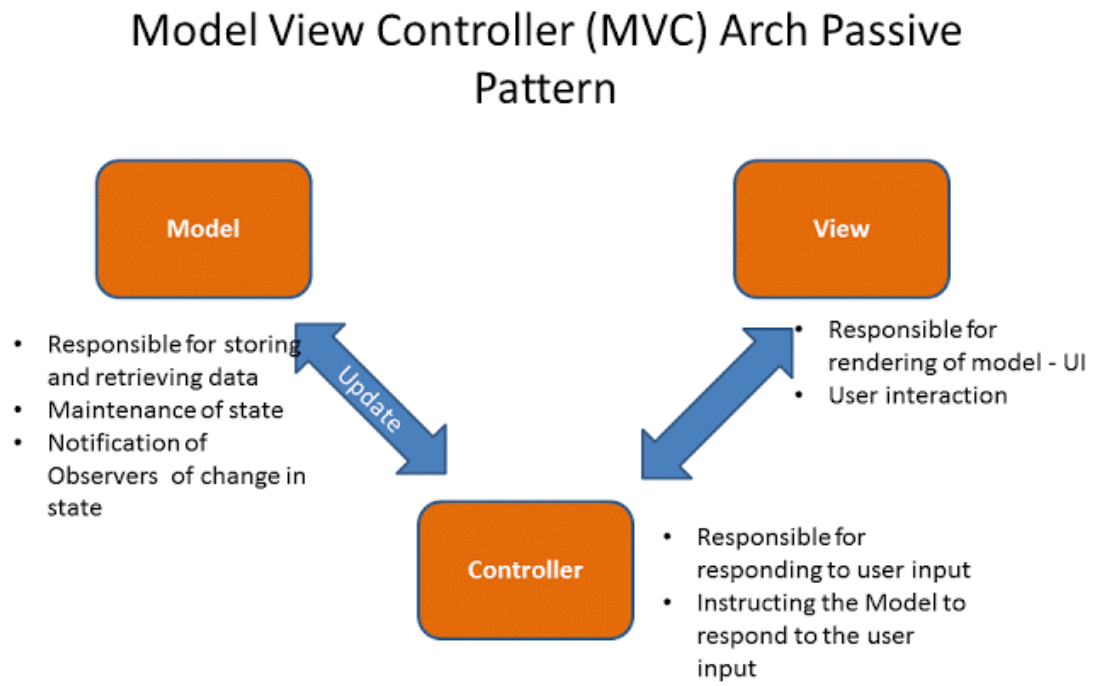
Spring MVC (Spring Web)

Model-View-Controller(MVC) Design Pattern



HTTP

- Açılımı Hytper-Text transfer protocol
- Internette kullanılan ana protokol

Farklı HTTP methodları

- Farklı operasyonlar için farklı HTTP methodları kullanılması daha uygundur. 4 tane en yaygın kullanılan methodu vardır
  1. GET: Bir veriyi değiştirmeden erişmek için kullanılır
  2. POST: Yeni bir veri oluşturmak için kullanılır
  3. PUT: Mevcut verinin güncellenmesi için kullanılır
  4. DELETE: Bir veriyi silmek için kullanılır

Bir HTTP isteğinin kısımları

- 4 kısımdan oluşur:
  1. Request satırı: HTTP methodunu, isteğin yollandığı URL'i ve protokol versiyonunu barındırır
  2. Header'lar: Request hakkındaki yan bilgiler
  3. Header'ların bittiğini belirten bir satır boşluk
  4. Request body: İsteğe bağlı bir alan. Asıl yollanmak istenen veri bu kısma konur.

