Seq(HttpOrigin("http://spray.io"))))
  responseAs[String] === "beep"
}
```

respondWithMediaType Overrides the media-type of the response returned by its inner route with the given one.

Signature

```
def respondWithMediaType(mediaType: MediaType): Directive0
```

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by overriding the `entity.contentType.mediaType` with the given one if the entity is non-empty. Empty response entities are left unchanged.

If the given media-type is not accepted by the client the request is rejected with an `UnacceptedResponseContentTypeRejection`.

Note: This directive removes a potentially existing `Accept` header from the request, in order to “disable” content negotiation in a potentially running `Marshaller` in its inner route. Also note that this directive does *not* change the response entity buffer content in any way, it merely overrides the media-type component of the entities `Content-Type`.

Example

```
import MediaTypes._

val route =
  path("foo") {
    respondWithMediaType(`application/json`) {
      complete("[]") // marshalled to `text/plain` here
    }
  }

Get("/foo") ~> route ~> check {
  mediaType === `application/json`
  responseAs[String] === "[]"
}

Get("/foo") ~> Accept(MediaRanges.`text/*`) ~> route ~> check {
  rejection === UnacceptedResponseContentTypeRejection(ContentType(`application/json`)) : Nil
}
```

respondWithSingletonHeader Adds a given HTTP header to all responses coming back from its inner route only if a header with the same name doesn't exist yet in the response.

Signature

```
def respondWithSingletonHeader(responseHeader: HttpHeader): Directive0
```

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by potentially adding the given `HttpHeader` instance to the headers list. The header is only added if there is no header instance with the same name (case insensitively) already present in the response. If you'd like to add more than one header you can use the `respondWithSingletonHeaders` directive instead.

Example

```
val respondWithMuppetHeader =
  respondWithSingletonHeader(RawHeader("Funky-Muppet", "gonzo"))

val route =
  path("foo") {
    respondWithMuppetHeader {
      complete("beep")
    }
  } ~
  path("bar") {
    respondWithMuppetHeader {
      respondWithHeader(RawHeader("Funky-Muppet", "kermit")) {

```

```

        complete("beep")
      }
    }
  }

  Get("/foo") ~> route ~> check {
    headers.filter(_.is("funky-muppet")) === List(RawHeader("Funky-Muppet", "gonzo"))
    responseAs[String] === "beep"
  }

  Get("/bar") ~> route ~> check {
    headers.filter(_.is("funky-muppet")) === List(RawHeader("Funky-Muppet", "kermit"))
    responseAs[String] === "beep"
  }

```

respondWithSingletonHeaders Adds the given HTTP headers to all responses coming back from its inner route only if a respective header with the same name doesn't exist yet in the response.

Signature

```

def respondWithSingletonHeaders(responseHeaders: HttpHeader*): Directive0
def respondWithSingletonHeaders(responseHeaders: List[HttpHeader]): Directive0

```

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by potentially adding the given `HttpHeader` instances to the headers list. A header is only added if there is no header instance with the same name (case insensitively) already present in the response. If you'd like to add only a single header you can use the `respondWithSingletonHeader` directive instead.

Example See the `respondWithSingletonHeader` directive for an example with only one header.

respondWithStatus Overrides the status code of all responses coming back from its inner route with the given one.

Signature

```

def respondWithStatus(responseStatus: StatusCode): Directive0

```

Description This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by unconditionally overriding the status code with the given one.

Example

```

val route =
  path("foo") {
    respondWithStatus(201) {
      complete("beep")
    }
  }

  Get("/foo") ~> route ~> check {
    status === StatusCodes.Created
    responseAs[String] === "beep"
  }

```

RouteDirectives

The `RouteDirectives` have a special role in spray's routing DSL. Contrary to all other directives (except most *FileAndResourceDirectives*) they do not produce instances of type `Directive[L <: HList]` but rather "plain" routes of type `Route`. The reason is that the `RouteDirectives` are not meant for wrapping an inner route (like most other directives, as intermediate-level elements of a route structure, do) but rather form the actual route structure leaves.

So in most cases the inner-most element of a route structure branch is one of the `RouteDirectives` (or *FileAndResourceDirectives*):

complete Completes the request using the given argument(s).

Signature

```
def complete[T :ToResponseMarshaller] (value: T): StandardRoute
def complete(response: HttpResponse): StandardRoute
def complete(status: StatusCode): StandardRoute
def complete[T :Marshaller] (status: StatusCode, value: T): StandardRoute
def complete[T :Marshaller] (status: Int, value: T): StandardRoute
def complete[T :Marshaller] (status: StatusCode, headers: Seq[HttpHeader], value: T): StandardRoute
def complete[T :Marshaller] (status: Int, headers: Seq[HttpHeader], value: T): StandardRoute
```

The signature shown is simplified, the real signature uses magnets. ¹

Description `complete` uses the given arguments to construct a `Route` which simply calls `requestContext.complete` with the respective `HttpResponse` instance. Completing the request will send the response "back up" the route structure where all logic that wrapping directives have potentially chained into the responder chain is run (see also *The Responder Chain*). Once the response hits the top-level `runRoute` logic it is sent back to the underlying *spray-can* or *spray-servlet* layer which will trigger the sending of the actual HTTP response message back to the client.

Example

```
val route =
  path("a") {
    complete(HttpResponse(entity = "foo"))
  } ~
  path("b") {
    complete(StatusCodes.Created, "bar")
  } ~
  (path("c") & complete("baz")) // `&` also works with `complete` as the 2nd argument

Get("/a") ~> route ~> check {
  status === StatusCodes.OK
  responseAs[String] === "foo"
}

Get("/b") ~> route ~> check {
  status === StatusCodes.Created
  responseAs[String] === "bar"
}

Get("/c") ~> route ~> check {
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

```

status === StatusCodes.OK
responseAs[String] === "baz"
}

```

failWith Bubbles up the given error through the route structure where it is dealt with by the closest `handleExceptions` directive and its `ExceptionHandler`.

Signature

```
def failWith(error: Throwable): StandardRoute
```

Description `failWith` explicitly raises an exception that gets bubbled up through the route structure to be picked up by the nearest `handleExceptions` directive. If no `handleExceptions` is present above the respective location in the route structure *The runRoute Wrapper* will handle the exception and translate it into a corresponding `HttpResponse` using the in-scope `ExceptionHandler` (see also the *Exception Handling* chapter).

There is one notable special case: If the given exception is a `RejectionError` exception it is *not* bubbled up, but rather the wrapped exception is unpacked and “executed”. This allows the “tunneling” of a rejection via an exception.

Example

```

val route =
  path("foo") {
    failWith(new RequestProcessingException(StatusCodes.BandwidthLimitExceeded))
  }

Get("/foo") ~> sealRoute(route) ~> check {
  status === StatusCodes.BandwidthLimitExceeded
  responseAs[String] === "Bandwidth limit has been exceeded."
}

```

redirect Completes the request with a redirection response to a given target URI and of a given redirection type (status code).

Signature

```
def redirect(uri: Uri, redirectionType: Redirection): StandardRoute
```

Description `redirect` is a convenience helper for completing the request with a redirection response. It is equivalent to this snippet relying on the `complete` directive:

```

complete {
  HttpResponse(
    status = redirectionType,
    headers = Location(uri) :: Nil,
    entity = redirectionType.htmlTemplate match {
      case ""      => HttpEntity.Empty
      case template => HttpEntity(`text/html`, template format uri)
    })
}

```

Example

```

val route =
  pathPrefix("foo") {
    pathSingleSlash {
      complete("yes")
    } ~
    pathEnd {
      redirect("/foo/", StatusCodes.PermanentRedirect)
    }
  }

Get("/foo/") ~> route ~> check {
  responseAs[String] === "yes"
}

Get("/foo/") ~> route ~> check {
  status === StatusCodes.PermanentRedirect
  responseAs[String] === ""The request, and all future requests should be repeated using <a href="/

```

reject Explicitly rejects the request optionally using the given rejection(s).

Signature

```

def reject: StandardRoute
def reject(rejections: Rejection*): StandardRoute

```

Description `reject` uses the given rejection instances (which might be the empty `Seq`) to construct a `Route` which simply calls `requestContext.reject`. See the chapter on *Rejections* for more information on what this means.

After the request has been rejected at the respective point it will continue to flow through the routing structure in the search for a route that is able to complete it.

The explicit `reject` directive is used mostly when building *Custom Directives*, e.g. inside of a `flatMap` modifier for “filtering out” certain cases.

Example

```

val route =
  path("a") {
    reject // don't handle here, continue on
  } ~
  path("a") {
    complete("foo")
  } ~
  path("b") {
    // trigger a ValidationRejection explicitly
    // rather than through the `validate` directive
    reject(ValidationRejection("Restricted!"))
  }

Get("/a") ~> route ~> check {
  responseAs[String] === "foo"
}

```



```
Get("/b") ~> route ~> check {
  rejection === ValidationRejection("Restricted!")
}
```

SchemeDirectives

Scheme directives can be used to extract the Uri scheme (i.e. “http”, “https”, etc.) from requests or to reject any request that does not match a specified scheme name.

scheme Rejects a request if its Uri scheme does not match a given one.

Signature

```
def scheme(schm: String): Directive0
```

Description The `scheme` directive can be used to match requests by their Uri scheme, only passing through requests that match the specified scheme and rejecting all others.

A typical use case for the `scheme` directive would be to reject requests coming in over http instead of https, or to redirect such requests to the matching https URI with a `MovedPermanently`.

For simply extracting the scheme name, see the `schemeName` directive.

Example

```
val route =
  scheme("http") {
    extract(_.request.uri) { uri
      redirect(uri.copy(scheme = "https"), MovedPermanently)
    }
  } ~
  scheme("https") {
    complete(s"Safe and secure!")
  }

Get("http://www.example.com/hello") ~> route ~> check {
  status === MovedPermanently
  header[Location] === Some(Location(Uri("https://www.example.com/hello")))
}

Get("https://www.example.com/hello") ~> route ~> check {
  responseAs[String] === "Safe and secure!"
}
```

schemeName Extracts the value of the request Uri scheme.

Signature

```
def schemeName: Directive1[String]
```

Description The `schemeName` directive can be used to determine the Uri scheme (i.e. “http”, “https”, etc.) for an incoming request.

For rejecting a request if it doesn’t match a specified scheme name, see the `scheme` directive.

Example

```
val route =
  schemeName { scheme =>
    complete(s"The scheme is '${scheme}')}
}

Get("https://www.example.com/") ~> route ~> check {
  responseAs[String] === "The scheme is 'https'"
}
```

SecurityDirectives

authenticate Authenticates a request by checking credentials supplied in the request and extracts a value representing the authenticated principal.

Signature

```
def authenticate[T](auth: Future[Authentication[T]])(implicit executor: ExecutionContext): Directive[?]
def authenticate[T](auth: ContextAuthenticator[T])(implicit executor: ExecutionContext): Directive[?]
```

The signature shown is simplified, the real signature uses magnets. ¹

Description On the lowest level, `authenticate`, takes either a `Future[Authentication[T]]` which authenticates based on values from the lexical scope or a value of type `ContextAuthenticator[T]` = `RequestContext Future[Authentication[T]]` which extracts authentication data from the `RequestContext`. The returned value of type `Authentication[T]` must either be the authenticated principal which will be supplied to the inner route or a rejection to reject the request with if authentication failed .

Both variants return futures so that the actual authentication procedure runs detached from route processing and processing of the inner route will be continued once the authentication finished. This allows longer-running authentication tasks (like looking up credentials in a database) to run without blocking the `HttpService` actor, freeing it for other requests. The `authenticate` directive itself isn’t tied to any HTTP-specific details so that various authentication schemes can be implemented on top of `authenticate`.

Standard HTTP-based authentication which uses the `WWW-Authenticate` header containing challenge data and `Authorization` header for receiving credentials is implemented in subclasses of `HttpAuthenticator`.

HTTP Basic Authentication `spray` supports `HTTP basic authentication` through the `BasicHttpAuthenticator` and provides a series of convenience constructors for different scenarios with `BasicAuth()`. Make sure to use basic authentication only over SSL because credentials are transferred in plaintext.

Implementing a `UserPassAuthenticator` The most generic way of deploying HTTP basic authentication uses a `UserPassAuthenticator` to validate a user/password combination. It is defined like this:

```
type UserPassAuthenticator[T] = Option[UserPass] Future[Option[T]]
```

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

Its job is to map a user/password combination (if existent in the request) to an authenticated custom principal of type `T` (if authenticated).

```
def myUserPassAuthenticator(userPass: Option[UserPass]): Future[Option[String]] =
  Future {
    if (userPass.exists(up => up.user == "John" && up.pass == "p4ssw0rd")) Some("John")
    else None
  }

val route =
  sealRoute {
    path("secured") {
      authenticate(BasicAuth(myUserPassAuthenticator _, realm = "secure site")) { userName =>
        complete(s"The user is '$userName'")
      }
    }
  }

Get("/secured") ~> route ~> check {
  status === StatusCodes.Unauthorized
  responseAs[String] === "The resource requires authentication, which was not supplied with the requ
  header[HttpHeaders.`WWW-Authenticate`].get.challenges.head === HttpChallenge("Basic", "secure site
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~>
  addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] === "The user is 'John'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status === StatusCodes.Unauthorized
    responseAs[String] === "The supplied authentication is invalid"
    header[HttpHeaders.`WWW-Authenticate`].get.challenges.head === HttpChallenge("Basic", "secure sit
  }
}
```

From configuration There are several overloads to configure users from the configuration file. Obviously, this is neither a secure (plaintext passwords) nor a scalable approach. If you don't pass in a custom config users are configured from the `Configuration` path `spray.routing.users`.

```
def extractUser(userPass: UserPass): String = userPass.user
val config = ConfigFactory.parseString("John = p4ssw0rd")

val route =
  sealRoute {
    path("secured") {
      authenticate(BasicAuth(realm = "secure site", config = config, createUser = extractUser _)) { u
        complete(s"The user is '$userName'")
      }
    }
  }

Get("/secured") ~> route ~> check {
  status === StatusCodes.Unauthorized
}
```

```

responseAs[String] === "The resource requires authentication, which was not supplied with the requ
header[HttpHeaders.`WWW-Authenticate`].get.challenges.head === HttpChallenge("Basic", "secure site
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~>
  addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] === "The user is 'John'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status === StatusCodes.Unauthorized
    responseAs[String] === "The supplied authentication is invalid"
    header[HttpHeaders.`WWW-Authenticate`].get.challenges.head === HttpChallenge("Basic", "secure site
  }

```

From LDAP (todo)

authorize Guards access to the inner route with a user-defined check.

Signature

```

def authorize(check: Boolean): Directive0
def authorize(check: RequestContext Boolean): Directive0

```

Description The user-defined authorization check can either be supplied as a `Boolean` value which is calculated just from information out of the lexical scope, or as a function `RequestContext Boolean` which can also take information from the request itself into account. If the check returns `true` the request is passed on to the inner route unchanged, otherwise an `AuthorizationFailedRejection` is created, triggering a 403 `Forbidden` response by default (the same as in the case of an `AuthenticationFailedRejection`).

In a common use-case you would check if a user (e.g. supplied by the `authenticate` directive) is allowed to access the inner routes, e.g. by checking if the user has the needed permissions.

Example

```

def extractUser(userPass: UserPass): String = userPass.user
val config = ConfigFactory.parseString("John = p4ssw0rd\nPeter = pan")
def hasPermissionToPetersLair(userName: String) = userName == "Peter"

val route =
  sealRoute {
    authenticate(BasicAuth(realm = "secure site", config = config, createUser = extractUser _)) { use
      path("peters-lair") {
        authorize(hasPermissionToPetersLair(userName)) {
          complete(s"$userName' visited Peter's lair")
        }
      }
    }
  }

```

```

val johnsCred = BasicHttpCredentials("John", "p4ssw0rd")
Get("/peters-lair") ~>
  addCredentials(johnsCred) ~> // adds Authorization header
  route ~> check {
    status === StatusCodes.Forbidden
    responseAs[String] === "The supplied authentication is not authorized to access this resource"
  }

val petersCred = BasicHttpCredentials("Peter", "pan")
Get("/peters-lair") ~>
  addCredentials(petersCred) ~> // adds Authorization header
  route ~> check {
    responseAs[String] === "'Peter' visited Peter's lair"
  }

```

optionalAuthenticate Authenticates a request by checking credentials supplied in the request and extracts a value representing the authenticated principal, or *None* if no credentials were supplied.

Signature

```

def optionalAuthenticate[T](auth: Future[Authentication[T]]) (implicit executor: ExecutionContext): Directive[T]
def optionalAuthenticate[T](auth: ContextAuthenticator[T]) (implicit executor: ExecutionContext): Directive[T]

```

The signature shown is simplified, the real signature uses magnets. ¹

Description The `optionalAuthenticate` directive is similar to the `authenticate` directive but always extracts an `Option` value instead of rejecting the request if no credentials could be found.

Authentication vs. Authorization *Authentication* is the process of establishing a known identity for the user, whereby ‘identity’ is defined in the context of the application. This may be done with a username/password combination, a cookie, a pre-defined IP or some other mechanism. After authentication the system believes that it knows who the user is.

Authorization is the process of determining, whether a given user is allowed access to a given resource or not. In most cases, in order to be able to authorize a user (i.e. allow access to some part of the system) the users identity must already have been established, i.e. he/she must have been authenticated. Without prior authentication the authorization would have to be very crude, e.g. “allow access for *all* users” or “allow access for *noone*”. Only after authentication will it be possible to, e.g., “allow access to the statistics resource for `_admins_`, but not for regular *members*”.

Authentication and authorization may happen at the same time, e.g. when everyone who can properly be authenticated is also allowed access (which is often a very simple and somewhat implicit authorization logic). In other cases the system might have one mechanism for authentication (e.g. establishing user identity via an LDAP lookup) and another one for authorization (e.g. a database lookup for retrieving user access rights).

5.7.9 Complete Examples

The `/examples/spray-routing/` directory of the *spray* repository contains a number of example projects for *spray-routing*, which are described here.

¹ See The Magnet Pattern for an explanation of magnet-based overloading.

on-spray-can

This examples demonstrates how to run *spray-routing* on top of the *spray-can HTTP Server*. It implements a very simple web-site and shows off various features like streaming, stats support and timeout handling.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project on-spray-can" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project on-spray-can` and `run` sequentially "inside" of SBT.)

