Lazy vs. Eager Loading Strategies in JPA 2.1

Patrycja Wegrzynowicz
CTO, Yon Labs
JavaOne 2018

About Me

- 20+ professional experience
  - Software engineer, architect, head of software R&D
- Author and speaker
  - JavaOne, Devoxx, JavaZone, Jazoon, TheServerSide Java Symposium, OOPSLA, ASE, others
- Top 10 Women in Tech 2016 in Poland
- Founder and CTO of Yon Labs/Yonita
  - Consulting, trainings and code audits
  - Automated detection and refactoring of software defects
  - Security, performance, concurrency, databases
- Twitter @yonlabs

YONITA

About Me

- 20+ professional experience
  - Software engineer, architect, head of software R&D
- Author and speaker
  - JavaOne, Devoxx, JavaZone, Jazoon, TheServerSide Java Symposium, OOPSLA, ASE, others
- Top 10 Women in Tech 2016 in Poland
- Founder and CTO of Yon Labs/Yonita
  - Consulting, trainings and code audits
  - Automated detection and refactoring of software defects
  - Security, performance, concurrency, databases
- Twitter @yonlabs

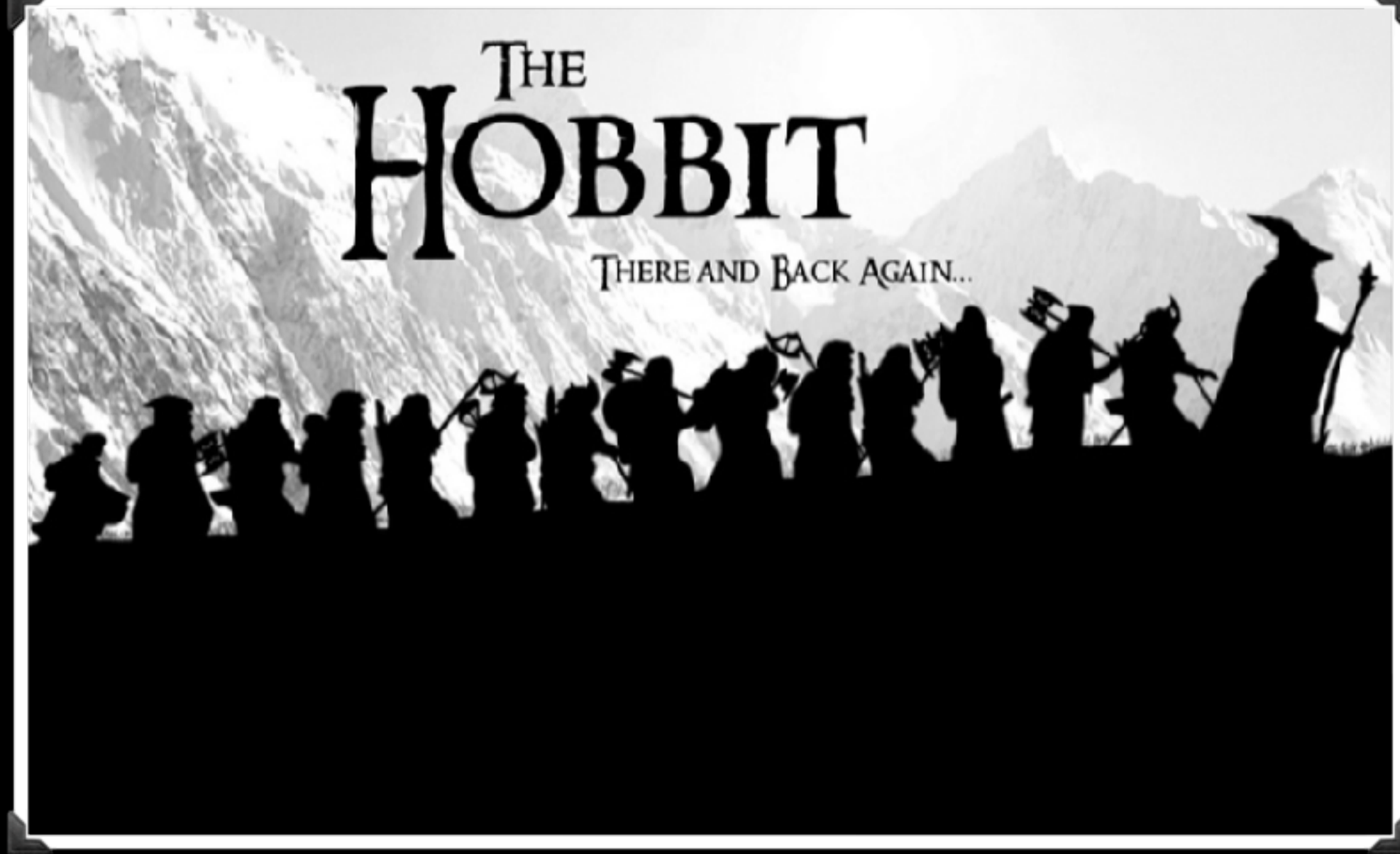
YONITA

Agenda

- Motivation
  - JPA
  - Performance
- Loading Strategies
- Projections and Aggregation
- Lazy vs. Eager
- Entity Graphs

