1 – SOAP vs Restful ?

SOAP standardında, web servisi tanımlayan format olan WSDL ile Java, .Net ve SAP ABAP ortamlarında çok kolay hem server hem client üretilebilir. Bu yöntem "contract-first design" olarak adlandırılmaktadır. Program arayüzünü tanımlarken netlik ve basitlik hedeflerini güzel dengelediğini düşünüyorum.

REST servislerinin avantajı basitliğindedir. Server-Client ilişkisini netleştirmek, dokümante etmek ve otomatik kod oluşturma için Swagger, RAML gibi standartlar çıkıyor. Dokümantasyon konusunda çok işe yarıyorlar. Ancak "contract-first design" mantığıyla tüm mesaj yapısını tanımlarken karmaşıklıklarıyla bu standartların SOAP'tan farkı kalmamaktadır. Bu noktada REST servislerin SOAP'tan avantajlı olduğu tek nokta JSON kullanmak olabilir.

JSON'ın XML'e göre avantajları konusunda genellikle katılmadığım bir görüş performans avantajının önemi. JSON hem daha az yer kaplayarak network'te hem de yapısının basitliğiyle encode/decode(CPU) işlemlerinde daha hızlı. Ancak performans ile ilgili somut rakamlar, entegrasyonun mesaj boyutu ve frekansı incelenmeden karar verilmemelidir. Çoğu entegrasyonun bu performans farkına ihtiyaç duyduğunu düşünmüyorum. Performansın çok önemli olduğu entegrasyonlarda belki de Protocol Buffers kullanılmalı?

SOAP tercih edilmeli:

Kurumsal projelerde

Veri hataları ve servisin nasıl kullanılacağı ile ilgili iletişim azaltılmak istendiğinde

.Net, Java, SAP ABAP ile iletişim kuran entegrasyonlarda

REST tercih edilmeli:

Basit yapılar

Farklı dillerde yazılımcılar tarafından anlaşılması istenen entegrasyon Ör: Python, Ruby, PHP, C client yazmak daha kolaydır.

JSON kullanmanın çok avantajlı olması. Ör: Web sitesi arayüzü veya mobil uygulama tarafından kullanılacak bir servis.

Bunların dışında halihazırda var olan entegrasyonlarla tutarlılık ve yazılımcıların tercihleri değerlendirilmelidir.

Günümüzde kurumsal servislerde çoğunlukla SOAP tercih edilmektedir.

2 - Difference between acceptance test and functional test ?

Fonksiyonel test : Bu bir doğrulama faaliyetidir; doğru çalışan bir ürün mü ürettik? Yazılım iş gereksinimlerini karşılıyor mu?

Bu tür testler için, bu senaryonun "gerçek dünyada" var olma olasılığı düşük olsa bile, düşünebildiğimiz tüm olası senaryoları kapsayan test senaryolarımız var. Bu tür testleri yaparken maksimum kod kapsamını hedefliyoruz. O zaman yakalayabileceğimiz herhangi bir test ortamını kullanıyoruz, kullanılabilir olduğu sürece "üretim" kalibreli olması gerekmiyor.

Kabul testi : Bu bir doğrulama faaliyetidir; doğru olanı mı inşa ettik? Müşterinin gerçekten ihtiyacı olan bu mu?

Bu genellikle müşteri ile işbirliği içinde veya dahili bir müşteri temsilcisi (ürün sahibi) tarafından yapılır. Bu tür testler için, yazılımın kullanılmasını beklediğimiz tipik senaryoları kapsayan test senaryoları kullanıyoruz. Bu test, "üretim benzeri" bir ortamda, müşterinin kullanacağıyla aynı veya ona yakın donanım üzerinde gerçekleştirilmelidir. Bu, "ility'lerimizi" test ettiğimiz zamandır:

Güvenilirlik, Kullanılabilirlik : Bir stres testi ile doğrulanmıştır.

Ölçeklenebilirlik : Bir yük testi ile doğrulanmıştır.

Kullanılabilirlik : Bir inceleme ve müşteriye gösterim yoluyla doğrulanmıştır. Kullanıcı arayüzü beğenilerine göre yapılandırılmış mı? Müşteri markasını tüm doğru yerlere yerleştirdik mi? İstedikleri tüm alanlara/ekranlara sahip miyiz?

Güvenlik (aka, Güvenlilik, tam uyum sağlamak için) : Gösterim yoluyla doğrulandı. Bazen bir müşteri, güvenlik denetimi ve/veya izinsiz giriş testi yapmak için dışarıdan bir firmayı işe alır.

Sürdürülebilirlik : Yazılım güncellemelerini/yamalarını nasıl sunacağımızın gösterilmesi yoluyla doğrulanmıştır.

Yapılandırılabilirlik : Müşterinin sistemi ihtiyaçlarına göre nasıl değiştirebileceğinin gösterilmesi yoluyla doğrulanır.

3 - What is Mocking ?

