CHAPTER 1
Using the Uniform Interface

HTTP is an application-level protocol that defines operations for transferring representations between clients and servers. In this protocol, methods such as `GET`, `POST`, `PUT`, and `DELETE` are operations on resources. This protocol eliminates the need for you to invent application-specific operations such as `createOrder`, `getStatus`, `updateStatus`, etc. How much you can benefit from the HTTP infrastructure largely depends on how well you can use HTTP as an application-level protocol. However, a number of techniques including SOAP and some Ajax web frameworks use HTTP as a protocol to transport messages. Such usage makes poor use of HTTP-level infrastructure. This chapter presents the following recipes to highlight various aspects of using HTTP as an application protocol:

Recipe 1.1, "How to Keep Interactions Visible"
: Visibility is one of the key characteristics of HTTP. Use this recipe to learn how to maintain visibility.

Recipe 1.2, "When to Trade Visibility"
: There are cases when you may need to forgo visibility to meet application needs. Use this recipe to find some scenarios.

Recipe 1.3, "How to Maintain Application State"
: Use this recipe to learn the best way to manage state.

Recipe 1.4, "How to Implement Safe and Idempotent Methods on the Server"
: Maintaining safety and idempotency helps servers guarantee repeatability for requests. Use this recipe when implementing servers.

Recipe 1.5, "How to Treat Safe and Idempotent Methods in Clients"
: Follow this recipe to implement clients for safety and idempotency principles.

Recipe 1.6, "When to Use GET"
: Use this recipe to learn when to use `GET`.

Recipe 1.7, "When to Use POST"
: Use this recipe to learn when to use `POST`.

Recipe 1.8, "How to Create Resources Using POST"
: Use this recipe to learn how to create new resources using the `POST` method.

Recipe 1.9, "When to Use PUT to Create New Resources"
: You can use either `POST` or `PUT` to create new resources. This recipe will discuss when using `PUT` is better.

Recipe 1.10, "How to Use POST for Asynchronous Tasks"
: Use this recipe to learn how to use the `POST` method for asynchronous tasks.

Recipe 1.11, "How to Use DELETE for Asynchronous Deletion"
: Use this recipe to learn how to use the `DELETE` method for asynchronous deletion of resources.

Recipe 1.12, "When to Use Custom HTTP Methods"
: Use this recipe to learn why custom HTTP methods are not recommended.

Recipe 1.13, "When and How to Use Custom HTTP Headers"
: Use this recipe to learn when and how to use custom HTTP headers.

1.1 How to Keep Interactions Visible

As an application protocol, HTTP is designed to keep interactions between clients and servers visible to libraries, servers, proxies, caches, and other tools. Visibility is a key characteristic of HTTP. Per Roy Fielding (see Appendix A for references), visibility is "the ability of a component to monitor or mediate the interaction between two other components." When a protocol is visible, caches, proxies, firewalls, etc., can monitor and even participate in the protocol.

Problem

You want to know what visibility means and what you can do to keep HTTP requests and responses visible.

Solution

Once you identify and design resources, use `GET` to get a representation of a resource, `PUT` to update a resource, `DELETE` to delete a resource, and `POST` to perform a variety of potentially nonidempotent and unsafe operations. Add appropriate HTTP headers to describe requests and responses.

Discussion

Features like the following depend entirely on keeping requests and responses visible:

Caching
: Caching responses and automatically invalidating cached responses when resources are modified

Optimistic concurrency control
: Detecting concurrent writes and preventing resource changes when such operations are based on stale representations

Content negotiation
: Selecting a representation among alternatives available for a given resource

Safety and idempotency
: Ensuring that clients can repeat or retry certain HTTP requests

When a web service does not maintain visibility, such features will not work correctly. For instance, when the server's usage of HTTP breaks optimistic concurrency, you may be forced to invent application-specific concurrency control mechanisms on your own.

Maintaining visibility lets you use existing HTTP software and infrastructure for features that you would otherwise have to build yourself.

HTTP achieves visibility by virtue of the following:

- HTTP interactions are stateless. Any HTTP intermediary can infer the meaning of any given request and response without correlating them with past or future requests and responses.
- HTTP uses a uniform interface consisting of `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, and `TRACE` methods. Each method in this interface operates on one and only one resource. The syntax and the meaning of each method do not change from application to application or from resource to resource. That is why HTTP is known as a *uniform interface*.
- HTTP uses a MIME-like envelope format to encode representations. This format maintains a clear separation between headers and the body. Headers are visible, and except for the software that is creating the message and the software that is processing the message, every piece of software in between can treat the body as completely opaque.