Databases

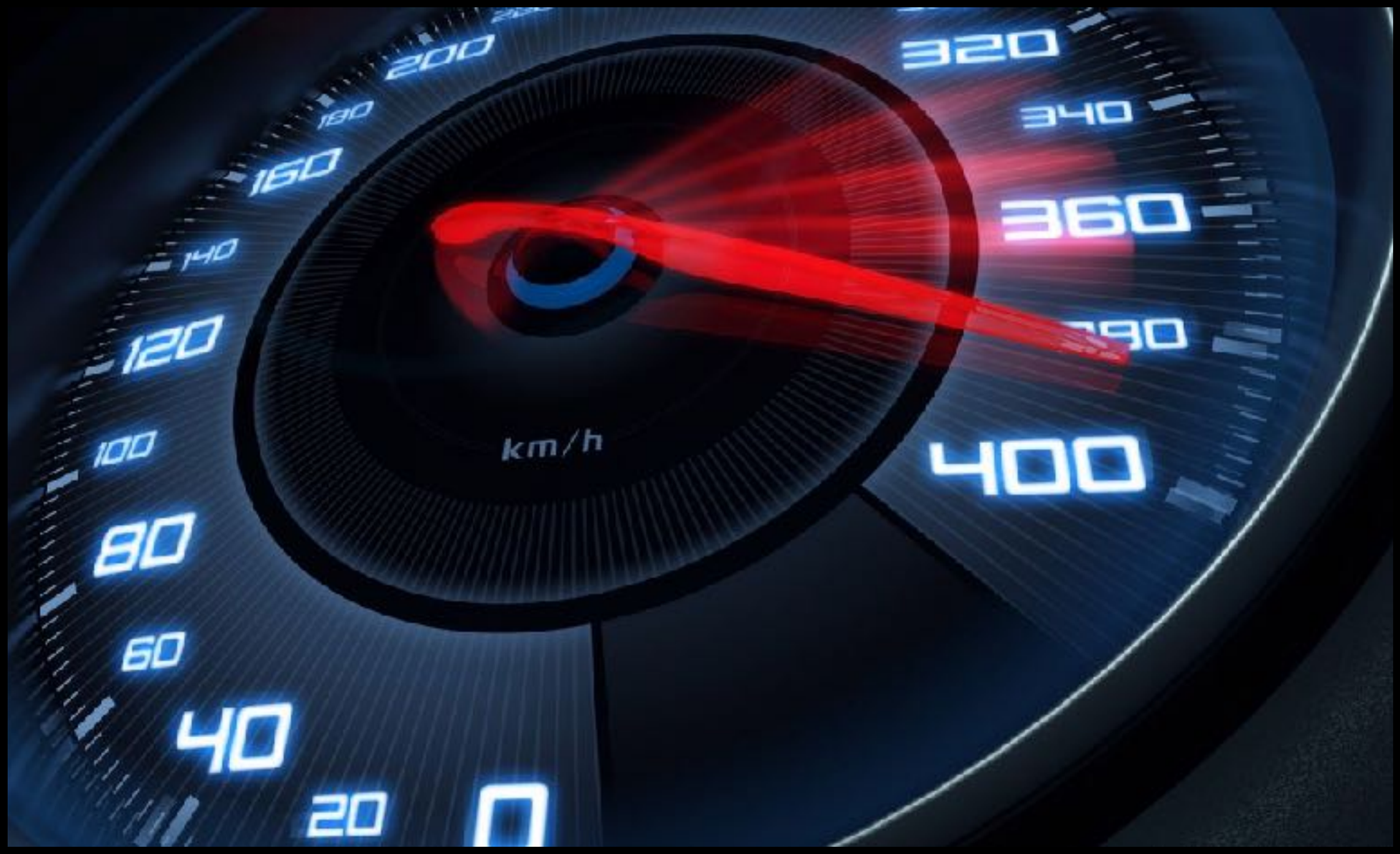
YONITA



Databases

The Mordor of Java Developers

YONITA



Performance

YONITA

Hibernate Puzzle #1
Heads of Hydra

```
@Entity
public class Hydra {
   private Long id;
   private List<Head> heads = new ArrayList<Head>();

   @Id @GeneratedValue
   public Long getId() {…}
   @OneToMany(cascade=CascadeType.ALL)
   public List<Head> getHeads() {
      return Collections.unmodifiableList(heads);
   }
}


// new EntityManager and new transaction:
// creates and persists the hydra with 3 heads


// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

How Many Queries in 2nd Tx?

```
@Entity
public class Hydra {
   private Long id;
   private List<Head> heads = new ArrayList<Head

   @Id @GeneratedValue
   public Long getId() {…}
   @OneToMany(cascade=CascadeType.ALL)
   public List<Head> getHeads() {
      return Collections.unmodifiableList(heads)
   }
}


// new EntityManager and new transaction:
// creates and persists the hydra with 3 heads


// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

(a)1 select
(b)2 selects
(c)1+3 selects
(d)2 selects, 1 delete, 3 inserts
(e)None of the above

How Many Queries in 2nd Tx?

(a) 1 select

(b) 2 selects

(c) 1+3 selects

(d) 2 selects, 1 delete, 3 inserts

(e) None of the above

During commit hibernate checks whether the collection property is dirty (needs to be re-created) by comparing Java identities (object references).

Puzzle #1 Heads of Hydra Another Look

```
@Entity
public class Hydra {
   private Long id;
   private List<Head> heads = new ArrayList<Head>();

   @Id @GeneratedValue
   public Long getId() {…}
   @OneToMany(cascade=CascadeType.ALL)
   public List<Head> getHeads() {
      return Collections.unmodifiableList(heads);
   }
}


// new EntityManager and new transaction:
// creates and persists the hydra with 3 heads


// new EntityManager and new transaction
// during find only 1 select
Hydra found = em.find(Hydra.class, hydra.getId());
// during commit 1 select (heads),1 delete (heads),3 inserts (heads)
```

Lessons Learned

- Expect unexpected ;-)
- Prefer field access mapping
- Operate on collection references returned by hibernate
  - Don’t change collection references unless you know what you’re doing

Lessons Learned

- Expect unexpected ;-)
- Prefer field access mapping
- Operate on collection references returned by hibernate
  - Don’t change collection references unless you know what you’re doing
  - `List<Head> newHeads = new List<>(hydra.getHeads());`
    `hydra.setHeads(newHeads);`

Other Providers?

