

HTTP is your architecture

spot-media.de

Kore Nordmann <kore@qafoo.com>

Tobias Schlitt <toby@qafoo.com>

September 27, 2010



About us

- ▶ Long time FLOSS developers
- ▶ Open source enthusiast
- ▶ Authors, conference speakers
- ▶ Recently founded Qafoo GmbH - passion for software quality
- ▶ PMC members of Apache Zeta Components

Architecture

LCoDC\$SS

Do you know this term before?
HTTP: [Fie00]



Architecture

LCoDC\$SS

- ▶ Who heard of this term before?

FTP [Fie00]

Architecture

LCoDC\$SS

- ▶ Who heard of this term before?
 - ▶ This **is** HTTP. [Fie00]

Architecture

LCoDC\$SS



Architecture

Layered CoDC\$SS



Architecture

Layered Code on Demand C\$SS



Architecture

Layered Code on Demand Client \$S Server



Architecture

Layered Code on Demand Client Cached S
Server



Architecture

Layered Code on Demand Client Cached
Stateless Server

Outline

HTTP

Layered

Conclusion



HTTP methods

- ▶ Well known methods

- ▶ GET
- ▶ POST

- ▶ Less known / used methods

- ▶ PUT
- ▶ DELETE

- ▶ Not so known methods

- ▶ CONNECT
- ▶ TRACE
- ▶ MKCOL
- ▶ PROPFIND
- ▶ PROPPATCH

... and any you want. . .



HTTP methods

- ▶ Well known methods
 - ▶ GET
 - ▶ POST
- ▶ Less known / used methods
 - ▶ PUT
 - ▶ DELETE

Other known methods

- ▶ HEAD
- ▶ OPTIONS
- ▶ TRACE
- ▶ CONNECT
- ▶ PATCH
- ▶ COPY
- ▶ MOVE
- ▶ LOCK
- ▶ UNLOCK
- ▶ MKCOL
- ▶ PROPFIND
- ▶ PROPPATCH
- ▶ REPORT
- ▶ SOURCE
- ▶ COPY
- ▶ MOVE
- ▶ LOCK
- ▶ UNLOCK
- ▶ MKCOL
- ▶ PROPFIND
- ▶ PROPPATCH
- ▶ REPORT
- ▶ SOURCE

... and any you want. ...

HTTP methods

- ▶ Well known methods
 - ▶ GET
 - ▶ POST
- ▶ Less known / used methods
 - ▶ PUT
 - ▶ DELETE
- ▶ Mostly unknown methods
 - ▶ HEAD
 - ▶ OPTIONS
 - ▶ TRACE
 - ▶ CONNECT
 - ▶ WebDAV
 - ▶ MKCOL
 - ▶ PROPSET
 - ▶ PROPGET
 - ▶ Use any you want. . .



“Safe” HTTP methods

- ▶ “[..] GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval.” [RF99]
- ▶ ... so it is “safe” for spiders to call them.

- ▶ `Cache-Control: public` - all bots should use method="GET"
- ▶ `Cache-Control: no-cache` - if content is modified, the result can be cached.
- ▶ `Cache-Control: no-store` - do not use that automatically
- ▶ `Cache-Control: no-cache` - Squid
- ▶ `Cache-Control: no-cache` - Company application proxies
- ▶ `Cache-Control: no-cache` - Shared Code on Demand Client Cached Stateless Server

“Safe” HTTP methods

- ▶ “[..] GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval.” [RF99]
- ▶ ... so it is “safe” for spiders to call them.
 - ▶ Search forms should use `method="GET"`

Cacheable: If the response is modified, the result can be cached.

