안녕하세요. 오늘 HTTP에 대해 발표할 발표자 이승민입니다. 발표의 내용은 주로 제가 공부를 하며 이해를 했던 부분으로 구성을 했습니다. HTTP에 대해 처음 공부하는 만큼 기본적인 사항에 집중하고 제가 이해하는 것을 기반으로 발표를 만들려고 노력했습니다. 그런데 웹을 구성하는 제일 중요한 구성요소가 HTTP, HTML, URL라고 해서 향상된 고급 개념에 대해서는 발표가 끝난 후에도 계속 셀프 스터디를 진행할 예정입니다. 목차를 보시면 기본 HTTP 개념과 SSL에 대해 발표를 한다고 써뒀는데, SSL은 보안 쪽에 가깝지만 기본 사항과 마찬가지로 중요하다고 생각하여 추가하게 되었습니다.

HTTP는 hyper text transfer protocol로, hyper text를 전송하는 규약입니다. 인터넷을 쓰다 보면 네트워크는 그냥 잘 작동이 된다고 생각하게 되는데 사실 네트워크에도 계층이 있고, 이 계층에는 데이터를 잘 전송하기 위해 지켜야 할 프로토콜이 있습니다. presentation이나 session을 넣은 조금 더 복잡한 네트워크 구조도 있지만 여기서는 어플리케이션 레이어, 트렌스포트 레이어, 네트워크 레이어, 링크 레이어, 피지컬 레이어가 있는 구조를 가져왔습니다. HTTP는 최상위층인 어플리케이션을 사용하며 트렌스포트레이어는 또 다른 중요한 전송 프로토콜인 TCP/UDP를 사용합니다. 네트워크 레이어는 IP 프로토콜은 사용하고 링크 레이어는 이더넷이며 피지컬레이어는 말 그대로 물리적인 하드웨어 선입니다. HTTP가 전송한다는 hyper text란 일반 텍스트와는 다르게 다른 문서들과 이어져있는 텍스트로, 한 페이지에서 링크를 눌렀을 때 다른 페이지로 갈 수 있는 텍스트를 말합니다. 그런데 HTTP가 HTTP라고 해서 하이퍼텍스트만을 전송하는 게 아니라 이미지나 html 문서, 동영상들과 같은 타입의 데이터도 전송을 합니다. 이 HTTP 프로토콜을 이용해서 소통을 하는 구조는 클라이언트와 서버로 구성되어있습니다. HTTP에는 리퀘스트와 리스폰스라는 두가지 타입이 존재하는데 클라이언트는 언제나 필요한 것이 있으면 서버와의 커넥션을 열어 리퀘스트 메세지를 보냅니다. 서버는 연결을 기다리고 있는 상태로, 클라이언트에게 연결요청이 오면 연결을 허용하고 리퀘스트에 담겨있는 요구사항을 읽어 그에 맞는 리스폰스 메시지를 작성합니다. 토렌트와 같은 peer to peer 형식은 양끝단이 둘 다 클라이언트일 수 있지만 그런 경우가 아니고서는 보통 클라이언트와 서버 구조로 소통을 합니다. 또한 클라이언트는 언제든지 IP 주소가 바뀌어도서버와 연결을 시작하는데 문제가 없지만, 서버 같은 경우는 클라이언트가 찾아올 수 있게 주소가 고정이 되어있어야 합니다. 그리고 언제 클라이언트의 요청이 올 지 모르기 때문에 항상 전원이 켜져 동작하고 있는 상태를 유지해야 합니다. 리퀘스트 메시지와 리스폰스 메시지에 대한 더 자세한 사항은 뒤에서 더 살펴보도록 하겠습니다.