Consider an HTTP request to update a resource:

```
# Request
PUT /movie/gone_with_the_wind HTTP/1.1 ❶
Host: www.example.org ❷
Content-Type: application/x-www-form-urlencoded

summary=...&rating=5&... ❸

# Response
HTTP/1.1 200 OK ❹
Content-Type: text/html;charset=UTF-8 ❺
Content-Length: ...
```

```
<html>❻
 ...
</html>
```

❶ Request line containing HTTP method, path to the resource, and HTTP version

❷ Representation headers for the request

❸ Representation body for the request

❹ Response status line containing HTTP version, status code, and status message

❺ Representation headers for the response

❻ Representation body for the response

In this example, the request is an HTTP message. The first line in this message describes the protocol and the method used by the client. The next two lines are request headers. By simply looking at these three lines, any piece of software that understands HTTP can decipher not only the intent of the request but also how to parse the body of the message. The same is the case with the response. The first line in the response indicates the version of HTTP, the status code, and a message. The next two lines tell HTTP-aware software how to interpret the message.

For RESTful web services, your key goal must be to maintain visibility to the extent possible. Keeping visibility is simple. Use each HTTP method such that it has the same semantics as specified by HTTP, and add appropriate headers to describe requests and responses.

Another part of maintaining visibility is using appropriate status codes and messages so that proxies, caches, and clients can determine the outcome of a request. A status code is an integer, and the status message is text.

As we will discuss in Recipe 1.2, there are cases where you may need to trade off visibility for other characteristics such as network efficiency, client convenience, and separation of concerns. When you make such trade-offs, carefully analyze the effect on features such as caching, idempotency, and safety.

1.2 When to Trade Visibility

This recipe describes some common situations where trading off visibility may be necessary.

Problem

You want to know common situations that may require you to keep requests and responses less visible to the protocol.

Solution

Whenever you have multiple resources that share data or whenever an operation modifies more than one resource, be prepared to trade visibility for better abstraction of information, loose coupling, network efficiency, resource granularity, or pure client convenience.

Discussion

Visibility often competes with other architectural demands such as abstraction, loose coupling, efficiency, message granularity, etc. For example, think of a person resource and a related address resource. Any client can submit `GET` requests to obtain representations of these two resources. For the sake of client convenience, the server may include data from the address resource within the representation of the person resource as follows:

```
# Request to get the person
GET /person/1 HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<person>
  <name>John Doe</name>
  <address type="home">
    <street>1 Main Street</street>
    <city>Bellevue</city>
    <state>WA</state>
  </address>
</person>

# Request to get the address
GET /person/1/address HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<address type="home">
  <street>1 Main Street</street>
  <city>Bellevue</city>
  <state>WA</state>
</address>
```

Let's assume that the server allows clients to submit `PUT` requests to update these resources. When a client modifies one of these resources, the state of the related resource also changes. However, at the HTTP level, these are independent resources. Only the server knows that they are dependent. Such overlapping data is a common cause of reduced visibility.

One of the important consequences of reduced visibility is caching (see Chapter 9). Since these are two independent resources at the HTTP level, caches will have two copies of the address: one as an independent address representation and the other as part of the person representation. This can be inefficient. Also, invalidating one

representation from the cache will not invalidate the other representation. This can leave stale representations in the cache.

In this particular example, you can eliminate the overlap between these resources by including a reference to the address from the person resource and avoid including address details. You can use links (see Chapter 5) to provide references to other resources.

Although providing a link may minimize overlaps, it will force clients to make additional requests.

In this example, the trade-off is between visibility and client convenience and, potentially, network efficiency. A client that always deals with person resources can make a single request to get information about the person as well as the address.

Here are some more situations where you may need to give up visibility for other benefits:

Client convenience
: Servers may need to design special-purpose coarse-grained composite resources for the sake of client convenience (e.g., Recipe 2.4).

Abstraction
: In order to abstract complex business operations (including transactions), servers may need to employ controller resources to make changes to other resources (e.g., Recipe 2.6). Such resources can hide the details used to implement business operations.

Network efficiency
: In cases where a client is performing several operations in quick succession, you may need to combine such operations into batches to reduce network latency (e.g., Recipes 11.10 and 11.13).

In each of these cases, if you focus only on visibility, you may be forced to design your web service to expose all data as independent resources with no overlaps. A web service designed in that manner may lead to fine-grained resources and poor separation of concerns between clients and servers. For an example, see Recipe 2.6. Other scenarios such as copying or merging resources and making partial updates (see Chapter 11) may also require visibility trade-offs.

Provided you are aware of the consequences early during the design process, trading off visibility for other benefits is not necessarily bad.

1.3 How to Maintain Application State

Often when you read about REST, you come across the recommendation to "keep the application state on the client." But what is "application state" and how can you keep that state on the client? This recipe describes how best to maintain state.

Problem

You want to know how to manage state in RESTful web services such that you do not need to rely on in-memory sessions on servers.