4. Browse to <http://127.0.0.1:8080/>
5. Alternatively you can access the service with `curl`:

```
curl -v 127.0.0.1:8080/ping
```

6. Stop the service with:

```
curl -v 127.0.0.1:8080/stop
```

on-jetty

This examples demonstrates how to run *spray-routing* on top of *spray-servlet*. It implements a very simple web-site and shows off various features like streaming, stats support and timeout handling.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project on-jetty" container:start shell
```

4. Browse to <http://127.0.0.1:8080/>
5. Alternatively you can access the service with `curl`:

```
curl -v 127.0.0.1:8080/ping
```

6. Stop the service with:

```
container:stop
```

simple-routing-app

This examples demonstrates how to use the *SimpleRoutingApp* trait.

Follow these steps to run it on your machine:

1. Clone the *spray* repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project simple-routing-app" run
```

(If this doesn't work for you your SBT runner cannot deal with grouped arguments. In this case you'll have to run the commands `project simple-routing-app` and run sequentially "inside" of SBT.)

4. Browse to <http://127.0.0.1:8080/>

5. Alternatively you can access the service with `curl`:

```
curl -v 127.0.0.1:8080/ping
```

6. Stop the service with:

```
curl -v 127.0.0.1:8080/stop
```

5.7.10 Minimal Example

This is a complete, very basic *spray-routing* application:

```
import spray.routing.SimpleRoutingApp

object Main extends App with SimpleRoutingApp {
  implicit val system = ActorSystem("my-system")

  startServer(interface = "localhost", port = 8080) {
    path("hello") {
      get {
        complete {
          <h1>Say hello to spray</h1>
        }
      }
    }
  }
}
```

It starts a *spray-can HTTP Server* on localhost and replies to GET requests to `/hello` with a simple response.

5.7.11 Longer Example

The following is a *spray-routing* route definition that tries to show off a few features. The resulting service does not really do anything useful but its definition should give you a feel for what an actual API definition with *spray-routing* will look like:

```

import scala.concurrent.duration.Duration
import spray.routing.HttpService
import spray.routing.authentication.BasicAuth
import spray.routing.directives.CachingDirectives._
import spray.httpx.encoding._

trait LongerService extends HttpService with MyApp {

  val simpleCache = routeCache(maxCapacity = 1000, timeToIdle = Duration("30 min"))

  val route = {
    path("orders") {
      authenticate(BasicAuth(realm = "admin area")) { user =>
        get {
          cache(simpleCache) {
            encodeResponse(Deflate) {
              complete {
                // marshal custom object with in-scope marshaller
                getOrdersFromDB
              }
            }
          }
        }
      } ~
      post {
        // decompresses the request with Gzip or Deflate when required
        decompressRequest() {
          // unmarshal with in-scope unmarshaller
          entity(as[Order]) { order =>
            // transfer to newly spawned actor
            detach() {
              complete {
                // ... write order to DB
                "Order received"
              }
            }
          }
        }
      }
    }
  } ~
  // extract URI path element as Int
  pathPrefix("order" / IntNumber) { orderId =>
    pathEnd {
      // method tunneling via query param
      (put | parameter('method ! "put")) {
        // form extraction from multipart or www-url-encoded forms
        formFields('email, 'total.as[Money]).as[Order] { order =>
          complete {
            // complete with serialized Future result
            (myDbActor ? Update(order)).mapTo[TransactionResult]
          }
        }
      } ~
      get {
        // JSONP support
        jsonpWithParameter("callback") {
          // use in-scope marshaller to create completer function
          produce(instanceOf[Order]) { completer => ctx =>

```



```

        processOrderRequest(orderId, completer)
      }
    }
  } ~
  path("items") {
    get {
      // parameters to case class extraction
      parameters('size.as[Int], 'color ?, 'dangerous ? "no")
        .as(OrderItem) { orderItem =>
          // ... route using case class instance created from
          // required and optional query parameters
        }
    }
  }
} ~
pathPrefix("documentation") {
  // cache responses to GET requests
  cache(simpleCache) {
    // optionally compresses the response with Gzip or Deflate
    // if the client accepts compressed responses
    compressResponse() {
      // serve up static content from a JAR resource
      getFromResourceDirectory("docs")
    }
  }
} ~
path("oldApi" / Rest) { pathRest =>
  redirect("http://oldapi.example.com/" + pathRest, StatusCodes.MovedPermanently)
}
}
}

```

5.8 spray-servlet

spray-servlet is an adapter layer providing (a subset of) the *spray-can HTTP Server* interface on top of the Servlet API. As one main application it enables the use of *spray-routing* in a servlet container.

5.8.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-servlet* depends on

- *spray-http*
- *spray-util*
- *spray-io* (only required until the upgrade to Akka 2.2, will go away afterwards)
- akka-actor 2.1.x (with ‘provided’ scope, i.e. you need to pull it in yourself)
- the Servlet-3.0 API (with ‘provided’ scope, usually automatically available from your servlet container)

5.8.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-servlet* into your classpath. You might also want to check out:

- The *xsbt-web-plugin* for simplifying the development process
- The *Getting Started* chapter for info on the *spray* project template for *spray-servlet*

5.8.3 Configuration

Just like Akka *spray-servlet* relies on the *typesafe config* library for configuration. As such its JAR contains a `reference.conf` file holding the default values of all configuration settings. In your application you typically provide an `application.conf`, in which you override Akka and/or *spray* settings according to your needs.

Note: Since *spray* uses the same configuration technique as Akka you might want to check out the [Akka Documentation on Configuration](#).

This is the `reference.conf` of the *spray-servlet* module:

```
#####
# spray-servlet Reference Config File #
#####

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

spray.servlet {

  # The FQN (Fully Qualified Name) of the class to load when the
  # servlet context is initialized (e.g. "com.example.ApiBoot").
  # The class must have a constructor with a single
  # `javax.servlet.ServletContext` parameter and implement
  # the `spray.servlet.WebBoot` trait.
  boot-class = ""

  # If a request hasn't been responded to after the time period set here
  # a `spray.http.Timedout` message will be sent to the timeout handler.
  # Set to `infinite` to completely disable request timeouts.
  request-timeout = 30 s

  # After a `Timedout` message has been sent to the timeout handler and the
  # request still hasn't been completed after the time period set here
  # the server will complete the request itself with an error response.
  # Set to `infinite` to disable timeout timeouts.
  timeout-timeout = 500 ms

  # The path of the actor to send `spray.http.Timedout` messages to.
  # If empty all `Timedout` messages will go to the "regular" request handling actor.
  timeout-handler = ""

  # A path prefix that is automatically "consumed" before the request is
  # being dispatched to the HTTP service route.
  # Can be used to match servlet context paths configured for the application.
  # Make sure to include a leading slash with your prefix, e.g. "/foobar".
  # Set to `AUTO` to make spray-servlet pick up the ServletContext::getContextPath.
```

```

root-path = AUTO

# Enables/disables the addition of a `Remote-Address` header
# holding the clients (remote) IP address.
remote-address-header = off

# Enables/disables the returning of more detailed error messages to
# the client in the error response.
# Should be disabled for browser-facing APIs due to the risk of XSS attacks
# and (probably) enabled for internal or non-browser APIs.
# Note that spray will always produce log messages containing the full error details.
verbose-error-messages = off

# The maximum size of the request entity that is still accepted by the server.
# Requests with a greater entity length are rejected with an error response.
# Must be greater than zero.
max-content-length = 5 m

# Enables/disables the inclusion of `spray.servlet.ServletRequestInfoHeader` in the
# headers of the HTTP request sent to the service actor.
servlet-request-access = off

# Enables/disables the logging of warning messages in case an incoming
# message (request or response) contains an HTTP header which cannot be
# parsed into its high-level model class due to incompatible syntax.
# Note that, independently of this settings, spray will accept messages
# with such headers as long as the message as a whole would still be legal
# under the HTTP specification even without this header.
# If a header cannot be parsed into a high-level model instance it will be
# provided as a `RawHeader`.
illegal-header-warnings = on

# Sets the strictness mode for parsing request target URIs.
# The following values are defined:
#
# `strict`: RFC3986-compliant URIs are required,
#           a 400 response is triggered on violations
#
# `relaxed`: all visible 7-Bit ASCII chars are allowed
#
# `relaxed-with-raw-query`: like `relaxed` but additionally
#                           the URI query is not parsed, but delivered as one raw string
#                           as the `key` value of a single Query structure element.
#
uri-parsing-mode = relaxed
}

```

5.8.4 Basic Architecture

The central element of *spray-servlet* is the `Servlet30ConnectorServlet`. Its job is to accept incoming HTTP requests, suspend them (using Servlet 3.0 `startAsync`), create immutable *spray-http* `HttpRequest` instances for them and dispatch these to a service actor provided by the application.

The messaging API as seen from the application is modeled as closely as possible like its counterpart, the *spray-can HTTP Server*.

In the most basic case, the service actor completes a request by simply replying with an `HttpResponse` instance to

the request sender:

```
def receive = {
  case HttpRequest(...) => sender ! HttpResponse(...)
}
```

5.8.5 Starting and Stopping

A *spray-servlet* application is started by the servlet container. The application JAR should contain a `web.xml` similar to this one from the *simple-spray-servlet-server* example.

The `web.xml` registers a `ServletContextListener` (`spray.servlet.Initializer`), which initializes the application when the servlet is started. The `Initializer` loads the configured `boot-class` and instantiates it using the default constructor, which must be available. The boot class must implement the `WebBoot` trait, which is defined like this:

```
/**
 * Trait that must be implemented by the Boot class.
 */
trait WebBoot {

  /**
   * The ActorSystem the application would like to use.
   */
  def system: ActorSystem

  /**
   * The service actor to dispatch incoming HttpRequests to.
   */
  def serviceActor: ActorRef
}
```

A very basic boot class implementation is [this one](#) from the *simple-spray-servlet-server* example.

The boot class is responsible for creating the Akka `ActorSystem` for the application as well as the service actor. When the application is shut down by the servlet container the `Initializer` shuts down the `ActorSystem`, which cleanly terminates all application actors including the service actor.

5.8.6 Message Protocol

Just like in its counterpart, the *spray-can HTTP Server*, all communication between the connector servlet and the application happens through actor messages.

Request-Response Cycle

As soon as a new request has been successfully read from the servlet API it is dispatched to the service actor created by the boot class. The service actor processes the request according to the application logic and responds by sending an `HttpResponse` instance to the sender of the request.

The `ActorRef` used as the sender of an `HttpRequest` received by the service actor is unique to the request, i.e. each request will appear to be sent from different senders. *spray-servlet* uses these sender `ActorRefs` to coalesce the response with the request, so you cannot send several responses to the same sender. However, the different response parts of a chunked response need to be sent to the same sender.

Caution: Since the `ActorRef` used as the sender of a request is an `UnregisteredActorRef` it is not reachable remotely. This means that the service actor needs to live in the same JVM as the connector servlet.

Chunked Responses

Alternatively to a single `HttpResponse` instance the handler can choose to respond to the request sender with the following sequence of individual messages:

- One `ChunkedResponseStart`
- Zero or more `MessageChunks`
- One `ChunkedMessageEnd`

The connector servlet writes the individual response parts into the servlet response `OutputStream` and flushes it. Whether these parts are really rendered “to the wire” as chunked message parts depends on the servlet container implementation. The Servlet API has not dedicated support for chunked responses.

Request Timeouts

If the service actor does not complete a request within the configured `request-timeout` period a `spray.http.Timedout` message is sent to the timeout handler, which can be the service actor itself or another actor (depending on the `timeout-handler` config setting). The timeout handler then has the chance to complete the request within the time period configured as `timeout-timeout`. Only if the timeout handler also misses its deadline for completing the request will the connector servlet complete the request itself with a “hard-coded” error response (which you can change by overriding the `timeoutResponse` method of the `Servlet30ConnectorServlet`).

Send Confirmations

If required the connector servlet can reply with a “send confirmation” message to every response (part) coming in from the application. You request a send confirmation by modifying a response part with the `withAck` method (see the *ACKed Sends* section of the *spray-can* documentation for example code). Confirmation messages are especially helpful for triggering the sending of the next response part in a response streaming scenario, since with such a design the application will never produce more data than the servlet container can handle.

Send confirmations are always dispatched to the actor, which sent the respective response (part).

Closed Notifications

The Servlet API completely hides the actual management of the HTTP connections from the application. Therefore the connector servlet has no real way of finding out whether a connection was closed or not. However, if the connection was closed unexpectedly for whatever reason a subsequent attempt to write to it usually fails with an `IOException`. In order to adhere to same message protocol as the *spray-can HTTP Server* the connector servlet therefore dispatches any exception, which the servlet container throws when a response (part) is written, back to the application wrapped in an `Tcp.ErrorClosed` message.

In addition the connector servlet also dispatches `Tcp.Closed` notification messages after the final part of a response has been successfully written to the servlet container. This allows the application to use the same execution model for *spray-servlet* as it would for the *spray-can HTTP Server*.

5.8.7 HTTP Headers

The connector servlet always passes all received headers on to the application. Additionally the values of the `Content-Length` and `Content-Type` headers are interpreted by the servlet itself. All other headers are of no interest to it.

Also, if your `HttpResponse` instances include a `Content-Length` or `Content-Type` header they will be ignored and *not* written through to the servlet container (as the connector servlet sets these response headers itself).

Note: The `Content-Type` header has special status in *spray* since its value is part of the `HttpEntity` model class. Even though the header also remains in the `headers` list of the `HttpRequest` *spray*'s higher layers (like *spray-routing*) only work with the `Content-Type` value contained in the `HttpEntity`.

5.8.8 Accessing HttpServletRequest

If your application needs access to the `javax.servlet.http.HttpServletRequest`, the `spray.servlet.servlet-request-access` setting can be set to `on`. This results in the connector servlet adding an additional request header of type `spray.servlet.ServletRequestInfoHeader`. This allows the service actor (or directives) to access members of `HttpServletRequest` that are not in `HttpRequest`. This is necessary when working with container managed security and access to the authenticated principal is required (via `getUserPrincipal`) or when accessing an authenticated client SSL certificate (via `getAttribute("javax.servlet.request.X509Certificate")`).

5.8.9 Differences to spray-can

Chunked Requests Since the Servlet API does not expose the individual request parts of chunked requests to a servlet there is no way *spray-servlet* can pass them through to the application. The way chunked requests are handled is completely up to the servlet container.

Chunked Responses *spray-can* renders `ChunkedResponseStart`, `MessageChunks` and `ChunkedMessageEnd` messages directly to “the wire”. Since the Servlet API operates on a somewhat higher level of abstraction *spray-servlet* can only write these messages to the servlet container one by one, with `flush` calls in between. The way the servlet container interprets these calls is up to its implementation.

Closed Messages The Servlet API completely hides the actual management of the HTTP connections from the application. Therefore the connector servlet has no way of finding out whether a connection was closed or not. In order to provide a similar message protocol as *spray-can* the connector servlet therefore simply assumes that all connections are closed after the final part of a response has been written, no matter whether the servlet container actually uses persistent connections or not.

Timeout Semantics When working with chunked responses the semantics of the `request-timeout` config setting are different. In *spray-can* it designates the maximum time, in which a response must have been *started* (i.e. the first chunk received), while in *spray-servlet* it defines the time, in which the response must have been *completed* (i.e. the last chunk received).

HTTP Pipelining & SSL Support Whether and how HTTP pipelining and SSL/TLS encryption are supported depends on the servlet container implementation.

5.8.10 Packaging a WAR file

If you use the `xsbt-web-plugin` you can very easily package your project into a WAR file with the `package` command provided by the plugin.

5.8.11 Example

The `/examples/spray-servlet/` directory of the `spray` repository contains a number of example projects for `spray-servlet`.

simple-spray-servlet-server

This example implements a very simple web-site built on top of `spray-servlet`. It shows off various features like streaming and timeout handling.

Follow these steps to run it on your machine:

1. Clone the `spray` repository:

```
git clone git://github.com/spray/spray.git
```

2. Change into the base directory:

```
cd spray
```

3. Run SBT:

```
sbt "project simple-spray-servlet-server" container:start shell
```

4. Browse to `http://127.0.0.1:8080/`

5. Alternatively you can access the service with `curl`:

```
curl -v 127.0.0.1:8080/ping
```

6. Stop the service with:

```
container:stop
```

5.9 spray-testkit

One of `sprays` core design goals is good testability of the created services. Since actor-based systems can sometimes be cumbersome to test `spray` fosters the separation of processing logic from actor code in most of its modules.

For services built with `spray-routing` `spray` provides a dedicated test DSL that makes actor-less testing of route logic easy and convenient. This “route test DSL” is made available with the `spray-testkit` module.

5.9.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) `spray-testkit` depends on

- `spray-http` (with ‘provided’ scope)
- `spray-httpx` (with ‘provided’ scope)
- `spray-routing` (with ‘provided’ scope)
- `spray-util`
- akka-actor 2.1.x (with ‘provided’ scope, i.e. you need to pull it in yourself)
- akka-testkit 2.1.x (with ‘provided’ scope, i.e. you need to pull it in yourself)
- scalatest (with ‘provided’ scope, for the `ScalatestRouteTest`)

- `specs2` (with 'provided' scope, for the `Specs2RouteTest`)

5.9.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-testkit* into your classpath. However, since you normally don't need to have access to *spray-testkit* from your production code, you should limit the dependency to the `test` scope:

```
libraryDependencies += "io.spray" %% "spray-testkit" % version % "test"
```

Currently *spray-testkit* supports the two most popular scala testing frameworks, `scalatest` and `specs2`. Depending on which one you are using you need to mix either the `ScalatestRouteTest` or the `Specs2RouteTest` trait into your test specification.

