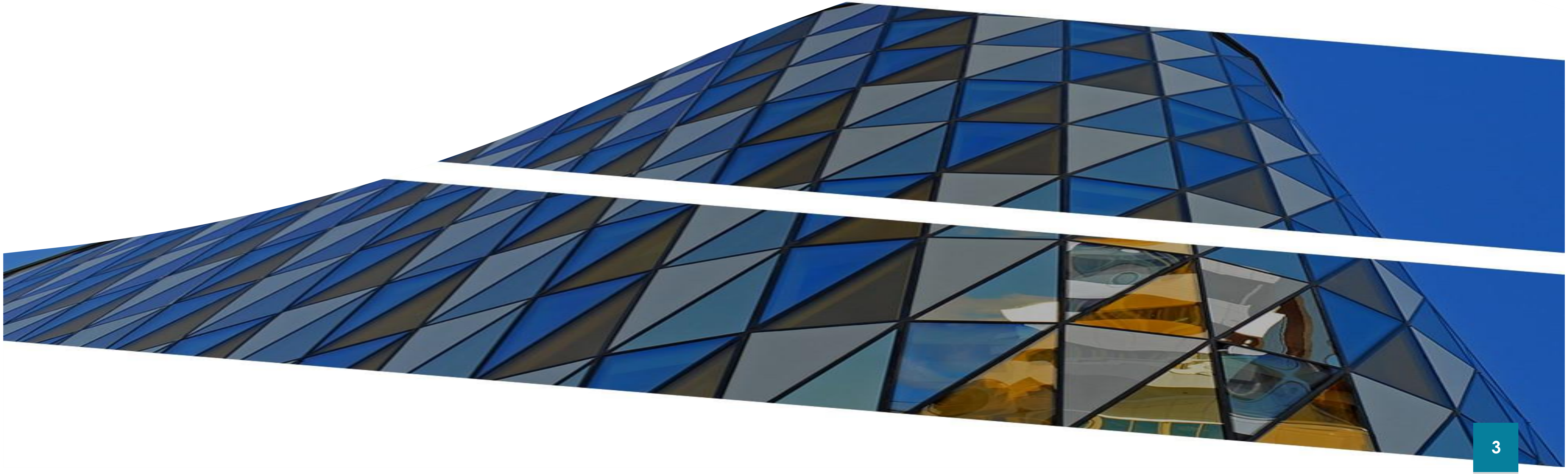
SOA. REST. DDD. Working with SQL Databases in Go

Where to Find The Code and Materials?

https://github.com/iproduct/coursegopro

SOA & REST

Developing horizontally scalable web services



Service Oriented Architecture (SOA) – Definitions

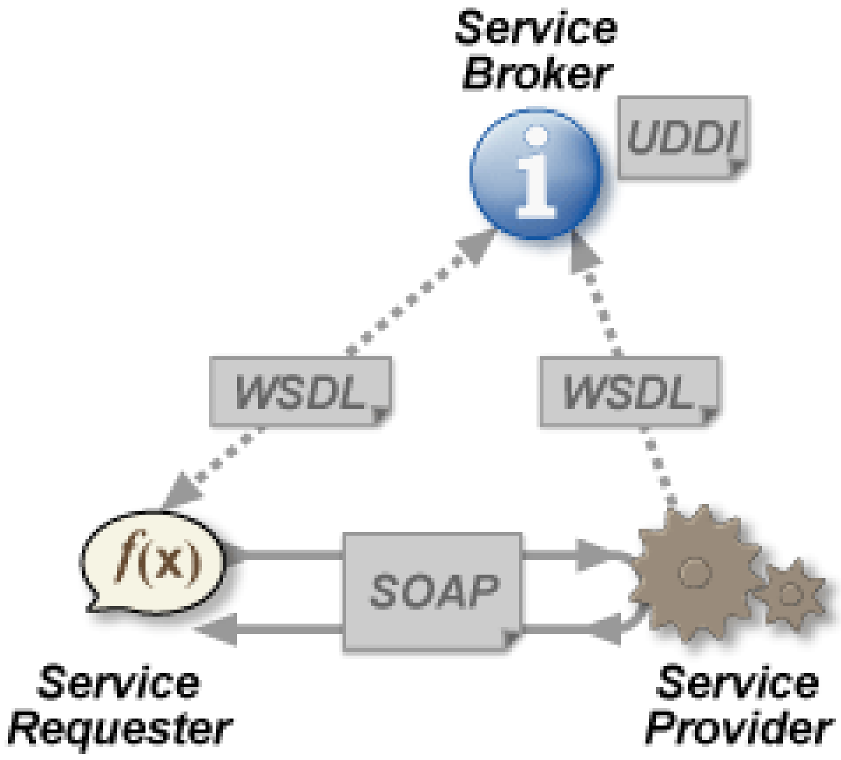
Thomas Erl: SOA represents an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services. SOA can establish an abstraction of business logic and technology, resulting in a loose coupling between these domains. SOA is an evolution of past platforms, preserving successful characteristics of traditional architectures, and bringing with it distinct principles that foster service-orientation in support of a service-oriented enterprise. SOA is ideally standardized throughout an enterprise, but achieving this state requires a planned transition and the support of a still evolving technology set.

References: Erl, Thomas. serviceorientation.org – About the Principles, 2005–06

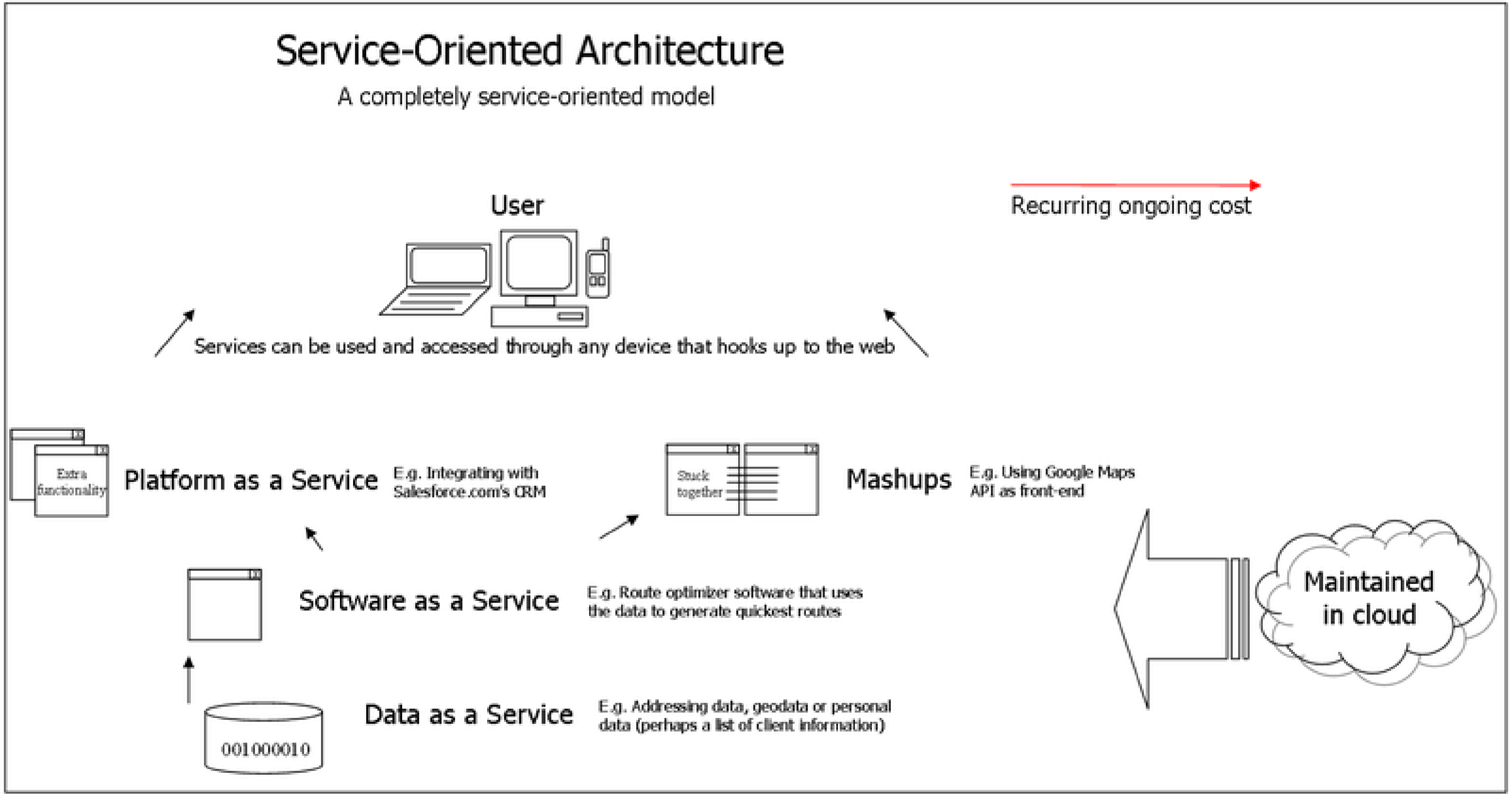
Classical Web Services - SOAP + WSDL

- Web Services are:

- components for building distributed applications in SOA architectural style
- communicate using open protocols
- are self-descriptive and self-content
- can be searched and found using UDDI or ebXML registries (and more recent specifications – WSIL & **Semantic Web Services**)



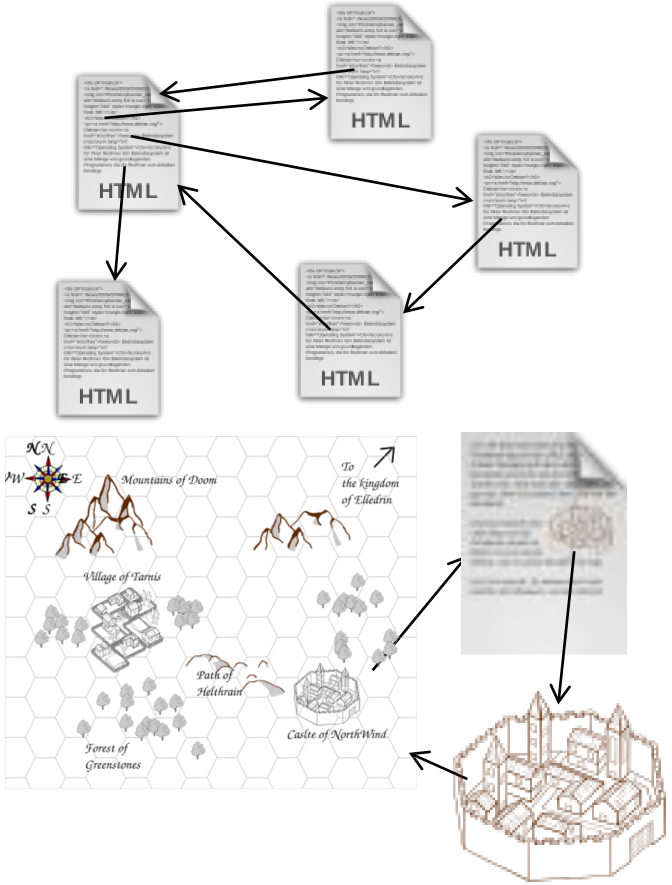
Service Oriented Architecture (SOA)



Source: http://en.wikipedia.org/wiki/File:SOA_Detailed_Diagram.png,
Author: JamesLWilliams2010, License: Creative Commons Attribution 3.0 Unported

Hypertext & Hypermedia

- Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text
- HTTP is the protocol to exchange or transfer hypertext
- Hypermedia - extension of the term hypertext, is a nonlinear medium of information which includes multimedia (text, graphics, audio, video, etc.) and hyperlinks of different media types (e.g. image or animation/video fragment can be linked to a detailed description).



REST Architecture

According to Roy Fielding [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

- Client-Server
- Stateless
- Uniform Interface:
  - Identification of resources
  - Manipulation of resources through representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)
- Layered System
- Code on Demand (optional)

Representational State Transfer (REST)

- REpresentational State Transfer (REST) is an architecture for accessing distributed hypermedia web-services
- The resources are identified by URIs and are accessed and manipulated using an HTTP interface base methods (GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH)
- Information is exchanged using representations of these resources
- Lightweight alternative to SOAP+WSDL -> HTTP + Any representation format (e.g. JavaScript Object Notation – JSON)

HTTP Request Structure

GET */context/Servlet* **HTTP/1.1**

Host: *Client_Host_Name*

Header2: *Header2_Data*

…

HeaderN: *HeaderN_Data*

<Празен ред>

POST */context/Servlet* **HTTP/1.1**

Host: *Client_Host_Name*

Header2: *Header2_Data*

…

HeaderN: *HeaderN_Data*

<Празен ред>

POST_Data

HTTP Response Structure & JSON

HTTP/1.1 200 OK

Content-Type: application/json

Header2: Header2_Data

…

HeaderN: HeaderN_Data

<Празен ред>

```
[{ "id":1,
   "name":"Novelties in Java EE 7 ...",
   "description":"The presentation is ...",
   "created":"2014-05-10T12:37:59",
   "modified":"2014-05-10T13:50:02",
 },
 { "id":2,
   "name":"Mobile Apps with HTML5 ...",
   "description":"Building Mobile ...",
   "created":"2014-05-10T12:40:01",
   "modified":"2014-05-10T12:40:01",
}]
```

Representational State Transfer (REST)

- Identification of resources – URIs
- Representation of resources – e.g. HTML, XML, JSON, etc.
- Manipulation of resources – through these representations
- Self-descriptive messages - Internet media type (MIME type) provides enough information to describe how to process the message. Responses also explicitly indicate their cacheability.
- Hypermedia as the engine of application state (aka **HATEOAS**)
- Application contracts are expressed as **media types** and [semantic] link relations (**rel** attribute - RFC5988, "Web Linking")

Multipurpose Internet Mail Extensions (MIME)

- Different types of media are represented using different text/binary encoding formats – for example:
  - Text -> plain, html, xml …
  - Image (Graphics) -> gif, png, jpeg, svg …
  - Audio & Video -> mp3, ogg, webm …
- Multipurpose Internet Mail Extensions (MIME) allows the client to recognize how to handle/present the particular multimedia asset/node:

  Media Type Media SubType (format)

  - Ex.: Content-Type: text/plain
- More examples for standard MIME types: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types
- Vendor specific media (MIME) types: application/vnd.*+json/xml

Hypermedia As The Engine Of Application State (HATEOAS) – New Link Header (RFC 5988) Example

```
Content-Length →1656

Content-Type →application/json

Link →<http://localhost:8080/polling/resources/polls/629>; rel="prev";
  type="application/json"; title="Previous poll",
  <http://localhost:8080/polling/resources/polls/632>; rel="next";
  type="application/json"; title="Next poll",
  <http://localhost:8080/polling/resources/polls>; rel="collection";
  type="application/json"; title="Polls collection",
  <http://localhost:8080/polling/resources/polls>; rel="collection up";
  type="application/json"; title="Self link",
  <http://localhost:8080/polling/resources/polls/630>; rel="self"
```

Advantages of REST

- Scalability of component interactions – through layering the client server-communication and enabling load-balancing, shared caching, security policy enforcement;
- Generality of interfaces – allowing simplicity, reliability, security and improved visibility by intermediaries, easy configuration, robustness, and greater efficiency by fully utilizing the capabilities of HTTP protocol;
- Independent development and evolution of components – dynamic evolvability of services, without breaking existing clients.
- Fault tolerant, Recoverable, Secure, Loosely coupled

Richardson's Maturity Model of Web Services

According to **Leonard Richardson** [Talk at QCon, 2008 - http://www.crummy.com/writing/speaking/2008-QCon/act3.html]:

- Level 0 – POX: Single URI (XML-RPC, SOAP)
- Level 1 – Resources: Many URIs, Single Verb (URI Tunneling)
- Level 2 – HTTP Verbs: Many URIs, Many Verbs (CRUD – e.g Amazon S3)
- Level 3 – Hypermedia Links Control the Application State = HATEOAS (Hypertext As The Engine Of Application State) === **truely** RESTful Services

Simple Example: URLs + HTTP Methods

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as http://api.example.com/comments/	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Individual element, such as http://api.example.com/comments/11	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it does not exist, create it.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Source: https://en.wikipedia.org/wiki/Representational_state_transfer

Web Application Description Language (WADL)

- XML-based file format providing machine-readable description of HTTP-based web application resources – typically RESTful web services
- WADL is a W3C Member Submission
  - Multiple resources
  - Inter-connections between resources
  - HTTP methods that can be applied accessing each resource
  - Expected inputs, outputs and their data-type formats
  - XML Schema data-type formats for representing the RESTful resources
- But WADL resource description is **static**

Cross-Origin Resource Sharing (CORS)

- Enables execution of cross-domain requests for web resources by allowing the server to decide which scripts (from which domains – Origin) are allowed to receive/manipulate such resources, as well as which HTTP Methods (GET, POST, PUT, DELTE, etc.) are allowed.
- In order to implement this mechanism, when the HTTP methods differs from simple GET, a prefight OPTIONS request is issued by the web browser, in response to which the server returns the allowed methods, custom headers, etc. for the requested web resource and script Origin.