아까 HTTP가 하이퍼텍스트 뿐 아니라 여러타입의 데이터를 전송한다고 말씀을 드렸는데 사실 그냥 데이터 자체를 전송하지 않고 인코딩 공정을 거치게 됩니다. 이때 어떤 형식의 파일인지 표현하기 위해 MIME-type형싱의 라벨링을 하게 됩니다. MIME 타입은 원래 전자이메일에서 사

용이 되는 형식이었는데 그 이전에 HTTP에서 사용하던 프로토콜이 ASCII 로 쓰여진 텍스트 파일만을 인코딩 할 수 있었기 때문에 ASCII가 아닌 이미지, 동영상과 같은 바이너리 타입의 데이터를 전송하기 위해 MIME을 사용하기 시작했습니다. 표에 여러가지 MIME 타입이 쓰여있는데 형식은 주타입 슬레쉬 부타입으로 이루어져있습니다. MIME 타입이 굉장히 중요한 이유는 리퀘스트 메시지와 리스폰스 메시지를 작성할 때 쓰는 헤더로 서버로부터 받아들일 수 있는 데이터 타입을 명시해둔 accept content와 바디의 데이터가 어떤 형식인지 써둔 content type 헤더에서 사용하는 타입이 MIME 타입이기 때문입니다. 표에는 나와있지 않는데 얼마전에 http로 요청을 보냈을 때 에러가 났고, 그때 멀티파트라는 타입이 있다는 사실을 알게 되었습니다. 멀티파트는 단어 그대로 여러가지의 형식을 가지고 있는 것인데 예를 들어 html 형식 사용자의 이름과 사진이 들어있어 텍스트와 이미지가 동시에 들어있는 상황과 같습니다. 표에서 json이 application/json과 text/json 두개가 있는데 application/json이 공식 MIME 타입이고, text/json은 application/json이 등록되기 전에 사용한 비공식 타입이라고 합니다. HTTP를 보면서 json에 대해 상당히 많이 들었기 때문에 json에 대해 추가로 몇가지를 알아보았습니다. Json은 자바스크립트 오브젝트 노테이션의 약자로 자바스크립트 기법으로 데이터를 기술한 데이터 표현형식 입니다. Json의 특징은 json이전에 많이 사용이 되던 xml과는 다르게 데이터 표현이 간결하고 자바 스크립트를 사용하는 웹 브라우저와 호환성이 좋다고 합니다. Application/json을 보면 뒤에 uft-8이라는 단어가 나오는데 이는 유니코드 인코딩 형식이라고 합니다. Uft-8은 url에 ascii가 아닌 한글 같은 언어를 넣을 때 인코딩을 하기 위해서도 사용이 되는 중요한 기능이고 추후에 더 깊게 공부를 하려고 합니다.

URL은 유니폼 리소스 아이덴티 파이어의 약자입니다. 웹 상의 셀 수 없이 많은 리소스는 서버 안에 저장되어 있거나 클라이언트의 리퀘스트에 따라서 서버에서 클라이언트로 전송되고 있습니다. 이런 리소스들을 하나씩 구분할 수 있도록 사용되는 것이 URI입니다. HTTP를 공부하기 전에는 URI가 얼마나 중요한 역할을 하는지 알 수 없었지만 URI가 없다면 리소스를 가리키는 지표가 없기 때문에 웹 상에 존재하는 그 리소스에 접근을 하는 방법이 없습니다. URI와 URL 개념이 많이 헷갈렸으나 결국 url은 유니폼 리소스 로케이션으로 리소스의 주소를 나타내고 있기 때문에 하나의 식별자로 기능을 할 수 있고, 따라서 URL이 URI의 하위개념이라고 이해를 하게 되었습니다. URI는 특정한 문법을 가지고 있습니다. 우선은 스키마 또는 스킴이라고 불리는 부분이 있는데, 이는 사용할 프로토콜의 종류를 알려줍니다. -그 뒤로 호스트명이 나오는데 호스트명은 DNS에 가서 이름을 해석할 수 있는 도메인이나 ip주소로 되어있어야 합니다. 제가 공부할 때 봤던 url 형식입니다. 최상위 도메인인 com 뒤에 오는 숫자는 포트번호이고, search는 경로를 나타내고 있습니다. ?는 구분문자로 경로와 쿼리파라메터를 구분합니다. 쿼리는 &로 이어집니다. 여기서 #은 프레그먼트라는데 처음에는 무슨 용도인지 잘 이해가 가지 않았으나, 이것은 html element의 아이디였습니다. #이 뒤에 왔을 경우에는 긴 웹페이지에서 그 아이디를 가진 엘리멘트로 이동하여 사용자가 열람을 원하는 부분으로 바로 접근할 수 있습니다. 그런데 이런 프레그먼트는 사실 http 리퀘스트에서 사용되지는 않고 브라우저에서만 사용하는 기능입니다.   
 유닉스 계열의 우분투를 사용했던 적이 있는데 그때 이용했던 절대경로와 상대경로의 개념이 URI에서도 있었습니다. 절대경로는 가고자하는 경로의 전체 uri를 써두는 것으로만 알고 있었는데있었는데변 있을 때 마다 모든 명세를 바꿔줘야 하는 불편함이 있었습니다. 상대경로는 이런 불편함은 없지만 항상 기준이 되는 url이 있어야 상대경로로 주어진 경로를 제대로 찾아가 리소스를 얻을 수 있습니다. 이 기준 url은 base url라고 하며, 상대경로를 사용하기 위해서는 꼭 주어져야 해서 html문서 안에 따로 명시해둔다고 합니다.