Mocking(mocklama), popüler yazılım metodolojisi olan TDD ve özelde birim testlerinin (unit test), test ettikleri sistemi izole etmede kullandığı yöntemlerden biridir. Bu yöntemler, geniş anlamıyla test dublörleri (test double) olarak tanımlanabilir. Test dublörleri, test edilen sistemin bağımlı olduğu diğer birimlerin yerini tutar. Bu izolasyona birim testlerinde ihtiyaç duyulmasının temelde iki sebebi vardır:

Birim testleri, genelde test ettikleri sistemin kendisi ile ilgili varsayımları doğrulamak için yazılır.

Test dublörleri, davranış ve kullanım şekillerine göre çeşitlenir. Bunlardan en çok kullanılanları dummy, fake, stub, spy ve mock‘tur denebilir. Bu çeşitliliğe sebep olan genel faktörler, bu dublörlerin beklenen işi yapıp yapmadığı ve yaparken nasıl bir davranış gösterdiği ile ilgilidir.

NOT: 1. madde ile ilgili olarak; entegrasyonu testleri, izole halde çalışan bağımsız sistemlerin bir araya getirildiğinde ortaya çıkan durumlarını inceler. Birim testlerinin istenilen çalışma sürelerinin kısa olması da bir diğer faktör olarak sayılabilir.

Mock’ların genel kullanım şekli, yerine geçtiği bağımlılık üzerinde, test edilen sistemin yapması beklenen işlemlerin yapılıp yapılmadığını doğrulamak olarak tanımlanabilir. Mock nesnelerinin casus nesnelerinden (spy) farkı, her ikisi de üzerlerinde yapılan işlemleri takip ederken, mocklar bu işlemi testlerin doğrulama (assert) kısmına da entegre ederler.

Mocklama işlemi genelde kütüphaneler yardımıyla, test metodlarının içinde, veya tekrar eden test ayarlarının yapıldığı bir özelleştirme metodunu, satır arası kodlar ile yapılır. Yani çoğu zaman mocklanan tipden devralan bir tip yazılmaz. Mock kütüphaneleri genelde bu işi, dilin reflection kütüphanesinden faydalanarak, çalışma zamanında, ayarlanan kurulumu sağlayacak vekil tipler üreterek sağlarlar. Bunun getirdiği bir avantaj, bütün ön koşulları yerine getiren bir dublör birimi yazmadan, her test metodu için yalnızca beklenen davranışı sağlayan satır içi ayarlamalar yapılmasına olanak sağlamasıdır.

Mock kütüphanelerinin diğer bir avantajı ise, genellikle kendi içlerinde doğrulama (assertion) mekanizmaları bulundurmaları ve mocklanan nesne üzerindeki beklentilerin karşılanıp karşılanmamasına göre, çağırıldıkları testin başarı durumunu etkilemeleridir. Mock kütüphanelerinin yaptığı bu işi, elle yazılan mock tiplerinde geliştiricinin kendisi yapması gerekebilir.

Mocklama işlemi yapılırken genelde bağımlı olan birimle ilgili yapılması beklenen çağırmalarla ilgili ayarların yapılmasının yanı sıra, bazen test edilen sistemin doğru çalışması için, bu çağırmaların geriye değerler döndürmesi gerekmektedir. Bu durumları çözmek için, geliştirilen veya ayarlanan mock tipinin, test edilen tipi rahatsız etmeyecek bir değer döndürmesi gerekebilir. Mock kütüphaneleri böyle durumlarda yapılan ayarlamalarda geriye değer döndürmeye de olanak sağlar.

Dezavantajlar

Mock nesnelerini doğru yerlerde kullanmanın çeşitli avantajları olduğu gibi, dezavantajları da olabiliyor.

Bir dezavantaj test ortamında mock kullanılacak bir sistemde neredeyse her şeyin birer arayüz (interface) üzerine inşa edilmesi gerekebiliyor. Bu da kimi zaman over-engineering olarak nitelendirilen probleme yol açabiliyor. Arayüz gerekmeyen yerlerde sınıfların kullanılması da sadece mocklamanın yapılabilmesi için normal şartlarda son ve kesin halinde olan metodların (örneğin C# için), virtual olarak işaretlenmesini gerektirebiliyor. Sınıflardan devralarak yazılan mocklar için bir başka sorun ise constructor'ların her durumda çağrılması, örneğin veritabanına constructor'ında bağlantı sağlayan bir sınıfın üzerine inşa edilen bir mock nesnesi de kullanmayacağı halde yine bu bağlantıyı kurmak zorunda kalabiliyor.

Mock ile ilgili bir başka dezavantaj, test ortamını kimi zaman karmaşıklaştırması olabilir. Bağımlı olunan sınıf yirmi satırlık bir kod barındırabilirken, test ortamının gerekliliklerini sağlamak için yüzlerce satır kod ile bahsedilen sınıfı taklit eden bir mock objesi yazmak gerekebiliyor. Bu karmaşıklığı çözmek için, "kendini tekrar etme" prensibini kullanmak gerekebiliyor. Bunun için test metodlarında parametreler ve bu parametreleri çözümleyen "fixture" kütüphanelerini kullanılabilir. Örneğin, .Net Framework için AutoFixture kütüphanesi, AutoMoq adında bir alt kütüphane ile, test metodlarına yer alan arayüzleri Moq kütüphanesinin Mock nesnelerini kullanarak çözümleyebiliyor.

4 - What is a reasonable code coverage % for unit tests (and why) ?