HTTP Headers with CORS Simple Requests

- HTTP GET simple request

GET /crossDomainResource/ HTTP/1.1

Referer: http://sample.com/crossDomainMashup/

Origin: http://sample.com

- HTTP GET response

Access-Control-Allow-Origin: http://sample.com

Content-Type: application/xml

…

HTTP Headers with CORS POST/PUT/DELETE Requests

- HTTP OPTIONS preflight request

```
OPTIONS /crossDomainPOSTResource/ HTTP/1.1
Origin: http://sample.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: MYHEADER
```

- HTTP response

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://sample.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: MYHEADER
Access-Control-Max-Age: 864000
```

RESTful Patterns and Best Practices

According to **Cesare Pautasso** [http://www.jopera.org/files/SOA2009-REST-Patterns.pdf]:

- Uniform Contract
- Content Negotiation
- Entity Endpoint
- Endpoint Redirection
- Distributed Response Caching
- Entity Linking
- Idempotent Capability

REST Antipatterns and Worst Practices

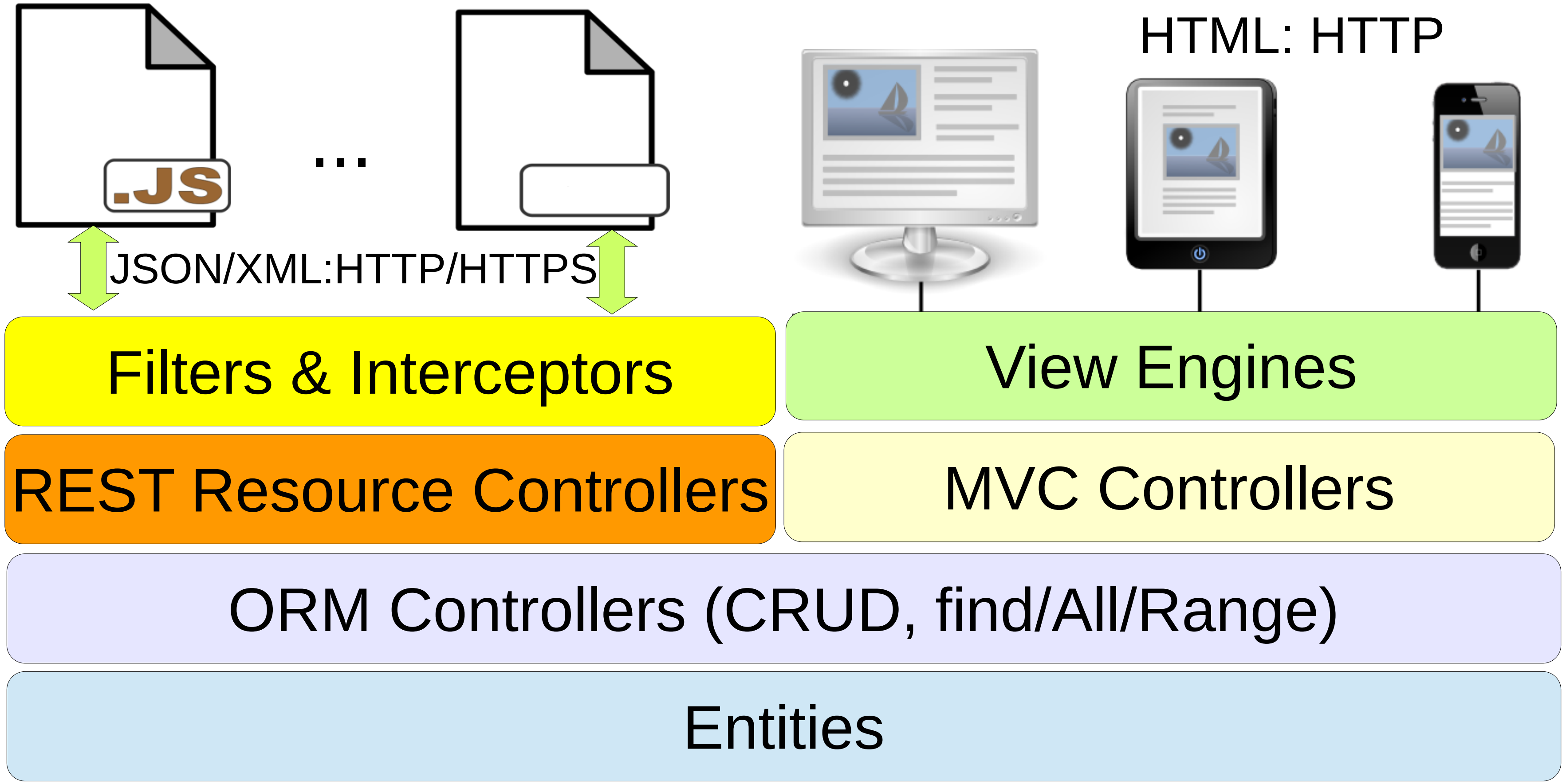
According to **Jacob Kaplan-Moss** [http://jacobian.org/writing/rest-worst-practices/]:

- Conflating models and resources
- Hardcoded authentication
- Resource-specific output formats
- Hardcoded output formats
- Weak HTTP method support (e.g. tunell everything through GET/POST)
- Improper use of links
- Couple the REST API to the application

Web Service Architectures

Coping with the Complexity – Domain Driven Design

N-Tier Web Architectures



Domain Driven Design (DDD)

We need tools to cope with the complexity inherent in most real-world design problems.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

Domain Driven Design (DDD) – back to basics: domain objects, data and logic.

Described by Eric Evans in his book:
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

Domain Driven Design (DDD)

Main concepts:

- Entities, value objects and modules
- Aggregates and Aggregate Roots [Haywood]:
  value < entity < aggregate < module < BC
- Repositories, Factories and Services:
  application services <-> **domain services**
- Separating interface from implementation

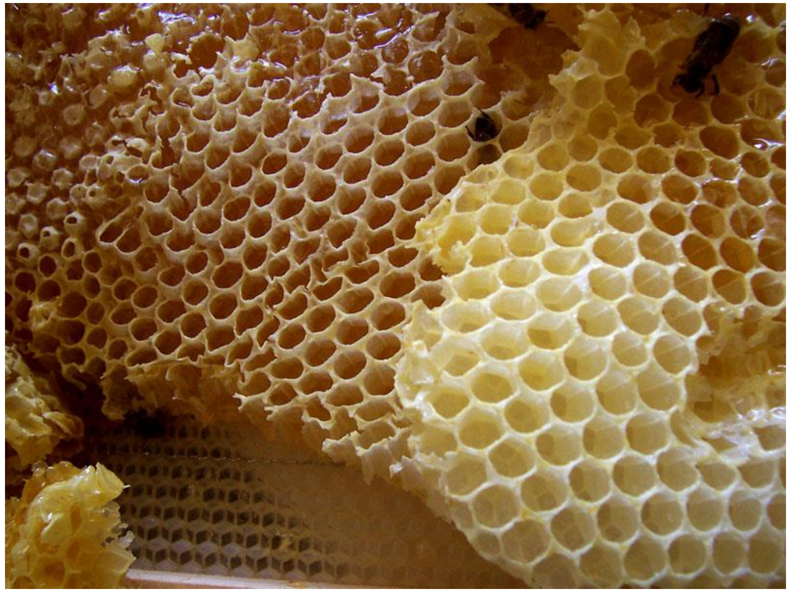
Domain Driven Design (DDD) & Microservices

- Ubiquitous language and Bounded Contexts
- DDD Application Layers:
- Infrastructure, Domain, Application, Presentation
- Hexagonal architecture :

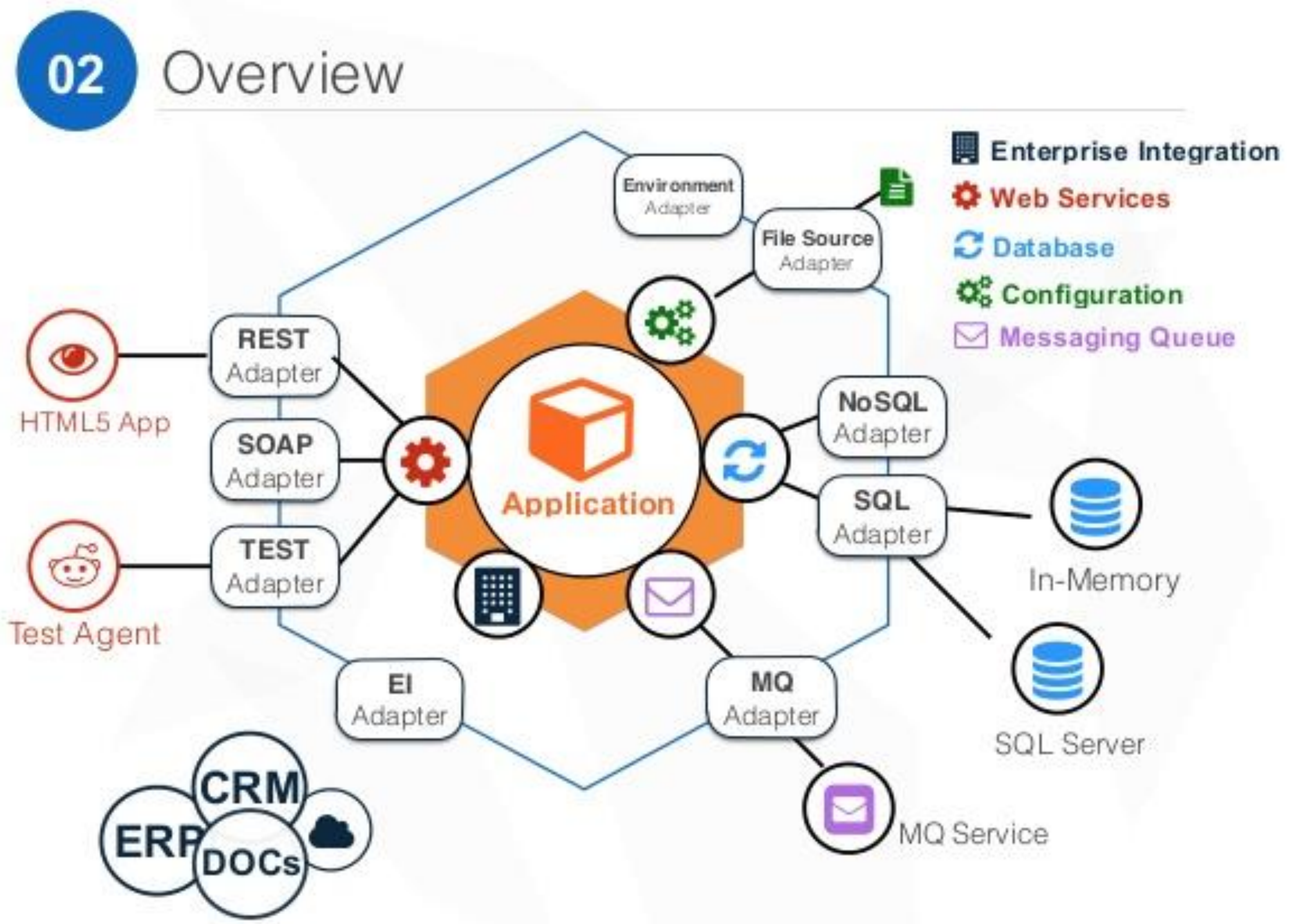
  OUTSIDE <-> transformer <->

  (application <-> domain)

  [A. Cockburn]



Hexagonal Architecture



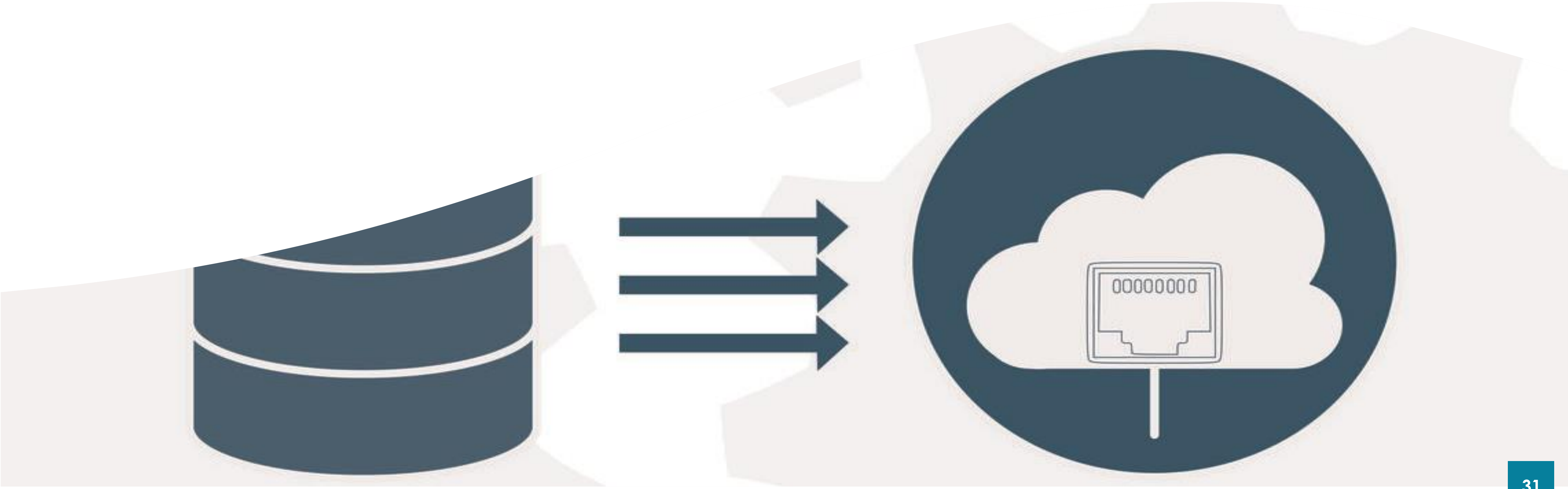
Source: https://www.slideshare.net/fabricioepa/hexagonal-architecture-for-java-applications

Hexagonal Architecture Design Principles

- Allows an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.
- As events arrive from the outside world at a port, a technology-specific adapter converts it into a procedure call or message and passes it to the application
- Application sends messages through ports to adapters, which signal data to the receiver (human or automated)
- The application has a semantically sound interaction with all the adapters, without actually knowing the nature of the things on the other side of the adapters

Source: http://alistair.cockburn.us/Hexagonal+architecture

Introduction to Databases and Distributed Data Processing



Databases, DBMSs and DB Models

- Database - an organized collection of data, generally stored and accessed electronically from a computer system. Can be developed using formal design and modeling techniques.
- DataBase Management System (DBMS) – software that interacts with end users, applications, and the database to capture and analyze the data, providing core facilities to create and administer databases.
- DBMSs can be classified according to the database models that they support:
  - In 1980s relational databases became dominant, modelling data as rows and columns in a series of tables, and the vast majority use Structured Query Language (SQL) for writing and querying data.
  - In the 2000s, non-relational databases became popular, referred to as NoSQL because they use different query languages.

Relational Databases

- "Relational database" term – invented by E. F. Codd at IBM in 1970, paper: "A Relational Model of Data for Large Shared Data Banks".
- Present the data to the user as relations (a presentation in tabular form, i.e. as a collection of tables with each table consisting of a set of rows and columns)
- Provide relational operators to manipulate the data in tabular form.
- As of 2009, most commercial relational DBMSs employ SQL as their query language.