리퀘스트의 형식을 보도록 하겠습니다. api를 쓰면서도 리스폰스 메시지만 봤었지 리퀘스트 메시지의 형식은 볼 일이 없었기 때문에 이번을 기회로 좀 더 자세히 보려고 노력했습니다. 클라이언트에서 서버로 보내지는 리퀘스트 메시지는 이런 형식으로 쓰여집니다. (실제 예시 페이지 넘기기) 실제 리퀘스트 메시지의 형식입니다. 리퀘스트나 리스폰스가 이런 형태다, 하고 이론적으로 접근을 할 때는 주의를 기울이지 않았었는데, 실제로 api를 써보며 HTTP의 전송형태를 보니 이전까지 주위를 기울이지 않았던 헤더가 통신에서 굉장히 중요하다는 사실을 경험하게 되었습니다. 예시로 저는 리퀘스트 바디에 서버에 전달할 파라메터를 작성했는데 content-type 헤더는 json으로 명시를 해두고 실제로 전송한 데이터는 json이 아니라 ascii 형식이나 큰 바이너리 데이터 전송을 지원하는 form data 형식이라서 서버 측에서 응답으로 415 상태코드를 받았어야 했습니다. 여기서의 헤더는 클라이언트가 받아들일 수 있는 데이터 타입을 명시한 accept, 받아들일 수 있는 언어 타입을 명시한 accept-language가 나와있습니다. 이 헤더들을 보다보면 ‘q’라는 단어를 볼 수 있는데 무슨 뜻인지 검색을 해보니 q는 relative quality factor로, 여러 옵션이 있을 때 클라이언트가 선호하는 옵션을 표시하기 위해 존재했습니다. >> keep alive와 connection 헤더는 열려진 연결의 지속에 관련된 헤더입니다. 제가 자세히 본 부분은 Accept-encoding입니다. Accept-encoding은 클라이언트가 받아들일 수 있는 인코딩 타입입니다. 서버는 클라이언트에게 크기가 큰 데이터를 바디에 채워서 보내야 하는 때가 있는데 이때 사용하는 컨텐츠 인코딩을 사용합니다. 흥미로웠던 부분은 데이터가 인코딩이 되었을 때 원본 데이터의 content ㅣlength는 압축이 된 크기로 변경이 되는데, 나중에 클라이언트가 리스폰스를 받았을 때 디코딩을 하기 위해서 content type은 원본 데이터 정보를 그대로 사용하며 content-encoding이라는 헤더를 사용하여 데이터가 인코딩이 되었음을 알린다는 것 이었습니다. Content-encoding을 보다보니 transfer encoding이라는 헤더를 발견하게 되었습니다. 이는 트랜스퍼 인코딩을 위한 헤더였는데, 컨텐츠 인코딩이 리스폰스의 바디 부분만을 인코딩했다면 트렌스퍼 인코딩은 기본 헤더를 제외한 메시지 자체를 인코딩하는 형식이었습니다. 이런 인코딩을 하는 이유는 서버가 후가공을 해서 클라이언트에 보내야 하는 다이나믹 리소스의 경우는 처음부터 리소스의 크기를 알 수 없어 content length를 설정할 수 없는데, 원래 프로토콜에서는 클라이언트가 데이터의 끝을 알 수 있도록 서버에서 content length 헤더를 설정해줘야 합니다. 요즘의 웹페이지들은 대부분이 static 이 아니고 dynamic한 형식이고 서버에서 동적으로 생성이 되기 때문에 전송당시에 데이터 전체의 길이를 알 수 없습니다. 때문에 이 문제를 해결해야 하는데 이때의 해결책이 chunk로 나눠서 인코딩을 하는 transfer encoding입니다. 청크 인코딩은 메시지 전체를 정해진 크기의 덩어리로 잘라서 데이터를 다 전송할 때 까지 덩어리를 계속 클라이언트에게 보내는 기법인데 이러면 데이터의 끝부분에 보낼 데이터를 다 보냈다는 표시만 하면 되고 길이는 알지 못해도 되기 때문에 동적 리소스를 전송할 수 있습니다. 만약 서버가 transfer 인코딩을 했다면 리스폰스에 transfer encoding 이라는 헤더를 꼭 추가해 줘야 합니다. 콘텐츠 인코딩에서 클라이언트가 받을 수 있는 인코딩 타입을 accept encoding으로 명시한 것 처럼 transfer encoding에서도 클라이언트는 자기가 받을 수 있는 transfer encoding 타입을 TE 헤더에 넣어둡니다. 언급된 헤더 말고도 리퀘스트에서만 쓰이는 헤더, 리스폰스에서 쓰이는 헤더, 바디 부분에서 쓰이는 엔티티 헤더 등 중요한 헤더들이 많이 있는데 전부 다 살펴보지는 못해서 좀 더 공부를 해야 할 필요를 느꼈습니다.

