Report

No .2

By Amir Lotfi

Contents

Rest's Resource Naming Conventions

منابع یک مجموعه از نگاشتهایی به موجودیتهای یک سرویس، سرور، تولید کننده یا ارائه دهنده خدمات هستند. موجودیتها خود شامل انواع مختلف فایلها یا حتی سرویسها میباشد، هر چیزی که مصرف کننده یا کاربر بدنبال آن هستند. طبیعتاً این منابع باید به نحوی نامگذاری بشوند تا از اصول REST پیروی کنند. نامگذاری صحیح، استفاده از یک ارائه دهنده خدمات را راحتتر میکند. مجموعه قوانین تجربی در این زمینه به ما کمک میکند تا معماری قابل قبول داشته باشیم.

1 .Using Noun for Resources

فرض کنیم که در یک سیستم ما منابعی را به کاربران معمولی نگاشت میکنیم و آنها را users مینامیم. عملیاتهای CRUD مانند PUT, GET را در URI قرار نمیدهیم، مثلاً :

```
Not So Good:
```

- `/createUser [well! This gonna invoke PUT method, right ?!]`
- `/getUser [what if we want to add a user, should we build a new endpoint?!]`

```
Better be :
```

- `/users [ invoking GET method will give us the users or POST will generate new one ]`
- `/users/{userId} [if we invoke GET on users, we can use 'user.entity' primary key to fetch specific user]`

مورد دوم، دست توسعه دهنده را برای اعمال متدهای بیشتر روی یک مسیر و زیرشاخه های آن بازتر میگذارد. **در واقع بهتر است که مسیرها به صورت یک درخت باشند تا جنگل!**

2 .Use HTTP Methods for Actions

دوباره از مثال بالا استفاده میکنیم، ما یک مسیر users داریم و متدهای مختلف را روی همین مسیر یا روی **زیرشاخه** های آن پیاده سازی میکنیم. در وضعیت اول، ما برای هر متد باید یک مسیر مجزا تعریف کنیم، اما در روش دوم متدهای مختلف روی مسیر و زیرمجموعه های آن تعریف میشود.

3 .Plural Nouns for Collections

برای مجموعه از منابع که در یک دسته قرار میگیرند، بهتر است که از اسم جمع استفاده کنیم؛

Not so Good:

- /product
- /student

Better Be:

- /products
- /students

4 .Parameters for Filtering, Sorting & Pagination

در یک مسیر REST URI از متغییرها زمانی استفاده میکنیم که بخواهیم داده ها را در قالب خاص، مرتب یا محدود، دریافت کنیم.

[using parameters for pagination]

/products?page=1&size=20

[using parameters for filtering or sorting]

/students?avg=19

5 .Multi-word Naming

برای نامگذاری مسیرهای بزرگ، بهتر است که از استاندارد snake-case یا camelCase استفاده کنیم و همینطور، از خط زیر پرهیز کنیم، خط فاصله قابل قبول است.

No so Good:

- /head_of_department
- /HeadOfDepartment

Better Be:

- /head-of-deparment [this much more preferred over camelCase]
- /headOfDepartment [not so good, still we can do this]

Primary Key Creation Best Practice

کلید شناسه یکتا برای ردیابی آنی یک سطر در جدول بانک اطلاعاتی است. عمده کلیدهایی که برای جداول استفاده میشوند، در تئوری متعلق به گروه کلیدهای طبیعی هستند؛ درواقع یک سری ستون، یک یا چند، را بعنوان کلید در نظر میگیرند. دو مشکل عمده این روش دارد :

۱. کلیدهای طبیعی وابسته به ساختار جدول هستند، در صورتی که جداول تغییر کنند این کلیدها نیز تغییر میکنند.
۲. کلیدهای ترکیبی خود افزونگی داده را ایجاد میکنند و افزونگی منجر به تبدیل جداول اصلی به زیرمجموعه از جداول میانی خواهد شد زیرا باید از روشهای نرمالیزه کردن جداول استفاده کنیم.

از این سو طراحان از نوع دیگر کلیدها استفاده میکنند، به آنها کلیدهای جانشین یا surrogate میگویند. این کلیدها ظاهر عددی دارند و عمدتاً یک تابع محاسباتی در پشت سر دارند که از روی محتویات جدول، هش، یا از روی الگوریتمهای از پیش نوشته شده، دنبالهای عددی، استفاده میکنند. این به عنوان یک ستون تنها نقش کلید را ایفا میکنند، معنی خاصی ندارند و توسط برنامه استفاده نمیشوند. در رابطه با JPA روش مذکور با انوتیشین GeneratedValue پیاده سازی میشود.