Solution

Encode application state into URIs, and include those URIs into representations via links (see Chapter 5). Let clients use these URIs to interact with resources. If the state is large or cannot be transported to clients for security or privacy reasons, store the application state in a durable storage (such as a database or a filesystem), and encode a reference to that state in URIs.

Discussion

Consider a simplified auto insurance application involving two steps. In the first step, the client submits a request with driver and vehicle details, and the server returns a quote valid for a week. In the second step, the client submits a request to purchase insurance. In this example, the application state is the quote. The server needs to know the quote from the first step so that it can issue a policy based on that quote in the second request.

Application state is the state that the server needs to maintain between each request for each client. Keeping this state in clients does not mean serializing some session state into URIs or HTML forms, as web frameworks like ASP.NET and JavaServer Faces do.

Since HTTP is a stateless protocol, each request is independent of any previous request. However, interactive applications often require clients to follow a sequence of steps in a particular order. This forces servers to temporarily store each client's current position in those sequences outside the protocol. The trick is to manage state such that you strike a balance between reliability, network performance, and scalability.

The best place to maintain application state is within links in representations of resources, as in the following example:

```
# Request
POST /quotegen HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded
```

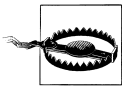
```
fname=...&lname=...&..

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<quote xmlns:atom="http://www.w3.org/2005/Atom">
  <driver>
    ...
  </driver>
  <vehicle>
    ...
  </vehicle>
  <offer>
    ...
    <valid-until>2009-10-02</valid-until>
    <atom:link href="http://www.example.org/quotes/buy?quote=abc1234"
          rel="http://www.example.org/rels/quotes/buy"/> ❶
  </offer>
</html>
```

❶ A link containing application state

In this example, the server stores the quote data in a data store and encodes its primary key in the URI. When the client makes a request to purchase insurance using this URI, the server can reinstate the application state using this key.

Choose a durable storage such as a database or a filesystem to store application state. Using a nondurable storage such as a cache or an in-memory session reduces the reliability of the web service as such state may not survive server restart. Such solutions may also reduce scalability of the server.

Alternatively, if the amount of data for the quote is small, the server can encode the state within the URI itself, as shown in the code below.

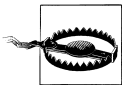


When you store application state in databases, use database replication so that all server instances have access to that state. If the application state is not permanent, you may also need to clean up the state at some point.

```
# Request
GET /quotegen?fname=...&lname=...&... HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<quote xmlns:atom="http://www.w3.org/2005/Atom">
  <driver>
```

```
    ...
  </driver>
  <vehicle>
    ...
  </vehicle>
  <offer>
    ...
    <valid-until>2009-08-02</valid-until>
    <atom:link href="http://www.example.org/quotes/buy?fname=...&lname=...&..."
          rel="http://www.example.org/quotes/buy"/>
  </offer>
</html>
```

Since the client will need to send all that data back in every request, encoding the application state in links may reduce network performance. Yet it can improve scalability since the server does not need to store any data, and it may improve reliability since the server does not need to use replication. Depending on your specific use case and the amount of state, use a combination of these two approaches for managing application state, and strike a balance between network performance, scalability, and reliability.



When you store application state in links, make sure to add checks (such as signatures) to detect/prevent the tampering of state. See Recipe 12.5 for an example.

1.4 How to Implement Safe and Idempotent Methods on the Server

Safety and idempotency are guarantees that a server must provide to clients in its implementation for certain methods. This recipe discusses why these matter and how to implement safety and idempotency on the server.

Problem

You want to know what idempotency and safety mean, and what you can do to ensure that the server's implementation of various HTTP methods maintain these two characteristics.

Solution

While implementing `GET`, `OPTIONS`, and `HEAD` methods, do not cause any side effects. When a client resubmits a `GET`, `HEAD`, `OPTIONS`, `PUT`, or `DELETE` request, ensure that the server provides the same response except under concurrent conditions (see Chapter 10).

Discussion

Safety and idempotency are characteristics of HTTP methods for servers to implement. Table 1-1 shows which methods are safe and which are idempotent.

Table 1-1. Safety and idempotency of HTTP methods

Method	Safe?	Idempotent?
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No

Implementing safe methods

In HTTP, safe methods are not expected to cause side effects. Clients can send requests with safe methods without worrying about causing unintended side effects. To provide this guarantee, implement safe methods as read-only operations.

Safety does not mean that the server must return the same response every time. It just means that the client can make a request knowing that it is not going to change the state of the resource. For instance, both the following requests *may be* safe:

```
# First request
GET /quote?symb=YHOO HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8

15.96

# Second request 10 minutes later
GET /quote?symb=YHOO HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8

16.10
```

In this example, the change in response between these two requests may have been triggered by some other client or some backend operation.