Cacheable: Can be used by proxies that automatically

cache responses / Squid

Cacheable: Company application proxies

Cacheable: Content Delivery Network / Code on Demand Client **Cacheless Stateless Server**

“Safe” HTTP methods

- ▶ “[..] GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval.” [RF99]
- ▶ ... so it is “safe” for spiders to call them.
 - ▶ Search forms should use `method="GET"`
- ▶ Since nothing is modified, the result can be cached.
 - ▶ Proxies can use that automatically
 - ▶ Varnish / Squid
 - ▶ Company application proxies

Code on Demand Client Cached Stateless Server

“Safe” HTTP methods

- ▶ “[..] GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval.” [RF99]
- ▶ ... so it is “safe” for spiders to call them.
 - ▶ Search forms should use `method="GET"`
- ▶ Since nothing is modified, the result can be cached.
 - ▶ Proxies can use that automatically
 - ▶ Varnish / Squid
 - ▶ Company application proxies
 - ▶ **Layered** Code on Demand Client **Cached** Stateless Server

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[...] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[...] refers to an already existing resource, the server SHOULD be considered as a modified version of the resource.” [RF99]
 - ▶ Examples
 - ▶ Updating a users account data using PUT `/users/42`
 - ▶ Creation of resources with known identifiers
- ▶ POST is used to create a new resource
 - ▶ “[...] used to request that the origin server accept the entity enclosed in the request as a new subordinate[.]” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource

- ▶ “[...] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
- ▶ “[...] refers to an already existing resource, the server SHOULD be considered as a modified version of the resource” [RF99]
- ▶ Examples

- ▶ Updating a users account data using PUT `/users/42`
- ▶ Creation of resources with known identifiers

- ▶ POST appends to an existing resource

- ▶ “[...] request that the origin server accept the entity enclosed in the request as a new subordinate[.]” [RF99]
- ▶ Examples [RF99]
- ▶ Annotation of existing resources
- ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
- ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[...] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[...] refers to an already existing resource, the server SHOULD be considered as a modified version of the resource” [RF99]
 - ▶ Examples
 - ▶ Updating a users account data using PUT /users/42
 - ▶ Creation of resources with known identifiers
- ▶ POST appends to an existing resource
 - ▶ “[...]is used to request that the origin server accept the entity enclosed in the request as a **new subordinate[...]**” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[...] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[...] refers to an already existing resource, the server SHOULD be considered as a modified version of the resource” [RF99]
 - ▶ Examples
 - ▶ Creating a users account data using PUT /users/42
 - ▶ Updating of resources with known identifiers
- ▶ POST appends to an existing resource
 - ▶ “[...]is used to request that the origin server accept the entity enclosed in the request as a **new subordinate[...]**” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[..] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[..] refers to an already existing resource, the PUT SHOULD be considered as a modified version of that resource.” [RF99]
 - ▶ Examples
 - ▶ Storing a users account data using PUT /users/42
 - ▶ Updating of resources with known identifiers
- ▶ POST appends to an existing resource
 - ▶ “[..]is used to request that the origin server accept the entity enclosed in the request as a **new subordinate[..]**” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[..] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[..] refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version [..]” [RF99]

EXAMPLES

- ▶ PUT is used to update a user's account data using PUT /users/42
- ▶ PUT is used to create new resources with known identifiers
- ▶ POST appends to an existing resource
 - ▶ “[..] is used to request that the origin server accept the entity enclosed in the request as a **new subordinate[..]**” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

POST vs. PUT

- ▶ PUT creates or replaces a resource
 - ▶ “[..] requests that the enclosed entity be stored under the supplied Request-URI.” [RF99]
 - ▶ “[..] refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version [..]” [RF99]
 - ▶ Examples
 - ▶ Updating a users account data using PUT /users/42
 - ▶ Creation of resources with known identifiers
- ▶ POST appends to an existing resource
 - ▶ “[..]is used to request that the origin server accept the entity enclosed in the request as a **new subordinate[..]**” [RF99]
 - ▶ Examples [RF99]
 - ▶ Annotation of existing resources
 - ▶ Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
 - ▶ Extending a database through an append operation.

Idempotent methods

$$f(x) = f(f(x))$$

GET and HEAD are idempotent but POST has to be idempotent

Resending the request again, should not change anything

Idempotent methods PUT and DELETE

It is not safe to just resend request, if one failed due to network problems

Idempotence is a useful property in all messaging systems

Idempotent methods

$$f(x) = f(f(x))$$

- ▶ Everything but POST has to be idempotent
 - ▶ Executing the request again, should not change anything.