Örnek HTTP request

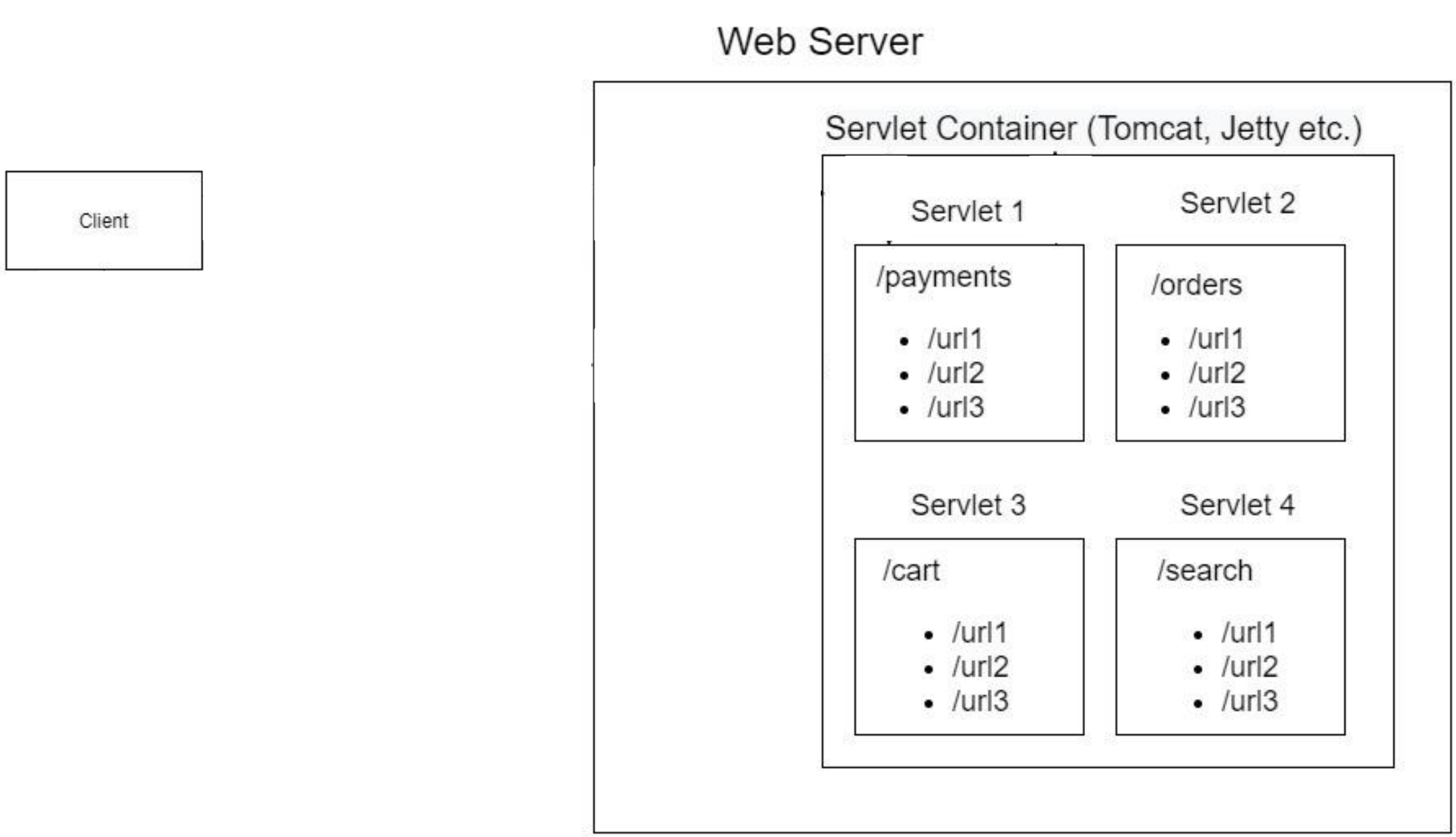
```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

Tomcat

- Tomcat 8080 port'unda çalışıp 80 port'unu dinleyen bir uygulamadır. 80 portuna bir istek geldiğinde tomcat bu isteği işler, ve bunu ilgili servlet'e paslar.
- Tomcat aynı zamanda bir 'servlet container' olarak da adlandırılır, zira birden fazla servlet'i barındırabilir.

Servlet

- Servlet özünde web server'ı ile bizim uygulamamız arasında köprü kuran bir class'tır. Request'in işlenmesindeki bütün pis işleri yapıp, bize rahatça anlayabileceğimiz ve kullanabileceğimiz bir formatta sunar.
- Spring tam olarak bu noktada geliyor. Servlet tek bir class değildir, java camiası tarafından tanımlanan bir standarttır. Bir servlet'i implement etmenin tek bir yöntemi yoktur, bu standartlara uyan her class bir servlet'tir. Spring'in de DispatcherServlet adında bir servlet implementation'ı vardır ve bize application yazma konusunda yardımcı olur.
- Servlet, kendisine gelen bir request'in hangi fonksiyona gitmesi gerektiğine karar ve spring'in diğer elemanları ile çalışarak bizim elimize işlenmesi kolay bir halde sunar.


Client
Web Server
Servlet Container (Tomcat, Jetty etc.)
Servlet 1
/payments
/url1
/url2
/url3
Servlet 2
/orders
/url1
/url2
/url3
Servlet 3
/cart
/url1
/url2
/url3
Servlet 4
/search
/url1
/url2
/url3

Request alma

- Bir web application’ın bel kemiği request alma ve bunlara cevap verebilmeden oluşur. Biz de ilk bölümde spring ile bir HTTP request’i nasıl handle edilire bakacağız.

@Controller ve @RequestMapping

- @Controller annotation'ı, client'tan gelen istekleri karşılamakla görevli class'ların başına konur. Stereotype anntation'dır
- @RequestMapping, gelen request'in URL'ine bağlı olarak hangi class/fonksiyona gitmesi gerektiğini spring'e söylemek için kullanılır

- Request'i www.test.com/index/welcome adresine yollamak gerekir.