حال که کلید به این شیوه تولید میشود، برای عملکرد بهتر دیتابیس، بهتر است که روی ستونهایی شاخصگذاری انجام دهیم. حافظه بیشتری مصرف میشود اما سرعت اجرا افزایش خواهد یافت، به مثال زیر توجه کنید :

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

@Table(name = "ent_product", indexes = Array.apply(
  new Index(columnList = "name", unique = true),
  new Index(columnList = "price", unique = false)
))
@Entity
class ScalaProductEntity
{

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sc_ent_prod_gen")
  @SequenceGenerator(name = "sc_ent_prod_gen", initialValue = 1, allocationSize = 1)
  var id: Integer = _

  @Column
  var name: String = _

  @Column
  var price: Integer = _

}
```

اولاً کلید این جدول که ستون عددی id را شامل میشود، هیچ معنا و مفهومی در دنیای بیرون ندارد و از روی یک دنباله عددی مقدار دهی میشود. دوماً با تغییر هر کدام از ستونها جدول، کلید بدون تغییر باقی میماند و سوماً، روی ستونهایی که کلید کاندید بودند در اندیس گذاری استفاده میکنیم؛ در این جدول، نام محصول را یک کلید کاندید و اندیس یکتا در نظر گرفتیم.

Entity Creation

در ساختار بانکهای اطلاعاتی جدولی مبتنی بر SQL، جداول قابلیت ارثبری ندارد؛ هر جدول عمدتاً یک ماهیت مستقل در دنیایی بیرون را نگاشت میکند. در چهارچوب OOP ارثبری ابزاری برای افزایش بهره وری یک کلاس است. حال از این ابزار به چهار شیوه مختلف میتوان ارثبری را هم در سطح کلاس و هم در سطح زبان پرس و جوی بانک اطلاعاتی یا کوئری پیاده سازی کرد.

@MappedSuperClass

در این روش میتوانیم یک سلسله مراتب که ساختار منطقی دارد را روی جداول در نظر بگیریم؛ این قالب بر روی دیتابیس هیچ اثری ندارد، بلکه از سوی Spring مدیریت میشود :

```
@MappedSuperclass
class GroovyBaseEntity implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "groovy_generator")
    protected Integer id
}

@MappedSuperclass
class GroovyPersonBaseEntity extends GroovyBaseEntity {

    @Column(name = "name", length = 32)
    protected String name
    @Column(name = "surname", length = 32)
    protected String surname

}

@SequenceGenerator(name = "groovy_generator", sequenceName = "gv_student_seq", initialValue = 1, allocationSize = 1)
@Entity
class GroovyStudentEntity extends GroovyPersonBaseEntity {
    @Column(name = "student_id", length = 32)
    private String studentId
}

@SequenceGenerator(name = "groovy_generator", sequenceName = "gv_professor_seq", initialValue = 1, allocationSize = 1)
@Entity
class GroovyProfessorEntity extends GroovyPersonBaseEntity {
    @Column(name = "professor_id", length = 32)
    private String professorId
}
```

@Inheritance via SINGLE_TABLE

به ازای هر کلاس، تنها یک جدول تشکیل خواهد شد؛ این جدول خصوصیتهای تمام زیرکلاسهایش در سلسله مراتب ارثبری را دارد. در این قالب میتوانیم از کوئریهای چندریختی نیز استفاده کنیم، یک جدول اما با کوئریهای مختلف.

```
@MappedSuperclass
public class BaseEntity
        implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "groovy_generator")
    protected Integer id;
}

@MappedSuperclass
class PersonBaseEntity
        extends BaseEntity
{

    @Column(name = "name", length = 32)
    protected String name;
    @Column(name = "surname", length = 32)
    protected String surname;

}

interface EmployeeType
{
    String STUDENT   = "STUDENT";
    String PROFESSOR = "PROFESSOR";
}

@Entity
@SequenceGenerator(name = "groovy_generator", sequenceName = "uni_emp_seq", initialValue = 1, allocationSize = 1)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE", length = 32, discriminatorType = DiscriminatorType.STRING)
class UniversityEmployee
        extends PersonBaseEntity
{