PUT and DELETE

It is safe to just resend request, if one failed due to network problems

Idempotence is a useful property in all messaging systems

Idempotent methods

$$f(x) = f(f(x))$$

- ▶ Everything but POST has to be idempotent
 - ▶ Executing the request again, should not change anything.
 - ▶ This includes PUT and DELETE

It is possible to just resend request, if one failed due to network problems

Idempotence is a useful property in all messaging systems

Idempotent methods

$$f(x) = f(f(x))$$

- ▶ Everything but POST has to be idempotent
 - ▶ Executing the request again, should not change anything.
 - ▶ This includes PUT and DELETE
- ▶ Really useful to just resend request, if one failed due to network problems

Idempotence is a useful property in all messaging systems

Idempotent methods

$$f(x) = f(f(x))$$

- ▶ Everything but POST has to be idempotent
 - ▶ Executing the request again, should not change anything.
 - ▶ This includes PUT and DELETE
- ▶ Really useful to just resend request, if one failed due to network problems
- ▶ Idempotence is a useful property in all messaging systems

HTTP method fail in PHP

- ▶ `$_GET` contains the request parameters
- ▶ `$_POST` contains the request body

NOTE: HTTP requests may contain a body and request parameters

GET

you want to use something like `$request->parameters`

`$request->body`

HTTP method fail in PHP

- ▶ `$_GET` contains the request parameters
- ▶ `$_POST` contains the request body
- ▶ All HTTP requests may contain a body **and** request parameters
 - ▶ Yes, even GET

Don't want to use something like `$request->parameters`
or `$request->body`

HTTP method fail in PHP

- ▶ `$_GET` contains the request parameters
- ▶ `$_POST` contains the request body
- ▶ All HTTP requests may contain a body **and** request parameters
 - ▶ Yes, even GET
- ▶ You may want to use something like `$request->parameters` and `$request->body`

Do you speak HTTP?

- ▶ “The methods GET and HEAD MUST be supported by all general-purpose servers.” [RF99]

“All other methods are OPTIONAL.” [RF99]

“If any of the above methods are implemented, they MUST be implemented with the same semantics as those specified in [RFC99].”

“If your website is not HTTP, if you are using POST for a search form,

then the search results are not bookmarkable or linkable...”

Do you speak HTTP?

- ▶ “The methods GET and HEAD MUST be supported by all general-purpose servers.” [RF99]
- ▶ “All other methods are OPTIONAL;” [RF99]

Even if the above methods are implemented, they MUST be implemented with the same semantics as those specified in [RFC99]

Even if your website is not HTTP, if you are using POST for a search form,

the search results are not bookmarkable or linkable...

Do you speak HTTP?

- ▶ “The methods GET and HEAD **MUST** be supported by all general-purpose servers.” [RF99]
- ▶ “All other methods are **OPTIONAL**;” [RF99]
- ▶ “however, if the above methods are implemented, they **MUST** be implemented with the same semantics as those specified in section 9.” [RF99]

“If your website is not HTTP, if you are using POST for a search form,

then the search results are not bookmarkable or linkable...”

Do you speak HTTP?

- ▶ “The methods GET and HEAD MUST be supported by all general-purpose servers.” [RF99]
- ▶ “All other methods are OPTIONAL;” [RF99]
- ▶ “however, if the above methods are implemented, they MUST be implemented with the same semantics as those specified in section 9.” [RF99]
 - ▶ Sorry, your website is not HTTP, if you are using POST for a search form.
 - ▶ And the search results are not bookmarkable or linkable. . .