- EclipseLink
  - `1 select`
- OpenJPA
  - `IllegalAccessError`
  - `not able to enhance the class, in both modes: runtime & build-time enhancing`
- Datanucleus
  - `1 select`
- ‘A Performance Comparison of JPA Providers’

Lessons Learned

- A lot of depends on a JPA provider!
- JPA is a spec
  - A great spec, but only a spec
  - It says what to implement, not how to implement
- You need to tune performance of an application in a concrete environment

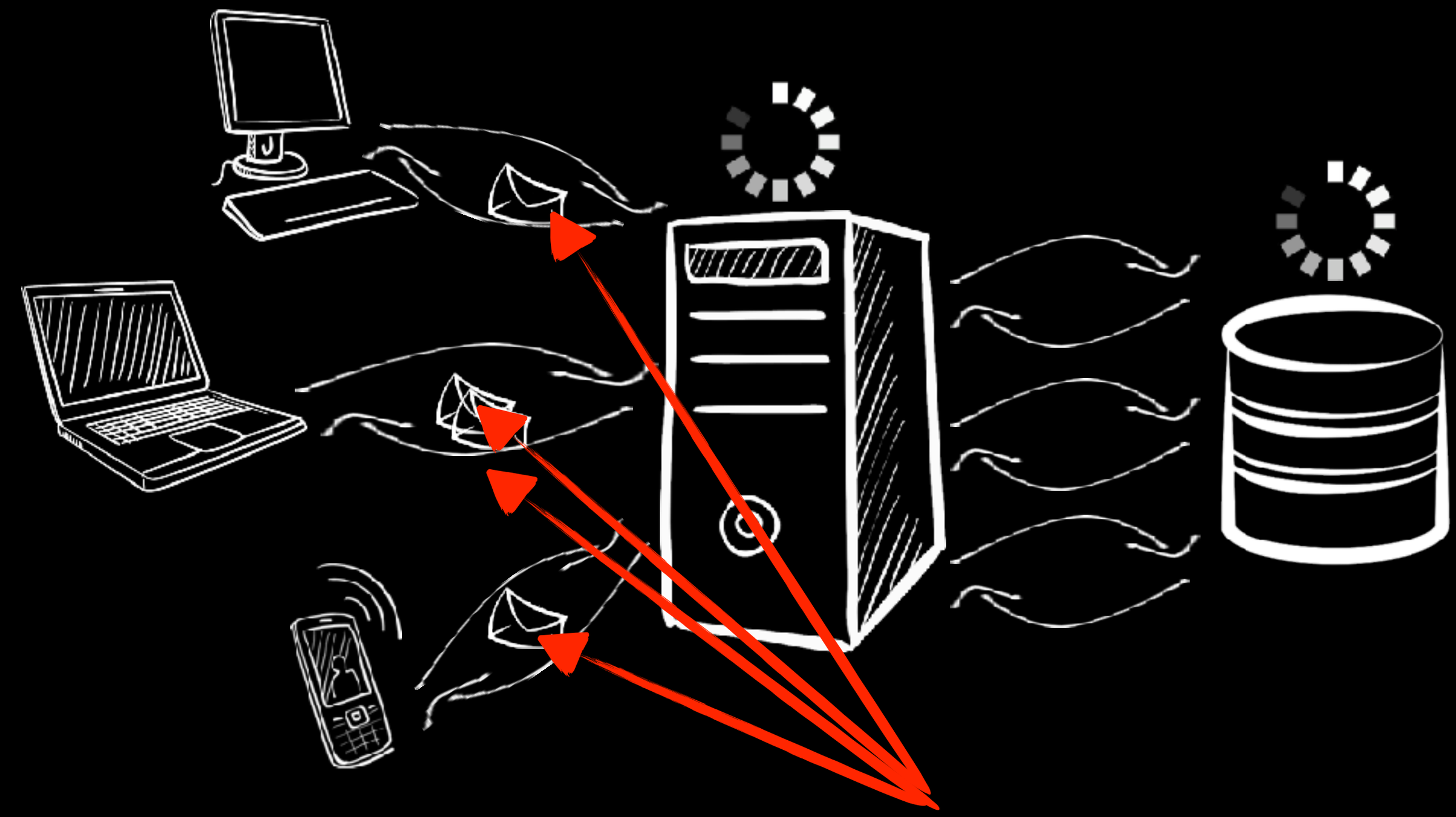
Performance Tuning

- What is performance?
- Why are loading strategies so important?