%100 . Her şeyin test edildiğinden emin olmak istediğiniz için bunu seçebilirsiniz. Bu size test kalitesi hakkında herhangi bir fikir vermez, ancak bazı kalite testlerinin her ifadeye (veya şubeye vb.) dokunduğunu söyler. Yine, bu güven derecesine geri döner: Kapsamınız %100'ün altındaysa , kodunuzun bazı alt kümelerinin test edilmediğini biliyorsunuz .

Bazıları bunun aptalca olduğunu iddia edebilir ve sadece kodunuzun gerçekten önemli olan kısımlarını test etmelisiniz. Ayrıca, kodunuzun yalnızca gerçekten önemli olan kısımlarını da korumanız gerektiğini savunuyorum. Test edilmemiş kodlar da kaldırılarak kod kapsamı iyileştirilebilir.

%99 (veya %95, doksanların yükseklerindeki diğer rakamlar.) %100'e benzer bir güven düzeyi ifade etmek istediğiniz , ancak ara sıra test edilmesi zor olan köşe hakkında endişelenmemek için kendinize biraz marj bırakın. kod.

%80 . Bu numarayı birkaç kez kullanımda gördüm ve nereden geldiğini tam olarak bilmiyorum. 80-20 kuralının garip bir şekilde kötüye kullanılabileceğini düşünüyorum ; genel olarak, buradaki amaç , kodunuzun çoğunun test edildiğini göstermektir. (Evet, %51 aynı zamanda "en çok" olur, ancak %80 çoğu insanın en çok neyi kastettiğini daha iyi yansıtır .) Bu, "iyi test edilmiş"in yüksek bir öncelik olmadığı (sizin yapmadığınız) orta düzeydeki durumlar için uygundur. Düşük değerli testler için çaba harcamak istemezsiniz), ancak yine de bazı standartların olmasını istediğiniz bir öncelik için yeterlidir.

Pratikte %80'in altındaki sayıları görmedim ve bunların yerleştirileceği bir durumu hayal etmekte zorlanıyorum. Bu standartların rolü, doğruluk konusundaki güveni artırmaktır ve %80'in altındaki rakamlar özellikle güven verici değildir. (Evet, bu özneldir, ancak yine de fikir, standardı belirlediğinizde bir kez öznel seçim yapmak ve ardından ileriye dönük nesnel bir ölçüm kullanmaktır.)

5 – HTTP/POST vs HTTP/PUT ?

POST ve PUT ikiside sunucuya veri göndermek için kullanılan HTTP metodlarıdır. POST sadece belirli bir kaynağa veri göndermek için kullanılır ve veri ile ne iş yapılacağı server'a bağlıdır. PUT ise aynı kaynağa aynı adres ile erişilir ve eğer içerik var ise gelen veriler ile değiştirilir , eğer içerik yok ise yeni içerik yaratılır. PUT ile daha çok server'a dosya bazlı içerikler göndermek için tercih edilir.

6 - What are the Safe and Unsafe methods of HTTP ?

Güvenli yöntemler, kaynakları değiştirmeyen HTTP yöntemleridir. Örneğin, bir kaynak URL'sinde GET veya HEAD kullanılması , kaynağı ASLA değiştirmemelidir. Ancak, bu tamamen doğru değil. Bunun anlamı: kaynak gösterimini değiştirmeyecek. Güvenli yöntemlerin bir sunucu veya kaynak üzerindeki şeyleri değiştirmesi hala mümkündür, ancak bu farklı bir temsili yansıtmamalıdır.

Bu, blog gönderisini gerçekten silecekse, aşağıdakilerin yanlış olduğu anlamına gelir:

GET /blog/1234/delete HTTP/1.1

Güvenli yöntemler, kaynağa herhangi bir etkisi olmadan önbelleğe alınabilen, önceden getirilebilen yöntemlerdir.

Idempotent yöntemler

Idempotent bir HTTP yöntemi, farklı sonuçlar olmadan birçok kez çağrılabilen bir HTTP yöntemidir. Yöntemin yalnızca bir kez veya on kez çağrılması fark etmez. Sonuç aynı olmalıdır. Yine, bu sadece sonuç için geçerlidir, kaynağın kendisi için değil. Bu yine de manipüle edilebilir (güncelleme zaman damgası gibi, bu bilgilerin (geçerli) kaynak gösteriminde paylaşılmaması şartıyla.

Aşağıdaki örnekleri göz önünde bulundurun:

```
a = 4;
a++;
```

İlk örnek idempotenttir: Bu ifadeyi kaç kez çalıştırırsak çalıştıralım, a her zaman 4 olacaktır. İkinci örnek idempotent değildir. Bunu 10 kez çalıştırmak, 5 kez çalıştırmaktan farklı bir sonuca yol açacaktır. Her iki örnek de a'nın değerini değiştirdiğinden, ikisi de güvenli olmayan yöntemlerdir.