```
@Controller
@RequestMapping("/index")
public class TestController {

    @RequestMapping("/welcome")
    public String index() {
        return "welcome.html";
    }
}
```

Özelleşmiş Request Mappings

- @GetMapping
- @PostMapping
- @DeleteMapping
- @PutMapping
- Bunlar şuna eşittir:
- @RequestMapping(path =‘/something’, method=RequestMethod.GET)

@RequestParam

- URL'deki query paramterlerini almak için kullanılır(ör: /test?name=hebele&surname=hubele)
- Almak istediğiniz parametrenin adını annotation'a vermeniz gerekir, veya önüne koyduğunuz variable'ın isminin bire bir aynı olması gerekir.

```
@Controller
public class TestController {

    @RequestMapping("/test")
    public void test(@RequestParam String name, @RequestParam String surname) {
        System.out.println(name + ' ' + surname);
    }
}
```

@PathVariable

- URL'in kendisindeki template halindeki değişkenleri almak için kullanılır
- Örnek: /courses/15

```
@Controller
public class TestController {

    @RequestMapping("/courses/{courseNumber}")
    public void test(@PathVariable Long courseNumber) {
        System.out.println(courseNumber);
    }
}
```

Uygulama

- Aşağıdaki dependency'leri ekleyin:
  - Spring Web Starter
  - Lombok
  - Validation
  - Spring Boot Devtools
- /courses/15?name=testName&instructor=testPerson
- '15 testName testPerson' yazan bir controller function'ı yazın

Request'lere cevap verme

- View ile
- JSON ile

View ile cevap

```
@Controller
public class TestController {

    @RequestMapping("/index")
    public String index() {
        return "index";
    }
}
```

JSON ile cevap

- İlk olarak, JSON nedir?
- Açılımı Javascript Object Notation

```
{
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

JSON dönme

- Direk bir POJO dönebilirsiniz. Spring arkada sizin objelerinizi JSON'a dönüştürür.

```
class Student {
    private String name;
    private Long age;
    private String email;

    public String getName() {
        return name;
    }

    public Long getAge() {
        return age;
    }

    public String getEmail() {
        return email;
    }
}
```

```
{
   "name": "test",
   "age": 24,
   "email": "something@something.com"
}
```

@ResponseBody

```
@Controller
public class TestController {

    @RequestMapping("/welcome")
    @ResponseBody
    public Student index() {
        return new Student( name: "asdsa", age: 45L, email: "asdas");
    }
}
```

ResponseEntity

- Status code, header, content type gibi request'in body'si ile ilgili olmayan diğer parametrelerin oluşturulmasını kolaşlaştırmak için kullanılır.
- Döneceğimiz objeyi ResponseEntity ile sararak meta data bilgileri kolaylıkla client'a dönebiliriz
- Ayrıca içinde bulunan builder ile tüm değerleri tek tek atamak yerine tek satırda cevabımızın meta data'larını atayabiliriz

ResponseEntity kullanımı

```java
@RestController
public class TestController {

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String test() {
        return "Ben bir testim!";
    }

}
```

```java
@RestController
public class TestController {

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public ResponseEntity test() {
        return ResponseEntity.status(HttpStatus.OK)
                .header( headerName: "hataVarMi",  ...headerValues: "false")
                .eTag("tag")
                .contentType(MediaType.APPLICATION_JSON)
                .body("Ben bir testim!");
    }

}
```

Request’ler ile beraber JSON alma

- Nasıl JSON dönüleceğini gördük. Fakat gelen request’lerdeki JSON’ları nasıl alırız?
- Bunun için @RequestBody kullanıyoruz.
- Bir handler function parametresinin başına @RequestBody koyduğunuz zaman, spring, request’in body’sindeki içeriği otomatik olarak sizin verdiğiniz class’a dönüştürmeye çalışır. Önemli olan nokta JSON ile yolladığınız key isimlerini dönüştürmek istediğiniz class’taki field isimleri ile aynı olmak ZORUNDA.

```json
{
  "name": "test",
  "age": 24,
  "email": "something@something.com"
}
```

```java
@RestController
public class TestController {

    @RequestMapping("/welcome")
    public void index(@RequestBody Student student) {
        System.out.println(student.getName() + ' ' + student.getAge() + ' ' + student.getEmail());
    }
}
```