Request Handling

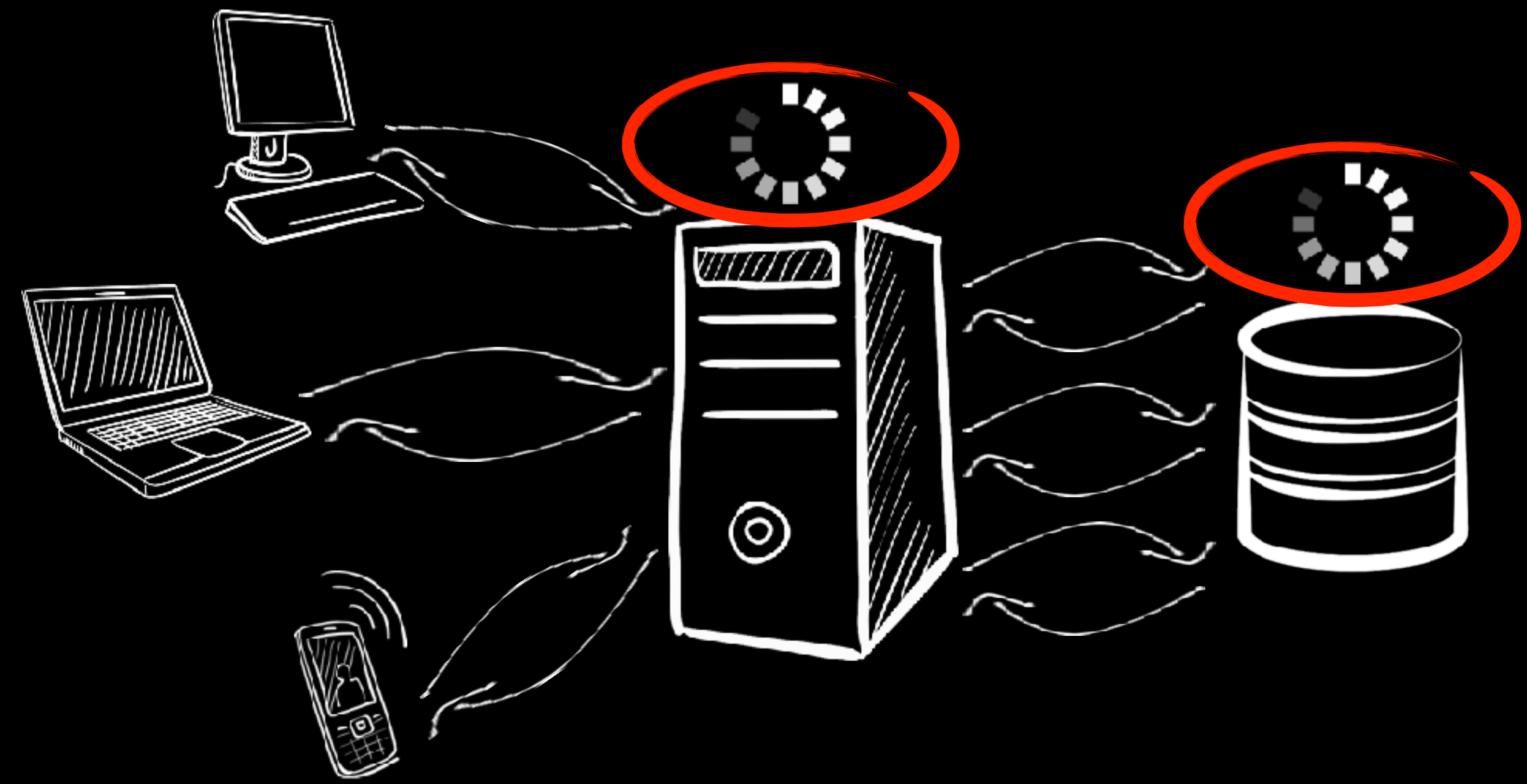
YONITA

Performance: Throughput



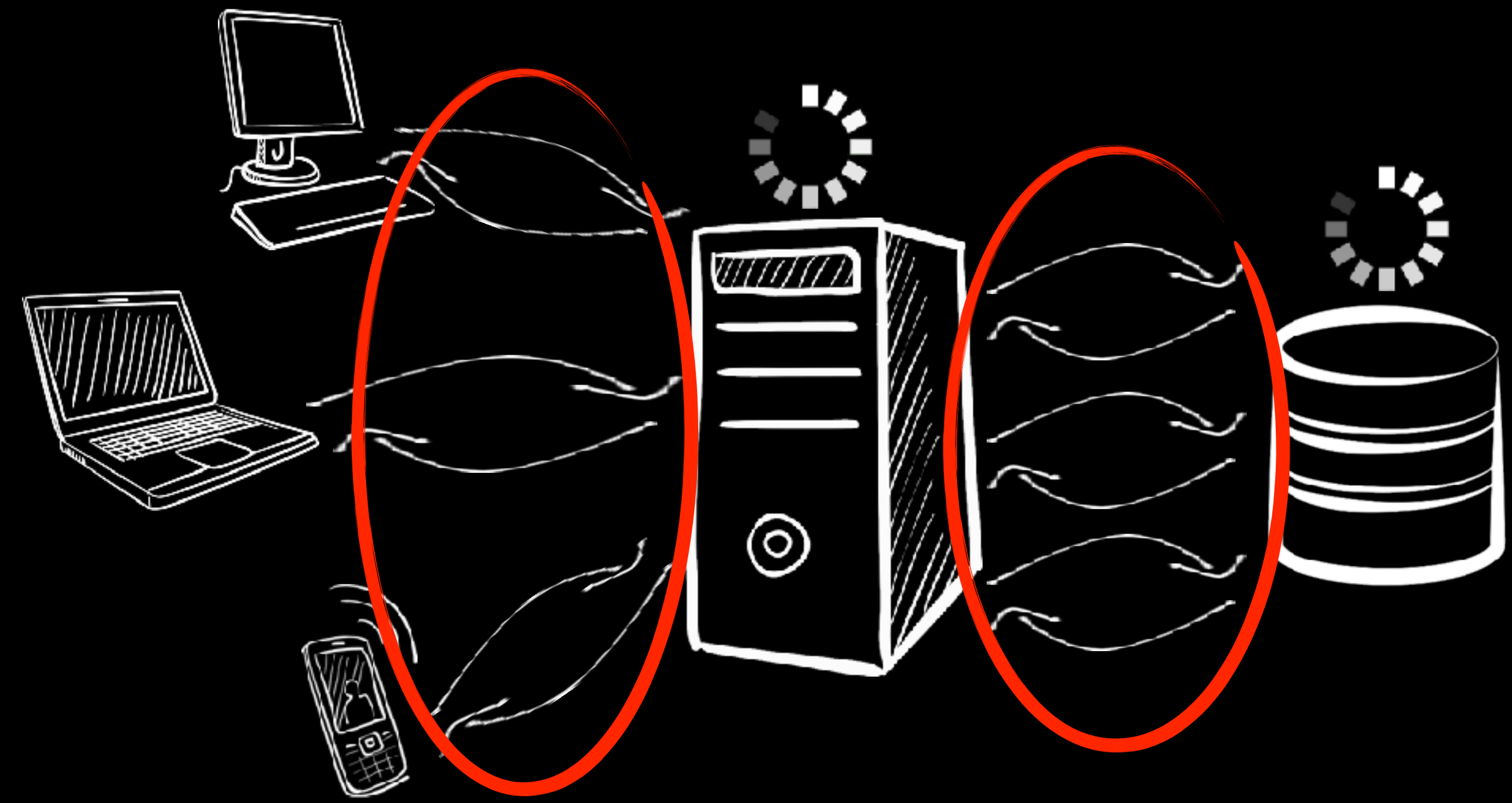
YONITA

Performance: Execution Time



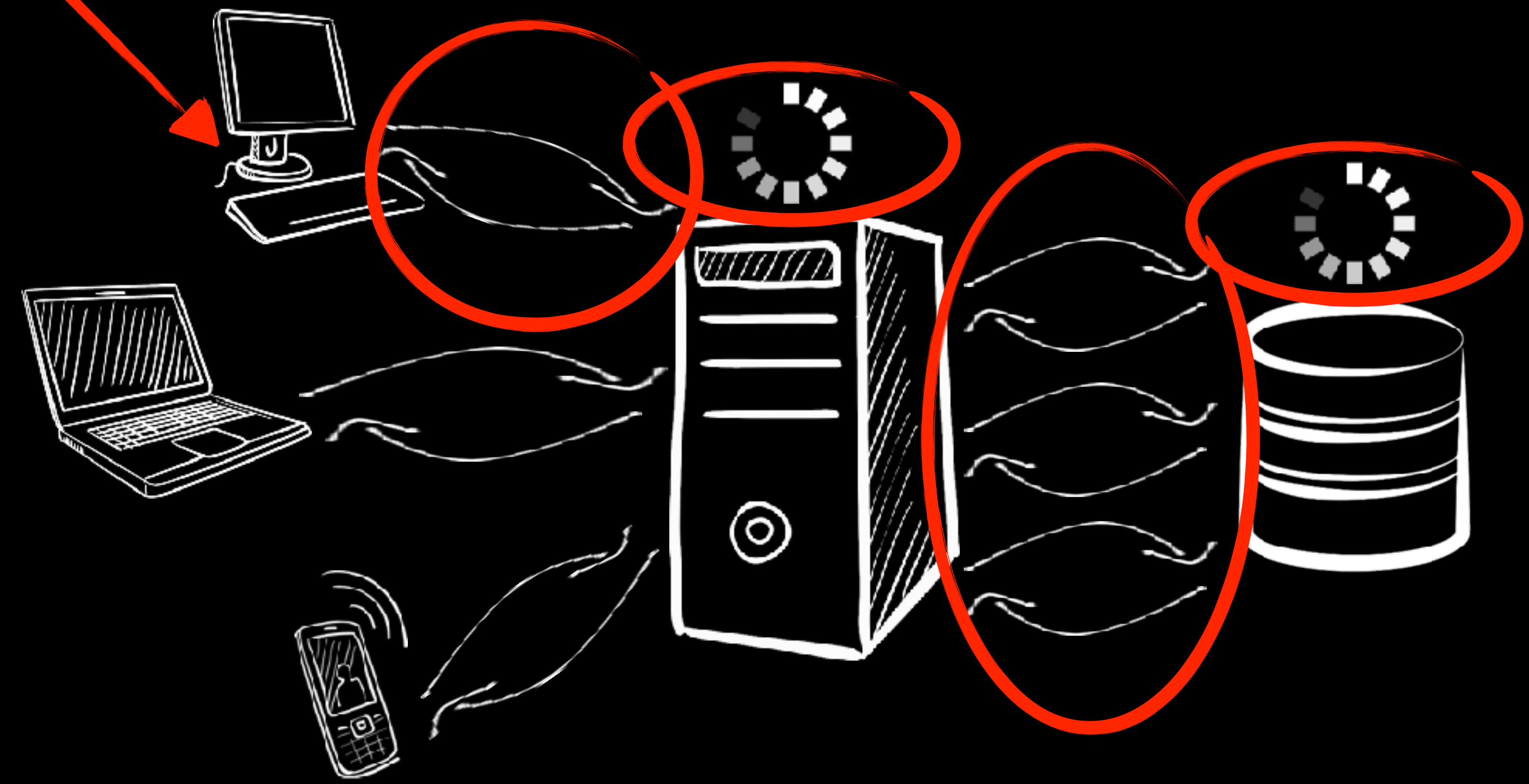
YONITA

Performance: Latency



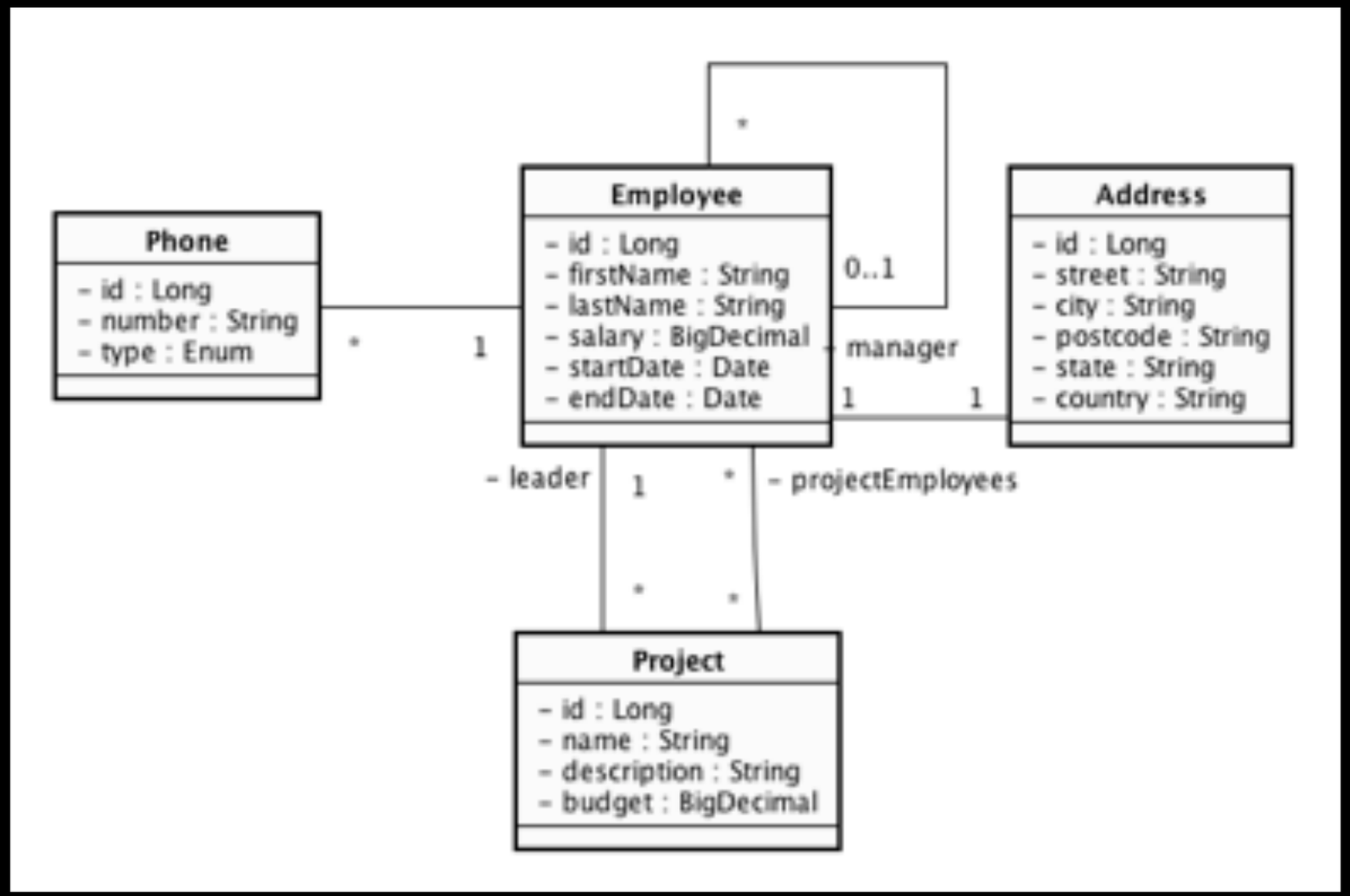
YONITA

Performance: Response Time



YONITA

Example



YONITA

Employee Entity

```
@Entity
 class Employee {
   @Id @GeneratedValue
   private Long id;
   private String firstName;
   private String lastName;
   private BigDecimal salary;
   @OneToOne @JoinColumn(name = "address_id")
   private Address address;
   @Temporal(TemporalType.DATE)
   private Date startDate;
   @Temporal(TemporalType.DATE)
   private Date endDate;
   @ManyToOne @JoinColumn(name = "manager_id")
   private Employee manager;
   // …
}
```

Sum of Salaries By Country Select All (1)

```
TypedQuery<Employee> query = em.createQuery(
    "SELECT e FROM Employee e", Employee.class);
List<Employee> list = query.getResultList();


// calculate sum of salaries by country
// map: country->sum
Map<String, BigDecimal> results = new HashMap<>();
for (Employee e : list) {
    String country = e.getAddress().getCountry();
    BigDecimal total = results.get(country);