Bağımsızlık, hataya dayanıklı bir API oluşturmada önemlidir. BIR ISTEMCININ BIR KAYNAĞI POST aracılığıyla güncellemek istediğini varsayalım . POST bağımsız bir yöntem olmadığından, onu birden çok kez çağırmak yanlış güncellemelere neden olabilir . POST isteğini sunucuya gönderirseniz , ancak bir zaman aşımı alırsanız ne olur? Kaynak gerçekten güncellendi mi? Sunucuya istek gönderilirken veya istemciye yanıt gönderilirken zaman aşımı

oldu mu? Güvenli bir şekilde tekrar deneyebilir miyiz, yoksa önce kaynağa ne olduğunu anlamamız mı gerekiyor? İdempotent yöntemleri kullanarak, bu soruyu yanıtlamak zorunda değiliz, ancak sunucudan gerçekten bir yanıt alana kadar isteği güvenle yeniden gönderebiliriz.

Güvenli yöntemlerle uğraşırken de dikkatli olun: GET gibi görünüşte güvenli bir yöntem bir kaynağı değiştirecekse, sizinle sunucu arasındaki herhangi bir ara katman istemci proxy sisteminin bu yanıtı önbelleğe alması mümkün olabilir. Bu kaynağı aynı URL üzerinden değiştirmek isteyen başka bir istemci (örneğin: http :// örnek .org / api / makale / 1234 / delete), sunucuyu aramaz , bilgiyi doğrudan önbellekten döndürür. Güvenli olmayan (ve bağımsız olmayan) yöntemler hiçbir zaman ara yazılım proxy'leri tarafından önbelleğe alınmaz

7 - How does HTTP Basic Authentication work ?

HTTP temel kimlik doğrulama, bir sunucunun bir istemciden kimlik doğrulama bilgisi (bir kullanıcı kimliği ve parola) isteyebileceği basit bir sorgulama ve yanıt mekanizmasıdır. İstemci, kimlik doğrulama bilgilerini bir Yetkilendirme başlığında sunucuya iletir. Kimlik doğrulama bilgileri, base-64 kodlamasındadır.

Not HTTP temel kimlik doğrulama şeması, yalnızca web istemcisi ile sunucu arasındaki bağlantı güvenli olduğunda güvenli kabul edilebilir. Bağlantı güvenli değilse, şema, yetkisiz kullanıcıların bir sunucu için kimlik doğrulama bilgilerini keşfetmesini önlemek için yeterli güvenliği sağlamaz. Bir parolanın ele geçirilebileceğini düşünüyorsanız, kullanıcı kimliğini ve parolayı korumak için SSL şifrelemeli temel kimlik doğrulamasını kullanın.

Bir istemci, sunucunun kimlik doğrulama bilgisi beklediği bir istekte bulunursa, sunucu bir 401 durum kodu, bir kimlik doğrulama hatasını gösteren bir neden ifadesi ve bir WWW-Authenticate başlığı ile bir HTTP yanıtı gönderir. Çoğu web istemcisi, bu yanıtı son kullanıcıdan bir kullanıcı kimliği ve parola isteyerek gerçekleştirir.

HTTP temel kimlik doğrulaması için WWW-Authenticate üstbilgisinin biçimi şöyledir:

WWW-Authenticate: Basic realm="Sitemiz"

WWW-Authenticate başlığı, kullanıcı kimliğinin ve parolanın uygulanacağı kaynak kümesini tanımlayan bir bölge özniteliği içerir. Web istemcileri bu

dizeyi son kullanıcıya görüntüler. Her bölge farklı kimlik doğrulama bilgileri gerektirebilir. Web istemcileri, son kullanıcıların her istek için bilgileri yeniden yazmalarına gerek kalmaması için her bölge için kimlik doğrulama bilgilerini depolayabilir.

Web istemcisi bir kullanıcı kimliği ve parola aldığında, orijinal isteği bir Yetkilendirme başlığıyla yeniden gönderir. Alternatif olarak, istemci orijinal isteğini yaptığında Yetkilendirme başlığını gönderebilir ve bu başlık sunucu tarafından kabul edilebilir, böylece sorgulama ve yanıt sürecinden kaçınılabilir.

Yetkilendirme başlığının biçimi:

Yetkilendirme: Temel kullanıcı kimliği:şifre

8 - Define RestTemplate in Spring ?

RestTemplate, client tarafında senkronize HTTP isteklerini yürütmek için Spring kütüphanesi içindeki default sınıftır. Rest servislerin yaygınlaşmasından bu yana çoğu geliştirici, Rest servislerini çağırmak için spring-boot-starter-web paketindeki Spring'in geleneksel RestTemplate'iyle çalışmaya alıştı. Spring ayrıca spring-boot-starter-webflux paketinde WebClient adlı bir sınıfa sahiptir. Bu yazımız, RestTemplate'den WebClient'e geçiş yapmanız gerekip gerekmediğine karar vermenize yardımcı olacaktır.

RestTemplate/WebClient Avantajları ve Dezavantajları

RestTemplate

RestTemplate thread-safe bir yapıdadır.

RestTemplate servlet yapısı üzerine inşa edilmiştir. Bu yüzden thread-per-request yaklaşımını izler. Sonuç olarak uygulama, thread-pool'u tüketecek veya tüm kullanılabilir belleği kaplayacak birçok thread oluşturacaktır. Bu sebeple performans problemleri ortaya çıkacaktır.