```
dvdrental=# select title, release_year, length, replacement_cost from film
dvdrental-#   where length > 120 and replacement_cost > 29.50
dvdrental-#   order by title desc;
          title          | release_year | length | replacement_cost
-------------------------+--------------+--------+------------------
 West Lion               |         2006 |    159 |            29.99
 Virgin Daisy            |         2006 |    179 |            29.99
 Uncut Suicides          |         2006 |    172 |            29.99
 Tracy Cider             |         2006 |    142 |            29.99
 Song Hedwig             |         2006 |    165 |            29.99
 Slacker Liaisons        |         2006 |    179 |            29.99
 Sassy Packer            |         2006 |    154 |            29.99
 River Outlaw            |         2006 |    149 |            29.99
 Right Cranes            |         2006 |    153 |            29.99
 Quest Mussolini         |         2006 |    177 |            29.99
 Poseidon Forever        |         2006 |    159 |            29.99
 Loathing Legally        |         2006 |    140 |            29.99
 Lawless Vision          |         2006 |    181 |            29.99
 Jingle Sagebrush        |         2006 |    124 |            29.99
 Jericho Mulan           |         2006 |    171 |            29.99
 Japanese Run            |         2006 |    135 |            29.99
 Gilmore Boiled          |         2006 |    163 |            29.99
 Floats Garden           |         2006 |    145 |            29.99