Implementing idempotent methods

Idempotency guarantees clients that repeating a request has the same effect as making a request just once. Idempotency matters most in the case of network or software

failures. Clients can repeat such requests and expect the same outcome. For example, consider the case of a client updating the price of a product.

```
# Request
PUT /book/gone-with-the-wind/price/us HTTP/1.1
Content-Type: application/x-www-form-urlencoded

val=14.95
```

Now assume that because of a network failure, the client is unable to read the response. Since HTTP says that `PUT` is idempotent, the client can repeat the request.

```
# Request
PUT /book/gone-with-the-wind/price/us HTTP/1.1
Content-Type: application/x-www-form-urlencoded

val=14.95

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<value>14.95</value>
```

For this approach to work, you must implement all methods except `POST` to be idempotent. In programming language terms, idempotent methods are similar to "setters." For instance, calling the `setPrice` method in the following code more than once has the same effect as calling it just once:

```
class Book {
    private Price price;
    public void setPrice(Price price) {
        this.price = price;
    }
}
```

Idempotency of DELETE

The `DELETE` method is idempotent. This implies that the server must return response code `200` (`OK`) even if the server deleted the resource in a previous request. But in practice, implementing `DELETE` as an idempotent operation requires the server to keep track of all deleted resources. Otherwise, it can return a `404` (`Not Found`).

```
# First request
DELETE /book/gone-with-the-wind HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK

# Second request
DELETE /book/gone-with-the-wind HTTP/1.1
Host: www.example.org

# Response
```

```
HTTP/1.1 404 Not Found
Content-Type: text/html;charset=UTF-8

<html>
  ...
</html>
```

Even when the server has a record of all the deleted resources, security policies may require the server to return a `404` (`Not Found`) response code for any resource that does not currently exist.

1.5 How to Treat Safe and Idempotent Methods in Clients

Problem

You want to know how to implement HTTP requests that are idempotent and/or safe.

Solution

Treat `GET`, `OPTIONS`, and `HEAD` as read-only operations, and send those requests whenever required.

In the case of network or software failures, resubmit `GET`, `PUT`, and `DELETE` requests to confirm, supplying `If-Unmodified-Since` and/or `If-Match` conditional headers (see Chapter 10).

Do not repeat `POST` requests, unless the client knows ahead of time (e.g., via server's documentation) that its implementation of `POST` for any particular resource is idempotent.

Discussion

Safe methods

Any client should be able to make `GET`, `OPTIONS` and `HEAD` requests as many times as necessary. If a server's implementation causes unexpected side effects when processing these requests, it is fair to conclude that the server's implementation of HTTP is incorrect.

Idempotent methods

As discussed in Recipe 1.4, idempotency guarantees that the client can repeat a request when it is not certain the server successfully processed that request. In HTTP, all methods except `POST` are idempotent. In client implementations, whenever you encounter a software or a network failure for an idempotent method, you can implement logic to retry the request. Here is a pseudocode snippet:

```
try {
    // Submit a PUT request
```

```
        response = httpRequest.send("PUT", ...);
        if(response.code == 200) {
            // Success
            ...
        }
        else if(response.code >= 400) {
            // Failure due to client error
            ...
        }
        else if(response.code >= 500) {
            // Failure due to server error
            ...
        }
        ...
    }
    catch(NetworkFailure failure) {
        // Retry the request now or later
        ...
    }
```

In this example, the client implements logic to repeat the request only in the case of network failures, not when the server returned a `4xx` or `5xx` error. The client must continue to treat various HTTP-level errors as usual (see Recipe 3.14).

Since `POST` is not idempotent, do not apply the previous pattern for `POST` requests unless told by the server. Recipe 10.8 describes a way for servers to provide idempotency for `POST` requests.

1.6 When to Use GET

The infrastructure of the Web strongly relies on the idempotent and safe nature of `GET`. Clients count on being able to repeat `GET` requests without causing side effects. Caches depend on the ability to serve cached representations without contacting the origin server.

Problem

You want to know when and when not to use `GET` and the potential consequences of using `GET` inappropriately.

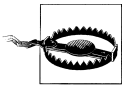
Solution

Use `GET` for safe and idempotent information retrieval.

Discussion

Each method in HTTP has certain semantics. As discussed in Recipe 1.1, the purpose of `GET` is to get a representation of a resource, `PUT` is to create or update a resource, `DELETE` is to delete a resource, and `POST` is either to create new resources or to make various other changes to resources.

Of all these methods, GET can take the least amount of misuse. This is because GET is both safe and idempotent.



Do not use GET for unsafe or nonidempotent operations. Doing so could cause permanent, unexpected, and undesirable changes to resources.