Spring 5 ile birlikte bakım moduna geçmiştir. Muhtemelen ilerleyen sürümlerde desteklenmeyecektir.

WebClient

WebClient thread-safe çünkü immutable’dır.

Senkron ve asenkron iletişimi destekler

Reaktif yapı üzerine kurulmuştur.

Spring 5 ile birlikte hayatımıza girmiştir. Bu sebeple ilerleyen sürümlerde desteklenecektir.

Bu iki yaklaşım arasındaki belirgin farkları görebilmek için, birçok eşzamanlı client isteğiyle performans testleri yapmamız gerekir. Belirli sayıda client isteğinden sonra RestTemplate’in thread-pool yoracağı yada tüm memory tüketerek önemli bir performans düşüşüne sebep olduğunu görürüz. Öte yandan reaktif yöntem, istek sayısına bakılmaksızın sürekli performanslı bir şekilde çalışmaya devam edecektir.

RestTemplate/WebClient Implementasyonları

RestTemplate

RestTemplate uygulanması oldukça basit ve sade bir yapıya sahiptir. Servis adresini, hangi HTTP yöntemi ile operasyonun gerçekleştirileceğini ve dönüş türünün ne olacağını veriyoruz.

```
ResponseEntity<Foo> response = restTemplate.exchange(uri, HttpMethod.GET, null, new ParameterizedTypeReference<Foo>(){});      List<Foo> result = response.getBody();
```

WebClient

WebClient ise biraz karmaşık gelebilir. Öncelikle hedef adresle bir WebClient örneği oluştururuz,

```
this.webClient = WebClient.builder().baseUrl("http://localhost:8080").build();
```

ardından gereksinimlerimize göre gerekli parametreleri yada headerları ekleyerek onu modifiye ederiz.

```
public Flux<Foo> getFoos() {
```

```
 return webClient.get().uri("/foo")
                         .retrieve()
                         .bodyToFlux(Foo.class);
}public Mono<Foo> getFooById(int id) {
 return webClient.get().uri("/foo/{id}",id)
                         .retrieve()
                         .bodyToMono(Foo.class);
}public Mono<Foo> createRecipe(Mono<Foo> foo) {
 return webClient.post().uri("/foo")
                         .body(foo,Foo.class)
                         .retrieve()
                         .bodyToMono(Foo.class);
}public Mono<Void> deleteFoo(int id) {
 return webClient.delete().uri("/foo/{id}", id)
                           .retrieve()
                           .bodyToMono(Void.class);
}public Mono<Foo> getFooByTitle(String title) {
 Map<String, String> requestParameters = new HashMap<>();
 requestParameters.put("title", title);
 return webClient.get().uri("/foo", requestParameters)
                         .retrieve()
                         .bodyToMono(Foo.class);
}
```

Diyelim harici servisimiz reaktif değil. Ancak yine de WebClient kullanmak istiyoruz. O zaman Flux ve Mono bizim için pek işe yaramaz, bu yüzden onları açmak zorunda kalacağız. Bu sebeple block() kullanabiliriz.

    Bunun reaktif bir ortamda kullanılmaması gerektiğini unutmayın.

```
public List<Foo> getFoos() {

 Mono<List<Foo>> fooListStream = webClient.get()
```

```
                                        .uri("/foo")
                                        .retrieve()
                                        .bodyToMono(
                    new ParameterizedTypeReference<List<Foo>>(){});

 List<Foo> fooList = fooListStream.block();

 return fooList;
}
```

9 – What is the difference between @Controller and @RestController ?

Bu iki annotasyon Spring Boot ‘ta Controller sınıfların oluşturulmasında kullanılır. Her iki annotasyon ile çalışan sınıfın URL ile istenen request’lere ait dönüş biçimlerini belirler.

Controller, Spring içerisinde uzun zamandır bulunan bir annotation. Diğer yandan ise @RestController, Spring 4.0 ile framework bünyesinde katıldı. Temel amacı adından da anladığımız gibi

REST endpoint’leri üretmeyi sağlamak. Aslında bunun bize sağladığı şey @Controller ve @ResponseBody notasyonlarını tek seferde vermesi. Bu sayede REST controller olmasını isteyeceğimiz her metoda @ResponseBody yazmaktan da kurtulmuş oluyoruz

10 – What is DNS Spoofing ? How to prevent ?

DNS spoofing diğer adıyla DNS önbellek zehirlenmesi, Alan Adı Sistemi (İngilizce: Domain Name System) verisini bozarak, DNS çözümleme önbelleğine bozuk verinin yerleştirildiği bir bilgisayar güvenliği saldırısıdır. Ad sunucusunun yanlış sonuç dönmesini sağlar, örneğin IP adresi. Böylece saldırgan, trafiği kendi bilgisayarına (ya da başka bir bilgisayara) yönlendirebilir.

DNS sunucularına yönelik birçok önbellek zehirlenmesi saldırısı, diğer DNS sunucuları tarafından kendilerine iletilen bilgilere daha az güvenerek ve sorgu ile doğrudan ilgili olmayan DNS kayıtlarını göz ardı ederek önlenebilir. Örneğin, BIND 9.5.0-P1 [1] ve daha üst sürümleri bu kontrolleri gerçekleştirir.[1] Kriptografik olarak güvenli rastgele sayıları ve DNS isteklerinin geldiği kaynak bağlantı noktasının rastgele olmasını sağlayan, bir 16-bit şifreleme özünü ve kaynak bağlantı noktası,

başarılı DNS zehirlenme saldırılarının olasılığını büyük ölçüde azaltabilir.