}


@Entity
@DiscriminatorValue(value = EmployeeType.STUDENT)
class StudentEntity
        extends UniversityEmployee
{
    @Column(name = "student_id", length = 32)
    private String studentId;
}

@Entity
@DiscriminatorValue(value = EmployeeType.PROFESSOR)
class ProfessorEntity
        extends UniversityEmployee
{
    @Column(name = "professor_id", length = 32)
    private String professorId;
}
```

@Inheritance via JOINED

در این قالب، کلید کاندید در کلاس مادر تبدیل به کلید اصلی و در زیرکلاسها تبدیل به کلید اصلی و خارجی خواهد شد. هر کلاس به جدولی مجزا نگاشت میشود و هر کلاس خصوصیتهای خود را نگاه میدارد و از مادران خود ستونی جدید به ارث نمیبرند. یکی از ایرادات این دسته بندی، پیوند همزمان چندین جدول با کوئریهای تودرتو است و این چندلایه بودن کوئریها خود را زمانی نشان میدهد که مستقیماً بر جدول مادر بدنبال رکورد باشیم.

```
@MappedSuperclass
public class BaseEntity
        implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "groovy_generator")
    protected Integer id;
}

@MappedSuperclass
class PersonBaseEntity
        extends BaseEntity
{

    @Column(name = "name", length = 32)
    protected String name;
    @Column(name = "surname", length = 32)
    protected String surname;

}

@Entity
@SequenceGenerator(name = "groovy_generator", sequenceName = "uni_emp_seq", initialValue = 1, allocationSize = 1)
@Inheritance(strategy = InheritanceType.JOINED)
class UniversityEmployee
        extends PersonBaseEntity
{

}


@Entity
@PrimaryKeyJoinColumn(name = "uni_emp_stud_id")
class StudentEntity
        extends UniversityEmployee
{
    @Column(name = "student_id", length = 32)
    private String studentId;
}

@Entity
@PrimaryKeyJoinColumn(name = "uni_emp_prof_id")
class ProfessorEntity
        extends UniversityEmployee
{
    @Column(name = "professor_id", length = 32)
    private String professorId;
}
```

@Inheritance via TABLE_PER_CLASS

در این قالب، همه خصوصیتهای کلاسهای مادر به فرزندان به ارث میرسد و هر کلاسی کلید اصلی خود را خواهد داشت. شبیه به انوتشین @MappedSuperClass عمل میکند اما کلاس مادر نیز یک موجودیت جدا خواهد بود. این قالب به نسبت دو نمونه دیگر، بسیار هزینه بر خواهد بود.

```
@MappedSuperclass
public class BaseEntity
        implements Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "groovy_generator")
    protected Integer id;
}

@MappedSuperclass
class PersonBaseEntity
        extends BaseEntity
{

    @Column(name = "name", length = 32)
    protected String name;
    @Column(name = "surname", length = 32)
    protected String surname;

}


@Entity
@SequenceGenerator(name = "groovy_generator", sequenceName = "uni_emp_seq", initialValue = 1, allocationSize = 1)
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
class UniversityEmployee
        extends PersonBaseEntity
{

}

@Entity
class StudentEntity
        extends UniversityEmployee
{
    @Column(name = "student_id", length = 32)
    private String studentId;
}

@Entity
class ProfessorEntity
        extends UniversityEmployee
{
    @Column(name = "professor_id", length = 32)
    private String professorId;
}
```

Entity Creation Best Practice

روشهای ساخت یک **موجودیت خوب** روی قواعد جدای از نامگذاری تمرکز دارند.

1 .Inheritance & Polymorphism Queries

در بخش Entity Creation درباره کوئریهای چندریختی صحبت کردیم؛ خارج از دسترسی و توانایهای بانک اطلاعات، میتوانیم قدرت برنامه نویسی شی گراء را روی جداول نیز پیاده سازی بکنیم تا منطق قابل فهمتری را روی دامنه اطلاعات پیاده سازی کنیم. ارثبری بر موجودیتها و چندریختی بر زبان پرس و جو دو قابلیت جالب Hibernate هستند.

2 .Validations

بانک اطلاعاتی وظیفه نگهداری از داده ها را برعهده دارد، آیا داده ها معتبر هستند ؟! مثلاً درج آدرس ایمیل در جدول؛ طبیعتاً بانک اطلاعاتی محدودیتهایی روی رشته ایمیل در نظر میگیرد که حداقل محدود به طول رشته است اما اعتبارسنجی آن وظیفه بانک اطلاعاتی نیست.

3 .Native Query with DTO

معمولاً همه ستونهای دیتابیس مورد استفاده لایه های بالاتر نیستند حذف آنها بهینه سازی بسیار عالی در سطح IO ایجاد میکند و البته کوئری های دستی قابلیت حمل کد را کاهش میدهند.