5.9.3 Usage

Here is an example of what a simple test with *spray-testkit* might look like:

```
import org.specs2.mutable.Specification
import spray.testkit.Specs2RouteTest
import spray.routing.HttpService
import spray.http.StatusCodes._

class FullTestKitExampleSpec extends Specification with Specs2RouteTest with HttpService {
  def actorRefFactory = system // connect the DSL to the test ActorSystem

  val smallRoute =
    get {
      pathSingleSlash {
        complete {
          <html>
            <body>
              <h1>Say hello to <i>spray</i>!</h1>
            </body>
          </html>
        }
      } ~
      path("ping") {
        complete("PONG!")
      }
    }

  "The service" should {

    "return a greeting for GET requests to the root path" in {
      Get() ~> smallRoute ~> check {
        responseAs[String] must contain("Say hello")
      }
    }

    "return a 'PONG!' response for GET requests to /ping" in {
      Get("/ping") ~> smallRoute ~> check {
        responseAs[String] === "PONG!"
      }
    }
  }
}
```



```

"leave GET requests to other paths unhandled" in {
  Get("/kermit") ~> smallRoute ~> check {
    handled must beFalse
  }
}

"return a MethodNotAllowed error for PUT requests to the root path" in {
  Put() ~> sealRoute(smallRoute) ~> check {
    status === MethodNotAllowed
    responseAs[String] === "HTTP method not allowed, supported methods: GET"
  }
}
}
}

```

The basic structure of a test built with *spray-testkit* is this (expression placeholder in all-caps):

```

REQUEST ~> ROUTE ~> check {
  ASSERTIONS
}

```

In this template *REQUEST* is an expression evaluating to an `HttpRequest` instance. Since both *RouteTest* traits extend the *spray-htpx Request Building* trait you have access to its mini-DSL for convenient and concise request construction.¹

ROUTE is an expression evaluating to a *spray-routing* `Route`. You can specify one inline or simply refer to the route structure defined in your service.

The final element of the `~>` chain is a `check` call, which takes a block of assertions as parameter. In this block you define your requirements onto the result produced by your route after having processed the given request. Typically you use one of the defined “inspectors” to retrieve a particular element of the routes response and express assertions against it using the test DSL provided by your test framework. For example, with *specs2*, in order to verify that your route responds to the request with a status 200 response, you’d use the `status` inspector and express an assertion like this:

```
status mustEqual 200
```

The following inspectors are defined:

¹ If the request URI is relative it will be made absolute using an implicitly available instance of `DefaultHostInfo` whose value is “`http://example.com`” by default. This mirrors the behavior of *spray-can* which always produces absolute URIs for incoming request based on the request URI and the `Host`-header of the request. You can customize this behavior by bringing an instance of `DefaultHostInfo` into scope.

Inspector	Description
body: HttpEntity.NonEmpty	Returns the contents of the response entity. If the response entity is empty a test failure is triggered.
charset: HttpCharset	Identical to contentType.charset
chunks: List[MessageChunk]	Returns the list of message chunks produced by the route.
closingExtension: String	Returns chunk extensions the route produced with a ChunkedMessageEnd response part.
contentType: ContentType	Identical to body.contentType
definedCharset: Option[HttpCharset]	Identical to contentType.definedCharset
entity: HttpEntity	Identical to response.entity
handled: Boolean	Indicates whether the route produced an HttpResponse for the request. If the route rejected the request handled evaluates to false.
header(name: String): Option[HttpHeader]	Returns the response header with the given name or None if no such header can be found.
header[T <: HttpHeader: ClassTag]: Option[T]	Identical to response.header[T]
headers: List[HttpHeader]	Identical to response.headers
mediaType: MediaType	Identical to contentType.mediaType
rejection: Rejection	The rejection produced by the route. If the route did not produce exactly one rejection a test failure is triggered.
rejections: List[Rejection]	The rejections produced by the route. If the route did not reject the request a test failure is triggered.
response: HttpResponse	The HttpResponse returned by the route. If the route did not return an HttpResponse instance (e.g. because it rejected the request) a test failure is triggered.
responseAs[T: Unmarshaller: ClassTag]: T	Unmarshals the response entity using the in-scope FromResponseUnmarshaller for the given type. Any errors in the process trigger a test failure.
status: StatusCode	Identical to response.status
trailer: List[HttpHeader]	Returns the list of trailer headers the route produced with a ChunkedMessageEnd response part.

5.9.4 Sealing Routes

The section above describes how to test a “regular” branch of your route structure, which reacts to incoming requests with HTTP response parts or rejections. Sometimes, however, you will want to verify that your service also translates *Rejections* to HTTP responses in the way you expect.

You do this by wrapping your route with the `sealRoute` method defined by the `HttpService` trait. The `sealRoute` wrapper applies the logic of the in-scope *ExceptionHandler* and *RejectionHandler* to all exceptions and rejections coming back from the route, and translates them to the respective `HttpResponse`.

The *on-spray-can* examples defines a simple test using `sealRoute` like this:

```
"return a MethodNotAllowed error for PUT requests to the root path" in {
  Put() ~> sealRoute(demoRoute) ~> check {
    status === MethodNotAllowed
    responseAs[String] === "HTTP method not allowed, supported methods: GET, POST"
  }
}
```

}

5.9.5 Examples

A full example of how an API service definition can be structured in order to be testable with *spray-testkit* and without actor involvement is shown with the *on-spray-can* example. [This](#) is its test definition.

Another great pool of examples are the tests for all the predefined directives in *spray-routing*. They can be found [here](#).

5.10 spray-util

The *spray-util* module contains a number of smaller helper classes that are used by all other *spray* modules, except *spray-http*, which is kept intentionally free of other *spray* dependencies.

5.10.1 Dependencies

Apart from the Scala library (see *Current Versions* chapter) *spray-util* only depends on *akka-actor* (with ‘provided’ scope, i.e. you need to pull it in yourself).

5.10.2 Installation

The *Maven Repository* chapter contains all the info about how to pull *spray-util* into your classpath.

Afterwards just import `spray.util._` to bring all relevant identifiers into scope.

5.10.3 Configuration

Just like Akka *spray-util* relies on the *typesafe config* library for configuration. As such its JAR contains a `reference.conf` file holding the default values of all configuration settings. In your application you typically provide an `application.conf`, in which you override Akka and/or *spray* settings according to your needs.

Note: Since *spray* uses the same configuration technique as Akka you might want to check out the [Akka Documentation on Configuration](#).

This is the `reference.conf` of the *spray-util* module:

```
#####
# spray-util Reference Config File #
#####

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

spray {

  # Always contains the deployed version of spray.
  # Referenced, for example, from the `spray.can.server.server-header` setting.
  version = "<VERSION>"
}
```

5.10.4 Pimps

spray-util provides a number of convenient “extensions” to standard Scala and Akka classes.

The currently available pimps can be found [here](#). Their hooks are placed in the `spray.util` package object, you bring them in scope with the following import:

```
import spray.util._
```

Side Note

Even though now officially somewhat frowned upon due to its arguably limited PC-ness we still like the term “pimps” for these, since it honors the origins of the technique (the “pimp-my-library” pattern, as it was originally coined by Martin Odersky in a [short article](#) in late 2006) and provides a very succinct and, in the scala community, well-known label for it.

5.10.5 LoggingContext

The `LoggingContext` is a simple `akka.event.LoggingAdapter` that can always be implicitly created. It is mainly used by *spray-routing* directives, which require a logging facility for an implicitly available `ActorRefFactory` (i.e. `ActorSystem` or `ActorContext`).

Project Info

6.1 Current Versions

Since *spray* heavily depends on [Akka](#) its releases are usually closely tied to specific Akka versions.

6.1.1 0.9.0

This version targets Scala 2.9.x and Akka 1.3.x and is therefore not recommended for new projects anymore. Its documentation doesn't live here but in the (old) [github wiki](#).

6.1.2 1.0.1

This is the latest version targeting Scala 2.9.3 and Akka 2.0.5. Its sources live in the [release/1.0](#) branch of the *spray* repository.

6.1.3 1.1.3 / 1.2.3 / 1.3.3

The current and stable *spray* release is a triple release, targeting both Scala 2.10 and Scala 2.11 as well as three Akka versions at the same time.

Please choose **1.1.3**, **1.2.3** or **1.3.3** depending on what Scala/Akka version you are targeting:

- *spray* **1.1.3** is built against Scala 2.10.5 and Akka 2.1.4.
Its sources live in the [release/1.1](#) branch of the *spray* repository.
- *spray* **1.2.3** is built against Scala 2.10.5 and Akka 2.2.5.
Its sources live in the [release/1.2](#) branch of the *spray* repository.
(Please note that Akka 2.2.3 or later is *required*, earlier Akka versions will *not* work!)
- *spray* **1.3.3** is built against Scala 2.10.5 and Akka 2.3.9 as well as Scala 2.11.6 and Akka 2.3.9.
Its sources live in the [release/1.3](#) branch of the *spray* repository.
Note: Contrary to version 1.1.3 and 1.2.3 the 1.3.3 release is published with **crosspaths enabled** since it targets two Scala versions at the same time!

For information about where to find the *spray* artifacts please check out the [Maven Repository](#) chapter.

6.1.4 Shapeless Versions

If you want to use `shapeless` as well as `spray-routing` in your application you need to select the version of `spray(-routing)` that was built against the `shapeless` release which you'd like to use.

- For `shapeless` 1.2.4 you should use `spray` **1.1.3**, **1.2.3** or **1.3.3** and the `spray-routing` module.
- For `shapeless` 2.0.0 you should use `spray` **1.3.2** (Scala 2.10 or Scala 2.11) and the `spray-routing-shapeless2` module instead of `spray-routing`.
- For `shapeless` 2.1.0 you should use `spray` **1.3.3** (Scala 2.10 or Scala 2.11) and the `spray-routing-shapeless2` module instead of `spray-routing`.

6.1.5 Nightly Builds

If you'd like to have access to the most recent changes and additions without having to build `spray` yourself you can rely on the nightly builds, which we are currently publishing for the `release/1.0`, `release/1.1`, `release/1.2` and `release/1.3` branches of the `spray` repository. Every day shortly past midnight UTC a new build is made available unless the respective branch has not seen any new commits since the last build.

In order to help you identify the exact commit from which a build is cut every artifact directory includes a `commit-<version>.html` file containing the commit hash with a link to the commit on github.

Nightly builds are available from the <http://nightlies.spray.io> repository.

6.2 Migration from M8

6.2.1 Big breaking changes

Replacement of `HttpBody` by `HttpData`

`HttpBody` was completely replaced by `HttpData`. `HttpData` is a data structure that abstracts over either `ByteString` buffers (`HttpData.Bytes`), data from files (`HttpData.FileBytes`), and combinations of both (`HttpData.Compound`). This allows `spray-can` to combine both heap byte buffers and data from files without ever having to load file data into a heap buffer. The companion object of `HttpEntity` contains lots of overloaded apply methods to create an `HttpEntity` without having to deal with `HttpData` directly.

Authentication

Authentication rejections were remodelled. There's now only one rejection type, `AuthenticationFailedRejection`, that contains a cause (`CredentialsMissing` or `CredentialsRejected`) and a list of headers to add to the response as a challenge for the client. Previously, challenge generation wasn't handled uniformly and challenges weren't generated at all in some cases. For your custom authenticator, you now have to generate the challenge headers directly in the `ContextAuthenticator` implementation. `HttpAuthenticator` now has a new abstract method to implement to return challenge headers.

Chunking support

With request chunk aggregation turned off (`spray.can.server.request-chunk-aggregation-limit = 0`) `spray-can` will deliver request chunks as is to the user-level handler. Previously, all chunks of a request came in subsequently without any interaction needed from the handler. Now the `ChunkedRequestStart` message has to be acknowledged with a `RegisterChunkHandler` message which contains the `ActorRef` of an actor to handle

the chunks before any actual `MessageChunks` will be delivered. This allows to separate the handling of multiple incoming request chunk streams to several chunk handlers.

PathMatcher infrastructure

The path matching infrastructure was changed by replacing automatic behavior with more explicit directives. Mainly, this affects uses of nested `path("")` directives. Previously, `path("")` would either match 1) nothing, 2) a single slash, or 3) two slashes before the path end. Since the question of whether to deliver identical content under two different URIs (adding a trailing slash to a URI results in a semantically *different* URI!) is one of continuous discussion (see for example: <http://googlewebmastercentral.blogspot.de/2010/04/to-slash-or-not-to-slash.html>) we want to make it easy for your to pick your strategy and be consistent about it. With the new version, you can use one of the new directives instead of `path("")` to be explicit about what to match:

- `pathEnd`, which matches just the path end without any trailing slashes
- `pathSingleSlash`, which matches the path only when it ends with a slash
- `pathEndOrSingleSlash`, which matches in both cases

Directives `path` and `pathPrefix` previously matched an optional beginning slash from the (remaining) path. The beginning slash is now required. If that's not what you want use `rawPathPrefix` instead.

Also:

- `PathMatcher.Slash` replaces the old `PathMatcher.Slash_!` which matches exactly one slash.
- `PathMatcher.PathEnd` now matches exactly the end of the path (before: an optional slash as well).
- `PathMatcher.Segments` also doesn't match a trailing slash any more.
- `PathMatcher.Empty` was renamed to `PathMatcher.Neutral`.

It is now easy to create path matchers for optional or repeated matches. Use `PathMatcher.?` to make a `PathMatcher` optional. Use `PathMatcher.repeat` to capture several matches.

(Un)marshalling

(TODO)

- Introduction of `ToResponseMarshaller` etc.
- `RequestContext.complete` overloads gone
- `CompletionMagnet` gone

6.2.2 Comprehensive list of breaking changes

The following list is an annotated version of all commits that were marked as breaking (!) since M8. You may find additional information on how to fix broken code by looking into the linked commits. Changes are often accompanied with test changes that show how code can be fixed.

spray-cache

- `ExpiringLruCache` uses `Duration` instead of `Long` for timeouts ²³⁹⁴⁷²⁰.
- `LruCache.apply` doesn't allow `Duration.Zero` any more but `Duration.Inf`, instead, for not specifying any timeout ²³⁹⁴⁷²⁰.

²³⁹⁴⁷²⁰ use `util.Timestamp` in `ExpiringLruCache`

spray-can

Server-side changes:

- Http handlers must answer `ChunkedRequestStart` message with `RegisterChunkHandler` command (see *chunking support*) [d738e54]
- `ServerSettings` timeout settings were moved into a separate class ⁹¹⁹³³¹⁸.

Client-side changes:

- new member `HostConnectorSettings.maxRedirects` to enable automatic redirection handling on the client side [98365ff]
- `spray.can.client.user-agent-header` vs. specifying a custom `User-Agent` header behavior changed: a custom `User-Agent` header now always overrides any configuration setting. The setting is used when there's no custom header specified [6fec00c].
- Connection failure and request timeout now produce dedicated exceptions on the client side [4b48875].
- `HostConnectorSettings.connectionSettings` are now read from `spray.can.client` instead of from `spray.can.host-connector.client` [9abbcf6].
- `spray.can.client.ssl-encryption` is gone. Instead, `Http.Connect` got a new `sslEncryption` parameter replacing the global setting. [e922cd4].
- `HostConnectorSetup` must now be created from hostname and port. Before it was created from an `InetSocketAddress` which wasn't enough to distinguish virtual hosts [a47f3b0].

Other changes:

- `keepOpenOnPeerClosed` is not supported for `Http` and was removed from `Http.Register` [f6b0292].
- `ClientConnectionSettings` and `ServerSettings` got a new member `maxEncryptionChunkSize` [0b5ef36].
- Lots of formerly public types belonging to spray's private API were marked as such [da29cdf].
- `spray.can.server.response-size-hint` was adapted to `spray.can.server.response-header-size-hint`. The same for `spray.can.client.request-size-hint` [ba1ae77].
- `Content-Length.length` is now a `Long` value. Also `spray.can.server.parsing.max-content-length` and `incoming-auto-chunking-threshold-size` [b2fee8d].
- `SetIdleTimeout` command now always resets the timeout [ab17f00].
- `SslTlsSupport` pipeline stage now publishes a `SSLSessionEstablished` event with session details [80982d4].
- New `ParserSettings.sslSessionInfoHeader` setting which enables the automatic addition of a synthetic `SSL-Session-Info` header to a request/response with SSL session information [e486900].
- `ClientConnectionSettings.userAgentHeader` is now modelled directly by an `Option[User-Agent]`. [da12531].

spray-http

- `Access-Control-Allow-Origin` and `Origin` header models now have members of newly introduced type `HttpOrigin` instead of the previous `Uri` which didn't completely match the model [015f3c6].
- `Renderer.seqRenderer` and related signatures changed [e058a43].

