Report

No .1

By Amir Lotfi

Contents

Rest Architecture

در تعریف استاندارد از سوی Roy Fielding یک **معماری** برای پیاده سازی سیستم های توزیعی مبتنی بر انواع داده های مورد نیاز کاربران، از جمله داده های تصویری، صوتی یا نوشتاری؛ در ادامه، معماری یعنی ساختاری که یک مجموعه سیستمی اعم از سخت افزار و نرم افزار با چه راه و روش هایی، با یکدیگر ارتباط برقرار کنند. امروزه در کنار REST از کلیدواژه هایی مثل Client & Server نیز استفاده میکنند. عموماً Client ها کاربرانی هستند که در یک سوی شبکه بدنبال منابعی میگردند که در حیطه کنترل Server ها قرار دارند. به نوعی میتوانیم آنها به ترتیب مصرف کننده و تولید کننده قلمداد کنیم.

طبیعتاً هر معماری مبتنی بر اصولی پیاده سازی میشود، در معماری REST این اصول در شش دسته طبقه بندی میشود :

1 .Uniform Interface

بنابر اصل همگانی یا (General Principle) نرم افزارهای خوب باید خارج از قید و بند و محدودیتهای خاص، طراحی و پیاده سازی شوند. نرم افزارها عمدتاً برای رفع نیازهای کلی و نه جزئی پیاده سازی مشوند و حتی بعد از زمان متصور حیات آنها، باقی میمانند. در رابطه با معماری REST چهار شرط برای حفظ یک رابط یکنواخت در نظر گرفته است :

- Identification of resources : واسط نرم افزاری، بر هر بستری، باید تمامی منابعی که در اختیار دارد و میگذارد را شناسه بندی کند.
- Manipulation of resources through representations : دسترسی به منابع باید در قالب و چهارچوب یکپارچه در دسترس مصرف کننده قرار بگیرد؛ این چهارچوب معمولاً شرایطی را فراهم می آورد تا منابع در سمت Server دستکاری بشوند. عمدتاً این چهارچوب برای هر منبع به صورت مستقل عمل میکند به این معنی که یک منبع میتواند تنها مصرف بشود و همزمان منبعی دیگر مصرف و از سوی کاربر دچار تغییر نیز بشوند.
- Self-descriptive messages : ارتباطات از Client به Server یا بالعلکس درقالب پیغامهایی انجام میشوند؛ زمانی که چهارچوبی برای دسترسی به منابع ایجاد میکنیم باید قالبی برای این پیغامها نیز در نظر بگیریم، این پیغامها باید به حد کافی اطلاعات داشته باشند تا عملیات روی منابع را بدرستی انجام دهند و از سوی دیگر باید پاسخهایی به کاربران یا مصرف کنندگان نیز ارسال کنند تا آنها حیطه و میزان توانایی تغییرات روی منابع را نیز بدانند.
- Hypermedia as the engine of application state : مصرف کنندگان یا Client تنها یک شناسه برای ورود به Server را دارند که آن عمدتاً Initial URL گفته میشود. کاربران با تنها یک شناسه ورودی به صورت پویا و با اکتشاف در سوی Server به دیگر منابع دسترسی پیدا میکنند؛ مثلا لینکهایی که در تبلیغات و پوسترهای فیلم های سینمایی بر روی وبسایتها قرار میگیرند، معمولاً به کاربران اجازه دانلود میدهند، این یعنی اکتشاف.

امروزه ساختار REST API عمدتاً بر روی پروتکل HTTPپیاده سازی شده است، به این معماری HTTP-based REST API نیز گفته میشود. منابع به واسطه استاندارد URI طبقه بندی میشوند و توابعی که در پروتکل HTTP تعریف شدند، مثل PUT, GET, POST, UPDATE نیز روشهای دسترسی به منابع را توصیف میکنند.

2 .Client & Server

در طراحی سیستمها، اصلی به نام Separate of Concern به این گونه تعریف میشود که، هر جزئی از سیستم باید نگران وظایف که باید انجام دهد باشد، هر اندازه اجزا در این مورد مرزبندی مشخصتری داشته باشند، میزان وابستگی کاهش میابد. در رابط Client & Server در صورتی که تولید کننده و مصرف کننده از ساختار یکدیگر خبر نداشته باشند، به طبع میزان وابستگیها نیز کمتر خواهد شد و در این صورت میتوانیم یک مصرف کننده **به اصطلاح** جهانی داشته باشیم؛ مثلا سرور را یک کامپیوتر در نظر بگیریم و مصرف کننده را یک تلفن همراه؛ با در نظر گرفتن اصل مذکور، کامپیوتر نیازی ندارد و **نباید** از ساختار مصرف کننده خود خبر داشته باشد در این صورت میتواند منابع خود را حتی در اختیار یک سیستم سومی، مثلاً کنسول بازی، بگذارد. از جمله مزیت این جداسازی، رشد و مقیاس پذیری سیستم سرور برای کلاینتهای بیشتر خواهد بود؛ البته این امکان نیز وجود دارد که تغییرات در هر دو سو، به مرور زمان تاثیری در نحوه ارتباط کلاینت و سرور داشته باشد، باید سیستم را همیشه بروز نگاه داشت.