```
@Entity
class StudentEntity
  extends Serializable
{
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  var id: Integer = _

  @Column(name = "studentId", length = 32)
  var studentId: String = _

  @Column(name = "name", length = 32)
  var name: String = _

  @Column(name = "surname", length = 32)
  var surname: String = _

  @Column(name = "address", length = 64)
  var address: String = _

}

object StudentEntity
{
  case class Read(id: String, name: String, surname: String)

}

trait StudentEntityRepository
  extends JpaRepository[StudentEntity, Integer]
{

  @Query("""SELECT u.studentId as studentId, u.name as name ,u.surname as surname from StudentEntity u WHERE studentId =
:studentId""")
  def findByStudentId(studentId: String): Optional[StudentEntity.Read]

}
```

4 .Auditability & Logging

داده ها مدام در حال تغییراند. در اکثر سیستمهای بزرگ، ردپای تمامی فعل و انفعلات سیستم را رصد میکنند و آنرا در جداولی جداگانه نیز ذخیره میکنند. رصد تغییرات، حفظ امنیت و داده کاوی اطلاعات عمده فعالیتهایی است که روی این جداول انجام میگیرد.

```
import org.slf4j.{Logger, LoggerFactory}
import org.springframework.data.annotation.{CreatedDate, LastModifiedDate}

import java.time.LocalDateTime
import javax.annotation.PreDestroy
import javax.persistence.{Column, Entity, EntityListeners, GeneratedValue, GenerationType, Id, PostRemove, PostUpdate, PreUpdate,
Table};

@EntityListeners(
  value = Array.apply(
    classOf[PreListener],
    classOf[PostListener]
  )
)
@Table(name = "student")
@Entity
class StudentEntity
  extends Serializable
{
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  var id     : Integer = _
  @Column(length = 32)
  var name   : String  = _
  @Column(length = 32)
  var surname: String  = _

  @LastModifiedDate
  var lastModifiedDate: LocalDateTime = _
  @CreatedDate
  var createdDate     : LocalDateTime = _
}

class PostListener
  extends Serializable
{
  val log: Logger = LoggerFactory.getLogger(getClass.getName)

  @PostUpdate
  def postUpdate(entity: Object): Unit = log.info("updated")

  @PostRemove
  def postRemove(entity: Object): Unit = log.info("destroyed")
}

class PreListener
  extends Serializable
{
  val log: Logger = LoggerFactory.getLogger(getClass.getName)

  @PreUpdate
  def preUpdate(entity: Object): Unit = log.info("updating")

  @PreDestroy
  def preDestroy(entity: Object): Unit = log.info("destroying")
}
```

Unit Testing

اگر هر بخشی از برنامه اعم از متد(ها)، کلاس(ها)، ماژول(ها) در یک محیط ایزوله، بدون اینکه نیازمندیهای آنها را در فرایند تست دخیل کنیم، درواقع Unit Testing را انجام دادیم. ایزوله سازی قطعه کد و اجرای تست روی آن وابستگی تست را کاهش میدهد و سرعت اجرای تست را افزایش میدهد؛ نیازمندیهای تست را در قالب شیهای پوچ یا Mock برای قطعه کد تامین میکنند. این اشیای پوچ رفتار شی را تقلید میکنند و این تقلید در دستان برنامه نویس است.

```
object TestObject
{

  class Dependency
  {
    def function(): Boolean = ???
  }

  class Dependant(dependency: Dependency)
  {
    def function(): Boolean = dependency.function()

  }
}

@ExtendWith(Array.apply(classOf[MockitoExtension]))
class DependantTest
{
  @Mock
  var dependency: TestObject.Dependency = _
  @InjectMocks
  var dependant : TestObject.Dependant  = _

  @Test
  def test(): Unit =
  {
    Mockito.when(dependency.function()).thenReturn(Boolean.TRUE)
    val result = dependant.function()
    Assertions.assertTrue(result)
  }