Bununla birlikte, yönlendiriciler, güvenlik duvarları, proxy'ler, ağ adresi çevirisi (NAT) gerçekleştiren diğer ağ geçidi cihazları veya daha özel olarak bağlantı noktası adresi çevirisi (PAT), bağlantı durumunu izlemek için kaynak bağlantı noktalarını yeniden yazabilirler. Kaynak bağlantı noktaları değiştirilirken PAT cihazları, ad sunucuları ve saplama çözümleyicileri [2] tarafından uygulanan kaynak bağlantı noktası rastgeleliğini kaldırabilir.

Güvenli DNS (DNSSEC), verilerin doğruluğunu belirlemek için güvenilir bir ortak anahtar sertifikasıyla imzalanan şifreli dijital imzaları kullanır. DNSSEC, önbellek zehirlenmesi saldırılarına karşı koyabilir ve 2010 yılında İnternet kök bölgesi sunucularına uygulandı, ancak 2008'den beri İnternet çapında yaygınlaşamamıştır.[1]

Bu tür bir saldırı, bağlantı kurulduktan sonra uçtan uca doğrulama gerçekleştirilerek taşıma katmanında veya uygulama katmanında hafifletilebilir. Bunun yaygın bir örneği, Taşıma Katmanı Güvenliği ve dijital imzaların kullanılmasıdır. Örneğin, HTTPS (güvenli HTTP sürümü) kullanarak, kullanıcılar sunucunun dijital sertifikasının geçerli olup olmadığını ve bir web sitesinin beklenen sahibine ait olup olmadığını kontrol edebilirler. Benzer şekilde, güvenli kabuk uzaktan oturum açma programı, oturuma devam etmeden önce bitiş noktalarındaki (biliniyorsa) dijital sertifikaları kontrol eder veya güncellemeleri otomatik olarak indiren uygulamalar imzalama sertifikasının bir kopyasını güncelleme ile bilgisayara gömebilir ve yazılım güncellemesinde gelen imzayı gömülü sertifikaya göre doğrulayabilir.

11 – What is content negotiation ?

Content Negotiation işleyişi HTTP protokolüne özgü bir kavramdır. Anlam olarak tercüme edecek olursak, client ve server arasında yapılan bir içerik anlaşması veya müzakeresidir diyebiliriz. Amacı, aynı URI ile farklı döküman türlerinde içerik sunabilmektir. Yani daha genel bir ifadeyle kaynak gösterim şeklinin kullanıcılar tarafından belirlenmesi diyebiliriz. Content Negotiation ile ilgili resmi dökümanlar buradaki w3.org sayfasında bulunmaktadır.Kullanıcıların sunucu kaynaklarına ulaşmak için kullandıkları internet tarayıcıları kendi yetenek türlerine göre kaynak türünü seçmek için HTTP protokolünde belirlenmiş olan kurallara uygun talepte bulunurlar. Örneğin bir X tarayıcısı JPEG dökümanlarını işleyebilecek yeteneği yoksa fakat PNG dökümanlarını işleyebiliyorsa sorgu sırasında bu isteğini HTTP Accept Header (istek başlığı) bilgisi olarak sunucuya iletir. Örneğin

Accept istek başlığı ile sunucuya kullanıcının medya türü tercihini Accept:image/png şeklinde belirtir.

Burada Accept istek başlıkları superset/subset biçiminde temsil edilmektedir. Örneğin Accep:image/png örneğinde “image“ superset’tir, “png“ ise subset’tir.

Örnek medya türleri için Accept Header bilgileri:

```
Accept: application/json
Accept: image/png
Accept: image/*
Accept: text/xml
```

Accept istek başlığında birden fazla tercih de belirtilebilir. Örneğin:

```
Accept: text/html, text/xml, image/jpeg, */*
```

Burada talebi gönderen taraf text/html, text/xml, image/jpeg ile açıkça belirtilmiş medya türlerinin tercih ettiğini, bunun yanında */* ile farklı medya türleri varsa onları da kabul edebileceğini ifade etmektedir. Burada sunulan bir dizi medya türü vardır. Bu medya türlerine üstünlük katsayısı vererek öncelik tercihi yapmak da mümkündür. Örneğin:

```
Accept: text/html, text/xml, image/jpeg, */* ; q=0.01
```

Üstünlük katsayısı 1.0 ve 0.0 arasında bir değer alır ve “q” ile gösterilir. Burada text/html gibi üstünlük katsayısı belirtilmemiş türlerin katsayısı otomatik olarak 1.0 atanır. Ancak */* kalıbı 0.01 gibi düşük bir öncelik olarak belirlenmiştir.

İnternet tarayıcıları medya tiplerine göre kendi öncelik katsayılarını belirlerler. İnternet tarayıcıları kullanmadan fiddler gibi bir araç yardımıyla HTTP talepleri oluşturup, Header seçeneklerini kendimiz belirleyebiliriz.