Outline

HTTP

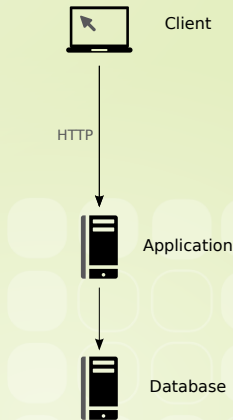
Layered

Conclusion



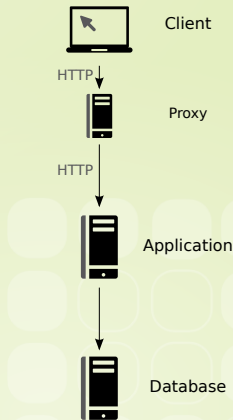
Layered architecture

- ▶ HTTP allows layered architectures
- ▶ But what is required to make this work?
 - ▶ Request semantic must be understood by the application
 - ▶ Application must be able to talk to the database



Layered architecture

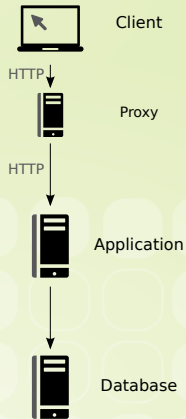
- ▶ HTTP allows layered architectures
 - But what is required to make this work?
 - The request semantic must be supported by the proxy
 - The response semantic must be supported by the application



Layered architecture

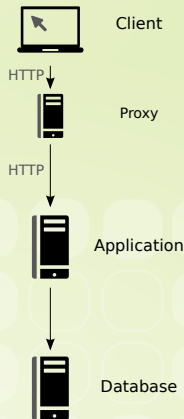
- ▶ HTTP allows layered architectures
- ▶ But what is required to make this work?

Application semantics must be supported by the underlying protocol. The protocol must be able to support the application semantics.



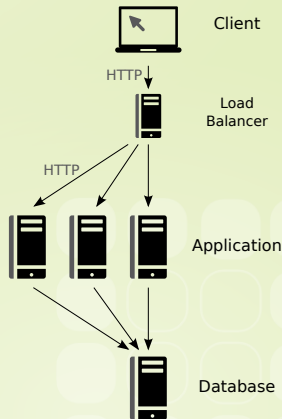
Layered architecture

- ▶ HTTP allows layered architectures
- ▶ But what is required to make this work?
 - ▶ Request semantic must be handled by the proxy



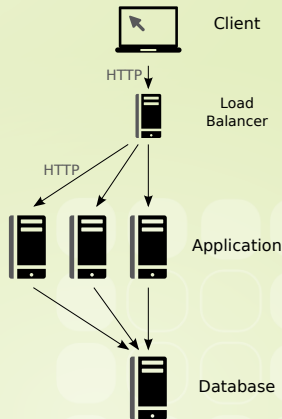
Layered architecture

- ▶ HTTP allows layered architectures
- ▶ But what is required to make this work?
 - ▶ Request semantic must be handled by the proxy



Layered architecture

- ▶ HTTP allows layered architectures
- ▶ But what is required to make this work?
 - ▶ Request semantic must be handled by the proxy
 - ▶ The server must be **stateless**



Stateless server

- ▶ No persistent connection
- ▶ Each request contains all information to be processed

is exchanged transparently
sessions and static data



Stateless server

- ▶ No persistent connection
- ▶ Each request contains all information to be processed
 - ▶ Cookies

cookies are exchanged transparently
sessions and static data

Stateless server

- ▶ No persistent connection
- ▶ Each request contains all information to be processed
 - ▶ Cookies
- ▶ Servers can be exchanged transparently

no sessions and static data

Stateless server

- ▶ No persistent connection
- ▶ Each request contains all information to be processed
 - ▶ Cookies
- ▶ Servers can be exchanged transparently
 - ▶ Mind the sessions and static data



Drawbacks / Benefits

- ▶ Drawbacks

- ▶ Users do have state – makes session handling

- ▶ Benefits

- ▶ App servers do not scale beyond a single node

- ▶ Session data can be hosted on a dedicated cluster

- ▶ App servers do not matter

- ▶ Can use plain random load balancing – massively reduces complexity of this layer

- ▶ HTTP scales, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

- ▶ Drawbacks

- ▶ Users **do** have state – makes session handling complicated.