}
```

Integration Test

سیستم در دنیای واقعی استفاده میشود و تست نویسی ایزوله به تنهایی سنگ محکی برای سیستم نیست، مثلاً ارتباط با دیتابیس که باید در محیط واقعی انجام گیرد. در واقع ایزوله بودن بخشهای برنامه را کنار میگذاریم و سیستم را به عنوان یک **کل** تست میکنیم. تفاوتی که بین Unit Testing و Integration Testing به چشم میخورد نحوه برخورد آنها با بخشهای برنامه است. در Unit Testing ما هر بخش را تنها به عنوان **یک بخش** تست میکنیم بدون اینکه به وابستگی آن بیاندیشیم در مقابل، در Integration Testing ما تمامی بخشهای برنامه را به عنوان **یک کل** مینگریم.

```
object TestObject
{

  class Dependency
  {
    def function(): Boolean =
    {
      val random    = new Random()
      val seed      = 20
      val (a, b, c) = (random.nextInt(seed), random.nextInt(seed), random.nextInt(seed))

      a == b + c
    }
  }

  class Dependant(dependency: Dependency)
  {
    def function(): Boolean = dependency.function()

  }
}


class DependantTest
{
  var dependency: TestObject.Dependency = new TestObject.Dependency()
  var dependant : TestObject.Dependant  = new TestObject.Dependant(dependency = dependency)

  @Test
  def test(): Unit =
  {
    val result = dependant.function()
    Assertions.assertTrue(result)
  }

}
```

Java’s Code Coverage Tools

این ابزارها دو وظیفه عمده دارند :

- دسته اول کد را به کدهای خام برنامه اضافه میکند، طبیعتاً نتیجه باید دوباره کامپایل بشود.
- دسته دوم با بایت-کدها مستقیماً کار میکنند؛ زمانی که برنامه اجرا میشود، یا حتی قبلتر از آن، کدها را رصد میکند. عمدتاً در تست نویسی استفاده میشود و نهایتاً گزارشهایی بیرون میدهد که به برنامه نویس کمک میکند درباره لایه ها و شاخه های مختلف برنامه اطلاعاتی بدست بیاورد، مثلاً چه تعداد از خطوط کد تست شدند. چه کلاسها،متدها یا خطوطی درگیر تست بودند.

در زبان جاوا از نسخه 1.5 به بعد، بسته Java Instrument API به ماشین مجازی اضافه شد که با استفاده از آن میتوانیم به کلاسها، کدهایی اضافه بکنیم و دوباره برنامه را کامپایل کنیم؛ با کامپایل دوباره میتوانیم کدها را رصد کنیم و از وضعیت آنها گزارشهایی تهیه کنیم. ابزارهای code coverage با این تکنیک به اطلاعات درون برنامه دسترسی پیدا میکنند و از وضعیت اجرای آنها مخصوصاً در حیطه تست، گزارش تولید میکنند. برنامه نویس با تجزیه و تحلیل این داده ها میتواند مشکلاتی را دریابد که در حالت عادی به آنها دسترسی نداشت.

```
class Dependency
{
  def function(branch: Boolean): String =
  {
    if (branch)
    {
      "do this"
    }
    else
    {
      "do that"
    }
  }
}

class Dependent(private val dependency: Dependency)
{
  def function(branch: Boolean): String =
  {
    val fnResult = dependency.function(branch)
    "post " + fnResult
  }
}

class ScalaTest
{

  val dependent: Dependent = Dependent(dependency = Dependency())