Most abuse of GET happens in the form of using this method for unsafe operations. Here are some examples:

```
# Bookmark a page
GET /bookmarks/add_bookmark?href=http%3A%2F%2F
    www.example.org%2F2009%2F10%2F10%2Fnotes.html HTTP/1.1
Host: www.example.org

# Add an item to a shopping cart
GET /add_cart?pid=1234 HTTP/1.1
Host: www.example.org

# Send a message
GET /messages/send?message=I%20am%20reading HTTP/1.1
Host: www.example.org

# Delete a note
GET /notes/delete?id=1234 HTTP/1.1
Host: www.example.org
```

For the server, all these operations are unsafe and nonidempotent. But for any HTTP-aware software, these operations are safe and idempotent. The consequences of this difference can be severe depending on the application. For example, a tool routinely performing health checks on a server by periodically submitting a GET request using the fourth URI shown previously will delete a note.

If you must use GET for such operations, take the following precautions:

- Make the response noncacheable by adding a `Cache-Control: no-cache` header.
- Ensure that any side effects are benign and do not alter business-critical data.
- Implement the server such that those operations are repeatable (i.e., idempotent).

These steps may help reduce the impact of errors for certain but not all operations. The best course of action is to avoid abusing GET.

1.7 When to Use POST

This recipe summarizes various applications of POST.

Problem

You want to know the potential applications of the POST method.

Solution

Use `POST` for the following:

- To create a new resource, using the resource as a factory as described in Recipe 1.8
- To modify one or more resources via a controller resource as described in Recipe 2.6
- To run queries with large inputs as described in Recipe 8.3
- To perform any unsafe or nonidempotent operation when no other HTTP method seems appropriate

Discussion

In HTTP, the semantics of method `POST` are the most generic. HTTP specifies that this method is applicable for the following.*

- Annotation of existing resources;
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
- Providing a block of data, such as the result of submitting a form, to a data-handling process;
- Extending a database through an append operation.

All such operations are unsafe and nonidempotent, and all HTTP-aware tools treat `POST` as such:

- Caches do not cache responses of this method.
- Crawlers and such tools do not automatically activate `POST` requests.
- Most generic HTTP tools do not resubmit `POST` requests automatically.

Such a treatment gives great latitude for servers to use `POST` as a general-purpose method for a variety of operations, including tunneling. Consider the following:

```
# An XML-RPC message tunneled over HTTP POST
POST /RPC2 HTTP/1.1
Host: www.example.org
Content-Type: text/xml;charset=UTF-8

<methodCall>
  <methodName>messages.delete</methodName>
  <params>
    <param>
      <value><int>1234</int></value>
    </param>
  </params>
</methodCall>
```

* From Sec 9.5 of RFC 2616 (*http://tools.ietf.org/html/rfc2616#section-9.5*).

This is an example of XML-RPC (*http://www.xmlrpc.com/*) tunneling an operation via the `POST` method. Another popular example is SOAP with HTTP:

```
# A SOAP message tunneled over HTTP POST
POST /Messages HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=UTF-8

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:ns="http://www.example.org/messages">
    <ns:DeleteMessage>
      <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
  </soap:Body>
</soap:Envelope>
```

Both these approaches misuse the method `POST`. For this example, the `DELETE` method is more appropriate:

```
# Using DELETE
DELETE /message/1234 HTTP/1.1
Host: www.example.org
```

When there is no such direct mapping between the application's operations and HTTP, using `POST` has less severe consequences than overloading other HTTP methods.

In addition, the following situations force you to use `POST` even when `GET` is the right method to use:

- HTML clients like browsers use the URI of the page as the `Referer` header while making requests to fetch any linked resources. This may leak any sensitive information contained in the URI to external servers.

  In such cases, if using Transport Layer Security (TLS, a successor to SSL) or if the encryption of any sensitive information in the URI is not possible, consider using `POST` to serve HTML documents.
- As discussed in Recipe 8.3, `POST` may be the only option when queries from clients contain too many parameters.

Even in these conditions, use `POST` only as the last resort.

1.8 How to Create Resources Using POST

One of the applications of `POST` is to create new resources. The protocol is similar to using the "factory method pattern" for creating new objects.

Problem

You want to know how to create a new resource, what to include in the request, and what to include in the response.

Solution

Identify an existing resource as a factory for creating new resources. It is common practice to use a collection resource (see Recipe 2.3) as a factory, although you may use any resource.

Let the client submit a `POST` request with a representation of the resource to be created to the factory resource. Optionally support the `Slug` header to let clients suggest a name for the server to use as part of the URI of the resource to be created.

After creating the resource, return response code `201` (`Created`) and a `Location` header containing the URI of the newly created resource.

If the response body includes a complete representation of the newly created resource, include a `Content-Location` header containing the URI of the newly created resource.