⁹¹⁹³³¹⁸ break out `ServerSettings` timeout settings into sub case class, closes #489

- in `Uri.Query` a `'='` is rendered even for empty values unless the special value `Query.EmptyValue` is used. Also, a query parsed from `?key=` will now be rendered the same way (previously, a trailing `'='` was always stripped) [[d2b8bba](#)].
- `(Multipart)FormData.fields` are now represented as `Seq` to be able to model duplicate fields [[ad593d1](#)].
- `HttpMessage.entityAccepted` was renamed to `HttpMessage.isEntityAccepted` [[5d78dae](#)].
- *Replacement of `HttpBody` by `HttpData`* [[c6f49cc](#)].
- Many charsets in `HttpCharsets` are not any more available as static values. Use `HttpCharset.getForKey("windows-1252")` to access a particular charset [[f625b5a](#)].
- `Uri.Query.apply` and `Uri.Host.apply` have a new charset parameter [[88a25f7](#)].
- `Uri.Query` has a new subtype `Uri.Query.Raw` which will be generated when parsing with mode `Uri.ParsingMode.RelaxedWithRawQuery` [[d8a9ee4](#)].
- `MediaRanges.custom` was renamed to `MediaRange.custom` [[a915b8f](#)].
- `HttpSuccess` and `HttpFailure` are not public API any more. Use `StatusCode.isSuccess` instead [[a9e0d2c](#)].
- `HttpIp` was replaced by `RemoteAddress` which also supports “unknown” addresses. `X-Forwarded-For.ips` member was renamed to `addresses`. `RemoteAddress.ip` member was renamed to `address` [[443b0d8](#)].

spray-routing

- `RequestContext.complete` overloads were removed in favor of using the marshalling infrastructure (see *(Un)marshalling*) [[4d787dc](#)].
- `CompletionMagnet` is gone in favor of the new `ToResponseMarshaller` infrastructure [[7a36de5](#)].
- `FieldDefMagnetAux`, `ParamDefMagnetAux`, and `AnyParamDefMagnetAux` are gone and replaced by a simpler construct [[d86cb80](#)].
- `RequestContext.marshallingContext` is gone. `produce directive` loses its status and header parameter which can be replaced by using an appropriate `ToResponseMarshaller` [[b145ced](#)].
- `AuthenticationFailedRejection` now directly contains challenge headers to return. There’s no need to implement a (fake) `HttpAuthenticator` to make use of the rejection (see *Authentication*) [[9c9b976](#)].
- `FileAndResourceDirectives.withTrailingSlash` and `filePath` are now private [[ab35761](#)].
- `decompressRequest` and `compressResponse` now always need parentheses. Also, encoding directives like the `compressResponse` automatically use the `autoChunkFileBytes` directives to avoid having to load potentially huge files into memory [[e3defb4](#)].
- `(h) require directives` can now take several rejections instead of an `Option` of only one [[9c11228](#)].
- `detachTo` is gone in favor of `detach()` which always needs parentheses. The underlying implementation is now `Future`-based and needs an (implicit or explicit) `ExecutionContext` or `ActorRefFactory` in scope [[ead4a70](#)].
- `PathMatcher.(flat)Map` were renamed to `PathMatcher.h(flat)Map`. `map` and `flatMap` were reintroduced for `PathMatcher1` instances [[8c91851](#)].
- `AuthenticationFailedRejection` and `AuthenticationRequiredRejection` were merged and remodelled. [[034779d](#)]

- `PathMatchers.Empty` was renamed to `PathMatchers.Neutral` [[ee7fe47](#)].
- `Slash_!` is gone and `Slash` got its semantics. `PathEnd` now just matches the end of the path. `PathDirectives` were adapted to have the same semantics as before [[1480e73](#)].
- `UserPassAuthenticator.cached` was renamed to `CachedUserPassAuthenticator.apply` ¹³²⁶⁰⁴⁶.
- `PathMatcher.apply` now takes a `Path` prefix instead of a `String` [[3ff3471](#)].
- `PathMatcher.Segments` doesn't match trailing slashes anymore. Implicit infrastructure for `PathMatcher.?` was changed [[8ee49d7](#)].
- `pathEnd` and `pathEndOrSingleSlash` were introduced to replace the former `path("")` (see *PathMatcher infrastructure*) [[f0cbf25](#)].

spray-htpdx

(TODO)

- [[ae17d18](#)]
- [[9d27559](#)]
- [[fad2ff2](#)]
- [[f8f5b6d](#)]
- [[ebaa580](#)]
- [[ebe3e97](#)]
- [[dd51be5](#)]
- [[f5b1535](#)]
- [[adf9170](#)]
- [[f5997f8](#)]

spray-io

(TODO)

- [[01c4aa9](#)]
- [[5f23219](#)]
- [[76345ba](#)]
- [[2c77d8f](#)]

spray-testkit

(TODO)

- [[6a99cb7](#)]
- [[72c9397](#)]
- [[680fde0](#)]

¹³²⁶⁰⁴⁶ move `UserPassAuthenticator.cached` to `CachedUserPassAuthenticator.apply`, fixes #352

- [\[3b4ac55\]](#)

spray-util

(TODO)

- [\[e234dd9\]](#)
- [\[b0b90b3\]](#)
- The settings infrastructure was reworked. `XxxSettings.apply(config: Config)` now takes the root config instead of the subconfig. For the old behavior use the new `fromSubConfig` instead [\[78d7e4a\]](#).

6.3 Maven Repository

The latest *spray* releases are available from [Maven Central](#), so no special resolver should be required to be able to pull them in. In addition, older artifacts (including milestones and RCs) are hosted in this repository:

<http://repo.spray.io>

If you use **SBT** you'll want to add the following resolver:

```
resolvers += "spray repo" at "http://repo.spray.io"
```

Nightly builds are available from <http://nightlies.spray.io>, to use them add this resolver:

```
resolvers += "spray nightlies repo" at "http://nightlies.spray.io"
```

6.3.1 Artifact Naming

All *spray* artifacts follow this naming scheme:

Group ID `io.spray`

Artifact ID Module Name

Version Release Version

So, for expressing a dependency on a *spray* module with **SBT** you'll want to add something like this to your project settings:

```
libraryDependencies += "io.spray" %% "spray-can" % "1.x.x" // versions 1.1.x and 1.2.x
```

or (note the %%):

```
libraryDependencies += "io.spray" %% "spray-can" % "1.3.x"
```

Make sure to replace the artifact name and version number with the one you are targeting! (see *Current Versions*)

6.4 Contributing

We value all kinds of contributions from the community, not just actual code. Maybe the easiest and yet very good way of helping us improve *spray* is to ask questions, voice concerns or propose improvements on the [Mailing List](#). Or simply tell us about you or your organization using *spray* by sending us a small statement for inclusion on the [References](#) page.

If you do like to contribute actual code in the form of bug fixes, new features or other patches this page gives you more info on how to do it.

6.4.1 Building *spray*

Since *spray* is open-source and hosted on github you can easily build it yourself.

Here is how:

1. Install *SBT* (the master branch is currently built with *SBT* 0.12.3).
2. Check out the *spray* source code from the [github repository](#). Pick the branch corresponding to the version you are targeting (check the *Current Versions* chapter for more info on this).
3. Run `sbt compile test` to compile the suite and run all tests.

6.4.2 Contributing documentation

The site, i.e. what you see here at <http://spray.io>, is built with *spray* itself. It uses *sphinx* to generate documentation from *reStructured text* files located in the `docs` folder of the *spray* checkout. The documentation is served from the `site` sub-project of the *sbt* build. If you want to contribute documentation make sure you can build and view the site locally. Follow these instructions to get the site project working locally:

- Follow the instructions in the above section about *building spray*.
- Install *sphinx* (in Debian / Ubuntu install the `python-sphinx` package, for OS/X see this [mailing list thread](#)).
- Find the path of `sphinx-build` (in Ubuntu it's probably `/usr/bin/sphinx-build`)
- Set the `SPHINX_PATH` environment variable to that path.
- Run `sbt`
- In the *sbt* shell use `project site` to change into the site project.
- Use `compile` to build the site, this will take some time when running for the first time (~ 1 - 3 minutes).
- Use `re-start`¹ to start the local site server.
- Browse to <http://localhost:8080>
- Use `~ products` in *sbt* to let it monitor changes to the documentation sources automatically.
- Edit the documentation files inside the `docs` subdirectory. After saving a file, it will be automatically picked up by *sbt*, then it will be regenerated and be available in the browser after ~ 1 - 5 seconds with a refresh of the page.

6.4.3 git Branching Model

The *spray* team follows the “standard” practice of using the `master` branch as main integration branch, with WIP- and feature branches branching of it. The rule is to keep the `master` branch always “in good shape”, i.e. having it compile and test cleanly.

Additionally we maintain release branches for older and possibly future releases.

¹ `re-start` is a task from the *sbt-revolver* plugin which starts a project in the background while you can still use the *sbt* shell for other tasks.

6.4.4 git Commit Messages

We follow the “imperative present tense” style for commit messages (more info [here](#)) and begin each message with the module name(s) touched by the commit followed by a colon. Additionally, in order to make it easier for us (and everyone else) to track the effects of changes onto the public API we also explicitly classify every commit into exactly one of three categories:

Category	Description	Marker
Neutral	Only touches things “under the hood” and has no effect on <i>spray</i> ’s public API.	=
Extending	Extends the API by adding things. In rare cases this might break code due to things like identifier shadowing but is generally considered a “safe” change.	+
Breaking	Changes or removes public API elements. Will definitely break user code relying on these parts of the API surface.	!

Apart from the actual Scala interfaces the public API surface covered by these categories also includes configuration settings (most importantly the `reference.conf` files).

The category that a commit belongs to is indicated with a respective marker character that the commit’s message must start with (followed by a space char), e.g. `= testkit: clean up imports`. Note that *all* commits must carry exactly one of the markers listed in the table above, with one exception: merge commits that don’t introduce any changes themselves do not have to carry a marker. Instead, they start with “Merge”. Requiring the marker makes sure that the committer has actively thought about the effects of the commit on the public API surface.

Also, all commits in the “Extending” and especially in the “Breaking” category should contain a dedicated paragraph (in addition to the summary line) explaining in what way the change breaks the API and why this is necessary/beneficial. These paragraphs form the basis of release-to-release migration guides.

6.4.5 Issue Tracking

Currently the *spray* team uses the [Issues Page](#) of the projects [github repository](#) for issue management. If you find a bug and would like to report it please go there and create an issue.

If you are unsure, whether the problem you’ve found really is a bug please ask on the [Mailing List](#) first.

6.4.6 Contributor License Agreement (CLA)

Contributions to the project, no matter what kind, are always very welcome. However, we would like to make sure that we as the project maintainers as well as the contributors are properly covered with regard to the legal aspects of their contributions. This is why we can only accept patches if the patch is your original work and you license your work to the *spray* project under the [project’s open source license](#).

In order to provide a proper legal foundation for this we need to ask you to sign our CLA, which is a direct adaptation of the [Apache Foundation’s Individual Contributor License Agreement V2.0](#).

If you have not already done so, please download, complete and sign a copy of the CLA and then scan and [email](#) us a PDF file! If you prefer you can also snail-mail us the original, please ask for the mailing address.

6.4.7 Pull Requests

If you’d like to submit a code contribution please fork the [github repository](#) and [send us pull request](#) against the `master` branch (or the respective release branch, depending on what version your patch is targeting). Please keep in

mind that we might ask you to go through some iterations of discussion and refinements before merging and that you will need have signed a CLA first!

6.5 Changelog

6.5.1 Version 1.1.3 (2015-03-24)

- spray-can:
 - fix server-side `IllegalStateException` when `pipelining-limit > 1` (#958)
 - preserve order of headers during message parsing (#961)
 - introduce `spray.can.server.verbose-error-logging` settings (#977)
 - allow user-specified `Content-Length` for unchunked responses to non-transparent HEAD requests (#965)
 - fix chunked response to non-transparent HEAD request rendering `Content-Length: 0` (#964)
- spray-http:
 - fix handling of empty authority in `Uris` (#1014)
 - make `spray.http.RangeUnit.Bytes` a *case object* (#954)
 - add `Uri::toHttpRequestTargetOriginForm` (#1019)
 - catch `IllegalArgumentException`'s early when parsing headers (#992)
 - fix bug in `Authentication` header parser rules (#1006)
- spray-httpx:
 - switch XML parser to safer default settings to prevent XXE attacks
 - make `MultipartsMarshallers.multipartPartsMarshaller` public (#998)
 - fix `NoEncodingCompressor` (#936)
- spray-io:
 - fix SSL/TLS encryption stopping processing in certain renegotiation scenarios (#1010)
 - fix unnecessary reverse DNS query when connecting by IP with SSL/TLS (#993)
- spray-routing:
 - replace usages of `asInstanceOf` in `NumberMatcher` (#959)
 - fix incorrect behavior of `anyParam(s)` directive with default values (#981)
- upgraded to latest versions of all dependencies where easily possible
- smaller fixes and improvements across the board (e.g. error messages and scaladocs)

6.5.2 Version 1.1.2 (2014-10-08)

- spray-caching:
 - add API for accessing cache key set
- spray-can:
 - add support for rendering of 'Connection: Upgrade' response header

- fix incorrect error message in response parser
- prevent URIs with non-http scheme earlier in request-level client API (#879)
- add support for *Transfer-Encoding: identity* in server and client (#861)
- fix `StackOverflowException` in `spray.can.rendering.toTcpWriteCommand` (#877)
- spray-http:
 - fix URI encoding of non-BMP chars
 - fix *Cache-Control* directives parser (#862)
 - change `Uri.Query` rendering to also encode semicolons
 - add missing alias for *US-ASCII* charset
 - add q-Value to `LanguageRange` (#838)
 - accept comma as cookie separator (#869)
 - allow `0` as special value *If-Modified-Since* and similar headers (#942)
 - remove obsolete *LinkDirective* model
- spray-htpx:
 - unwrap *MalformedContent* error in *ResponseTransformation*
 - add support for new Play twirl with *PlayTwirlSupport* (#887)
 - change default charset for `FormData` marshalling to UTF-8
 - improve encoding of `FormData` fields
 - allow encoding of chunked message with empty-entity `Chunked*Start` (#852)
 - accept illegal headers in multipart parts as *RawHeader* instances (#859)
 - make multipart unmarshaller strip `Content-Type` header (#875)
- spray-routing:
 - fix *listDirectoryContents* not rendering line separators (#831)
 - fix logging users LDAP passwords on error
 - fix OSGi import package statement (wrt. `spray-caching`, `spray-can`, `spray-io` and `shapeless`)
 - fix `UnsupportedOperationException` in *PathMatchers.valueMap2PathMatcher*
- spray-util:
 - remove unnecessary reflection in *PimpedRegex.groupCount*

6.5.3 Version 1.1.1 (2014-03-14)

- spray-can:
 - fix redirects not reusing same connection (#719)
 - add `spray.can.{client,server}.ssl-tracing` flag to turn on `SslTlsSupport` `DEBUG` logging (#750)
 - enabled redirection for any 307 responses
 - avoid parsing attempt after complete input has been consumed
 - don't require use of `CommandWrapper` with `SetRequest/TimeoutTimeout` (#731)

- don't crash if responder to `HttpRequest` is null (#788)
- abort connection on client-side request timeout (#785)
- add client-side support for *Raw-Request-URI* header (#654)
- dispatch acks for `MessageChunk/ChunkedMessageEnd` to sender (#773)
- spray-http:
 - add models for *Allow*, *Link*, *If-(None-)-Match*, *If-(Un)Modified-Since*, *If-Range*, *ETag*, *Last-Modified*, *Content-Range* and *Range* headers
 - add *image/webp* media type (#809)
 - fix relative URI resolution (#740)
 - render Cookies as cookie-pairs in Cookie-headers (ignore other attributes)
 - fix broken *MediaRange::isXXX* implementations (#804)
 - improve *DateTime* from-String parsing to also support full ISO8601 strings
- spray-htpx:
 - fix *Json4sSupport* and *Json4sJacksonSupport* to not extend from *MetaMarshaller*
 - always use ROOT locale for xml parsing exceptions
 - marshal an empty Stream into a response with an empty entity (#789)
 - allow *UrlEncodedFormDataUnmarshaller* to extract duplicate fields (#823)
- spray-io:
 - drop exception (debug logged) in *SslTlsSupport* during `finishClose` (#823)
 - fix *BackPressureHandling* losing acks in particular situations (#798)
 - try to ignore/tone down warnings in *BackPressureHandling* (#771)
- spray-routing:
 - add *CacheConditionDirectives* (*conditional* directive)
 - add *conditional* support to *FileAndResourceDirectives*
 - add *RangeDirectives* and *UnsatisfiableRange* rejection
 - add range support to *FileAndResourceDirectives*
 - add (*optional*)*headerValueByType* directive
 - add *filter/hfilter* to *Directive*
 - add *Allow* header to method rejections
 - add missing support for *RequiredValueReceptacles* in *formField* and *anyParam* directives (#754)
 - handle exceptions in *onComplete* directive
 - better failure msg when binding in *SimpleRoutingApp.startServer* fails (#819)
- spray-servlet:
 - introduce *root-path = AUTO* and make it the default (#701)
 - add termination awaiting to *Initializer::contextDestroyed*
- spray-testkit: allow overriding of *ActorSystem* creation in *RouteTest* (#786)

- spray-util:
 - make testing agnostic to native newline characters (#632)
 - replace underscore with hyphen in *Utils.actorSystemNameFrom* (#786)

6.5.4 Version 1.1.0 (2013-11-29)

- spray-http: added some more helpers around form data and *BodyPart* construction
- spray-io:
 - fixed *Terminated* dead letters from IO layer (#704)
 - fixed *Tcp.SO.reuseAddress* not being applied for outgoing connections (#712)
- spray-routing:
 - fixed problem with *compressResponseIfRequested* allowing unused parameters
- spray-testkit: added *NoAutoHtmlLinkFragments* helper trait

6.5.5 Version 1.1-RC4 (2013-11-20)

- spray-can:
 - fixed occasional *DeathPactException* in *HttpHostConnector* (#699)
- spray-io:
 - fixed *SslTlsSupport* sending out multiple acks in some cases (#689)
 - fixed noisy *DeathPactException* when connection attempt fails (#692)
 - added verification of correct *clientMode* of *SSLEngines* created by **SSLServiceProvider.fromFunc* (#657)
- spray-routing:
 - made *FutureDirectives* and *authenticate(Future)* parameters call-by-name (#694)
 - applied small change as a workaround for <http://youtrack.jetbrains.com/issue/SCL-6214>

6.5.6 Version 1.1-RC3 (2013-11-13)

- spray-can:
 - fixed *IllegalStateException* (“Unexpected slot state: Idle”) in host-level API (#652)
 - fixed client-side request-timeout not cancelled after receiving *ChunkedResponseStart* (#670)
 - fixed negative ‘openConnections’ in *StatsSupport* (#666)
 - fixed possible excessive memory use when parsing incoming chunked messages (#662)
 - fixed server sometimes sending events to wrong receivers
 - improved “Cannot establish effective request URI” error message
- spray-http:
 - fixed OAuth2 bearer token parser problem (#671)
 - added q-value support to *HttpCharset* and *HttpEncoding*

- improved content negotiation logic (#677)
- made auth scheme and “realm” parameter parsing case-insensitive (#668)
- smaller fixes
- spray-htpx:
 - fixed *TwirlSupport* XML marshalling
 - added *DelegatingToResponseMarshallingContext* and *withEntityMapped/withResponseMapped* helpers
- spray-io:
 - fixed *IllegalStateExceptions* in *SslTlsSupport* (#667)
- spray-routing:
 - fixed *HttpService::runRoute* not handling *ConnectionClosed* events (#675)
 - fixed *listDirectoryContents* producing incorrect links when under *pathPrefix* (#684)
 - added *parameterless / modifier* for *PathMatcher*
- spray-util:
 - fixed *SimpleStash* to also stash message senders