Uygulama

- Bir Person class'ı oluşturun. Bu class'ta isim, soy isim, ve yaşı bulunsun.
- 3 tane rastgele Person oluşturun ve bunları postman'den bir controller'a bir list of objects olarak yollayın
- Daha sonra controller'da her kişinin yaşına 1 ekleyin, ve değiştirilmiş listeyi tekrar geri dönün

Java Bean Validation

- Gelen request'teki parametrelerin belli koşullara uyup uymadığını kontrol ederiz.
- Request objesi henüz elimize gelmeden bazı kriterlere uyduğundan emin oluruz, ve kriterlere uymayan request'ler için client'a özelleştirilmiş mesajlar dönebiliriz

Constraint annotation'ları

Annotation	Açıklaması
@Null, @NotNull	Null check'leri
@AssertTrue, @AssertFalse	Boolean check'leri
@Min, @Max	Değer check'leri
@DecimalMin, @DecimalMax	Decimal numberlar için değer check'i
@Negative, @NegativeOrZero, @Positive, @PositiveOrZero	Pozitif-negatif check'leri
@Size	Bir String'in uzunluğunun belli aralıkta olup olmadığına bakar
@Past, @PastOrPresent, @Future, @FutureOrPresent	Tarih kontrolleri için kullanılır
@Pattern	String'in verilen pattern'da olup olmadığını kontrol eder
@NotEmpty	String'in null veya boş olmadığından emin olur
@NotBlank	@NotEmpty ile benzer
@Email	Email kontrolü

Constraint kullanımı

```java
public class User {

    @NotBlank(message = "Ad boş olamaz!")
    private String name;
    @NotBlank(message = "Soy ad boş olamaz!")
    private String surname;
}
```

```java
@Component
public class MyFirstBean {

    @Size(max = 255, min = 10, message = "Dönülen mesajın uzunluğu 10 ve 255 arası olmalı!")
    public String print() {
        return "hebele";
    }
}
```

```java
@Component
public class MyFirstBean {

    public void print(@Max(value = 255, message = "Değer en fazla 255 olabilir") Integer value) {
        System.out.println(value);
    }
}
```

@Valid annotation’ı

```java
@RestController
public class TestController {

    @PostMapping("/test")
    public void addUser(@Valid User user) {
        System.out.println(user);
    }
}
```

Custom validator

- Bazen, kendi validation mantığımızı kendimizin yazması gerekebiliyor.
- Bunun için 2 farklı çözüm düşünülebilir.
  1. @AssertTrue ve @AssertFalse ile class içerisindeki fonksiyonlar ile geçerleme yapmak
  2. Yeni bir annotation oluşturup bu annotation'a bir class ile bir logic tanımlamak

@AssertTrue ve @AssertFalse

```java
public class User {

    @NotBlank(message = "Ad boş olamaz!")
    private String name;
    @NotBlank(message = "Soy ad boş olamaz!")
    private String surname;

    @AssertTrue
    public boolean isNameValid() {
        return name.length() < 255;
    }
}
```

Yeni annotation

```java
@Documented
@Constraint(validatedBy = TCKimlikNoValidator.class)
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TCKimlikNo {
}
```

```java
public class TCKimlikNoValidator implements ConstraintValidator<TCKimlikNo, String> {

    public void initialize(TCKimlikNo constraint) {
    }

    public boolean isValid(String tcKimlikNo, ConstraintValidatorContext context) {
        return false;
    }
}
```

```java
public class User {

    @TCKimlikNo
    private String tcKimlikNo;
}
```

Uygulama

- Bir User class'ı oluşturun
- İçerisinde şu alanlar bulunmalı: İsim, soyisim, yaş, e-mail adresi, TC kimlik numarası, doğum tarihi, ikametgah adresi ve kullanıcı adı olmalı
- İsim, soyisim, kullanıcı adı, e-mail adresi boş olmamalı
- Yaşı, 12 ile 100 arası olmalı
- E-mail adresi valid olmalı
- TC kimlik numarası formata uygun olmalı(Annotation ile yapılmalı)
- Doğum tarihi bugün veya daha öncesinde olmalı
- İkametgah adresi 250 karakterden kısa olmalı
- Kullanıcı adı «admin» olmamalı

Exception handling

- Application'dan fırlatılan exception'ları düzgün bir şekilde handle edebilmek için spring'in geliştirdiği bazı yöntemler var. Bunları iki ana kısıma bölebiliriz:
  1. Controller'a özgü exception handler'lar
  2. Global exception handler'lar