    if (total == null) total = BigDecimal.ZERO;
    total = total.add(e.getSalary());
    results.put(country, total);
}
```

Sum of Salaries by Country Select Join Fetch (2)

```
TypedQuery<Employee> query = em.createQuery(
    "SELECT e FROM Employee e
     JOIN FETCH e.address", Employee.class);
List<Employee> list = query.getResultList();


// calculate sum of salaries by country
// map: country->sum
Map<String, BigDecimal> results = new HashMap<>();
for (Employee e : list) {
    String country = e.getAddress().getCountry();
    BigDecimal total = results.get(country);
    if (total == null) total = BigDecimal.ZERO;
    total = total.add(e.getSalary());
    results.put(country, total);
}
```

Sum of Salaries by Country Projection (3)

```
Query query = em.createQuery(
    "SELECT e.salary, e.address.country
     FROM Employee e");
List<Object[]> list = (List<Object[]>) query.getResultList();

// calculate sum of salaries by country
// map: country->sum
Map<String, BigDecimal> results = new HashMap<>();
for (Object[] e : list) {
    String country = (String) e[1];
    BigDecimal total = results.get(country);
    if (total == null) total = BigDecimal.ZERO;
    total = total.add((BigDecimal) e[0]);
    results.put(country, total);
}
```

Sum of Salaries by Country Aggregation JPQL (4)

```
Query query = em.createQuery(
     "SELECT SUM(e.salary), e.address.country
      FROM Employee e
      GROUP BY e.address.country");
List<Object[]> list = (List<Object[]>) query.getResultList();

// already calculated!
```

Comparison 1-4 (Hibernate)
100000 Employees, Different DB Locations

	Local DB (ping: ~0.05ms)	North California (ping: ~38ms)	EU Frankfurt (ping: ~420ms)
(1) Select All (N+1)	26756ms	**2-3 hours**	**~1 day**
(2) Select Join Fetch			
(3) Projection			
(4) Aggregation JPQL			

Comparison 1-4
100000 Employees, Different DB Locations

	Local DB (ping: ~0.05ms)	North California (ping: ~38ms)	EU Frankfurt (ping: ~420ms)
(1) Select All (N+1)	26756ms	**2-3 hours**	**~1 day**
(2) Select Join Fetch	4854ms	18027ms	25096ms
(3) Projection			
(4) Aggregation JPQL			

Comparison 1-4
100000 Employees, Different DB Locations

	Local DB (ping: ~0.05ms)	North California (ping: ~38ms)	EU Frankfurt (ping: ~420ms)
(1) Select All (N+1)	26756ms	**2-3 hours**	**~1 day**
(2) Select Join Fetch	4854ms	18027ms	25096ms
(3) Projection	653ms	2902ms	5006ms
(4) Aggregation JPQL			