6.5.7 Version 1.1-RC2 (2013-10-30)

- spray-can:
 - changed *ClientConnectionSettings::userAgentHeader* from *String* to *Option[‘User-Agent’]* (#458)
 - relaxed parser to allow several identical *Content-Type* headers (#463)
 - improved *HttpHostConnector* for better performance with large numbers of connections (#643)
- spray-http:
 - fixed *DateTime* to allow years up to 9999 (#502)
 - fixed exception leaking from *HttpCharset.custom* (#637)
 - remodeled *HttpIp* to *RemoteAddress* (#638)
- spray-htpx:
 - made play dependency optional in OSGi *Import-Package* statement
 - added *MarshallerM* helper type for abstracting over monadicmarshallers
- spray-io:
 - fixed broken inbound bytes buffering in *SslTlsSupport* (#633)
 - removed too tight assertions, replaced remaining asserts (#631)
- spray-routing:
 - changed *path* directive and *Segments* matcher to never match trailing slashes
 - added *pathEnd*, *pathSingleSlash* and *pathEndOrSingleSlash* directive (#628)
 - added *PathMatcher::repeated* modifier (#636)

6.5.8 Version 1.1-RC1 (2013-10-22)

- Upgraded to Scala 2.10.3 and latest versions of all dependencies
- spray-caching: added a size method to `Cache[V]`
- spray-can:
 - “privatized” all classes/objects not meant to be part of public API
 - replaced `InetSocketAddress` in `HostConnectorSetup` with hostname/port pair (#394)
 - moved `client.ssl-encryption` setting from `reference.conf` into `Http.Connect` message (#396)
 - when creating `HostConnectorSettings` expect client settings at `spray.can.client` (#408)
 - introduced dedicated exceptions for connection failure and request timeout for host-level API
 - made `Content-Length` a long value (#443)
 - add auto-chunking for responses without content-length (#455)
 - broke out `ServerSettings` timeout settings into sub case class (#489)
 - change rendering of default `User-Agent` (#462)
 - added support for custom status codes (#564)
 - added explicit registration of chunk handler for incoming message chunks (#473)
 - added client-side redirection following (#132)
 - removed of `Http.Register::keepOpenOnPeerClosed` (#401)
 - a ton of other fixes, additions and smaller improvements
- spray-http:
 - added `startsWith` and `dropChars` method to `Uri.Path` model
 - added support for `Accept` header extensions and media-type parameters (#310)
 - fixed raw queries performing %-decoding and not being rendered as raw (#330)
 - made only standard charsets available as constants (#340)
 - added model for CORS headers
 - allowed empty `Host` headers
 - fixed `Location` header parsing to accept relative URIs (#484)
 - introduced `HttpData` model replacing the byte array in `HttpBody` and `MessageChunk` (#365)
 - added `CONNECT` method and support for custom HTTP methods (#428)
 - added `StatusCode::allowsEntity` member
 - made multipart form-data more flexible but have it adhere to the RFC more strictly
 - introduced a distinction between `”?key=”` and `”?key”` in queries (#460)
 - added `effectivePort` method on `Uri`
 - many other fixes, additions and smaller improvements
- spray-htpx:
 - flexibilized `RequestBuilding` and `ResponseTransformation` by generalizing the `~>` operator
 - added `mapHeaders`, `removeHeader` and `removeHeaders` to `RequestBuilding`

- moved `unmarshal` and `unmarshalUnsafe` to `Unmarshaller` object
- changed default `StringMarshaller` to prefer UTF-8 encoding over ISO-8859-1 (#498)
- added default `marshallers` for `ByteString` and `HttpData`
- changed default charset for `application/x-www-form-urlencoded` to UTF-8 (#526)
- added support for custom `Content-Disposition` headers
- added support for non-strict `FormDataUnmarshaller`
- added `FromMessageUnmarshaller`, `FromRequestUnmarshaller` and `FromResponseUnmarshaller`
- extended `MarshallingContext` with support for specifying additional HTTP headers
- introduced `ToResponseMarshaller` and supporting infrastructure
- added `Unmarshaller.oneOf` to create negotiating `Unmarshaller` (#581)
- added support for play-json (un)marshalling
- many other fixes, additions and smaller improvements
- spray-io:
 - upgraded to latest upstream Akka IO additions and fixes
 - significantly improved `SslTlsSupport` (#544)
 - smaller fixes, additions and improvements
- spray-routing:
 - added `headerValueByName(Symbol)` and `optionalHeaderValueByName(Symbol)` overload
 - added `rejection` parameter to `Directive::hrequire` and `Directive::require`
 - added `separateOnSlashes` helper for `PathMatchers` (#334)
 - moved `UserPassAuthenticator.cached` to `CachedUserPassAuthenticator.apply` (#352)
 - redefined `PathMatchers.Empty` as `PathMatchers.Neutral` with explicit type annotation (#339)
 - renamed `PathMatcher.(flat)map => h(flat)map`, introduced `map/flatMap` (#274)
 - added `detach` directive which executes its inner route in a future, removed `detachTo` (#240)
 - added `scheme` directives
 - added `PathMatchers.Segments` which will match all remaining segments as `List[String]`
 - added some higher-level compression/decompression directives
 - added `autoChunkFileBytes` directive
 - fixed `getFromDirectory` and `getFromResourceDirectory` not working properly for URIs with encoded chars
 - added `requestInstance` and `requestUri` directives (#525)
 - removed layer of `*Aux` classes by type aliases for simplicity
 - removed `CompletionMagnet` in favor of new `ToResponseMarshaller`
 - removed superfluous `RequestContext::complete` overloads

- added `PathMatcher : ?` modifier, removed `Slash_!`
- many other fixes, additions and smaller improvements
- spray-servlet:
 - added support for providing the `javax.servlet.http.HttpServletRequest` in a special header
 - added `illegal-header-warnings` setting in analogy to spray-can (#553)
 - added `spray.servlet.uri-parsing-mode` setting (#574)
 - smaller fixes, additions and improvements

6.5.9 Version 1.1-M8 (2013-06-10)

- Upgraded to Scala 2.10.2, Akka 2.1.4 and latest versions of all dependencies
- general: added automatic source formatting via `sbt-scalariform`
- spray-caching: folded `Cache.fromFuture` into `Cache.apply`
- spray-can:
 - migrated from old spray-io onto new akka-io
 - completely rewrote client-side APIs for improved architecture and usability
 - completely rewrote HTTP parser for cleanliness and much better performance
 - rewrote HTTP renderer for cleanliness and performance
 - many fixes, additions and smaller improvements
- spray-client:
 - removed `HttpConduit` (now part of spray-can client-side APIs)
 - moved `HttpDialog` from spray-can into spray-client
 - adapted to new spray-can client layer
- spray-http:
 - major refactoring for cleanliness and performance
 - established serializability for all model classes
 - added custom RFC3986-compliant URI model incl. fast parser
 - many fixes, additions and smaller improvements
- spray-httpx:
 - added marshaller for `Try[T]`
 - added `Json4s` native and `Json4S` jackson support
 - many fixes, additions and smaller improvements
- spray-io:
 - migrated large majority of logic to new Akka I/O layer (for Akka 2.2)
 - backported new Akka I/O layer from Akka 2.2 to Akka 2.1
 - updated remaining logic
- spray-routing:

- added several new directives (most notably: FutureDirectives, AnyParamDirectives)
- many fixes, additions and smaller improvements
- spray-servlet:
 - fixed NPE during exception handling using *e.getMessage*
 - enabled dynamic timeout setting via SetRequestTimeout and SetTimeoutTimeout commands
 - smaller improvements
- spray-util: many small fixes and improvements

6.5.10 Version 1.1-M7 (2012-12-19)

- Upgraded to Scala 2.10.0-RC5, Akka 2.1.0-RC6
- spray-can:
 - fixed broken overridability of SprayCanHttpServerApp.system member
 - fixed request rendering to not suppress custom Date headers
 - fixed rendering of ‘Content-Length: 0’ request headers
 - added option for tagging log messages with custom prefixes, closed #153
 - added DefaultHttpClient extension, closed #166
 - added explicit naming for connection actors
- spray-http:
 - fixed non-public access to the HttpRequest URI and queryParams members through the ‘copy’ method
 - added HttpMessagePart extractor
 - improved Date header parser to allow UTC as well as GMT time zone label
 - improved language-tag parser to also accept tags according to BCP 47, closed #168
- spray-httpx: changed JSON rendering to always be UTF-8 encoded
- spray-io:
 - changed IOExtension.ioBridge to IOExtension.ioBridge()
 - added option for tagging log messages with custom prefixes, closed #153
 - added proper supervisor strategy for connection actors: stop (and close connection) on all exceptions
 - improved enabling/disabling of encryption on the connection (SslTlsSupport)
 - flexibilized connection tag provision
 - changed IOClient.Connected events to be sent with connection actor as sender
- spray-routing:
 - fixed rejection duplication in certain cases
 - changed default marshalling encoding to UTF-8
 - added SimpleRoutingApp trait
 - added *complete(Future[StatusCode])* overload
 - added *optionalHeaderValuePF* directive

- simplified and homogenized `ExceptionHandler` and `RejectionHandler` to both return `Routes`
- testkit: turned response `MatchError` into proper error message, closed #165
- util:
 - introduced `SprayActorLogging` trait
 - improved `LoggingContext` with configurability regarding logger name generation
 - improved `PimpedFuture.delay` to accept implicit `ActorRefFactory` instead of `ActorSystem`
- examples:
 - added `simple-routing-app` example
 - renamed `simple-on-jetty` and `simple-on-spray-can` examples to `on-jetty / on-spray-can resp.`
- smaller fixes and improvements

6.5.11 Version 1.1-M6 (2012-11-30)

- Upgraded to Scala 2.10.0-RC3, Akka 2.1.0-RC3, parboiled 1.1.4, shapeless 1.2.3 and spray-json 1.2.3
- Added ‘is-local-ActorRef’ assertions across modules (where applicable)
- spray-can:
 - removed superfluous `ssl-encryption` config setting from `HttpClient`
 - increase default `HttpServer` `pipelining-limit` from 1 to 8
 - introduced `SprayCanHttpServerApp` convenience trait, updated examples
 - smaller performance improvements
- spray-http:
 - added support for ‘Bearer’ scheme in `Authorization` header (OAuth 2.0), closes #155
 - renamed ‘`OtherHttpCredentials`’ to ‘`GenericHttpCredentials`’
 - improved `MediaType` model, added more default `MediaTypes`, closed #157
 - improved warmup
 - improved parser for ‘`X-Forwarded-For`’ header to also accept “unknown” elements
 - added `DateTime.fromIsoDateTimeString`
- spray-httpx: made the `(Un)marshaller[NodeSeq]` also accept ‘`application/xml`’
- spray-io:
 - turned `IOBridge` into `Actor`, added optional parallelism
 - general refactoring for cleanliness and clarity
 - improved flexibility of connection actor creation
 - smaller performance improvements
- spray-routing:
 - added `Directive.recover` and `.recoverPF` modifiers
 - introduced `HeaderDirectives` trait, reworked and improved header extraction directives
 - changed evaluation of ‘`complete`’ directive arguments to `call-by-name`

- fixed incomplete exception handling in Directive.unwrapFuture
- renamed getFromFileName directive to getFromFile
- introduced default, simple HttpServiceActor implementation
- smaller refactoring in PathMatcher API
- spray-servlet: fixed missing application of root-path setting
- spray-testkit: flexibilized ActorSystem provision for RouteTest, closed #162
- spray-util: changed log level of loggers installed via ‘installEventStreamLoggerFor’ from DEBUG to WARNING
- smaller fixes and improvements

6.5.12 Version 1.1-M5 (2012-11-09)

- Upgraded to Scala 2.10.0-RC2 and Akka 2.1.0-RC2
- spray-can:
 - fixed NPE in RequestChunkAggregation
 - removed stray logging statement from RequestChunkAggregation
 - fixed incorrect rendering of chunked responses to HEAD requests
- spray-http: fixed incorrect test for response encoding acceptance
- spray-httpx: removed dangerous StatusCodeMarshaller
- spray-io:
 - improved logging of errors in the SslTlsSupport pipeline stage
 - reworked connection closing, added ConfirmedClose reason, fixed #147
- spray-routing:
 - added option for disabling automatic file-chunking
 - removed HttpService.routingSettings method, provision of RoutingSettings now fully implicit
 - renamed Directive.map and .flatMap to .hmap/.hflatMap, re-added .map and .flatMap pimps for single-value directives
 - improved HttpService ‘runRoute’ wrapper to also work in “sub-route” actors
 - removed ‘filter’ directive, converted all applications to ‘extract(...).flatMap’
 - added ‘pathTest’ directive
 - improve rendering of directory listings
 - changed default redirection type to ‘302 Moved Permanently’
 - improved host directive
 - added ‘dynamicIf’ directives
 - improved and flexibilized DebuggingDirectives
 - add ‘complete’ overload taking a single StatusCode parameter
- spray-can & spray-servlet: increased default idle and request timeouts to 120s and 30s respectively
- spray-testkit: improved provision of custom RouteTestTimeout

- spray-util: added 'installDebuggingEventStreamLoggers' convenience helper
- smaller fixes and improvements

6.5.13 Version 1.1-M4 (2012-10-19)

- Upgraded to Scala 2.10.0-RC1 and Akka 2.1.0-RC1

6.5.14 Version 1.0-M4 (2012-10-19)

- Upgraded to parboiled 1.1.3 and spray-json 1.2.2
- routing: further flexibilized directive combination with StandardRoutes
- routing: (re)added 'complete' overload taking a Future[HttpResponse], closes #143
- routing: added new directives:
 - unmatchedPath
 - listDirectoryContents
 - getFromBrowseableDirectory / getFromBrowseableDirectories
- smaller fixes and improvements

6.5.15 Version 1.0-M3 (2012-10-12)

Largest refactoring in the history of the project. Lots of breaking changes. Most importantly:

- Completely new documentation at <http://spray.io>
- Renamed group id from "cc.spray" to "io.spray"
- Renamed packages from "cc.spray..." to simply "spray..."
- Completely refactored suite module structure (from 4 to 10+1 modules)
- Improved naming of many identifiers across the board
- Completely new architecture underneath routing DSL
- Completely new Marshalling and Unmarshalling infrastructure
- ...

6.5.16 Version 1.0-M2 (2012-05-16)

- spray-io
 - Added SslTlsSupport pipeline stage
 - Fixed SetTimeout messages not working
 - Fixed closing of already closed connections improperly handled
 - Fixed bug in wiring of certain pipeline stages
 - Dropped 'confirm-sends' config setting, added 'ack:Boolean' field to IoWorker.Send
 - Renamed 'SendCompleted' event to 'AckSend'

- Made IoWorkerThread daemonic
- Improved pipeline architecture for reduced call stack depth
- spray-can
 - Added SSL/TLS support to HttpServer and HttpClient
 - Added ‘ack-sends’ setting to client and server config
 - Added ‘transparent-head-requests’ server config setting
 - Added HttpClient.SetRequestTimeout message
 - Fixed HttpServer not handling ‘Expect: 100-continue’ headers
 - Fixed HttpClient not properly handling HEAD requests to resources containing a message body
 - Fixed #99 (getFromFile(Name) sometimes throws IllegalStateException when streaming file content)
 - Fixed SetTimeout messages not working
 - Enabled validity verification on HttpRequests and HttpResponses
 - Extended HttpDialog to also accept ActorContexts as dispatcher container
- spray-base
 - http: fixed custom mediatypes not being matched
 - http: fixed ArrayIndexOutOfBoundsException during header parsing of unregistered CustomMediaTypes
 - http: fixed IPv6 Host header parsing failure
 - Fixed #108 (FormDataUnmarshaller should be more resilient)
 - Fixed incorrect error message in HttpContentExtractor
 - Fixed several memory leaks in streamMarshaller
- spray-server
 - Fixed actorSystem dependency in DebuggingDirectives incorrectly named
 - Make ErrorHandling#responseForException public
- spray-client
 - Improve ‘unmarshal’ pipeline step to accept all 2xx responses instead of only 200
- general
 - Added basic, but still incomplete, sphinx-based documentation system for new spray website
 - Upgraded to Akka 2.0.1
 - Extended simple-http-server and simple-http-client examples with optional SSL/TLS support
 - Fixed EOL related test failures on Windows
 - Lots of smaller fixes and improvements

6.5.17 Version 1.0-M1 (2012-04-05)

- Moved spray-can sources into spray codebase
- Added spray-io component (and spray-util for common code)
- Upgraded all components and examples to Akka 2.0

6.5.18 Version 0.9.0 (2012-03-07)

- Fixed another problem in the request retry logic (spray-client)
- Fixed incorrect response status code for authentication failures with invalid credentials
- Fixed “LruCache implementations also caching exceptions”
- Readded time-to-live based expiration to ExpiringLruCache
- Closed #87 (wrap non-200 responses in special exception)
- Closed #88 (added PathElement PathMatcher)

6.5.19 Version 0.9.0-RC4 (2012-02-27)

- Fixed spray-client retries not always honoring Pipelined dispatch strategy
- Added missing location pointer to entity of generated redirection responses
- Added directives: reject, cookie, optionalCookie, setCookie, deleteCookie, headerValue, headerValuePF, clientIP, provide and transformRejections
- Made FilterResult and SprayRoute1 monadic
- Added another overload to the ‘authenticate’ directive
- Added ‘toOption’ pimp to Strings (in utils)

6.5.20 Version 0.9.0-RC3 (2012-02-22)

- Fixed #78 (spray-client: unexpected closing of connection not properly handled)

6.5.21 Version 0.9.0-RC2 (2012-02-17)

- Upgrade to Akka 1.3.1
- Fixed getFromResource directive to not serve “content” of resources ending with slash
- Made cacheResults directive honor ‘Cache-Control: max-age’ header
- Added default Marshaller for Option[T] (#81)
- Added TwirlSupport trait
- Improved DebuggingDirectives trait, added logRequestResponse directive
- Improved relaxed header parsing to also accept custom HttpCharsets

6.5.22 Version 0.9.0-RC1 (2012-02-03)

- Upgrades:
 - Scala 2.9.1
 - Akka 1.3
 - spray-json 1.1.0
 - spray-can 0.9.2

- SBT 0.11.2
- Changed dependency on akka-actor from scope ‘compile’ to scope ‘provided’
- Added support for chunked responses, including chunk compression and automatic file chunking
- Added new directives: completeWith, redirect, autoChunk, dynamic, transformUnchunkedResponse, transformChunkedResponse
- Added default Marshallers for Eithers, HttpResults, HttpExceptions, Streams, Futures and Array[Byte]
- Added support for JSON (de)serialization via lift-json
- Added support for template rendering via Scalate
- Added support for LDAP authentication
- Added support for asynchronous and cached authentication
- Added option for relaxed header parsing (issue #68)
- Added DebuggingDirectives trait
- Simplified custom rendering of Rejections to HttpResponses
- Improved LruCache implementation to use com.googlecode.concurrentlinkedhashmap
- Fixed #72 (Different HttpConduits sometimes and erroneously share connections)
- Fixed #59 (Factor out Rejection Conversion)
- Fixed #67 (‘cacheResults’ directive should honor ‘Cache-Control: no-cache’ request header)
- Fixed most occurrences of implicit ambiguities with SprayJsonSupport
- Fixed several bugs in header parsing and rendering
- Extended spray-example-spray-can to show off new streaming features
- Lots of other fixes, additions and improvements