Bir kaynaktan gelen dökümanın medya türü farklı olabileceği gibi dil seçimi de farklı olabilmektedir. Aynı şekilde Accept Header bilgisinde istediğimiz dili belirtebiliriz. Örneğin, Accept-Language: tr şeklinde.

İçerikle ilgili bir diğer Header bilgisi ise Content-Type şeklinde belirtilen ve kaynağa erişmek isteyen kullanıcının hangi medya türünde döküman istediğini bildiren içerik bilgisidir. Örneğin Content-Type:application/json şeklinde belirlenen bir Header bilgisi kullanıcının JSON veri istediğini belirtmektedir.

Günümüzde kaynak kullanımı sadece tarayıcılar tarafından değil, mobil uygulamalar tarafından da ağırlıklı olarak kullanıldığı göz önüne alındığında Content Negotiation kavramı özellikle REST servisler oluşturulurken dikkat edilmesi gereken konulardan bir tanesidir. Çünkü REST servisleri HTTP tabanlı çalıştıkları için servis geliştiriciler olarak HTTP dünyasını ve kurallarını iyi tanımamız gerekmektedir. Bu sayede neyi neden kullandığımızı bilerek ilerleyebiliriz.

12 – What is statelessness in RESTful Web Services ?

REST istemci-sunucu arasında hızlı ve kolay şekilde iletişim kurulmasını sağlayan bir servis yapısıdır. Açılımı Representational State Transfer olan bu ifadeyi Türkçe'ye Temsili Durum Transferi diye çevirebiliriz. Bu yazıda REST'in genel mimarisini, nasıl çalıştığını, artılarını, eksilerini ve SOAP ile arasındaki farklılıklarını irdelemeye çalışacağız.

restfulREST, servis yönelimli mimari üzerine oluşturulan yazılımlarda kullanılan bir veri transfer yöntemidir. HTTP üzerinde çalışır ve diğer alternatiflere göre daha basittir, minimum içerikle veri alıp gönderdiği için de daha hızlıdır. İstemci ve sunucu arasında XML veya JSON verilerini taşıyarak uygulamaların haberleşmesini sağlar. REST standartlarına uygun yazılan web servislerine RESTful servisler diyoruz. Yani RESTful denildiğinde aklınıza çok farklı bir kavram veya dünya gelmesin.

REST stateless'dır, yani durum bilgisini saklamaz. Biraz daha açalım; REST standartlarında istemci-sunucu arasında taşınan verilerde ekstra başlık bilgileri saklanmaz, istemciye ait detaylar bulunmaz, bu bilgiler istemci-sunucu arasında taşınmaz. Dolayısıyla servis yönelimli uygulamalarda REST bize lightweight bir çözüm yapısı sunar.

Teorik olarak SOAP bir protokol, REST ise bir kurallar dizisidir. Bu iki terimi doğrudan kıyaslamak biraz yanlış olsa da, SOAP ile geliştirilen bir

uygulama ile REST ile geliştirilen bir uygulamayı kıyaslamanın REST'in daha iyi anlaşılmasını sağlamak için uygun bir yol olacağını düşünüyorum. O zaman kıyaslayalım:

SOAP servisleri RPC(Remote Process Call yani uzaktaki bir prosedürün çağrılması) çalışma yöntemini kullanır, WS-* gibi güvenlik protokollerini içerisinde barındırır, state bilgisini request ve response'larda saklar. Ancak REST'te bu durum daha farklıdır. REST servisler doğrudan bir URL çağrılarak çalışır, arada ek bir bileşen, yöntem veya protokol kullanılmaz.

SOAP bir servisi uygulamanıza dahil edebilmeniz için servisin WSDL'ına ihtiyaç duyarsınız, proxy sınıfları oluşturmanız gerekir, uzaktaki metotları tetikleyecek bileşenlere ihtiyaç vardır. DISCO, UDDI vs. derken aslında işin arka planında baya detay olduğunu görürsünüz. Yani özet olarak istemci, SOAP bir servisle ilgili herşeyi bilmek zorundadır, belirli standartları yerine getirilmeden SOAP bir servisi çağıramaz. Ancak REST ile yazılmış bir servisle çalışmak için ihtiyacınız olan tek şey URL. Bir URL'yi çağırırsınız, URL size JSON veya XML döndürür, dönen cevabı parse edersiniz ve servis entegrasyonunuz tamamlanır. Yani teorik olarak istemci uygulama REST bir servisin yapısını ve detaylarını bilmek zorunda değildir. REST'in bu basit standartları dışında uyulması gereken bir kural yoktur, son derece esnek bir yapı vardır.

RESTful bir servisi çağırmak için karşınızdaki kurum size www.siteadi.com/product/12345 gibi bir URL verir ve bu adresi çağırdığınızda ID değeri 12345 olan ürünün detaylarının size JSON olarak döneceğini söyler. Size kalan tek iş bu URL'yi back-end'de WebRequest vb. sınıflarla veya client-side'da AJAX fonksiyonları vasıtasıyla çağırmak ve gelen JSON verisini uygun formatta görüntülemek olacaktır.