Discussion

Consider the case of creating an address resource for a user. You can take the user resource as a factory to create a new address:

```
# Request
POST /user/smith HTTP/1.1
Slug: Home Address ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
Slug: Home Address ❷

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>

# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address ❸
Content-Location: http://www.example.org/user/smith/address/home_address ❹
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/home_address"/>
  <street>1, Main Street</stret>
  <city>Some City</city>
</address>
```

❶ User resource acting as a factory to create a home address resource

❷ A suggestion for naming the URI of the new resource

❸ URI of the newly created resource

❹ URI of representation in the response

In this example, the request contains data to create a new resource, and a `Slug` header with a suggestion for the URI of the new resource. Note that the `Slug` header is specified by AtomPub (RFC 5023). This header is just a suggestion from the client. The server need not honor it. See Chapter 6 to learn about AtomPub.

The status code of the response `201` indicates that the server created a new resource and assigned the URI `http://www.example.org/user/smith/address/home_address` to it, as indicated by the `Location` response header. The `Content-Location` header informs the client that the body of the representation can also be accessed via the URI value of this header.

Along with the `Content-Location` header, you can also include the `Last-Modified` and `ETag` headers of the newly created resource. See Chapter 10 to learn more about these headers.

1.9 When to Use PUT to Create New Resources

You can use either HTTP `POST` or HTTP `PUT` to create new resources. This recipe discusses when to use `PUT` to create new resources.

Problem

You want to know when to use `PUT` to create new resources.

Solution

Use `PUT` to create new resources only when clients can decide URIs of resources. Otherwise, use `POST`.

Discussion

Here is an example of a client using `PUT` to create a new resource:

```
# Request
PUT /user/smith/address/home_address HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

```
# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Location: http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/home_address"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

❶ Client using PUT to create a new resource

Use PUT to create new resources only when the client can control part of the URI. For instance, a storage server may allocate a root URI for each client and let clients create new resources using that root URI as a root directory on a filesystem. Otherwise, use POST.

When using POST to create new resources, the server decides the URI for the newly created resource. It can control its URI naming policies along with any network security-level configurations. You can still let servers use information in the representation (such as the Slug header) while generating URIs for new resources.

When you support PUT to create new resources, clients must be able to assign URIs for resources. When using this method to create new resources, take the following into consideration:

- To let clients assign URIs, the server needs to explain to clients how URIs on the server are organized, what kind of URIs are valid, and what kind are not.
- You also need to consider any security and filtering rules set up on servers based on URI patterns and may want to restrict clients to use a narrow range of URIs while creating new URIs.

In general, any resource that can be created via PUT can equivalently be created by using POST with a factory resource. Using a factory resource gives the server more control without explaining its URI naming rules. An exception is the case of servers providing a filesystem-like interface for clients to manage documents. WebDAV (see Recipe 11.4) is an example.

1.10 How to Use POST for Asynchronous Tasks

HTTP is a synchronous and stateless protocol. When a client submits a request to a server, the client expects an answer, whether the answer is a success or a failure. But this does not mean that the server must finish processing the request before returning a response. For example, in a banking application, when you initiate an account

transfer, the transfer may not happen until the next business day, and the client may be required to check for the status later. This recipe discusses how to use this method to process requests asynchronously.

Problem

You want to know how to implement `POST` requests that take too long to complete.

Solution

On receiving a `POST` request, create a new resource, and return status code `202` (`Accepted`) with a representation of the new resource. The purpose of this resource is to let a client track the status of the asynchronous task. Design this resource such that its representation includes the current status of the request and related information such as a time estimate.

When the client submits a `GET` request to the task resource, do one of the following depending on the current status of the request:

Still processing
: Return response code `200` (`OK`) and a representation of the task resource with the current status.

On successful completion
: Return response code `303` (`See Other`) and a `Location` header containing a URI of a resource that shows the outcome of the task.

On task failure
: Return response code `200` (`OK`) with a representation of the task resource informing that the resource creation has failed. Clients will need to read the body of the representation to find the reason for failure.

Discussion

Consider an image-processing web service offering services such as file conversions, optical character recognition, image cleanup, etc. To use this service, clients upload raw images. Depending on the nature and size of images uploaded and the current server load, the server may take from a few seconds up to several hours to process each image. Upon completion, client applications can view/download processed images.

Let's start with the client submitting a `POST` request to initiate a new image-processing task:

```
# Request
POST /images/tasks HTTP/1.1
Host: www.example.org
Content-Type: multipart/related; boundary=xyz

--xyz
Content-Type: application/xml;charset=UTF-8
```

```
...

--xyz
Content-Type: image/png

...

--xyz--
```

In this example, the client uses a multipart message, with the first part containing an XML document describing the kind of image-processing operations the server needs to perform and the second part containing the image to be processed.