Comparison 1-4
100000 Employees, Different DB Locations

	Local DB (ping: ~0.05ms)	North California (ping: ~38ms)	EU Frankfurt (ping: ~420ms)
(1) Select All (N+1)	26756ms	**2-3 hours**	**~20-24 hours**
(2) Select Join Fetch	4854ms	18027ms	25096ms
(3) Projection	653ms	2902ms	5006ms
(4) Aggregation JPQL	182ms	353ms	1198ms

YONITA

Performance Tuning: Data

- Data and processing close to each other
  - Large distance to data => long round-trip => high latency
- Get your data in bigger chunks
  - Many small queries => many round-trips => huge extra time on transport => high latency
- Don't ask about the same data many times
  - Extra processing time + extra transport time

Performance Tuning: Data

- Architecture and design
  - Components, communication, algorithms
- Loading
  - Model
  - Strategies
- Caching
  - Cache as much as you can

JPA Loading Strategies
Load Your Data Smartly

- Loading strategy: No JPA ;-)
  - No JPA-managed entities!
  - Projection and aggregation
- Loading strategy: LAZY for sure
- Loading strategy: EAGER for sure
- Loading strategy: It depends
  - My favourite ;-)

Loading Strategy: No JPA Managed Entities

- Listing and reporting anti-patterns
  - The same model used for different contexts
    - Business context vs. reporting context
  - Too much data loaded
  - Heavy processing on the Java side
- Use projection and aggregation in JPA!

Projection

JPQL => Data Transfer Object

```
Query query = em.createQuery(
”SELECT new com.yonita.jpa.EmployeeDto(
    e.salary, e.address.country)
FROM Employee e”);

// List<EmployeeDto>
List list = query.getResultList();
```

Projection & Aggregation
JPQL => Data Transfer Object

```
Query query = em.createQuery(
"SELECT new com.yonita.jpa.CountryStatDto(
    sum(e.salary), e.address.country)
FROM Employee e
GROUP BY e.address.country");

// List<CountryStatDto>
List list = query.getResultList();
```

Projection & Aggregation
SQL => Data Transfer Object

```
@SqlResultSetMapping(
   name = "countryStatDto",
   classes = {
      @ConstructorResult(
         targetClass = CountryStatDto.class,
         columns = {
         @ColumnResult(name = "ssum", type = BigDecimal.class),
         @ColumnResult(name = "country", type = String.class)
         }
      )
   }
)
```

Projection & Aggregation
SQL => Data Transfer Object

```
Query query = em.createNativeQuery(
"SELECT SUM(e.salary), a.country
FROM employee e
JOIN address a ON e.address_id = a.id
GROUP BY a.country", "countryStatDto");

// List<CountryStatDto>
List list = query.getResultList();
```

Projection Wrap-up

- JPA 2.0
  - Only JPQL query to directly product DTOs
- JPA 2.1
  - JPQL and native queries to produce DTOs
- Managed object
  - Sync with database
  - L1/L2cache
- Use cases for DTOs aka Direct Value Object
  - Reporing, statistics, history
  - Read-only data,UI data
  - Performance:
    - No need for managed objects
    - Rich (or fat) managed objects
    - Subset of attributes required
    - Gain speed
    - Offload an app server

Aggregation Wrap-up

- JPA 2.0
  - Selected aggregation functions: COUNT, SUM, AVG, MIN, MAX
- JPA 2.1
  - All functions as supported by a database
  - Call any database specific function using FUNCTION keyword
- Database specific aggregate functions
  - MSSQL:STDEV,STDEVP,VAR,VARP,...
  - MySQL:BIT_AND,BIT_OR,BIT_XOR,...
  - Oracle:MEDIAN,PERCENTILE,...
  - More...
- Use cases
  - Reporing, statistics, history
  - Performance:
    - Gain speed
    - Offload an app server to a database

Loading Strategy: EAGER for sure!

- We know what we want
  - Known range of required data in this execution path
- We want a little
  - Relatively small entity, no need to divide it into tiny pieces

Loading Strategy: Usually Better EAGER

- Network latency to a database
  - Lower number of round-trips to a database with EAGER loading

Loading Strategy: LAZY for sure!

- We don't know what we want
  - 'I'll think about that tomorrow' by Scarlett O'hara
  - Load only required data
- We want a lot
  - Divide and conquer
  - Load what's needed in the first place

Large Objects

- Lazy Property Fetching
  - @Basic(fetch = FetchType.LAZY)
- Recommended usage
  - Blobs
  - Clobs
  - Formulas
- Remember about byte-code instrumentation
  - Otherwise will not work
  - Silently ignores

Large Object

- Something smells here…
- Do you really need them?

Large Object

- Something smells here...
- Do you really need them?
- But, do you really need them?

Large Object

- Something smells here…
- Do you really need them?
- Ponder over your object model and use cases, otherwise it’s not gonna work

Large Collections

- Divide and conquer!
- Definitely lazy
- You don’t want a really large collection in memory
  - High memory consumption/multithreaded environment => frequent garbage collections => slow server
- Batch size
  - JPA provider specific configuration

Hibernate Puzzle #2
Plant a Tree

```
@Entity
public class Forest {
   @Id @GeneratedValue
   private Long id;
   @OneToMany
   private Collection<Tree> trees = new HashSet<Tree>();
   public void plantTree(Tree tree) {
      return trees.add(tree);
   }
}

// new EntityManager and new transaction:
// creates and persists a forest with 10.000 trees


// new EntityManager and new transaction
Tree tree = new Tree(“oak”); em.persist(tree);
Forest forest = em.find(Forest.class, id); forest.plantTree(tree);
```