메소드를 공부하면서 처음에 헤맨 부분이 있었기 때문에 그 점에 대해 말씀을 드리고자 메소드 페이지를 작성했습니다. GET, HEAD, TRACE, OPTIONS, DELETE의 역할을 이해하는 것에는 문제가 없었지만 제가 헷갈렸던 부분은 post와 put 메소드인데 처음에 post와 put 설명을 봤을 때 두 메소드 모두 서버에 데이터를 보내 리소스를 만들거나 업데이트한다고 하여 둘의 차이점을 찾을 수 없었습니다. 눈에 띄는 차이는 put은 한 번 서버로부터 리퀘스트가 이뤄지면 몇 번을 더 요청을 하여도 서버 안의 리소스에 변화가 없는 메소드이며, post는 만약 한 번 요청을 하고 또 다시 요청을 하면 서버에 있는 리소스에 변화가 오는 아이딤포텐트 메소드라는 사실입니다. 예를 들어 어떠한 리소스를 만든다고 한다면 put으로는 두번의 put이 와도 하나의 리소스만 만들어지고 post를 두 번 사용했다면 두 개의 리소스가 만들어지는 것입니다. 리소스 생성의 관점에 집중을 한다면 post는 생성할 리소스의 위치를 url로 지정하지 않고, put은 정확한 리소스의 위치를 제공하여, 그 곳에 리소스를 만드는 다는 것입니다. 뚜렷한 차이점을 알아내기 위해 인터넷을 계속 검색하다가 오라일리 출판사에서 나온 http 완벽 가이드에 써 있는 put은 서버에 문서를 쓸 때 사용하는 메소드이고 post는 서버에 특정한 파라메터 즉 아이디나 비밀번호등의 데이터를 전달하여 후에 서버에서 게이트웨이로 나가 필요한 작업을 하거나 디비에서 정보를 찾아오는 용도로 사용하는 메소드라고 이해를 하게 되었습니다. 그런데 검색 중에 rest api를 설계할 때 지원할 메소드를 용도에 따라 잘 정해두는 것이 정말로 중요하다는 사항을 보게 되어었어서 이보다 더 구체적인 차이점을 후에 꼭 익혀둬야겠다고 생각했습니다. 메소드의 종류에는 멱등/비멱등의 개념 외에도 safe 메소드라는 종류가 있었는데 서버에 리퀘스트를 보낼 때 서버 안의 리소스에 변화를 주지 않은 열람용 메소드 get, head, trace를 제외한 다른 메소드들은 안전하지 않은 메소드로 보고 있습니다.