  @Test
  def function(): Unit =
  {
    val branch = true
    val result = dependent.function(branch)
    Assertions.assertThat(result).isEqualTo("post do this")
  }
}
```

با اجرای تست، همزمان افزونه Jacoco نیز اجرا میشود و مجموعه نتایج زیر را تولید میکند، ما تنها کد بالا را در نظر میگیریم. رنگ زد یعنی ما بخشی از کد را تست کردیم و هنوز باقی بخشها باقی مانده اند. بخشهایی که تست نشدند با رنگ قرمز و سبز نیز بخشهای تحت پوشش تست هستند. اعدادی که در ستونها یادداشت میشوند، دو به دو باید تفسیر بشوند. مثلا Missed/Line نسبت خطوطی که تست نشدند. متغییر Cov بیانگر درصد خطوط کد یا زیرشاخه های تحت پوشش تست هستند به آنها Code Coverage و Branch Coverage نیز گفته میشود.

project

Sessions

project

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
us.core.pr.domain.crud.impl		6%		0%	15	18	83	92	12	15	0	3
us.core.pr.domain.entity.listener.impl		15%		n/a	46	48	61	73	46	48	11	13
us.core.pr.domain.entity		74%		43%	32	101	35	159	18	85	1	10
us.core.pr.builder		0%		0%	4	4	17	17	3	3	1	1
us.core.pr.service		81%		100%	13	27	30	131	13	23	0	3
us.core.pr.domain.dto.professor		36%		n/a	14	22	22	35	14	22	2	4
us.core.pr.domain.dto.college		35%		n/a	13	19	21	32	13	19	3	5
us.core.pr.domain.dto.student		39%		n/a	12	20	18	29	12	20	2	4
us.core.pr.repository.container.impl		0%		0%	4	4	6	6	2	2	1	1
us.core.pr.exception		10%		n/a	8	9	16	18	8	9	5	6
us.core.pr.validation.validators		0%		n/a	9	9	12	12	9	9	3	3
us.core.pr.domain.dto.mapper.impl.professor		45%		n/a	8	10	8	13	8	10	4	5
us.core.pr.domain.dto.mapper.impl.student		0%		n/a	8	8	8	8	8	8	4	4
us.core.pr.domain.dto.mapper.impl.college		0%		n/a	8	8	8	8	8	8	4	4
us.core.pr.exception.handler		17%		n/a	7	8	7	8	7	8	0	1
us.core.pr.repository.container.abstracts		0%		n/a	1	1	4	4	1	1	1	1
us.core.pr.exception.entity		0%		n/a	4	4	7	7	4	4	4	4
us.core.pr.validation.payload		0%		n/a	3	3	3	3	3	3	3	3
us.core.pr		82%		50%	2	8	3	11	1	7	0	3
us.core.pr.domain.dto.mapper.impl.course		84%		n/a	2	4	2	9	2	4	1	2
us.core.pr.domain.dto.mapper.factory.impl		42%		n/a	1	2	1	2	1	2	0	1
us.core.pr.utils		95%		n/a	1	4	1	12	1	4	0	1
us.core.pr.domain.entity.listener.abstracts		82%		n/a	1	5	1	8	1	5	1	3
us.core.pr.domain.dto.reporting		100%		n/a	0	28	0	44	0	28	0	4
us.core.pr.controller		100%		n/a	0	10	0	22	0	10	0	3
us.core.pr.configuratoin		100%		n/a	0	8	0	8	0	8	0	2
us.core.pr.service.abstracts		100%		n/a	0	3	0	12	0	3	0	3
us.core.pr.domain.crud.abstracts		100%		n/a	0	3	0	9	0	3	0	3
us.core.pr.controller.abstraction.abstracts		100%		n/a	0	3	0	9	0	3	0	3
us.core.pr.domain.dto.course		100%		n/a	0	5	0	8	0	5	0	1
us.core.pr.domain.dto.mapper.factory.abstracts		100%		n/a	0	1	0	1	0	1	0	1
Total	1,124 of 2,466	54%	31 of 54	42%	216	407	374	810	195	380	51	105

Created with JaCoCo 0.8.11.202310140853

project > us.core.pr

Source Files Sessions

us.core.pr

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Application		37%		n/a	1	2	2	3	1	2	0	1
Dependency		77%		50%	1	3	1	4	0	2	0	1
Dependent		100%		n/a	0	3	0	4	0	3	0	1
Total	7 of 41	82%	1 of 2	50%	2	8	3	11	1	7	0	3

Created with JaCoCo 0.8.11.202310140853

project > us.core.pr > Dependent

Sessions

Dependent

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
function(boolean)		100%		n/a	0	1	0	2	0	1
Dependent(Dependency)		100%		n/a	0	1	0	2	0	1
dependency()		100%		n/a	0	1	0	1	0	1
Total	0 of 24	100%	0 of 0	n/a	0	3	0	4	0	3

Created with JaCoCo 0.8.11.202310140853

project > us.core.pr > ScalaTest.scala

Sessions

ScalaTest.scala