How Many Queries in 2nd Tx?

```
@Entity
public class Forest {
   @Id @GeneratedValue
   private Long id;
   @OneToMany
   private Collection<Tree> trees = new Hash
   public void plantTree(Tree tree) {
      return trees.add(tree);
   }
}

// new EntityManager and new transaction:
// creates and persists a forest with 10.000 trees


// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

(a)1 select, 2 inserts
(b)2 selects, 2 inserts
(c)2 selects, 1 delete, 10.000+2 inserts
(d)2 selects, 10.000 deletes, 10.000+2 inserts
(e)Even more ;-)

How Many Queries in 2nd Tx?

(a) 1 select, 2 inserts
(b) 2 selects, 2 inserts
(c) 2 selects, 1 delete, 10.000+2 inserts
(d) 2 selects, 10.000 deletes, 10.000+2 inserts
(e) Even more ;-)

The combination of **OneToMany** and **Collection** enables a bag semantic. That's why the collection is re-created.

Plan a Tree Revisited

```
@Entity
public class Forest {
   @Id @GeneratedValue
   private Long id;
   @OneToMany
   private List<Tree> trees = new ArrayList<
   public void plantTree(Tree tree) {
      return trees.add(tree);
   }
}

// new EntityManager and new transaction:
// creates and persists a forest with 10.000 trees


// new EntityManager and new transaction
Tree tree = new Tree("oak"); em.persist(tree);
Forest forest = em.find(Forest.class, id); forest.plantTree(tree);
```

STILL BAG SEMANTIC

Use OrderColumn or IndexColumn for list semantic.

Plan a Tree Revisited

```
@Entity
public class Forest {
   @Id @GeneratedValue
   private Long id;
   @OneToMany
   private Set<Tree> trees = new HashSet<Tre
   public void plantTree(Tree tree) {
      return trees.add(tree);
   }
}

// new EntityManager and new transaction:
// creates and persists a forest with 10.000 trees


// new EntityManager and new transaction
Tree tree = new Tree(“oak”); em.persist(tree);
Forest forest = em.find(Forest.class, id); forest.plantTree(tree);
```

1. Collection elements loaded into memory
2. Unnecessary queries
3. Transaction and locking schema problems: version/ optimistic locking

Plan a Tree Revisited

```
@Entity
public class Forest {
   @Id @GeneratedValue
   private Long id;
   @OneToMany(mappedBy = “forest”)
   private Set<Tree> trees = new HashSet<Tre
   void plantTree(Tree tree) {
      return trees.add(tree);
   }
}

@Entity public class Tree {
   @Id @GeneratedValue
   private Long id;
   private String name;
   @ManyToOne private Forest forest;

   public void setForest(Forest forest) {
      this.forest = forest;
      forest.plantTree(this);
   }
}
```

Set semantic on the inverse side forces loading of all trees. (when parent/child synced in oo code)

Loading Strategy: It Depends

- You know what you want
  - But it's dynamic, depending on runtime parameters
- That was the problem in JPA 2.0
  - Fetch queries
  - Provider specific extensions
  - Different mappings for different cases
- JPA 2.1 comes in handy
  - Entity Graphs

Entity Graphs in JPA 2.1

- 'A template that captures the paths and boundaries for an operation or query'
- Fetch plans for query or find operations
- Defined by annotations
- Created programmatically

Entity Graphs in JPA 2.1

- Defined by annotations
  - @NamedEntityGraph, @NamedEntitySubgraph, @NamedAttributeNode
- Created programmatically
  - Interfaces EntityGraph, EntitySubgraph, AttributeNode

Entity Graphs in Query or Find

- Default fetch graph
  - Transitive closure of all its attributes specified or defaulted as EAGER
- javax.persistence.fetchgraph
  - Attributes specified by attribute nodes are EAGER, others are LAZY
- javax.persistence.loadgraph
  - Attributes specified by by attribute nodes are EAGER, others as specified or defaulted

Entity Graph Advantages

- Better hints to JPA providers
- Hibernate now generates smarter queries
  - 1 select with joins on 3 tables
  - 1 round-trip to a database instead of default N+1
- Dynamic modification of a fetch plan

Annotation Hell

```
@NamedEntityGraphs({
    @NamedEntityGraph(name="previewEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("body")
    }),
    @NamedEntityGraph(name="fullEmailEntityGraph", attributeNodes={
        @NamedAttributeNode("sender"),
        @NamedAttributeNode("subject"),
        @NamedAttributeNode("body"),
        @NamedAttributeNode("attachments")
    })
})
@Entity
public class EmailMessage { ... }
```

Query Usage

```
EntityGraph<EmailMessage> eg =
em.getEntityGraph("previewEmailEntityGraph");
List<EmailMessage> messages =
em.createNamedQuery("findAllEmailMessages")
        .setParameter("mailbox", "inbox")
        .setHint("javax.persistence.loadgraph", eg)
        .getResultList();
```

It Wouldn’t Be That Bad, If It Worked…

HHH-9298

Embedded NamedAttributeNode not supported in NamedEntityGraph

Type:	Bug	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	4.3.5, 5.0.3	Fix Version/s:	None
Component/s:	None		
Labels:	entitygraph		
Environment:	Wildfly 8.1.0		
Bug Testcase Reminder (view):	Bug reports should generally be accompanied by a test case!		
Last commented by a user?:	true		
Sprint:			

Description

I'm trying to deploy an app to Wildfly 8.1.0 (Hibernate 4.3.5), but Hibernate AttributeNodeImpl is throwing an exception - "Attribute x is not a managed type". I looked at the code and it's checking for Basic or Embedded annotation on lines 123-128 and throwing an exception if found.

My attribute is Embedded. According to JPA spec, Entity, MappedSuperclass and Embeddable are all

There’s that question…

YONITA

A fool with a tool is only a fool!

YONITA

Continuous Learning

YONITA

Q&A

- patrycja@yonita.com
- @yonlabs

YONITA