Controller’a özgü exception handling

```java
@RestController
public class TestController {

    @GetMapping("/test")
    public List<String> test() {
        throw new IllegalArgumentException();
    }

    @ExceptionHandler(value = {IllegalArgumentException.class})
    public String handleException(IllegalArgumentException exception) {
        return "Hatalı İstek!";
    }
}
```

Global Exception Handling

```java
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(value = {IllegalArgumentException.class})
    public String handleException(IllegalArgumentException exception) {
        return "Hatalı İstek!";
    }
}
```

Uygulama

- Validation hatası alındığında hangi exception fırlatıldığını bulun(postman'de yazıyor)
- Bu exception'ı global bir exception handler içerisinde yakalayın ve annotation içerisine yazdığınız mesajın veya mesajların geri dönülmesini sağlayın

RestTemplate

- Java ile request yollamak her zaman başa bela olmuştur
- Spring, bu sorunu çözmek için güzel bir yöntem sunmakta, RestTemplate, RequestEntity ve UriComponentsBuilder ile REST isteği atmak çok kolay hale geliyor
- RestTempleta'te pek çok farklı fonksiyonlar var, ama benim en faydalı bulduğum RequestEntity ve return type alan exchange fonksiyonu.

```java
@Component
public class MyFirstBean {

    public void sendRestRequest() {
        RestTemplate restTemplate = new RestTemplate();
        URI uri = UriComponentsBuilder.fromHttpUrl("http://www.google.com/maps")
                .pathSegment("1")
                .queryParam( name: "lon",  ...values: 45.458)
                .queryParam( name: "lat",  ...values: 48.5845)
                .build()
                .toUri();

        RequestEntity requestEntity = RequestEntity.get(uri).header( headerName: "key", ...headerValues: "value")
                .build();

        ResponseEntity<String> exchange = restTemplate.exchange(requestEntity, String.class);
        System.out.println(exchange);
    }
}
```

```java
@Component
public class MyFirstBean {

    public void sendRestRequest() {
        RestTemplate restTemplate = new RestTemplate();
        RequestEntity<Student> requestEntity = RequestEntity.post(URI.create("http://www.google.com/maps"))
                .header( headerName: "key", ...headerValues: "value")
                .contentType(MediaType.APPLICATION_JSON)
                .body(new Student());

        ResponseEntity<String> exchange = restTemplate.exchange(requestEntity, String.class);
        System.out.println(exchange);
    }
}
```

UriComponentsBuilder faydalı methodları

Fonksiyon adı	Açıklama
fromHttpUrl(String)	Verilen bir strin'i taban alarak URI oluşturmaya başlar
pathSegment(String…)	Mevcut URL'in sonuna bir / koyarak verilen string(ler)i append eder
queryParam(String, String)	Mevcut URL'sin sonuna verilen key-value ikilisi ile bir query parametresi ekler
queryParams(MultiValueMap)	Verilen map'teki tüm key-value'ları query parametresi olarak ekler
toUriString()	Builder'dan bir URI string'i oluşturur
fragment(String)	URL'in sonundaki fragment(#'ten sonraki değeri) atar
encode()	ASCII olmayan ve URL'de izin verilmeyen karakterleri UTF-8 ile encode ederek URL'de kullanılabilir hale getirir
build()	Build etme işlemini bitirir ve bir UriComponent döner

RequestEntity faydalı methodları

Fonksiyon ismi	Açıklama
get, post, put, delete, head, patch, options(URI)	RequestEntity'nin static builder methodları. RequestEntity oluşturmak için taban methodlar
header(String, String…)	Verilen key değerine value değer(ler)ini ekler
contentType(MediaType)	Gönderilen body'nin türünü ayarlar
contentLength(long)	Gönderilen isteğin uzunluğunu ayarlar
body(T)	Get harici isteklerde gönderilmek üzeri body'i ayarlar. Herhangi bir obje verilebilir
build()	RequestEntity'yi oluşturur

Uygulama

- İçerisinde userId, id, title ve body bulunduran Post class'ı ve içerisinde postId, id, name, email ve body bulunduran Comment class'ını oluşturun.
- Aşağıdaki request'leri RestTemplate, UriComponentsBuilder ve RequestEntity kullanarak jsonplaceholder'a yollayın ve sonuçlarını konsola basın.
  1. Tüm postları getirin
  2. İd'si 2 olan post'u getirin.
  3. Yeni rastgele bir post ekleyin
  4. userId'si 2 olan post'ları getirin
  5. İd'si 5 olan comment'i güncelleyin
  6. İd'si 1 olan post'u silin