6.5.23 Version 0.8.0 (2011-11-16)

- Upgrades:
 - Scala 2.9.1
 - Akka 1.2
 - spray-json 1.0.1
 - SBT 0.11.1 (many thx to Steffen for contributing the original buildfile)
- Support for ‘application/x-www-form-urlencoded’ as well as ‘multipart/form-data’ form content (many thx to Julien for contributing a lot of the implementation)
- Support for ‘multipart/mixed’ content (un)marshalling
- Support for spray-can as the first non-servlet-container web server
- Completely rewritten spray-client, now based on the new spray-can HttpClient
- Completely rewritten servlet connector layer for improved performance and stability
- Three new example projects:
 - spray-client-example
 - spray-example-simple

- spray-example-spray-can
- Unified SimpleParsers and Unmarshallers into a joint ‘Deserializer’ hierarchy
- Removed ‘optionalContent’ directive (use ‘content(as[Option[T]])’ instead)
- Renamed ‘spray-http’ module to ‘spray-base’
- Renamed cc.spray.marshalling to cc.spray.typeconversion
- Renamed SprayJsonMarshalling to SprayJsonSupport
- Moved encoding/decoding to for cc.spray.encoding
- Simplified case class extraction, now ‘as(T)’ rather than ‘as(instanceOf(T))’
- Simplified SprayTest infrastructure making the ‘DontDetach’ trait obsolete
- Lots of other fixes, additions and improvements

6.5.24 Version 0.7.0 (2011-07-27)

A great number of changes and improvements, most importantly:

- removed dependency on Akka Mist
- upgraded to Scala 2.9.0-1 and Akka 1.1.3
- split into spray-server, spray-client and spray-http
- added support for
 - gzip/deflate encodings
 - authentication/authorization (incl. HTTP Basic Auth)
 - proper server-side caching
 - even better path and parameter matching
 - case class extractions
 - spray-json (un)marshalling
 - Tomcat 6
- closed 27 tickets

6.5.25 Version 0.5.0 (2011-03-31)

first public release

6.6 Credits

spray was influenced by many projects, but the following ones especially stand out:

BlueEyes Another functional web service building toolkit for Scala. *sprays* HTTP model was originally based on the one from BlueEyes and the core ideas of the routing DSL are also heavily inspired by what the BlueEyes team made available.

Netty Netty is an excellent choice if you need to build asynchronous event-driven network applications with Java. *spray* adopted Nettys upstream/downstream pipelining approach for its *spray-io* module.

6.7 References

Since its inception in early 2011 organisations around the globe have been running applications using *spray* successfully in production.

For the ones who'd like to make their using *spray* public we have created this page. If you'd like your organisation to be listed here (along with a backlink) please provide us with the following details:

- Name of your organisation
 - Homepage URL
 - Short statement about how *spray* is used and/or your experiences (so far) with it.
 - Logo (optional)
-

VMware Thanks again for all your hard work on *spray*. The vCloud Integration Manager wouldn't have shipped (at least on time) without it, and we look forward to continuing to use it in our current project as well. :-)

Ebay We are using *spray* for an internal project here at ebay. It is quite small, and it was mostly as an excuse to learn about actor-based programming, but so far we have been very impressed by *spray*'s performance and simplicity and the fantastic support on the user group.

Movio Movio is a big data analytics platform for the cinema industry. We receive all our data through our Restful API, powered by *spray*. We have some complex requirements with routing and *spray*'s routing module provides the flexibility to dynamically manipulate routes. We also use *spray* as the REST backend to our application console which is a single page web app. *spray* is an awesome platform = akka + http + performance + simplicity.

Pongr At Pongr, *spray* powers the restful API used by our iPhone & Android apps and has been in production for a few months now. We use *spray-routing* on top of *spray-can*, wrapped into one fat jar. We are very happy with *spray*.

yardi.com We're using *spray* for a sizeable set of REST web services for our real-estate platform. Some of the services are a REST-based event bus, a notification/email system, a rights-management system (kind of like an authority/security system), and many others. We're getting excellent use out of *spray*, and want to pass on our thanks again for all the work!

Metafor Software We just went live with *spray* in production for the REST backend of our "single-page" web app a couple weeks ago. Very low volume, since we're still in semi-private beta, but I'm very happy with it so far. The immutable request/response model is a huge win for clarity and composability.

TimeTrade TimeTrade has used *spray* to build a number of core services within our online appointment scheduling system, and we plan to use it for even more. It also forms the basis of our stress testing framework. We've found it to be of excellent quality, and it represents the immutable model of HTTP in a very elegant way. The team behind it is very responsive and helpful.

nokta We are running *spray* in front of our ad-server at nokta.com, hosting ads for several high-volume websites (millions of visitors everyday). 3 servers are deployed with custom tarball with dependencies copied in it, and are running with custom init-scripts. We've started with play2 but switched to *spray* in the first week of production and never looked back. Besides technical merits the active community is what makes this project awesome.

Tinga We decided to use *spray* + Akka instead of using Play + Akka, as we'd like to have a clear separation between the API- and presentation layers. We are still in the earlier stage of building our infrastructure but are impressed by the community and the *spray* team's responsiveness.

telfish.com We are using *spray-can*, *spray-server* and *spray-json* for our customer-facing query API for more than 2 years now and are very happy with it. Our API code is robust, fast and easy to maintain.

Animatron LLC *spray* powers our internal REST services used to communicate components of our rendering engines and back-end servers. We faced some challenges but got excellent support from the development team. Keep the good work!

Zuberance We developed a new product completely on top of Scala/Akka/spray and are just finishing up beta.

system insights We converted from Jersey, which took about 2 weeks. The testability of *spray* is great – and runs fast.

Postarte.com Our web application is running on *spray-can*. We have a couple of static HTML web pages, images etc. served by simple spray routing Directive, but the most of the content is loaded via AJAX/REST/JSON. We use a Postgresql database with Prequel (enhanced a little bit).

TXOdds We're using spray.io for a REST interface for our fixtures management toolkit. Our experiences so far have been very promising and we are considering using it for the development of our new real-time single page web application. It is very easy to deploy and integrates very well with our existing scala codebase (akka etc.)

6.8 Sponsors

The following companies and organisations have kindly agreed to supply the *spray* development team with free open-source licenses of their excellent development tools:

Jetbrains We use IntelliJ IDEA Ultimate as the Scala IDE of our choice.

YourKit YourKit is kindly supporting open source projects with its full-featured Java Profiler. YourKit, LLC is the creator of innovative and intelligent tools for profiling Java and .NET applications. Take a look at YourKit's leading software products: [YourKit Java Profiler](#) and [YourKit .NET Profiler](#).

6.9 License

This software is licensed under the Apache 2 license, quoted below.

Copyright © 2011–2015 the spray project <<http://spray.io>>

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

[<http://www.apache.org/licenses/LICENSE-2.0>]

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

6.10 Community Projects

Over the course of the last years the active and vibrant *spray* community has created numerous additions and extensions to *spray* as well as examples and tutorials that, for various reasons, haven't always found their way into the main codebase.

As *spray* is now transitioning into [Akka](#) (see also [this post on the typesafe blog](#)) we have moved the list of community projects that formerly resided here into a dedicated section of the Akka “Community Projects” page at this address:

If you would like to add or update a project entry there please do so by clicking the “Edit” button on this page:
<https://github.com/akka/akka.github.com/blob/master/community/index.md>

7.1 spray on the Raspberry Pi

7.1.1 Introduction

As a spray newbie and general computer enthusiast I thought of playing around with [spray.io](#) and my Raspberry Pi (RPI) at home. In this blog post I want to show a small example of how to get started with [spray.io](#) on the RPI.

The [Raspberry Pi](#) is a credit-card sized computer with a huge number of use cases. It can be connected to a TV and equipped with [Raspbmc](#) (a special [XBMC](#)-Distribution for the RPI) and then be used as a Home Cinema PC (HTPC). A photo-enthusiast enhanced his camera and built a RPI-based mini-computer into a battery grip and called this [Camera Pi](#). Just recently I discovered that it is also possible to build a [custom GoogleTV](#) with the RPI. A very common use case is as a server in your network e.g. file server or web server.

So, as you can see, there are tons of things you can do with this little computer. However, due to its limited system resources (memory, CPU) it is sometimes considered too heavy-weight for JVM-based applications. A perfect challenge for me to see how well Scala, Akka and spray can scale *down* rather than up.

7.1.2 JVMs on the Raspberry Pi

Because of the [ARM](#)-based architecture of the RPI and the recently added support for the ARM architecture in popular JDKs, there are some things to consider in choosing the JDK. This sections gives an overview about the possible options.

The easiest way is to use the JDK bundled with your distribution. You can install it via the known package manager (e.g. `apt`). If you choose this way you end up with the *default* OpenJDK. This gives you a working JVM, which is a little bit slower because it doesn't have support for the specific instruction set of the ARM-architecture. The OpenJDK will fall back to a VM called [ZeroVM](#). This is an interpreter-only VM which is very portable but runs a little bit slower.

Additionally, there is the official version of the JDK 8 from Oracle, which is currently in early-access status. This version has dedicated support for the instruction set of the ARM and therefore is faster than the OpenJDK. A requirement for this JDK is the support of the [hardfp-api](#) in your OS, because Oracle's JDK depends on this. [Raspbian](#), a Debian-based distribution is an operating system with support for [hardfp-api](#) and therefore Oracle's JDK.

Furthermore, if you want to experiment, you can choose a completely different JDK. A candidate would be e.g. [Avian](#).

For this demo I will use Raspbian and the Oracle JDK. I explain the steps in the next paragraphs.

First, you need a working Linux-Distribution on your RPI. You can find the Raspian-images on the RPI [downloads](#)-page. I'll use the *Raspbian "wheezy"* image with support for [hardfp-api](#).

If your Raspbian installation is in place we can move on further to install the JDK. You can download the JDK from [here](#). There is an [installguide](#) which describes how to install the JDK on the RPI. This is basically a tar-file which you can then simply un-tar to a folder of your choice. For this demo it will be ok to just un-tar it in the home-folder of the user *pi*:

```
tar -xf jdk-8-ea-b36e-linux-arm-hflt-29_nov_2012.tar
```

This will create the folder `jdk1.8.0` in the current dir. Now, you can type `./jdk1.8.0/bin/java -version` to check that Java will run correctly and to see some version information.

7.1.3 spray-can on the Raspberry Pi

So, after our RPI runs a Linux and a JVM, I want to show you how to get started with [spray.io](#) on your RPI.

For this I created a [customized version](#) of the `spray-template` project from Github. You can clone this customized version with the command:

```
git clone https://github.com/matsluni/spray-template.git
```

Modifications on the `spray-template-project`

This customized version is based on the `spray-template-project` (1.2-M8). This version already uses Akka 2.2 including the new IO-module developed together with the `spray.io` team and is completely actor-based. At first I give it a telling name: `spray-can-rpi` (see `build.sbt`). Furthermore, the customized version includes the [assembly plugin](#) for `sbt` (see `build.sbt`). This is necessary because it is not possible to build the project on the RPI itself. Therefore we build it on our local system and transfer the complete JAR over to our RPI. This packaging is done by the assembly plugin.

There are further changes, which include a slightly modified Akka dispatcher config (see `application.conf`) to reduce the amount of threads to start by the akka-runtime. Otherwise Akka would start up to 64 threads which would kill the JVM of the RPI. Another minor change is to let `spray-can` listen to all interfaces of the RPI (see `Boot.scala`). This makes it possible to reach the demo-application from other hosts in the network including our local system.

Running the modified project on the RPI

After you cloned the git repository you can start `sbt` in this scala project. With the assembly-plugin it is very easy to package the JAR which contains all the dependencies we need for `spray.io`. From within `sbt` you can just type `assembly` to start the packaging process. If everything worked out you should have a JAR file in `./target/scala-2.10/` called `spray-can-rpi-assembly-0.1.jar`. This is the JAR file containing `spray.io` and everything it needs to run. This JAR can now transferred to the RPI. If you are on a Unix-like system you can copy it with `scp` or if you are on Windows you can use [WinSCP](#).

Now the time has come to start the `spray` app on the RPI. This is easy. If you transferred the JAR from your system to the home folder of the user *pi* where you also downloaded the JDK you can just enter:

```
~/jdk1.8.0/bin/java -Xss1M -Xms64M -jar spray-can-rpi-assembly-0.1.jar
```

This is a standard Java JAR start with modifications for the stacksize of 1 MB (`-Xss1M`) and the start heap size of 64 MB (`-Xms64M`).

If everything worked fine you should see something like:

```
[INFO] [05/18/2013 08:28:09.287] [on-spray-can-akka.actor.default-dispatcher-3] [akka://on-spray-can,
```

Now you can open your browser and direct it to the ip-address of your RPI and the correct port (the port is shown in the log output of `spray-can`) and you should see the welcome message of `spray-routing` and `spray-can`. This shows that `spray-can` now runs on the RPI and is happily answering your requests.

7.1.4 A little benchmark

After getting spray-can to work on the RPI I did some benchmarking to get an understanding of how much is possible with this setup. I ran my tests on a Model B with 256mb ram over LAN on a plain Raspbian installation. The full hardware-spec of the RPI can be [inspected here](#). For the actual benchmark I used a tool called `wrk` to run some requests against the RPI. I used it in the following way and with this parameters:

```
./wrk -c 30 -t 20 -d20s http://raspberrypi:8080/
```

This command will use 30 open connections with 20 threads and runs a 20 second test against the RPI. In this test I got around 400 requests/sec which is quite nice and shows that spray.io on the RPI is a really useful setup upon which one can implement real applications.

7.1.5 Conclusion

The goal of this blog post was to have a JDK-based HTTP-Server running on the RPI. I can say with spray.io this is possible. The customized version can be seen as a first step to build your own applications running on the RPI.

Furthermore, with this post I wanted to show some more things:

1. How easy it is to deploy spray.io on an embedded-like system like the RPI.
2. Scala can keep the promise to be a scalable language and platform, especially with the results from the little benchmark shown before.
3. Make myself more familiar with spray.io, Scala and the whole ecosystem to be able to build larger applications in the future.

For some feedback or other questions you can reach me via my twitter account [@matsluni](#).

Finally, I want to thank Mathias and Johannes from the spray.io team for this great piece of software, to make the guest post happen and also the support they gave me during the creation of this post.

7.2 spray 1.0-M8 / 1.1-M8 / 1.2-M8 released

Dear sprayers,

we – the *spray* team – are happy and proud to finally be able to announce the release of the much too long awaited **M8 milestone** of *spray* 1.0 / 1.1 / 1.2.

A lot has happened in and around the *spray* codebase since we released M7 in December, with the most notable change probably being the move onto the new Akka I/O layer that was jointly developed with the Akka team in the last months in order to provide an even better “Scala-all-the-way-through” actor-based I/O foundation, not just for HTTP. This new I/O layer will be officially released with Akka 2.2 (the first RC of which is already at this time), but for everyone who is not yet willing or able to upgrade from Akka 2.0 or 2.1 we provide a one-to-one backport of the new I/O abstraction with *spray-io* 1.0/1.1-M8.

But also on top of the raw I/O layer this milestone brings a lot of good stuff for your *spray* applications, for example:

- A completely new and even faster implementation of our *spray-can* HTTP parser. The work in this area was the basis for *spray*'s success in the latest round of the techempower benchmark, so upgrading to M8 should give your HTTP stack a nice performance boost.
- Fully rewritten *HTTP Client APIs*, which improve upon the experiences from earlier releases and give you the most powerful and flexible client-side HTTP abstractions available with *spray* to date.
- Much improved HTTP model in *spray-http*, with things like complete serializability, a very fast and fully RFC 3986-compliant URI parser as well as better and more efficient rendering abstractions.

The full list of changes, fixes and additions is too long for us to properly compile it, please check out the [Changelog](#) for more details.

Since this milestone is a triple release, here are the Scala and Akka versions that are targeted by the different artifacts (see also the [Current Versions](#) chapter):

<i>spray</i> version	Scala version	Akka version
1.0-M8.1	Scala 2.9.x	Akka 2.0.x
1.1-M8	Scala 2.10.x	Akka 2.1.x
1.2-M8	Scala 2.10.x	Akka 2.2.x

The development statistics for 1.1-M8 (since 1.1-M7):

- 334 commits
- 1.081 files changed
- 21.628 insertions
- 15.363 deletions
- 18 committers

As the name implies this release is still only a milestone until the 1.0/1.1/1.2 finals, but it will be the last one. In the coming weeks we are planning on burning through the [open tickets](#) and provide you with the first release candidate as soon as we can.

Until then we are, as always, very much looking forward to your feedback!

Happy spraying!

Cheers,
Mathias

7.2.1 Update 2013-06-11

We discovered a regression in *spray-routing* 1.0-M8 causing an NPE on first access. This has been fixed with the superseding release of **1.0-M8.1**.

7.3 Benchmarking spray

A few days ago the folks at [techempower](#) published [round 5](#) of their well-received current series of web framework benchmarks, the first one in which *spray* participates. The *techempower* benchmark consists of a number of different test scenarios exercising various parts of a web framework/stack, only one of which we have supplied a *spray*-based implementation for: the “JSON serialization” test. The other parts of this benchmark target framework layers (like database access), which *spray* intentionally doesn’t provide.

Here are the published results of the JSON test of [round 5](#) presented in an alternative visualization (but showing the exact same data):

The test was run between two identical machines connected via a GB-Ethernet link, a client machine generating HTTP requests with *wrk* as the load generator, and a server machine running the respective “benchmarkee”. In order to provide an indication of how performance varies with the underlying hardware platform all tests are run twice, once between two EC2 “m1.large” instances and once between two dedicated i7-2600K workstations.