리스폰스 메시지의 형태를 알아보도록 하겠습니다. 리스폰스의 형태는 이와 같습니다. 리스폰스 메시지에서 집중을 해서 봐야하는 헤더가 몇 개 있었습니다. 처음에 date는 중요치 않은 헤더라고 생각했는데 서버가 응답을 할 때 content length처럼 필수적으로 작성을 해줘야 하는 헤더였습니다. 클라이언트로부터 자주 리퀘스트가 들어오는 리소스는 middle agent인 프락시캐쉬 서버에 저장이 되는데 한 번 저장한 데이터를 영원히 사용할 경우도 있겠지만 그렇지 못하는 경우가 더 많습니다. 그래서 프락시 캐쉬서버도 클라이언트에게 최신의 변화가 적용이 되어있는 신선한 리스폰스를 전달하기 위해서 주기적으로 서버에 가서 저장된 리스폰스를 갱신하는데 이때 신선함 날짜를 계산하기 위해 처음 리스폰스가 만들어진 날짜를 하나의 기준으로 사용합니다. Last-modified와 Etag 또한 캐시 신선도 관리를 위한 헤더로 캐시 갱신의 조건절로 사용이 됩니다. Keep-alive 헤더는 리퀘스트 헤더에서도 나왔었는데, 파라미터인 timeout은 유효시간이었고, max는 받을 수 있는 리퀘스트의 최대치였습니다. Keep alive에서 흥미로웠던 사항은 저는 당연히 client와 server사이가 계속 연결이 된다고 생각하였으나, 실제로는 클라이언트와 서버 사이의 프록시들과 클라이언트, 서버가 모두 keep alive 헤더를 주고 받아 연결이 유지된다는 사실이었습니다. 다음으로 상태코드를 보겠습니다. 처음에 상태코드를 봤을 때는 100대는 정보, 200대는 성공, 300대는 리다이렉션, 400대는 클라이언트 문제 500대는 서버 문제라고 가볍게 생각하였습니다. 그런데 직접 rest api에 post 요청을 보내서 응답을 받아보니 상태코드는 제가 보낸 요청에 대한 서버의 처리를 단적으로 보여주는 아주 중요한 정보이고 특히 에러가 났을 때는 상태코드가 없었다면 직접 서버를 쳐다보지 않고 있는 이상, 무엇이 문제인지 전혀 알 수 없었음을 경험하였습니다. 제가 제일 많이 마주한 에러는 400에러와 415에러였는데 400에러는 서버에 리퀘스트를 보낼때 파라메터의 이름에 오타가 나서 나쁜 리퀘스트 형식이 되었고, 서버에 json 형식의 엔터티를 보내야 했었는데 form data, 즉 바이너리 데이터 형식을 보내게 되어 서버가 지원하지 않는 형식이라는 415 코드를 받았습니다. 상태코드를 받았을 때는 그때그때 검색을 할 수 있기 때문에 다행이라고 생각했을 때 300번대의 도저히 차이점을 모르겠는 상태코드들을 보게 되었습니다. 302와 303, 307이었습니다. 셋 다 URL이 임시적으로 바뀌었다며 location 헤더에 리다이렉션을 해야하는 URL을 넣어주는 역할인데 왜 똑 같은 상태코드를 세 개씩 만들었나 찾아보니 다른 점이 존재했습니다. 302는 post 리퀘스트를 리다이렉팅할 때 post 리퀘스트를 get으로 바꾸는 오류를 저지르고 있었습니다. 서버 자체가 리퀘스트 메소드를 바꾸는 것은 오류이기 때문에 302의 문제점을 해결하고자 303과 307을 만들어 냈습니다. 303 오류코드에서는 바뀐 url을 location에 주고, 클라이언트가 그 location 안에 있는 url로 다시 get 메소드를 보내야 합니다. Post 리퀘스트를 사용한 메소드 또한 같은 방법으로 동작합니다. 307 오류코드는 302가 원래 해야했던 역할을 수행합니다. 단 서버에서 리퀘스트의 메소드를 바꾸지 못하게 했기 때문에 post가 와도 그대로 post 리리퀘스트 형식을 유지할 수 있게 만들었습니다.

HTTP 연결도 정말 중요하지만 이해하는 부분에서 문제가 없었기 때문에 간단하게 제가 중요하게 생각했던 부분을 말씀드리고자 합니다. HTTP는 transfer 레이어의 프로토콜 중 도중에 누락된 데이터 없이 믿을 수 있게 전송을 하는 TCP와 함께 와야 합니다. 그런데 TCP가 확실성을 중요시하는 프로토콜이다보니 연결을 시작할 때 쓰리웨이 핸드세이킹을 하여 상대간의 연결이 확실함을 체크합니다. 이러다 보니 연결에 드는 시간이 길어지고, 웹페이지를 띄우기 위해 HTML과 이미지 등 여러 개의 리소스를 리퀘스트하고 받아야 하는 클라이언트의 입장에서는 너무나 늦은 속도의 통신이 되어버립니다. 그래서 생각해낸 연결 방식이 처음에는 병렬연결이었습니다. 원래는 하나의 시간대의 연결은 하나의 리퀘스트와 리스폰스만을 주고 받았다면 병렬 연결은 동시에 여러 커낵션을 열어서 리퀘스트와 리스폰스를 주고 받는 형식이었습니다. 그런데 병렬 연결을 통해서는 너무 많은 연결이 열려야 하고 이는 서버의 성능을 떨어뜨리기 때문에 새로운 방법을 강구하게 되었습니다. 그게 바로 지속 커넥션이었습니다. 한번 왔다갔다 하고 닫는 커넥션이 아니라 클라이언트나 서버가 연결을 닫자고 결정을하거나 시간이 지났을 때 연결이 닫히는 형태였습니다. 하나의 연결에 여러 번 데이터를 주고 받으면 연결을 시작하기 위한 쓰리웨이 핸드쉐이킹을 하지 않아도 되어 시간을 절약할 수 있었습니다. 지속 가능한 연결을 하기 위해서는 구 버전의 HTTP에서는 아까 말씀드렸던 keep alive 헤더를 사용했어야 했는데 HTTP/1.1 버전에서 부터는 모든 연결이 지속적임을 디폴트로 만들었습니다.