```
1.  package us.core.pr
2.
3.  import scala.util.Random
4.
5.
6.  class Dependency
7.  {
8.    def function(branch: Boolean): String =
9.    {
10.     if (branch)
11.     {
12.       "do this"
13.     }
14.     else
15.     {
16.       "do that"
17.     }
18.   }
19. }
20.
21. class Dependent(private val dependency: Dependency)
22. {
23.   def function(branch: Boolean): String =
24.   {
25.     val fnResult = dependency.function(branch)
26.     "post " + fnResult
27.   }
28. }
29.
```

Created with JaCoCo 0.8.11.202310140853

DTO

کلاسها و اشیایی که تنها داده ها را در خود ذخیره میکنند. این شی ها میان پروسه ها دست به دست میگردند و داده های مورد نیاز آنها را حمل میکنند. در گذشته، با پروتکل RMI برنامه نویسان جاوا میتوانستند کدی را که در سمت سرور، در قالب متد(ها) پیاده سازی شدند را فراخوانی کنند. این فراخوانی هزینه زیادی ایجاد میکرد و البته در دفعات زیاد هزینه نیز بیشتر میشد. در آن زمان از DTO به عنوان یک شی بسط دهنده داده ها استفاده میکردند که تنها اطلاعات را منتقل میکرد و برای کاهش فراخوانی های توابع در سوی سرور، عمدتاً فیلدهای بیشتری را در DTO بسط میدادند.

در زیر، دو کلاس Read, Delete در حکم dto عمل میکنند، آنها بخشی از داده ها را در خود ذخیره و به لایه های دیگر برنامه هدایت میکنند.

```
@Entity
class StudentEntity
  extends Serializable
{
  @Id
  var id       : Integer = _
  @Column
  var studentId: String  = _
  @Column
  var name     : String  = _
  @Column
  var surname  : String  = _
  @Column
  var address  : String  = _
  @Column
  var email    : String  = _
}

object StudentEntity
{
  object DTO
  {
    case class Read(var name: String, var surname: String)
      extends Serializable

    case class Delete(var studentId: String)
      extends Serializable

  }
}
```

Validation

سرورها به عنوان تولید کننده یا ارائه دهنده خدمات، یک مجموعه ای از داده ها را با مصرف کنندگان و کاربران خود رد و بدل میکنند، اغلب این داده ها در یک بانک اطلاعاتی ذخیره میشوند. داده ها در هر سروری باید از یک سری مجموعه دستور و العملها و فیلترها گذر کنند تا به نوعی **نرمالیزه** بشوند. مثلاً درگاه پرداختی شاپرک را در نظر بگیریم، شماره کارت الزاماً باید در ۱۶ رقم اعداد مثبت قرار بگیرد و یا سامانه فروشگاه دیجیکالا، تعداد کالاها باید مقداری مثبت باشند. محدودیتها مذکور، فیلترهایی هستند که داده های تبادلی را نرمالیزه میکنند.

در بستر Java Bean Validation API از یک سری انوتیشینها برای اعمال محدودیت در داده ها استفاده میشود، به مثال زیر توجه کنید؛ در این مثال، ستون age باید از ۱۸ بیشتر باشد و یا ستون email الزاماً باید از الگوی عبارات منظم (regex) استاندارد پیروی کند.

```
import javax.persistence.{Column, Entity, GeneratedValue, GenerationType, Id, Index, SequenceGenerator, Table}
import javax.validation.constraints.{Email, Min, NotNull} // validation package

@Table(
  name = "ent_sc_customer",
  indexes = Array.apply(
    new Index(columnList = "email", unique = true),
    new Index(columnList = "address", unique = true)
  )
)
@Entity
class ScalaCustomerEntity {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "ent_customer_seq_gen")
  @SequenceGenerator(name = "ent_customer_seq_gen", initialValue = 1, allocationSize = 1)
  var id: Integer = _

  @Column(name = "name", length = 64)
  @NotNull
  var name: String = _

  @Min(18)
  @Column(name = "age", length = 8)
  var age: Integer = _

  @Column(name = "address", length = 128)
  var address: String = _

  @Column(name = "email", length = 128)
  @Email(message = "invalid email type")
  var email: String = _
}
```

نکته حائز اهمیت، اولاً بخشی از این محدودیتها در سمت بانک اطلاعاتی اعمال میشوند، مثلا Min که حداقل مقدار یک ستون عددی را تعیین میکند اما محدودیتهایی هستند که از سوی برنامه روی داده ها اعمال سپس در بانک اطلاعاتی ذخیره میشوند، مثلا NotNull. در انتها، زمانی که محدودیتها نقض بشوند ، بسته به نوع محدودیت و محل اعمال آن، یک خطا پرتاب خواهد شد که باید توسط مدیریت شوند.