7.3.1 Analysis

In the graph above we compare the performance results on dedicated hardware with the ones on the EC2 machines. We would expect a strong correlation between the two, with most data points assembled around the trendline. The “bechmarkees” that are far off the trendline either don’t scale up or down as well as the “pack” or suffer from some configuration issue on their “weak” side (e.g. `cpoll_cppsp` and `onion` on `i7` or `gemini/servlet` and `spark` on the EC2). Either way, some investigation as to the cause of the problem might be advised.

In addition to plotting the average requests/sec numbers reported by `wrk` at the end of a 30 second run we have included an alternative projection of the request count, based on the average request latencies reported by `wrk` (e.g. 1 ms avg. latency across 64 connections should result in about 64K avg. req/s). Ideally these projected results should roughly match the actually reported ones (bar any rounding issues).

However, as you can see in the chart the two results differ substantially for some benchmarkees. To us this is an indication that something was not quite right during the respective test run. Maybe the client running `wrk` experienced some other load which affected its ability to either generate requests or measure latency properly. Or we are seeing the results of `wrk`’s somewhat “unorthodox” request latency sampling implementation. Either way, our confidence regarding the validity of the avg. request counts and the latency data would be higher if the two results were more closely aligned.

7.3.2 Take-Aways

The special value of this benchmark stems from the sheer number of different frameworks/libraries/toolsets that the techempower team has managed to include. Round 5 provides results for a very heterogeneous group of close to 70 (!) benchmarkees written in 17 different languages. As such it gives a good indication of the rough performance characteristics that can be expected from the different solutions. For example, would you have expected a Ruby on Rails application to run about 10-20 times slower than a good JVM-based alternative? Most people would have assumed a performance difference but the actual magnitude thereof might come as a surprise and is certainly interesting, not only for someone currently facing a technology decision.

For us as authors of an HTTP stack we look to such benchmarks from a slightly different angle. The main question for us is: How does our solution perform compared to alternatives *on the same platform*? What can we learn from them? Where do we still have potential for optimization that we appear to have left on the table? What effect on performance do the various architecture decisions have that one has to make when writing a library like *spray*?

As you can see from the graph above we can be quite satisfied with *spray*’s performance in this particular benchmark. It outperforms all other JVM-based HTTP stacks on the EC2 and, when looking at throughput projected from the latency data, even on dedicated hardware.

This shows us that our work on optimizing *spray*’s HTTP implementation is paying off. The version used in this benchmark is a recent *spray* 1.1 nightly build, which includes most (but not all) performance optimizations planned for the coming 1.0/1.1/1.2 triple release (1.0 for Akka 2.0, 1.1 for Akka 2.1 and 1.2 for Akka 2.2).

But, does this benchmark prove that *spray* is the fastest HTTP stack on the JVM?

Unfortunately it doesn’t. This one test exercises way to small a percentage of all the logic of the various HTTP implementations in order to be able to properly rank them. It gives an indication, but hardly more.

What’s missing?

7.3.3 Benchmarking Wish-List

Let’s look more closely at what the “JSON serialization test” of the techempower benchmark actually exercises. The client creates between 8 and 256 long-lived concurrent TCP connections to the server and fires as many test requests as possible across these connections. Each request hits the server’s NIC, bubbles up through the Linux kernel’s network stack, gets picked up by the benchmarkees IO abstraction and is passed on to the HTTP layer (where it is parsed and

maybe routed) before actually being handled by the “application logic”. In the case of this benchmark the application merely creates a small JSON object, puts it into an HTTP response and sends it back down the stack, where it passes all layers again in the opposite direction.

As such this benchmark tests how well the benchmarkee:

- interacts with the kernel with regard to “pulling out” the raw data arriving at a socket
- manages internal communication between its inner layers (e.g. IO <-> HTTP <-> Application)
- parses HTTP requests and renders HTTP responses
- serializes small JSON objects

It does all this using small requests with a fixed set of HTTP headers over a rather small number of long-lived connections. Also, it does it all at once without giving us a clue as to the potential strengths and weaknesses of the individual parts of the stack.

If we wanted to learn something deeper about how *spray* performs compared to its JVM-based competitors and where its strengths and weaknesses lie we’d have to setup a whole range of benchmarks that measure:

- raw IO performance:
1 to say 50K long-lived concurrent connections, minimal request and response sizes
- connection setup overhead:
varying number of per-request connections, minimal request and response sizes
- HTTP request parser performance:
varying number of request headers and header value sizes, varying entity sizes
- HTTP response renderer performance:
varying number of response headers and header value sizes, varying entity sizes
- HTTP chunking performance:
chunked requests and responses with varying number and size of message chunks
- HTTP pipelining performance:
varying number of request batch sizes
- SSL performance:
1 to say 50K long-lived connections, minimal request and response sizes
- WebSocket performance
- System- and JVM-level metrics (CPU utilization, GC-Activity, etc.)

If we had a benchmark suite producing numbers like these we’d feel much more comfortable in establishing a proper performance-based ranking of *spray* and its alternatives. And wouldn’t it be great if there was something like a “continuous benchmarking” infrastructure, that would automatically produce these benchmark results upon a simple `git push` into its repository?

Oh well... I guess our ever-growing todo-list just received one more item marked *important...* :)

Cheers,
Mathias

7.4 The Magnet Pattern

In this post I'd like to shine some light on one particular implementation technique that is used extensively under the hood of *sprays routing DSL* and which has turned out as a very valuable tool in our “pattern toolbox”. It solves certain kinds of problems in Scala API design and DSL development, especially (but not only) with regard to overloaded methods. We call it the *magnet pattern* and, hopefully, by the time you've finished reading this post this name will make sense to you.

Before diving into the details I'd like to make clear that we are certainly not the first ones to use this pattern and by no means claim its original “inventorship”. However, we find it interesting and valuable enough to give it a proper name and dedicate a somewhat lengthy blog post to its description. (In the process we'll touch on quite a few interesting edge cases of the current Scala language implementation, so I hope you'll learn something even if you consider yourself a somewhat seasoned Scala developer.)

(Update 2012-12-17: The following two paragraphs were added.)

There have been some questions as to how the technique presented here differs from type classes, which are an increasingly common and widely used mechanism in Scala (see [this paper](#) for some good background material). The short answer is: There is no real difference. Rather, we see magnets as a specific application of type classes, which are a broader, more general concept. Among other things type classes can be used for type-level computation or advanced generic programming as in *shapeless*. Most people probably first come to know them as a solution for associating logic with types in a way that allows for very loose coupling and *retroactive extension*. For example, *spray-json* uses type classes to attach JSON (de)serialization logic to types “from the outside”.

As shown in this post type classes can also be used to solve certain issues with regard to method overloading in Scala. Our intent is not to “rebrand” type classes, but rather to describe a way of using them for a specific purpose. In the case of method overloading we see value in labelling the combination of purpose and implementation technique with a dedicated name. With some kind of naming convention it's easier for someone reading a piece of code to derive intent and more quickly understand the purpose of a particular construct. This, describing a particular use case for type classes along with a proposal for naming the things involved, is what this post is all about.

7.4.1 The Problem

There are only two hard things in Computer Science: cache invalidation, naming things and off-by-1 errors.

—Phil Karlton (slightly adapted)

When you design a Scala API, be it in the context of a DSL or not, especially the “naming things” part can be one of the main challenges. Ideally the names you pick nicely capture the respective concepts of the domain you are modelling. If there are several ways the *same* domain concept can be applied, as is often the case, the representations of these alternatives in your API should also receive the same name. In your code this usually manifests itself as what is commonly called “method overloading”.

For example, in the domain of HTTP servers there is the concept of “finalizing” the processing of an HTTP request by sending some kind of response to the client. In *sprays routing DSL* this concept goes by the name “complete”. Thereby several ways of request “completion” are supported. Currently you can “complete” a request with either one of these seven sets of things:

- just a status code
- a custom object (that is to be marshalled as the response entity)
- a status code and a custom object
- a status code, a list of HTTP response headers and a custom object
- an `HttpResponse` object

- a `Future[HttpResponse]`
- a `Future[StatusCode]`

Since all of these are just different ways of achieving the same thing, namely “completing” the request, they should all be accessible via the same name in our DSL, namely `complete`. We do not want to clutter our DSL with seven different names like `completeWithStatus`, `completeWithStatusAndObject` and so on. Luckily, Scala allows for method overloading, so we could try to model our seven different completion alternatives with these method overloads:

```
def complete(status: StatusCode)
def complete[T :Marshaller](obj: T)
def complete[T :Marshaller](status: StatusCode, obj: T)
def complete[T :Marshaller](status: StatusCode, headers: List[HttpHeader], obj: T)
def complete(response: HttpResponse)
def complete(future: Future[HttpResponse])
def complete(future: Future[StatusCode])
```

Unfortunately though, method overloading in Scala comes with (at least) the following problems and inconveniences:

1. “Collisions” caused by type erasure
2. No lifting into a function (of all overloads at the same time)
3. Unavailability in package objects (before Scala 2.10)
4. Code duplication in case of many similar overloads
5. Limitations with regard to parameter defaults
6. Limitations with regard to type inference on arguments

The magnet pattern can solve all but the last two of these issues and we are going to discuss them in detail in a moment, but first let’s understand what the magnet pattern actually *is*.

7.4.2 Method Overloading Reloaded

The magnet pattern is an alternative approach to method overloading. Rather than defining several identically named methods with different parameter lists you define only one method with only one parameter. This parameter is called the *magnet*. Its type is the *magnet type*, a dedicated type constructed purely as the target of a number of implicit conversions defined in the magnets companion object, which are called the *magnet branches* and which model the various “overloads”.

Show me the Code

Let’s go back to our “complete” example from above to understand what this means. However, in order to focus on the key elements we are going to look only at the following three slightly adapted overloads:

```
def complete[T :Marshaller](status: StatusCode, obj: T): Unit
def complete(future: Future[HttpResponse]): Int
def complete(future: Future[StatusCode]): Int
```

For the sake of the example the return types are different from the actual *spray-routing* implementation but they serve well for illustrating the concepts. In order to model these three `complete` overloads with the magnet pattern we replace them with this single method definition:

```
def complete(magnet: CompletionMagnet): magnet.Result = magnet()
```

The `CompletionMagnet` is the following simple trait:


```
sealed trait CompletionMagnet {
  type Result
  def apply(): Result
}
```

The magnet branches are where the actual logic lives. They represent the different overload implementations we had before and are defined as implicit conversions to `CompletionMagnet` instances in the companion object:

```
object CompletionMagnet {
  implicit def fromStatusObject[T :Marshaller](tuple: (StatusCode, T)) =
    new CompletionMagnet {
      type Result = Unit
      def apply(): Result = ... // implementation using (StatusCode, T) tuple
    }
  implicit def fromHttpResponseFuture(future: Future[HttpResponse]) =
    new CompletionMagnet {
      type Result = Int
      def apply(): Result = ... // implementation using future
    }
  implicit def fromStatusCodeFuture(future: Future[StatusCode]) =
    new CompletionMagnet {
      type Result = Int
      def apply(): Result = ... // implementation using future
    }
}
```

That’s all we need in order to model method overloading with a magnet. All of the following calls will execute the logic in the respective magnet branch, just as if we had defined them with “regular” overloads:

```
complete(StatusCodes.OK, "All fine") // returns Unit
complete(someHttpResponseFuture) // returns Int
complete(someStatusCodeFuture) // returns Int
```

How does it work?

The magnet parameter on the single `complete` method we defined serves only as the “center of gravity” towards which the different magnet branches define implicit conversions. If you call `complete` with an argument that is not a `CompletionMagnet` instance itself, as is usually the case, the compiler looks for an implicit conversion that it can use to turn the argument you specified into an `CompletionMagnet`, so that your call becomes legal. Since implicit conversions defined in the companion object of any involved type are automatically in scope the compiler can “see” and select the matching magnet branch (if there is one) and we are set.

What is interesting is that this approach also works for “overloads” with more than one parameter just as well as different return types. If you call `complete` with several arguments the compiler looks for an implicit conversion that can produce a magnet instance from a *tuple* wrapping all arguments. This way overloads with up to 22 parameters (the maximum arity of tuples in scala) can be supported.

If the overloads differ in their return types, as in our example above, we can resort to *dependent method types* to model them. Dependent method types are available in Scala 2.9 as an experimental feature and thus need to be *explicitly enabled*. Even though they can be used for building powerful constructs there is nothing particularly dangerous or magical about them, so as of Scala 2.10 dependent method types are always enabled and do not even require a [SIP-18-style language import](#). What they allow you to do is to specify the return type of a method as “a function of” the method parameters, which is exactly what we are doing in the example above.

Implementation Notes

- If all overloads have the same return type there is no need for a `type` member on the magnet type. The central method with the magnet parameter (`def complete` in the example above) can then simply have the return type directly in its signature.
- If the magnet branch implementations share common logic you can of course factor it out, e.g. into private helpers on the magnet companion object. Another option would be to pull it up into the central method itself (`def complete` in the example above) and have the magnet only contribute the parts that differ between the overloads.
- Since it'll never be called from the outside the name of the abstract method in the magnet trait doesn't really matter. You might even want to mark it `private[module_name]`. Also the names of the implicit conversions on the magnet companion don't really matter. As you can see above we call them `from<source-type>` by convention.

7.4.3 Benefits

So, what does this alternative approach to method overloading give us? As it turns out it solves most of the problems with method overloading that we listed before. Of course, it also comes with a couple of drawbacks of its own, but first let's look into the advantages a bit deeper.

No Erasure-induced Collisions

Collisions caused by type-erasure probably pose the most severe problem of “traditional” method overloading on the JVM, since there is no clean work-around. It actually prevents us from implementing our `complete` overloads in the usual fashion, as can be seen by the following error the Scala compiler produces when we try to:

```
[error] ...: double definition:
[error] method complete:(future: scala.concurrent.Future[spray.http.StatusCode])Int and
[error] method complete:(future: scala.concurrent.Future[spray.http.HttpResponse])Int
[error] have same type after erasure: (future: concurrent.Future)Int
[error]   def complete(future: Future[StatusCode]) = { ...
[error]     ^
[error] one error found
```

The compiler is telling us that the last two of our overloads are a “double definition” because of type erasure. In order to understand what's going on we have to take a quick look at how methods are represented by the JVM. The JVM supports generics through [type erasure](#) (rather than [type reification](#) as Microsoft's CLR does, check out [this article](#) for more info on the difference). This means that all parameter types on generic types (in Java speak) are erased and non-existent on the JVM level. To the JVM our two overloads:

```
def complete(future: Future[HttpResponse]): Unit
def complete(future: Future[StatusCode]): Unit
```

both look like this:

```
def complete(future: Future): Unit
```

Since the compiler cannot produce two different implementation for the same method it has to give up.

This erasure-induced limitation to method overloading is not specific to Scala. Java and other JVM-based languages suffer from it as well. Theoretically we could hack our way around it by introducing “fake” return types for the colliding methods (since the return type is part of the method signature and therefore sufficient to discriminate between overloads), but in Scala we don't have to. With overloading via magnets we can remove the need to supply two

different implementations for the same (as seen by the JVM) method and nicely overcome the “collision problem” without having to compromise our API on the type level.

Full Function-Lifting

Scala supports a nice and easy notation for *lifting* a method into a function. Just follow the method name (without arguments) with a `_` as shown in this example:

```
scala> def twice(i: Int) = (i * 2).toString
twice: (i: Int)java.lang.String

scala> twice _
res0: Int => java.lang.String = <function1>
```

Now, if we overload the method like this:

```
def twice(i: Int) = (i * 2).toString
def twice(d: Double) = (d * 2).toString
```

it’d be nice if we could still simply say `twice _` and somehow lift both overloads at once, so that later on we could call the lifted function with either an `Int` or a `Double`. Unfortunately this is not supported, you have to decide at the “lifting point”, which overload to lift and you can only lift one.

With magnets this lifting of all overloads at once is no problem. In this case the type of `twice _` is `TwiceMagnet => String` and the “overloadedness” is retained. Only at the point where the lifted function is actually applied do you have to decide, which overload to choose. Just as in the unlifted case the compiler will supply the required implicit conversions at the call site.

Unfortunately this type of lifting only works when all overloads have the same return type and thus no dependent method types are required. For example, if we try to lift our `complete` overload from above with `complete _` the compiler will produce the following error:

```
error: method with dependent type (magnet: CompletionMagnet)magnet.Result
       cannot be converted to function value
       complete _
         ^
```

Package Object Support

Due to a long-standing Scala bug that was just recently fixed method overloading in package objects is not supported with any Scala version before 2.10. If you are searching for a solution for Scala 2.9 or earlier magnets might present a nice solution.

DRYness for many similar Overloads

Sometimes DSLs can require the definition of a larger number of very similar method overloads, which reduces DRYness and generally feels ugly. For example in *spray* 0.9 the `parameters` directive, which allows you to define the extraction of one or more request query parameters, was defined like this:

```
def parameters[A] (a: PM[A]): SprayRoute1[A] =
  parameter(a)

def parameters[A, B] (a: PM[A], b: PM[B]): SprayRoute2[A, B] =
  parameter(a) & parameter(b)

def parameters[A, B, C] (a: PM[A], b: PM[B], c: PM[C]): SprayRoute3[A, B, C] =
```

```

parameters(a, b) & parameter(c)
def parameters[A, B, C, D](a: PM[A], b: PM[B], c: PM[C], d: PM[D]): SprayRoute4[A, B, C, D] =
  parameters(a, b, c) & parameter(d)
...

```

Ideally, *spray* would have supported an arbitrary number of parameters like this but due to the duplication required we only defined nine. After we switched the implementation of the `parameters` directive to a combination of magnets and *shapeless*' `HLists` we can now support up to 22 parameters without any duplication.

The details of how exactly this is implemented in *spray-routing* are beyond the scope of this article, but in essence the solution looks like this: We define a single magnet branch for all tuples at once by making use of *shapeless*' support for automatically converting tuples to `HLists`. Since *shapeless* allows us to easily fold over `HLists` we can reduce the problem to a binary poly-function that specifies how two parameters are to be combined. This is pretty much as DRY as it gets.

Removal of implicit Parameters

sprays routing DSL tries to provide a way for API definition that is both highly concise and highly readable. As such it relies heavily on the one method in Scala that beats all others with regard to brevity: `apply`. If an object has only one clearly defined behavior or if there is a central one, which clearly outrivals all others in terms of importance, modeling this behavior as an `apply` method is the natural choice. Unfortunately, implicit parameter lists and `apply` occupy "the same namespace", which can lead to collisions. Since idiomatic Scala often times relies quite heavily on implicits (for instance when working with type classes) this can present a problem.