Upon ensuring that the contents are valid and that the given image-processing request can be honored, let the server create a new task resource:

```
# Response
HTTP/1.1 202 Accepted  ❶
Content-Type: application/xml;charset=UTF-8
Content-Location: http://www.example.org/images/task/1
Date: Sun, 13 Sep 2009 01:49:27 GMT

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request has been accepted for processing.</message>
  <ping-after>2009-09-13T01:59:27Z</ping-after> ❷
</status>
```

❶ Response code indicating that the server accepted the request for processing

❷ A hint to check for the status at a later time

The client can subsequently send a `GET` request to this task resource. If the server is still processing the task, it can return the following response:

```
# Request
GET /images/task/1 HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request is currently being processed.</message>
  <ping-after>2009-09-13T02:09:27Z</ping-after>
</status>
```

See Recipe 3.9 to learn the rationale behind the choice of the date-time value for the `ping-after` element.

After the server successfully completes image processing, it can redirect the client to the result. In this example, the result is a new image resource:

```
# Request
GET /images/task/1 HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 303 See Other ❶
Location: http://www.example.org/images/1
Content-Location: http://www.example.org/images/task/1

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>done</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request has been processed.</message>
</status>
```

❶ See the target resource for the result.

The response code `303` merely states that the result exists at the URI indicated in the `Location` header. It does not mean that the resource at the request URI (e.g., `http://www.example.org/images/task/1`) has moved to a new location.

This representation informs the client that it needs to refer to `http://www.example.org/images/1` for the result. If, on the other hand, the server fails to complete the task, it can return the following:

```
# Request
GET /images/task/1 HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>failed</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Failed to complete the request.</message>
  <detail xml:lang="en">Invalid image format.</detail>
  <completed>2009-09-13T02:10:00Z</completed>
</status>
```

1.11 How to Use DELETE for Asynchronous Deletion

This recipe outlines an approach for using `DELETE` for asynchronous tasks. This recipe is appropriate when resource deletion takes a significant amount of time for cleanup and archival tasks in the backend.

Problem

You want to know how to implement `DELETE` requests that take too long to complete.

Solution

On receiving a `DELETE` request, create a new resource, and return `202` (`Accepted`) with the response containing a representation of this resource. Let the client use this resource to track the status. When the client submits a `GET` request to the task resource, return response code `200` (`OK`) with a representation showing the current status of the task.

Discussion

Supporting asynchronous resource deletion is even simpler than creating or updating resources. The following sequence of steps illustrates an implementation of this recipe:

1. To begin, a client submits a request to delete a resource.

```
DELETE /users/john HTTP/1.1
Host: www.example.org
```

2. The server creates a new resource and returns a representation indicating the status of the task.

```
HTTP/1.1 202 Accepted
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/task/1" rel="self"/>
  <message xml:lang="en">Your request has been accepted for processing.</message>
  <created>2009-07-05T03:10:00Z</ping>
  <ping-after>2009-07-05T03:15:00Z</ping-after>
</status>
```

3. The client can query the URI `http://www.example.org/task/1` to learn the status of the request.

You can use the same approach for asynchronously updating a resource via the `PUT` method.

1.12 When to Use Custom HTTP Methods

There were several attempts to extend HTTP with new methods. The most prominent attempt was WebDAV (*http://www.webdav.org*). WebDAV defines several HTTP

methods, such as PROPFIND, PROPPATCH, MOVE, LOCK, UNLOCK, etc., for distributed authoring and versioning of documents (see Recipe 11.4). Other examples include PATCH (Recipe 11.9) for partial updates and MERGE (*http://msdn.microsoft.com/en-us/library/cc668771.aspx*) for merging resources.

Problem

You want to know the consequences of using custom HTTP methods.

Solution

Avoid using nonstandard custom HTTP methods. When you introduce new methods, you cannot rely on off-the-shelf software that only knows about the standard HTTP methods.

Instead, design a controller (see Recipe 2.6) resource that can abstract such operations, and use HTTP method POST.

Discussion

The most important benefit of extending methods is that they let servers define clear semantics for those methods and keep the interface uniform. But unless widely supported, extension methods reduce interoperability.

For example, WebDAV defines the semantics of MOVE as a "logical equivalent of a copy (COPY), followed by consistency maintenance processing, followed by a delete of the source, where all three actions are performed atomically." Any client can submit an OPTIONS request to determine whether a WebDAV resource implements MOVE. When necessary, if a resource supports this method, the client can submit a MOVE request to move a resource from one location to another.

```
# Request to discover supported methods
OPTIONS /docs/annual_report HTTP/1.1
Host: www.example.org

# Response
HTTP/1.1. 204 No Content
Allow: GET, PUT, DELETE, MOVE

# Move
MOVE /docs/annual_report HTTP/1.1
Host: www.example.org
Destination: http://www.example.org/docs/annual_report_2009

# Response
HTTP/1.1 201 Created
Location: http://www.example.org/docs/annual_report_2009
```

It is certainly possible to follow WebDAV's approach and design a new method, say, `CLONE`, to create a clone of an existing resource:

```
# Request to clone
CLONE /po/1234 HTTP/1.1
Host: www.example.org

# Clone created
HTTP/1.1 201 Created
Location: www.example.org/po/5678
```

Clients will then be able to discover support for this method and submit a `CLONE` request.