 Fantasia Park           |         2006 |    131 |            29.99
 Extraordinary Conquerer |         2006 |    122 |            29.99
 Everyone Craft          |         2006 |    163 |            29.99
 Dirty Ace               |         2006 |    147 |            29.99
 Clyde Theory            |         2006 |    139 |            29.99
 Clockwork Paradise      |         2006 |    143 |            29.99
 Ballroom Mockingbird    |         2006 |    173 |            29.99
(25 rows)
```

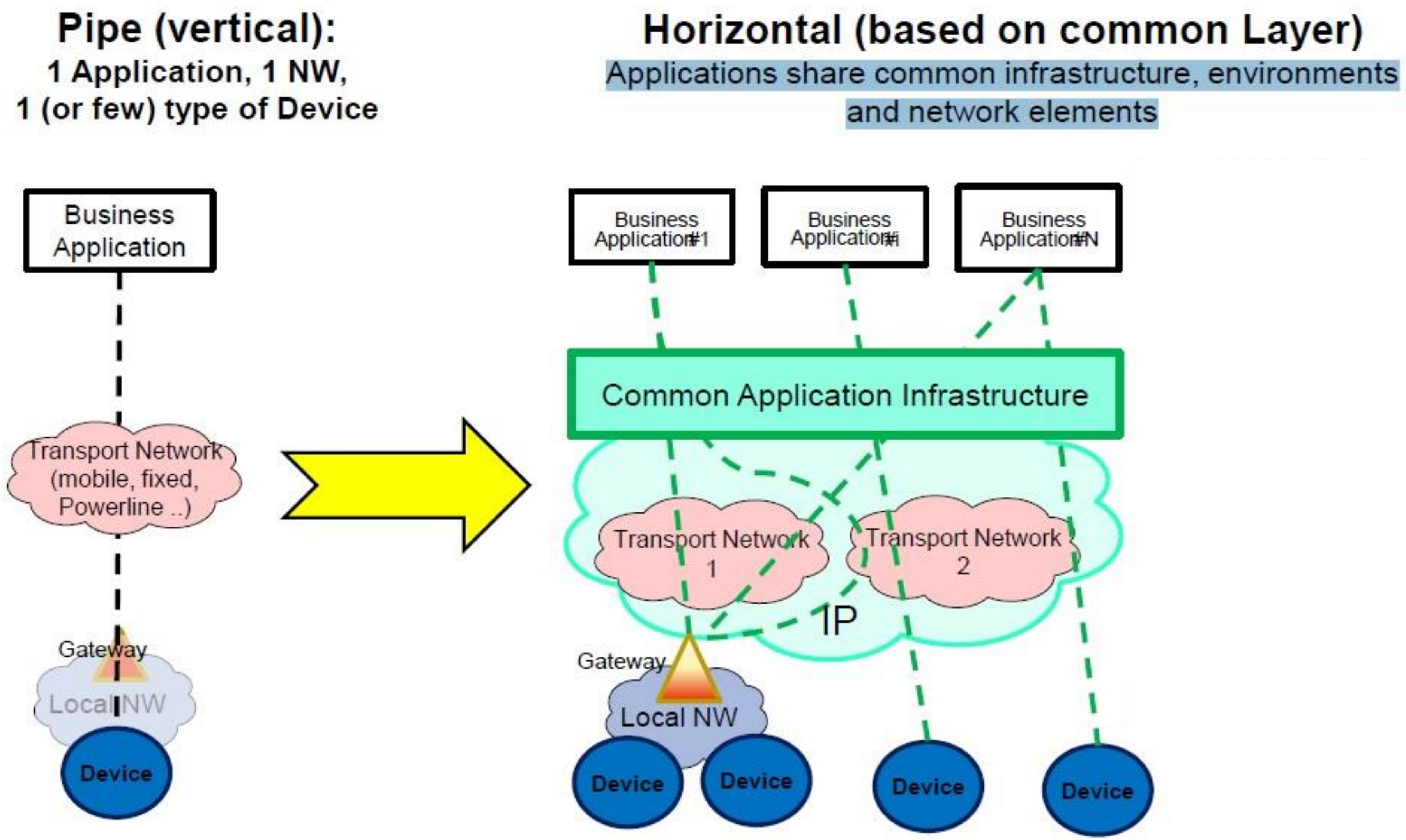
Examples: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, IBM DB2, SQLite

NoSQL and NewSQL Databases

- NoSQL databases – massively distributed, horizontally scalable, fast, do not require fixed table schemas, avoid join operations by storing denormalized data.
- CAP theorem: it is impossible for a distributed system to simultaneously provide consistency, availability, and partition tolerance guarantees –> eventual consistency = high availability and partition tolerance with a reduced level of data consistency.
- NewSQL is a class of modern relational databases that aims to provide the same scalable performance of NoSQL systems for online transaction processing (read-write) workloads while still using SQL and maintaining the ACID guarantees of a traditional database system.

Source: Wikipedia - https://en.wikipedia.org/wiki/Database

Vertical vs. Horizontal Scaling