프록시는 클라이언트와 서버 사이에서 캐시를 저장해주거나, 보안, 필터링, 접근제어등의 중요한 일을 하고 있는 중간관리자입니다. 흥미로웠던 부분은 프록시서버는 클라이언트에게 서버처럼 행동하고 서버에게 클라이언트처럼 행동하는 사실이었습니다. 프록시 서버에 캐쉬를 저장하기 때문에 리퀘스트에 대한 리스폰스를 받을 때 물리적으로 먼 원서버에게 까지 가면 시간이 한참 걸리는데 중간에 프록시가 있어서 시간을 단축할 수 있고, 심지어 네트워크망을 제공하는 kt, skt등의 isp(인터넷 서비스 프로바이더)들이 고객이 가입한 서비스에 차등을 두어 높은 서버스를 이용하는 고객의 경우에는 가까운 프록시로 연결을 해준다는 사실에 놀랐습니다. 프록시처럼 클라이언트와 서버 사이에서 서로를 연결해주는 역할을 하는 기능으로 게이트웨이가 있는데, 클라이언트와 서버가 각기 다른 프로토콜을 사용할 때 연결을 해주는 중요한 동작을 합니다. 그런데 게이트웨이에 대해서는 아직 공부를 하지 못하여 추후에 꼭 다시 공부를 하도록 하겠습니다.

캐시에 대해 읽다가 문뜩 쿠키에 대해서 생각을 하게 되었습니다. HTTP는 stateless하기 때문에 연결이 끊기면 그 전의 연결에 대한 클라이언트와 서버의 정보가 아무것도 없었는데 쿠키는 이런 http의 성질을 극복하고 사용자에게 편의를 제공하고자 만들어진 사실은 알았지만 빠른접근을 위해 특정한 정보를 저장하는 성격이 캐쉬와 유사해 보였습니다. 캐시와 쿠키를 제대로 정리하지 않으면 나중에 혹시라도 혼선이 생길까 따로 공부를 하였습니다. 캐시는 나중에 클라이언트가 또 찾을 가능성이 있는 리스폰스 속의 html, 이미지등의 모든 타입의 리소스를 일정한 시간동안 프록시 서버에 저장을 해 둡니다. 이런 캐시의 예측이 적중하여 히트가 발생하면 클라이언트에게 빠르게 리퀘스트에 대한 리스폰스를 줄 수 있어서 클라이언트가 웹페이지를 빠르게 불러올 수 있게 도움을 줍니다. 그런데 이 캐시는 cache control이나 eTag, last modified, if modified since expire 등의 헤더를 사용하여 일정 기간이 지난 후 캐시를 지닌 프록시 서버가 원서버에 신선한 리소스를 요청하여 캐시의 상태를 업데이트합니다. 쿠키는 유저를 구분하는 특별한 식별자를 만들어 유저 맞춤 서비스를 제공하거나, 일정기간동안 세션이 유지되게 하여 매순간마다 사용자가 로그인을 할 필요가 없게 만듭니다. 서버는 클라이언트가 리퀘스트를 보냈을 때 그 클라이언트를 위한 쿠키를 set cookie 헤더에 넣어서 보내고, 쿠키를 받은 클라이언트는 앞으로 cookie 헤더에 서버로부터 받은 cookie를 넣어 리퀘스트를 보냅니다. 때문에 서버는 매번 클라이언트를 알아 볼 수 있습니다. 쿠키는 캐시와는 다르게 프록시 서버에 저장하지 않고 클라이언트의 웹브라우저를 통해 사용자의 pc에 직접 저장이 됩니다. 쿠키를 업데이트하는 경우는 거의 없으며 일정 시간이 지나면 쿠키가 사라진다고 합니다. 쿠키는 일종의 개인정보이기 때문에 프록시 캐시에는 set cookie .헤더가 저장이 되면 안 됩니다. Security socket layer.