In reality, proxies, caches, and HTTP libraries will treat such methods as nonidempotent, unsafe, and noncacheable. In other words, they apply the same rules to such extension methods as `POST`, which is nonidempotent, unsafe, and most often noncacheable. This is because idempotency and safety are guarantees that the server must explicitly provide. For unknown custom methods, proxies, caches, and HTTP libraries cannot assume that the server provides such guarantees. Therefore, for most HTTP-aware software and tools, custom HTTP methods are synonymous with `POST`.

```
# Request to clone
POST /clone-orders HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded

id=urn:example:po:1234

# Clone created
HTTP/1.1 201 Created
Location: www.example.org/po/5678
```

Moreover, not all HTTP software (including firewalls) may support arbitrary extension methods. Therefore, use custom methods only when wide interoperability is not a concern.

Prefer `POST` over custom HTTP methods. Not every HTTP software lets you use custom HTTP methods. Using `POST` is a safer option.

1.13 When and How to Use Custom HTTP Headers

It is not uncommon to find HTTP servers using custom headers. Some well-known custom headers include `X-Powered-By`, `X-Cache`, `X-Pingback`, `X-Forwarded-For`, and `X-HTTP-Method-Override`. HTTP does not prohibit such extension headers, but depending on what clients and servers use custom headers for, custom headers may impede interoperability. This recipe discusses when and how to use custom HTTP headers.

Problem

You want to know the common conventions and best practices for using custom HTTP headers.

Solution

Use custom headers for informational purposes. Implement clients and servers such that they do not fail when they do not find expected custom headers.

Avoid using custom HTTP headers to change the behavior of HTTP methods. Limit any behavior-changing headers to the method `POST`.

If the information you are conveying through a custom HTTP header is important for the correct interpretation of the request or response, include that information in the body of the request or response or the URI used for the request. Avoid custom headers for such usages.

Discussion

Most websites using the WordPress blogging platform (*http://wordpress.org*) include the following HTTP headers in responses:

```
X-Powered-By: PHP/5.2.6-2ubuntu4.2
X-Pingback: http://www.example.org/xmlrpc.php
```

Such headers are not part of HTTP. The first header is generated by the PHP runtime that WordPress is built on. It indicates that the server is using a particular version of PHP on Ubuntu. The `X-Pingback` header contains a URI that clients can use to notify WordPress when a reference is made on some other server to the resource. Similarly, HTTP caching proxy Squid uses `X-Cache` headers to inform clients whether the representation in the response is being served from the cache.

Such usages are informational. Clients receiving those headers are free to ignore them without loss of functionality. Another commonly used informational header is `X-Forwarded-By`.

```
X-Forwarded-For: 192.168.123.10, 192.168.123.14
```

The purpose of this header is to convey the source of the request to the server. Some proxies and caches add this header to report the source of the request to the server. In this example, the server received a request from `192.168.123.10` via `192.168.123.14`. If all proxies and caches that the request is served through augment this header, then the server can determine the IP address of the client.

Although names of some custom headers start with `X-`, there is no established convention for naming these headers. When you introduce custom headers, use a convention such as `X-{company-name}-{header-name}`.

The following custom HTTP headers are not informational and may be required for the correct processing of requests or responses:

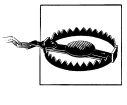
```
# A version number of the resource
X-Example-Version: 1.2

# An identifier for the client
X-Example-Client-Id: 12345

# An operation
X-Example-Update-Type: Overwrite
```

Avoid such usages. They weaken the use of URIs as resource identifiers and HTTP methods as operations.

Another commonly used custom header is `X-HTTP-Method-Override`. This header was initially used by Google as part of the Google Data Protocol (*http://code.google.com/apis/gdata/docs/2.0/basics.html*). Here is an example:

```
# Request
POST /user/john/address HTTP/1.1
X-HTTP-Method-Override: PUT
Content-Type: application/xml;charset=UTF-8

<address>
  <street>...</street>
  <city>...</city>
  <postal-code>...</postal-code>
</address>
```

In this case, the client uses `X-HTTP-Method-Override` with a value of `PUT` to override the behavior of the method used for the request, which is `POST`. The rationale for this extension was to tunnel the method `PUT` over `POST` so that any firewalls configured to block `PUT` will permit the request.



Instead of using `X-HTTP-Method-Override` to override `POST`, use a distinct resource to process the same request using `POST` without that header. Any HTTP intermediary between the client and the server may omit custom headers.