http://www.iot-week.eu/overview_2013/iot-week-2012/programme-1/tuesday-1/presentations/semantic-interoperability/K1_ETSI-M2M-oneM2M-and-the-need-for-semantics-IoTWeek2012.pdf, By Joerg Swetina (NEC)- ETSI M2M / oneM2M and the need for semantics

NoSQL and NewSQL Database Examples

Type	Notable examples of this type
Wide column	Accumulo, Cassandra, Scylla, HBase
Document:	Apache CouchDB, ArangoDB, BaseX, Clusterpoint, Couchbase, Cosmos DB, eXist-db, IBM Domino, MarkLogic, MongoDB, OrientDB, Qizx, RethinkDB
Key–value:	Aerospike, Apache Ignite, ArangoDB, Berkeley DB, Couchbase, Dynamo, FoundationDB, InfinityDB, MemcacheDB, MUMPS, Oracle NoSQL Database, OrientDB, Redis, Riak, SciDB, SDBM/Flat File dbm, ZooKeeper
Graph:	AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso
New SQL	CockroachDB, Citus, Vitess

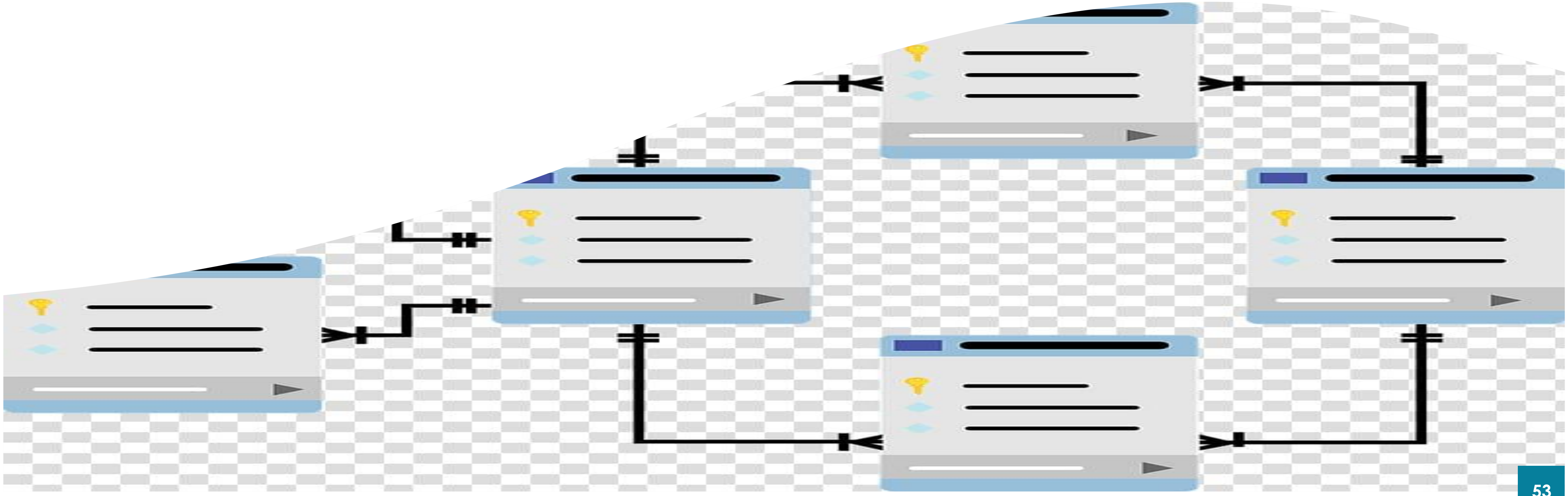
Source: Wikipedia - https://en.wikipedia.org/wiki/NoSQL

SQL and NoSQL Databases Comparison

Data model	Performance	Scalability	Flexibility	Complexity	Functionality
Key–value store	high	high	high	none	variable (none)
Column-oriented store	high	high	moderate	low	minimal
Document-oriented store	high	variable (high)	high	low	variable (low)
Graph database	variable	variable	high	high	graph theory
Relational database	variable	variable	low	moderate	relational algebra

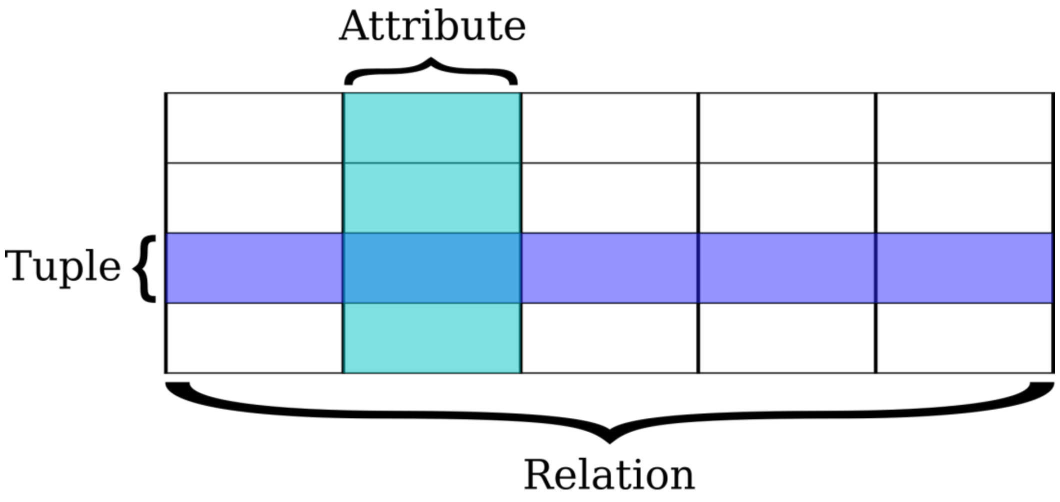
Source: Wikipedia - https://en.wikipedia.org/wiki/NoSQL

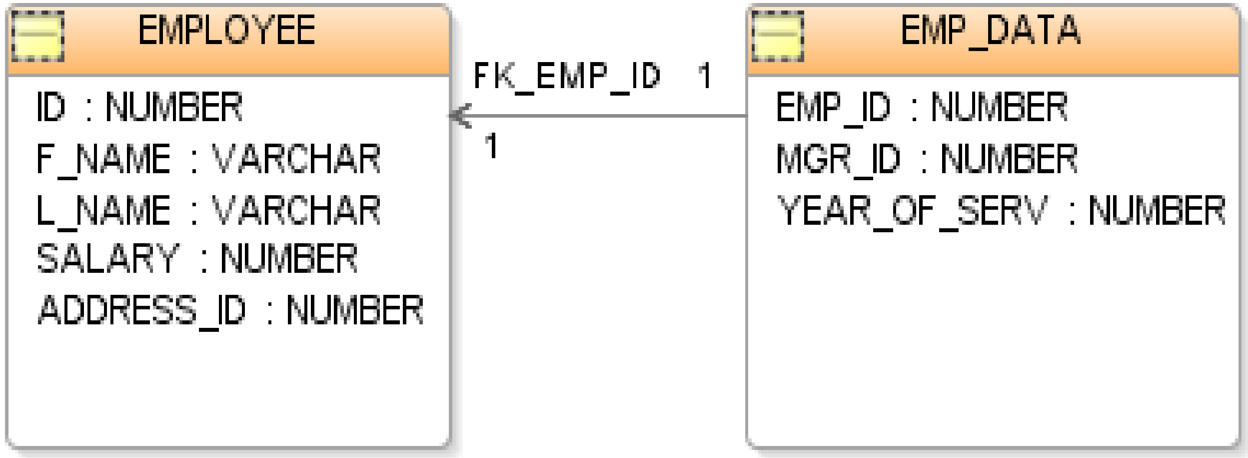
Relational Databases



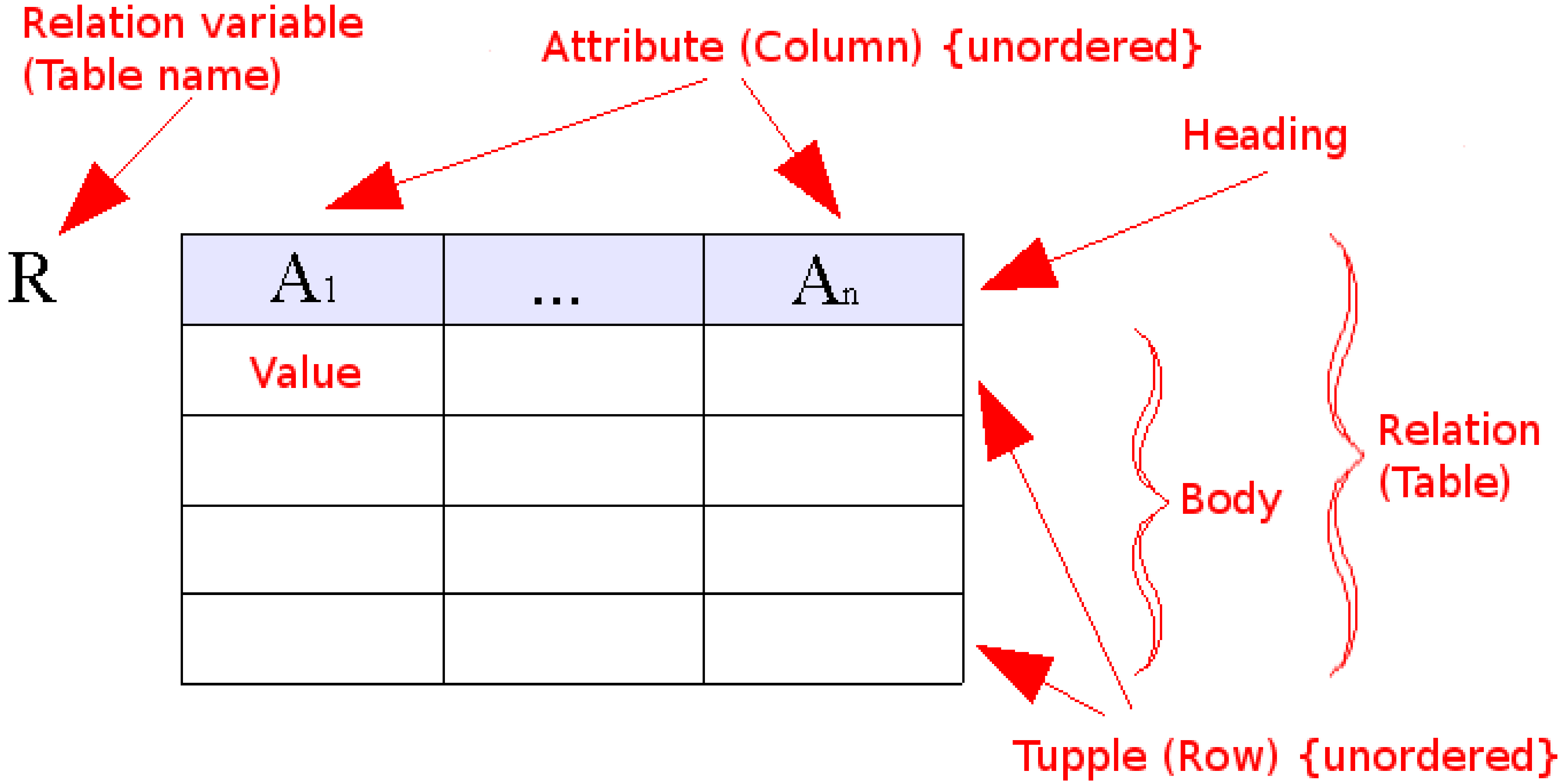
Relational Model

- relation ↔ table
- record, tuple ↔ row
- attribute ↔ column





Relational Model



Views. Domains. Constraints