- ▶ Benefits

- ▶ App servers do not scale beyond a single node

- ▶ Session data can be hosted on a dedicated cluster

- ▶ App servers do not matter

- ▶ Can use plain random load balancing – massively reduces complexity of this layer

- ▶ HTTP scales, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

► Drawbacks

- Users **do** have state – makes session handling complicated.

► Benefits

- App servers do not scale beyond a single node

► Session state can be hosted on a dedicated cluster

► Session servers do not matter

► Session servers use plain random load balancing – massively reduces complexity of this layer

► Session servers scale, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

► Drawbacks

- Users **do** have state – makes session handling complicated.

► Benefits

- App servers do not scale beyond a single node
- Session data can be hosted on a dedicated cluster

► Session servers do not matter

► Session servers use plain random load balancing – massively reduces the complexity of this layer

► Session servers scale, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

▶ Drawbacks

- ▶ Users **do** have state – makes session handling complicated.

▶ Benefits

- ▶ App servers do not scale beyond a single node
- ▶ Session data can be hosted on a dedicated cluster
- ▶ Failing servers do not matter

▶ **Simple** – use plain random load balancing – massively reduces complexity of this layer

▶ **Simple** – scales, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

▶ Drawbacks

- ▶ Users **do** have state – makes session handling complicated.

▶ Benefits

- ▶ App servers do not scale beyond a single node
- ▶ Session data can be hosted on a dedicated cluster
- ▶ Failing servers do not matter
- ▶ We can use plain random load balancing – massively reduces complexity of this layer

App servers do not scale, because it is build for this (Shared nothing architecture)

Drawbacks / Benefits

▶ Drawbacks

- ▶ Users **do** have state – makes session handling complicated.

▶ Benefits

- ▶ App servers do not scale beyond a single node
- ▶ Session data can be hosted on a dedicated cluster
- ▶ Failing servers do not matter
- ▶ We can use plain random load balancing – massively reduces complexity of this layer
- ▶ PHP scales, because it is build for this (Shared nothing architecture)

Embrace HTTP

- ▶ Use HTTP to communicate with backend services & subsystems
 - ▶ Webservices (REST, Soap, XMLRPC)
 - ▶ You can reuse your common infrastructure

Take the next level

Use HTTP to communicate with your database (CouchDB)
and try to eliminate layers where appropriate

Embrace HTTP

- ▶ Use HTTP to communicate with backend services & subsystems
 - ▶ Webservices (REST, Soap, XMLRPC)
 - ▶ You can reuse your common infrastructure
- ▶ Taking it to the next level
 - ▶ Use HTTP to communicate with your database (CouchDB)
 - ▶ Option to eliminate layers where appropriate

What is REST?

- ▶ What is REST actually?

- Describes services which follow the HTTP / LCoDCSSS
- Emphasizes the resources / concept character of URLs
- Even respects the Accept-* HTTP headers

What is REST?

- ▶ What is REST actually?
 - ▶ Describes services which follow the HTTP / LCoDCSSS

• Describes the resources / concept character of URLs

• Even respects the Accept-* HTTP headers

What is REST?

- ▶ What is REST actually?
 - ▶ Describes services which follow the HTTP / LCoDCSSS
 - ▶ Following the resources / concept character of URLs

even respects the Accept-* HTTP headers

What is REST?

- ▶ What is REST actually?
 - ▶ Describes services which follow the HTTP / LCoDCSSS
 - ▶ Following the resources / concept character of URLs
 - ▶ Sometimes even respects the Accept-* HTTP headers

Outline

HTTP

Layered

Conclusion



Thanks for listening

- ▶ HTTP and PHP are build like this for a reason
 - ▶ Scalabilty
 - ▶ Fault tolerance
- ▶ More about us:
 - ▶ <http://qafoo.com>



Bibliography I

- [Fie00] R. Fielding, *Architectural styles and the design of network-based software architectures*, Ph.D. thesis, University of California, Irvine, USA, 2000.
- [RF99] et al. R. Fielding, *Hypertext transfer protocol – http/1.1*, <http://tools.ietf.org/html/rfc2616>, June 1999.