3 .Stateless

هر درخواستی که از سوی مصرف کننده یا کلاینت به سوی تولید کننده یا سرور ارسال میشود باید یک بسته کامل از اطلاعاتی باشد که سرور با استفاده از آنها، درخواست کلاینت را پاسخ بدهد. سرور به هیچ وجه، نباید از درخواستهایی که از قبل ارسال شدند، دوباره استفاده کند. همیشه کلاینتها هستند که وضعیت درخواستهای قبلی را در خود ذخیره میکنند.

4 .Cacheable

مطمئناً کلاینتها درخواستهای تکراری به سوی سرور ارسال میکنند، پردازش این درخواستها فشاری مظاعف بر سرور خواهد بود و از سویی برای کاربران، انتظاری بیهوده. سرورها در این وضعیت درخواستها و پاسخهای تکراری را با سیاستهایی در سمت خود ذخیره میکنند، و کاربران نیز تا زمانی که درخواست جدیدی ارسال نکند، از این منابع بهره میبرد.

5 .Layered System

سیستم های چندلایه قابلیت ارتقا و مقیاس پذیری دارند بدون اینکه لایه ها از عملکرد یکدیگر خبر داشته باشند. یکی از معماریهای چندلایه که تا به امروز از آن استفاده شده است،MVC نام دارد.

6 .Code on Demand

در معماری REST، کلاینتها میتوانند یک سری کدها را از سوی سرور دانلود و روی سیستم خود اجرا کنند. در واقع سرور این اجازه را دارد که توانایی های بیشتری را به کلاینت در قالب کد اعطا کند.

@RequestBody Annotation

در فریمورک Spring زمانی که درخواستهای HTTP به سوی سرور ارسال میشوند، امکان دارد که داخل این درخواستها داده هایی در بخش HTTP Body ذخیره شده باشند. با انوتیشین مذکور میتوانیم این داده ها را به قالب یک شی جاوایی نگاشت کنیم. عمدتاً این داده ها به صورت JSON نگاشت میشوند. این انوتیشین را روی ورودی توابع قرار میدهند و نوع متغییر نیز نوع داده نهایی را مشخص میکند، به مثال زیر توجه کنید.

```
@RestController
@RequestMapping(path = "api/v1/professor")
public class ProfessorController
        extends AbstractProfessorController
{
    public ProfessorController(IProfessorService iProfessorService)
    {
        super(iProfessorService);
    }

    // add course to professor
    @PatchMapping(path = "/ac")
    public ResponseEntity<Object> addCourse(@ModelAttribute Update professor,
                                            @RequestBody Create course)
    {
        super.iProfessorService.addCourse(professor, course);
        return ResponseEntity.ok().build();
    }

    // get average
    @GetMapping(path = "/ga")
    public ResponseEntity<RpProfessorAVG> getAverage(
            @ModelAttribute Read read)
    {
        RpProfessorAVG rp = iProfessorService.getAverage(read);
        return ResponseEntity.ok(rp);
    }
}
```

JPA’s FetchType

جداول بر بستر Tabular SQL با استفاده از کلیدهای خارجی و جداول میانی با یکدگیر ارتباط میگیرند. در JPA بواسطه ساختمانهایی داده مثل لیستها و مجموعه ها، این ارتباطات را نگاشت میکنند، حال برای دسترسی به این ساختارها، دو روش عمده وجود دارد :

- Lazy : در این روش، دسترسی به اطلاعات تا زمانی که نیازی به آنها نباشد، به تعویق می افتد.
- Eager : دسترسی به اطلاعات درجا و بدون وقفه خواهد بود.

زمانی که حرف از دسترسی میزنیم، یعنی به اعضای لیست دسترسی داشته باشیم، در صورتی که برنامه درگیر با این قطعه کدها شود، یکی از سیاستهای بالا اتخاذ میشود. در این میان مزیتها و معایبی متوجه سیاستگذاری های بالا میباشد :

در دسترسی Lazy، زمان بارگذاری داده ها سریعتر صورت میگیرد پس حافظه کمتری نیز اشغال میشود. اما زمان دسترسی به داده ها خود یک وقفه در آینده ایجاد میکند.

در دسترسی Eager : داده ها در آن واحد در دسترس هستند اما زمان بارگذاری آنی بسیار طولانی خواهد بود و از سوی دیگر، داده های غیر ضروری نیز باگذاری میشوند.

ارتباطات میان جداول در سه دسته یک به یک، یک به چند، چند به یک و چند به چند تقسیم میشوند. بر بستر JPA هر کدام از این روشها با انوتیشینهایی مشخص میشوند و هر انوتیشین نیز یک سیاست پیش فرض برای بارگذاری داده ها اتخاذ میکند. دو مورد یک به چند و چند به چند با سیاست Lazy بارگذاری میشوند و از آن سو، ارتباطات چند به یک و یک به یک نیز با پیشفرض Eager.