Yazı için internetten grafik, resim vb. yazıya renk katacak bir unsur ararken aşağıdaki resmi buldum. REST ve SOAP arasındaki fark için çok anlamlı bir benzetme olmuş. İngilizce metni okumadan da resim çok şey anlatıyor bence.

restful-vs-soap

RESTful servisler veri iletiminde farklı HTTP metotlarını kullanmaktadır. Bunlar GET, POST, PUT, DELETE metotlarıdır. GET okuma, POST yeni kayıt ekleme(insert), PUT kayıt güncelleme(update), DELETE ise kayıt silme işlemi için kullanılır. Yapılan HTTP request'i için çağrılan URL ile beraber HTTP method bilgisi bahsi geçen 4 metottan biri olarak seçilir ve sunucu yapılan talebin kayıt üzerine nasıl etki edeceğini buna göre belirler.

Basit yapısı, kolay uygulanması, hızlı çalışması, esnek olması… bunlar RESTful servislerin artı yönleri. RESTful servislerinin bazı eksi yönleri de var tabii ki. Güvenlik bunlardan biri. SOAP servislerde standart lar gereği birçok güvenlik mekanizması otomatik olarak elinizin altındadır. Ancak RESTful servislerde güvenlik konuları geliştirilen yazılımın bir parçasıdır. İletişim seviyesinde güvenlik(transport level security) genellikle token aracılığıyla yapılır. İstemci kritik işlemleri çağırmadan önce bir login isteği gönderir. Bu istek sonucunda istemciye sessin token vb. bir değer verilir ve bundan sonra yapacağı istekler bu token değeri ile yapılır. Mesaj seviyesinde güvenlik(message level security) konusu da yine geliştirilen yazılımların içerisinde çözüm bulunması gereken bir konudur. Tabii ki bazı üçüncü parti araçları kullanarak hem iletişim hem de mesaj seviyesinde güvenlik fonksiyonlarını yazılımlarınıza daha kolay şekilde uygulayabilirsiniz.

13 - What is CSRF attack? How to prevent ?

CSRF (Cross Site Request Forgery) genel yapı olarak sitenin açığından faydalanarak siteye sanki o kullanıcıymış gibi erişerek işlem yapmasını sağlar.

Genellikle GET requesleri ve SESSION işlemlerinin doğru kontrol edilememesi durumlarındaki açıklardan saldırganların faydalanmasını sağlamaktadır.

.Aşağıdaki kodda saldırgan img etiketini kullanarak yani urli bir imaj urli olarak göstererek sisteme sanki daha önce kayıt olmuş bir kullanıcı gibi giriş yapmak istemektedir.

<img src=”abc.php?giris=dogru”>

Bu tür açıkları kapatmak için en pratik yol ise token kullanımıdır.

<?php

$_SESSION[“token”] = sha1(rand());

echo'<a href="abc.php?giris=dogru&token=.'$_SESSION["token"]."';

abc.php dosyasındaki session kontrolü ise şu şekilde olmalıdır:

```
<?php

if($_GET[“giris”]==dogru){

if(isset($_GET[“token”])&&$_GET[“token”]==$_SESSION[“token”]){

session_start();//Dogruysa oturumu baslat

}

else{

echo “token yanlış!”;

}

}
```

14 - What are the core components of the HTTP request and HTTP response ?

Dinlendirici API'leri anlamanın yolu HTTP request ve response'ların istemci ile sunucu arasındaki iletişimi oluşturan bileşenleri anlamaktır. HTTP çekme protokolü olarakda tanınabilir. İletişim herzaman istemciden başlar, istemci sunucuya bir HTTP Request gönderir. Buna karşılık sunucu, istemcinin isteklerine bağlı olarak bir HTTP Response gönderir.

HTTP Request

Her HTPP Request'i bir Message Header ve Optional (isteğe bağlı) body içerir. Bir HTTP Requestinde ilk satıra İstek Satırı denir. Tanımlayıcı olarakda adlandırılabilir. Örnek verirsek;

GET / home.html HTTP/1.1

POST / index.html HTTP/1.1

Bu iki örnek istek satırlarıdır. GET bir verb, home.html kısımları URI, sonrası ise kullanılan HTTP versiondur. İstek Satırından sonra Request'e bağlı Request Header'larına sahibiz. Bu Header'lar bize bazı özellikleri tanıtmakla yükümlüdür. Request name & Value'ler burda tanımlanır. name:value olarak istekde görünür. Birden fazla olması durumunda virgüller ile ayrılır. name:value,name:value olarak. Bunlardan sonra bir aralık gelir ve daha sonra optional olan Body gelir. Body, Request'in bilgisidir.

HTTP Response

Server HTTP Request aldıktan sonra Client'e HTTP Response gönderir. Bir cevap niteliğindedir. Response istenilen bilgilerde verebilir, 403,404 gibi bir takım Error'larda. Yani hatalar. Yapı olarak Request ile aunıdır. Response bir Message Header ve Optional Body ' den oluşur. Header'ın ilk satırına Status Line (Durum Satırı) denir.

Status Line : HTTP Version, Status Code, Reason Phrases

Durum Satırı : HTTP Versiyonundan, Durum Kodundan ve Durum Kodunu Açıklayan Neden Cümlesinden Oluşur.