For example, consider this snippet, which loosely resembles what we have in *spray*:

```

val post: Route => Route = ...

```

Here `post` defines some logic that modifies a `Route`. For this example it doesn't matter how `Route` is actually defined. All we care about is that we can use `post` to wrap a `Route` thereby producing another `Route`:

```

val route: Route =
  post {
    ... // some inner route
  }

```

The `post` modifier is only one of many modifiers that can be freely combined. Some of them are not modelled as *vals* but rather as *defs*, since they take some parameters. For example the `hosts` modifier filters requests according to some host name:

```

def host(hostName: String): Route => Route = ...

```

You could combine it with `post` like this:

```

val route: Route =
  host("spray.io") {
    post {
      ... // some inner route
    }
  }

```

The problem arises if a modifier method requires an implicit parameter list, for example if we wanted to flexibilize the `host` modifier to take any parameter that can be implicitly converted to a `String`:

```

def host[T](obj: T)(implicit ev: T => String): Route => Route = ...

```

At first glance this change doesn't look like it would hurt us but in fact it breaks our modifier composition! When we now write:

```
host("spray.io") {
  ... // some Route expression
}
```

the compiler will interpret our inner route expression not as an argument to the `Route => Route` function produced by `host`, but rather as an explicitly specified value for the implicit parameter. Clearly this is not what we want. We could fix this with an extra pair of parentheses like this:

```
(host("spray.io")) {
  ... // some Route expression
}
```

but as DSL designers this must leave us unsatisfied.

Luckily, the magnet pattern provides a nice solution. It allows us to push the implicit requirement “one level down”, so the combinability of our `host` modifier is fully restored:

```
def host(magnet: HostMagnet) = magnet()

sealed trait HostMagnet {
  def apply(): Route => Route
}

object HostMagnet {
  implicit def fromObj[T](obj: T)(implicit ev: T => String) =
    new HostMagnet {
      def apply() = ...
    }
}
```

Modelled in this way the implicit parameter list on the `host` method is removed, which prevents it from colliding with the `apply` method on the returned object (the `Route => Route` function in our case).

This example shows that the magnet pattern has certain applications outside of providing a mere alternative to method overloading. Because *sprays* routing DSL relies so heavily on functions and thereby `apply` calls, “removing” implicit parameter lists on DSL elements is crucial and the magnet pattern turns out to be a great asset in this regard.

7.4.4 Drawbacks

Of course, where there is light there must also be some darkness. The magnet pattern certainly isn't an exception in that regard. So let's look at what we have to pay in order to reap the benefits discussed above.

Verbosity

You probably already noticed that magnets come with a certain amount of extra verbosity. Having to introduce a dedicated type with companion object and anonymous classes for every magnet branch is no doubt a disadvantage. Apart from the additional lines this overhead increases code complexity, especially for other people reading your code. Someone not familiar with the pattern might scratch his head about why you chose to jump through all these extra hoops instead of simply resorting to “traditional” method overloading.

API “Obfuscation”

Somewhat related to the previous point, the magnet pattern might be perceived as actually “obfuscating” your APIs. While with “traditional” method overloading the API of a class or trait can be easily grasped from the method signatures the introduction of magnets pushes important parts of the API down into the “branches” on the magnet companion, where they are scattered across several implicit conversions. Also, since parameter lists with several elements are grouped together as tuples, where the individual members have no explicit name, important information with regard to the semantics of the individual parameters might be lost.

Another aspect of this is that the tools you might be relying on for inspecting a method signature at the call-site (like the “Parameter Info” view of your IDE) will not work anymore once you “magnetized” the method.

No named Parameters

Since parameters are not actually defined on the method itself you cannot address them by name, i.e. this doesn’t work (coming back to our example from the beginning):

```
complete(status = 200, obj = "All good")
```

Limited by-name Parameters

If you have several parameters on an overload and some of them are call-by-name you cannot transform that overload into a magnet branch and uphold the by-name property. E.g. this method cannot be directly “magnetized”:

```
def bar(a: Int, b: => String)
```

If you have only one single call-by-name parameter things might work as expected, depending on how exactly you’d like the parameter to be used, but there is a catch to watch out for!

Suppose we have this “traditional” definition:

```
def foo(s: => String): Unit = {  
  println(s)  
  println(s)  
}
```

We *can* “magnetize” it like this:

```
def foo(magnet: FooMagnet): Unit = magnet()  
  
sealed trait FooMagnet {  
  def apply()  
}  
object FooMagnet {  
  implicit def fromString(s: => String) =  
    new FooMagnet {  
      def apply(): Unit = {  
        println(s)  
        println(s)  
      }  
    }  
}
```

This compiles and, when we look at the following example, appears to be doing the same thing as its “unmagnetized” counterpart:

```
def string() = {
  print ("NOT-")
  "BAD"
}

foo(string())
```

This ends up printing “NOT-BAD” twice, as expected. Now if we move the body of the `string()` method directly into the argument expression of `foo` like this:

```
foo {
  print ("NOT-")
  "BAD"
}
```

you might be surprised to see the output being “NOT-BAD” and “BAD” instead. The `print("NOT-")` line is not actually executed during the second evaluation of the by-name parameter of the implicit `fromString`. How come?

The reason is discussed in Scala issue [SI-3237](#). In essence: The compiler has several options of how exactly to insert the implicit conversion and chooses the “wrong” one. Instead of generating this:

```
foo {
  FooMagnet.fromString {
    print ("NOT-")
    "BAD"
  }
}
```

it generates this:

```
foo {
  print ("NOT-")
  FooMagnet.fromString {
    "BAD"
  }
}
```

which is enough to make the types line up, but isn’t quite what we want. So, while “magnetizing” single by-name parameters works as expected if the argument is a single expression, the behavior of the magnetized version differs from the unmagnetized one if the argument consists of a block with several statements. Definitely something to be aware of!

Param List required

(2012-12-17: Updated after feedback with corrections, see post comments below)

The magnet pattern relies on the ability of the compiler to select one of potentially several magnet branches in order to make an otherwise illegal call work (type-wise). In order for this logic to actually kick in we need to “provoke” an initial type-mismatch that the compiler can overcome with an implicit conversion. This requires that we actually have a parameter list to work with. Overloads without a parameter list, like:

```
def foo: String
```

cannot be “magnetized”. Unfortunately this also renders the magnet pattern ineffective for removing implicit parameter lists that are not preceded by a non-implicit parameter list, something that we have to work around in several places in *spray-routing*.

Note that this does not mean that the parameter list cannot be empty. An overload like:

```
def foo(): String
```

can be turned into the following magnet branch without any problem:

```
implicit def fromUnit(u: Unit): FooMagnet = ...
```

No default Parameters

It's not hard to picture situations where combining method overloading with default parameters leads to apparent ambiguities that can quite significantly reduce the readability of your code. This for example:

```
def foo(a: Int, b: String = "") = ...
def foo(b: Int) = ...
```

is perfectly legal and compiles fine. However, the default parameter on the first overload will never actually kick in. Moreover, someone reading your code (like yourself 6 months down the road) might easily trip over which overload is actually being called by something like `foo(42)`.

Additionally, even in cases without risk of ambiguities, the Scala compiler currently only allows one of all overloads to define default parameters, otherwise you'll see a multiple overloaded alternatives of method `foo` define default arguments compiler error. As explained by [this answer](#) by Lukas Rytz on the scala-user mailing list the reason for this is a technical detail of how default parameters are currently implemented. So, potentially, this behavior could be changed in a future Scala version. (However, I certainly wouldn't count on it.)

Unfortunately, when implementing overloading with magnets, default parameters are not available at all. Instead you'll have to fall back to the old Java way of "unrolling" all defaults into their own overloads (i.e. magnet branches).

No Type Inference on Arguments

There are situations where method overloading prevents the compiler from inferring types in the way it otherwise would. Consider this example:

```
object Test {
  def foo(i: Int, f: String => String) = f(i.toString)
  def foo(d: Double, f: String => String) = f(d.toString)
}

Test.foo(42.0, _.trim)
```

This looks like a perfectly valid piece of code that the compiler should have no problem interpreting. Let's see what happens when we paste it into the REPL:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

object Test {
  def foo(i: Int, f: String => String) = f(i.toString)
  def foo(d: Double, f: String => String) = f(d.toString)
}

Test.foo(42.0, _.trim)

// Exiting paste mode, now interpreting.

error: missing parameter type for expanded function ((x$1) => x$1.trim)
Test.foo(42.0, _.trim)
           ^
```


The compiler cannot infer that the parameter of our anonymous function literal is a `String` even though there is obviously no other option. When we remove the first overload all is well and the snippet happily compiles. The reason for this phenomenon is buried in section “6.26.3 Overloading Resolution” of the Scala Language Specification. You might want to check out Jasons answer to [this Stackoverflow question](#) for some easier-to-understand explanation.

What we can see from this example is that method overloading can blind the compiler from “seeing” the argument type when several overloads define parameters with the same “shape” at the respective position. Unfortunately this is not only not improved by using magnets, it is even worsened.

Let’s look at an example (Scala 2.10 this time):

```
def foo(s: String): Unit = ???
def foo(f: String => String) = println(f(" Yay!"))

foo(_.trim)
```

Because the two overloads do not have the same “shape” this compiles and works as expected. Now the same thing magnetized:

```
def foo(magnet: FooMagnet) = magnet()

sealed trait FooMagnet {
  def apply()
}
object FooMagnet {
  implicit def fromString(s: => String) = new FooMagnet { def apply() = ??? }
  implicit def fromFunc(f: String => String) =
    new FooMagnet {
      def apply() = println(f(" Yay!"))
    }
}

foo(_.trim)
```

This doesn’t compile. We get the same missing parameter type for expanded function error as above, which shows us that the compiler is unable to infer that our function literal is to have the type `String => String`. When we think again about how the magnet pattern actually works this becomes clear. The compiler is looking for an implicit conversion from the type we specify to the magnet type. Since our `_.trim` argument does *not* have the type `String => String` (but rather some unqualified `Function1` type) the compiler cannot relate it to the respective magnet branch. Therefore it has no way of fully establishing the type of our function literal and gives up.

What this shows us is that the magnet pattern only works if the type of all arguments is fully known at the call site. Sometimes this can be inconvenient.

7.4.5 Conclusion

Stepping back, we can conclude that the magnet pattern offers a real alternative to “traditional” method overloading. It’s an alternative that is not per se better or worse. Rather, it’s simply different, with its own advantages and disadvantages. What is nice is that most of its properties are somewhat orthogonal to traditional overloading, the two solutions only share drawbacks in two areas (default parameters and type inference). For all other aspects one solution can overcome the issues of the other in that area, which gives us the choice to pick whatever technique best fits the requirements at hand. If you want you can even mix the two in one particular set of overloads. For example, you might choose to only use magnets for overcoming an erasure-induced collision on two overloads, and leave all others as is.

So, no matter whether you see immediate application opportunities for magnets in your own code or not, we think that the magnet pattern is a valuable technique to understand and master. If nothing else, having read about it will help you better comprehend what’s going on under the hood of *sprays routing DSL...*

Cheers,
Mathias

7.5 Welcome to the *spray* Blog

During the writing and continuous improvement of the *spray* documentation we've come across multiple things that don't quite fit the reference-type nature of the rest of this site. Sometimes this is because the material is kind of cross-cutting, like the description of a general technique used in several places of the *spray* implementation, and sometimes because it's more of a one-off highlighting of a specific feature or behavior. For this type of content a blog-style channel appears to be the better format, which is why we hereby solemnly unveil the official "*spray* blog".
*****tadaa*****

In addition to presenting our own posts we'd like to use this blog as an open platform for content from the community. Any kind of interesting material that's somehow, whether directly or more remotely, related to *spray* is very much welcome. It could be a small tidbit about a certain pattern you've discovered, an architecture-level description of a real-world API you've build or a maybe the results of a benchmark you've run. We'll happily serve up your article even if *spray* somehow did not make it to the top of your performance or overall satisfaction ranking... (well.... we'll see about that... ;^).

In any case, we are happy to hear from you, just pick any channel from the [Contact](#) page or type away in the posts comments section below. And if you really want to go for it fork the [spray codebase on github](#), add a post to the [/docs/blog/](#) folder and send us a pull request!

Cheers,
Mathias

Contact

8.1 Mailing List

In most cases the active and friendly [spray mailing list](#) is probably the best place for your needs with regard to support, feedback and general discussion.

Note: Your first post after signup is going to be moderated (for spam protection), but we'll immediately give you full posting privileges if your message doesn't unmask you as a spammer.

8.2 Twitter

If you [follow us on twitter](#) we'll keep you informed about new developments as well as general news around *spray*.

8.3 Email

If you feel that the mailing list is not appropriate for your request, you can also mail us under `info (at) spray (dot) io`

8.4 Commercial Support

For more details regarding commercial support, please [contact us](#).

- [ab17f00] use util.Timestamp instead of longs for timeout checking
- [f6b0292] get rid of Http.Register.keepOpenOnPeerClosed, fixes #401
- [0b5ef36] add max-encryption-chunk-size setting to ClientConnectionSettings and ServerSettings
- [98365ff] Implement redirection following (issue #132)
- [d738e54] require services to respond to ChunkedRequestStart with RegisterChunkHandler, fixes #473
- [da29cdf] “privatize” all classes/objects not meant to be part of public API
- [6fec00c] only render default User-Agent if no such header was explicit given, fixes #462
- [ba1ae77] upgrade to new HttpEntity / HttpData model
- [b2fee8d] make Content-Length a long value, fixes #443
- [4b48875] introduce dedicated exceptions for connection failure and request timeout for host-level API
- [9abbcf6] when creating HostConnectorSettings expect client settings at spray.can.client, fixes #408
- [e922cd4] move client.ssl-encryption setting from reference.conf into Http.Connect message, fixes #396
- [a47f3b0] replace InetAddress in HostConnectorSetup with hostname/port pair, fixes #394
- [80982d4] Publish SSLSessionEstablished event from SslTlsSupport upon successful SSL handshaking
- [e486900] Add SSLSessionInfo header to requests on server and responses on client
- [da12531] model user-agent-header value as User-Agent to fail fast, fixes #458
- [015f3c6] add HttpOrigin and use it for Access-Control-Allow-Origin and Origin headers, fixes #579
- [e058a43] allow creation of custom MediaTypes with “*” as a subtype when called by the parser, fixes #529
- [d2b8bba] introduce a distinction between “?key=” and “?key” in queries, fixes #460
- [ad593d1] make multipart form-data more flexible but have it adhere to the RFC more strictly
- [5d78dae] add CONNECT method and support for custom HTTP methods, closes #428
- [c6f49cc] introduce HttpData model replacing the byte array in HttpBody and MessageChunk, closes #365
- [f625b5a] add small extensions to Uri model
- [88a25f7] make only standard charsets available as constants, fixes #340
- [a915b8f] fix raw queries still performing %-decoding and not being rendered as raw, fixes #330

- [d8a9ee4] add support for Accept-Header extensions and media-type parameters, closes #310
- [a9e0d2c] support for custom status codes, fixes #564
- [443b0d8] remodel HttpIp to RemoteAddress, fixes #638
- [4d787dc] remove superfluous RequestContext::complete overloads
- [1480e73] improve PathMatcher infrastructure
- [7a36de5] CompletionMagnet: gone, streamlining completion API: accomplished
- [d86cb80] remove layer of *Aux classes by type aliases for simplicity
- [b145ced] upgrade to new ToResponseMarshaller, closes #293
- [9c9b976] AuthenticationFailedRejection now directly contains challenge headers to return, fixes #538
- [ab35761] fix getFromDirectory and getFromResourceDirectory not working properly for URIs with encoded chars
- [e3defb4] have encodeResponse automatically tie in autoChunkFileBytes
- [9c11228] small improvement of require and hrequire modifiers on directives
- [ead4a70] Added detach directive which executes its inner route in a future. Removed detachTo directive. Fixes #240.
- [8c91851] PathMatcher.(flat)map => h(flat)map, introduce map/flatMap, fixes #274
- [034779d] Render WWW-Authenticate header also for rejected credentials, fixes #188
- [ee7fe47] redefine PathMatchers.Empty as PathMatchers.Neutral with explicit type annotation, fixes #339
- [3ff3471] change PathMatcher.apply, add PathMatcher.provide method, cosmetic improvements
- [8ee49d7] add PathMatcher::repeated modifier, closes #636
- [f0cbf25] add pathEnd and pathEndOrSingleSlash directive, closes #628
- [ae17d18] create FormFile as an easy way to access uploaded file information for forms, fixes #327
- [9d27559] rename BodyPart.getName -> BodyPart.name, add BodyPart.dispositionParameterValue
- [fad2ff2] polish MediaType model, fix tests, smaller improvements
- [f8f5b6d] support content negotiation, fixes #167
- [ebaa580] enable FEOU and FSOD to be interchanged in the usual cases, fixes #426
- [ebe3e97] remove MetaUnmarshallers.scala, fold only member into FormDataUnmarshallers.scala
- [dd51be5] change default charset for application/x-www-form-urlencoded to utf8, fixes #526
- [f5b1535] decode should remove Content-Encoding header from message
- [adf9170] move unmarshal and unmarshalUnsafe to Unmarshaller and add unmarshaller method
- [f5997f8] flexibilize RequestBuilding and ResponseTransformation by generalizing the ~> operator
- [01c4aa9] major refactoring of SslTlsSupport, fixes #544
- [5f23219] improve DynamicPipelines trait
- [76345ba] abort connection on idle-timeout, fixes #539
- [2c77d8f] add support for compound write commands (Tcp.CompoundWrite)
- [6a99cb7] move result.awaitResult call from injectIntoRoute into check, fixes #205
- [72c9397] in RouteTests always convert URIs into absolute ones, fixes #464
- [680fde0] enable custom ExceptionHandlers in routing tests

[3b4ac55] small clean-up, remove duplication with httpx RequestBuilding

[e234dd9] remove SprayActorLogging and UtilSettings, simplify LoggingContext, fixes #421

[b0b90b3] Swap Duration.Undefined by Duration.Inf, fixes #440

[78d7e4a] improve *Settings infrastructure