@GeneratedValue Annotation

هر موجودیت به یک جدول نگاشت میشود و هر جدول نیازمند به یک یا مجموعه از ستونها بعنوان کلید اصلی داخلی است تا سیستم بان اطلاعات، سطرها را از یکدیگر تمیز دهد. طراحان دیتابیس مجبورند که با محاسباتی کلید را پیدا کنند که به دو مشکل عمده نیز برخورد میکنند :

- کلیدهای ترکیبی احتمال افزونگی داده ها را افزایش میدهد. طراحان دیتابیس مجبور هستند که ساختار جداول را با قوائد فرمهای استاندارد، نرمالیزه کنند که این جداول جدیدی را به دیتابیس اضافه میکند.
- مدیریت کلیدها در دیتابیس یک سری محدودیتهایی دارد، بعنوان مثال در دیتابیس MariaDB از خانواده دیتابیس MYSQL، نهایتاً کلیدی به طول ۷۶۷ بایت را ذخیره میکند و این یک سری محدودیتها را در ستونها بوجود میاورد، مخصوصاً اگر این ستونها داده های رشته ای را ذخیره کنند.

نهایتاً طراحان به این بخش سوق داده شدند، بهتر است که از اعداد به عنوان کلید اصلی جداول استفاده کنیم و باقی ستونها که کاندید کلید اصلی هستند را به عنوان کلید خارجی یا حتی شاخص (Index) در دیتابیس قرار دهیم. با انوتیشین @GeneratedValue و یک تولید کننده کلید یا Generator این اختیار را به دیتابیس میدهیم که کلید را برای جدول، اتوماتیک تولید کند.

کد زیر یک نمونه استفاده از این انوتیشین را نشان میدهد :

```
import javax.persistence.{Column, Entity, GeneratedValue, GenerationType, Id, SequenceGenerator,
Table}

@Table(name = "sc_ent_customer")
@Entity
class ScalaCustomerEntity
  extends Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "ScalaCustomerEntityGenerator")
  @SequenceGenerator(name = "ScalaCustomerEntityGenerator", sequenceName = "SC_PE_SEQ", ini-
tialValue = 1, allocationSize = 1)
  var id: Integer = _

  @Column
  var name: String = _


  @Column
  var price: String = _


}
```

متغییر strategy نحوه تولید کلید را مشخص میکند، چهار مدل پیاده سازی دارد :

- AUTO : مقداردهی وابسته به ORM و دیتابیس همزمان خواهد بود. هایبرنت بعنوان ORM پیشفرض بر اساس اشیای dialect که وابسته به دیتابیس نیز هستند، مقداردهی را کنترل میکند.
- IDENTITY : بعضی از دیتابیسها مانند MYSQL ستونها را به صورت AUTO-INCREAMENT مقداردهی میکنند؛ این مقداردهی تماماً در کنترل دیتابیس است.

- SEQUENCE : دیتابیسهایی مانند اوراکل دنباله های عددی تولید میکند که از سوی هایبرنت برای درج کلیدها فراخوانی میشود.
- TABLE : هایبرنت یک جدول در دیتابیس ایجاد میکند که مقدار بعدی کلید را در خود ذخیره و تولید میکند.

Record Pagination

زمانی که تعداد زیادی رکورد یا داده در بانک اطلاعاتی داریم، ترجیحاً بخشی از آنها را به کاربران نمایش میدهیم. به این عملیات، صفحه بندی میگوند، حال با شاخص هایی نیز میتوانیم رکوردها را در این صفحه ها مرتب کنیم. برای صفحه بندی باید از JpaRepository یا PagingAndSortingRepository استفاده کنیم. متدهای دسترسی به دیتابیس و جداول با ورودی Pageable مشخص شدند.

```
import org.springframework.data.domain.{Page, Pageable}
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.stereotype.Repository


@Repository
trait ScalaCustomerEntityRepository
  extends JpaRepository[ScalaCustomerEntity, Integer]
    with Serializable {
  def findAllByName(name: String, pageable: Pageable): Page[ScalaCustomerEntity]

}
```

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.data.domain.{Page, PageRequest}
import org.springframework.web.bind.annotation.{GetMapping, RequestMapping, RequestParam,
RestController}

@RestController
@RequestMapping(path = Array.apply("api/v1/customer"))
@Autowired
class ScalaCustomerEntityController
(
  var repository: ScalaCustomerEntityRepository
) {

  @GetMapping(params = Array.apply("name", "size", "page"))
  def getCustomer(@RequestParam name: String, @RequestParam size: Integer, @RequestParam page:
Integer): Page[ScalaCustomerEntity] = {
    val pageable = PageRequest.of(page, size);
    repository.findAllByName(name, pageable);
  }
}
```