- Def: Relations which store primary data are called **base relations** or **tables**. Other relations, which are derived from primary relations are **queries** and **views**.

- Def: **Domain** in database is a set of allowed values for a given attribute in a relation – an existing constraint about valid the type of values for given attribute.

- Def: **Constraints** allow more flexible specification of values that are valid for given attribute – e.g. from 1 to 10.

Keys

- **Key** consists of one or more attributes, such that:
  - 1) relation has no two records with the same values for these attributes
  - 2) there is no proper subset of these attributes with the same property
- **Primary Key** is an attribute (less frequently a group of attributes), which uniquely identifies each record (tupple) in the relation
- **Foreign key** is necessary when there exists a relation between two tables – usually it is an attribute in second table referring to primary key of the first table

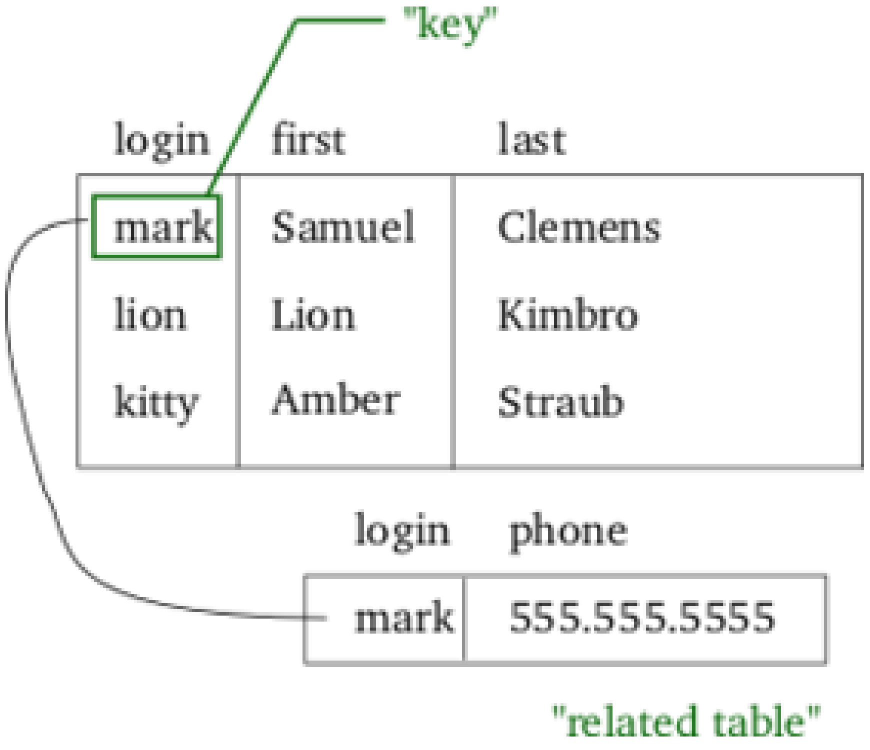
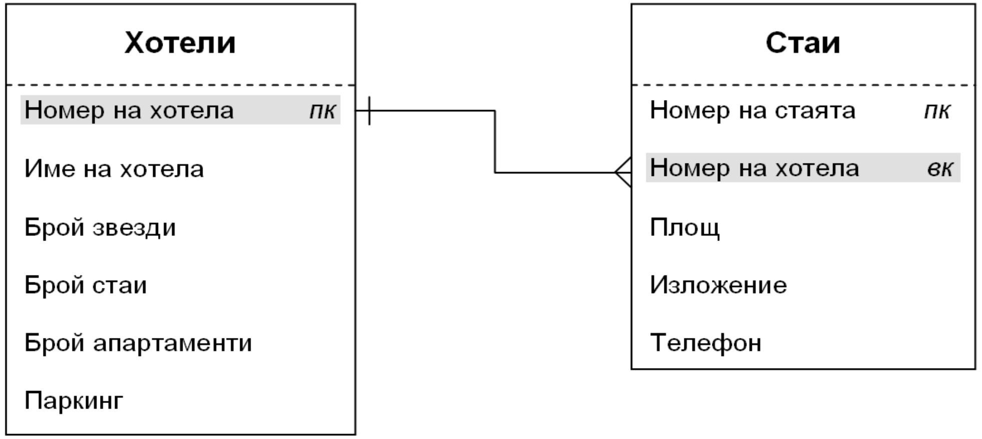
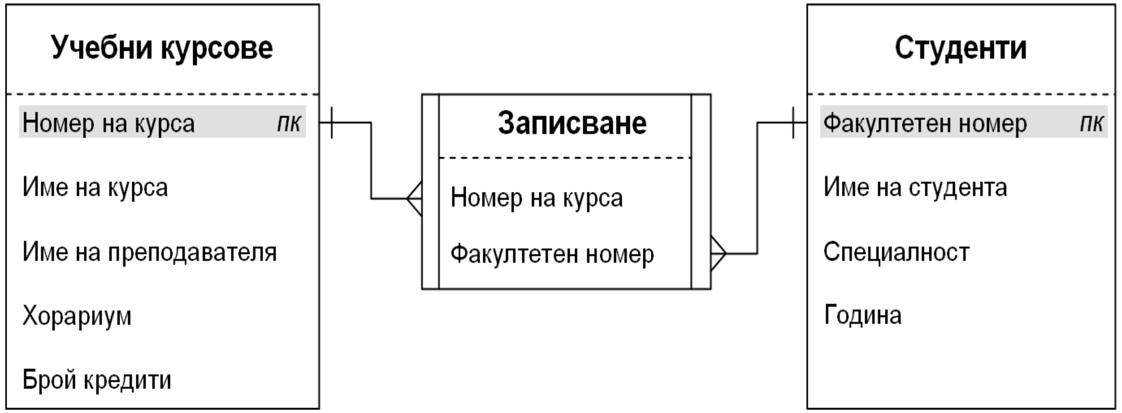


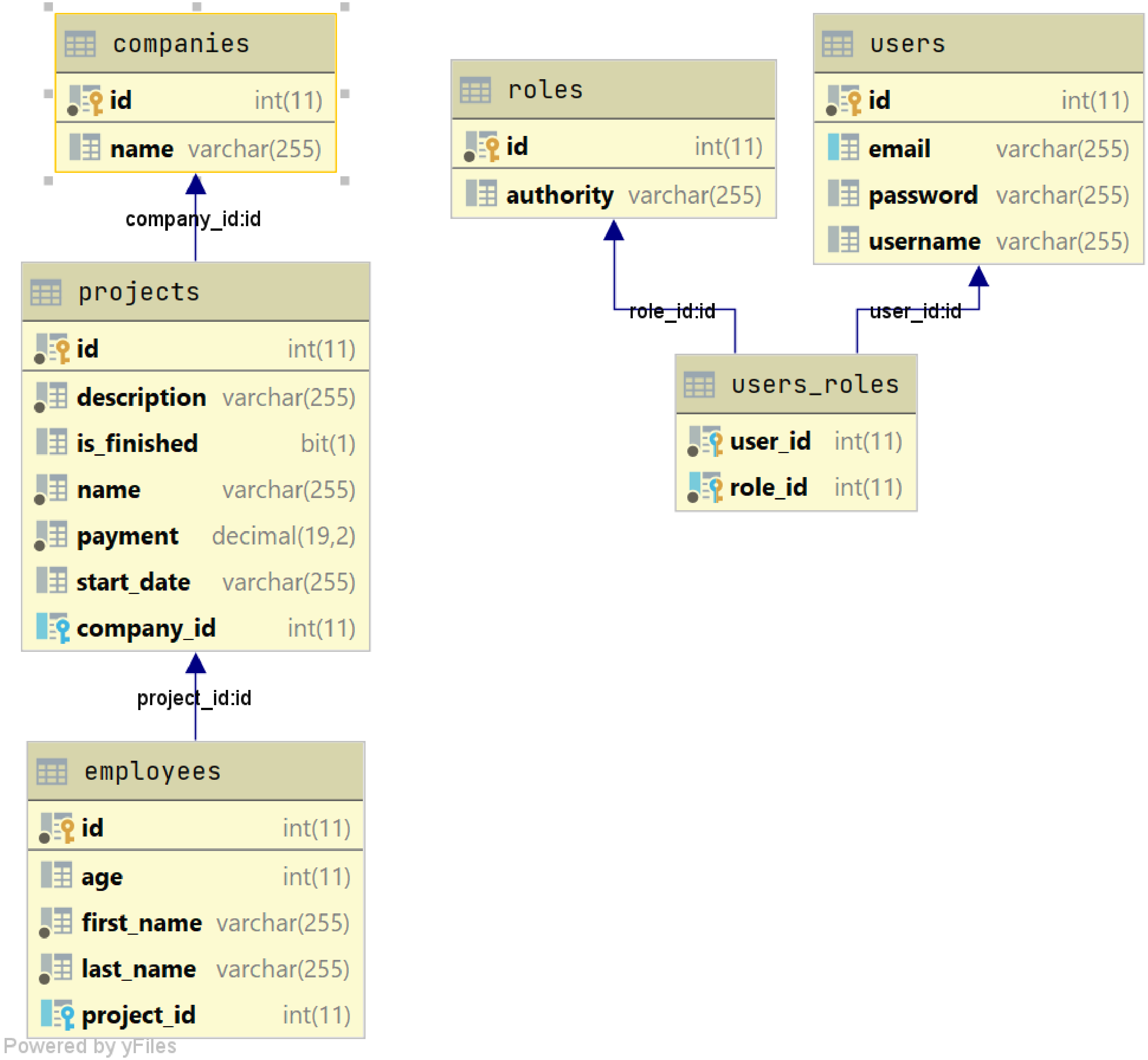
Table Relations. Cardinality

- Relationship is a dependency existing between two tables, when the records from first table can be connected somehow with records from second one.
- Cardinality:
  - One to one (1:1),
  - One to many (1:N),
  - Many to one (N:1)
  - Many to many (M:N)

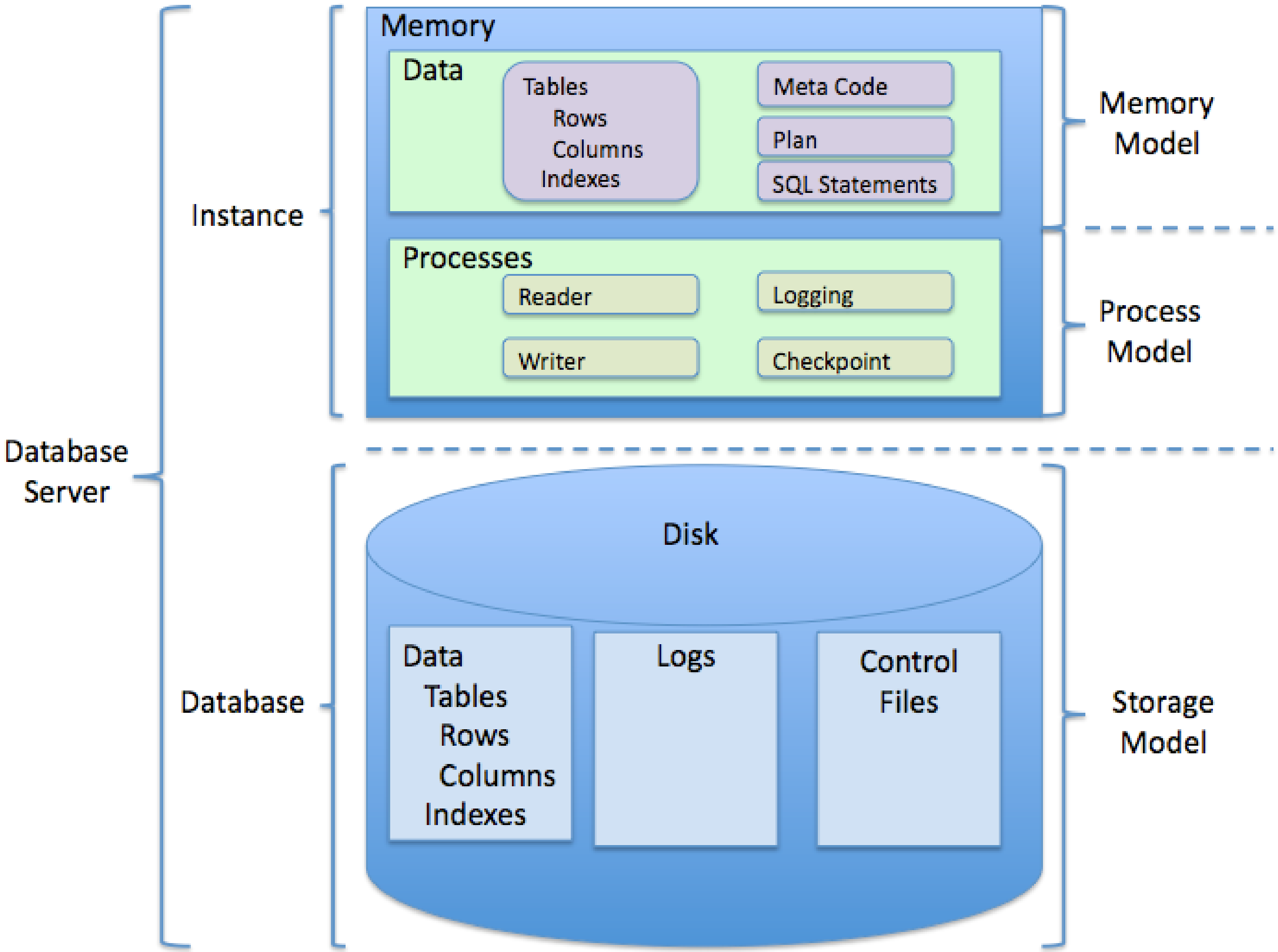




Relational Schema Example ER Model



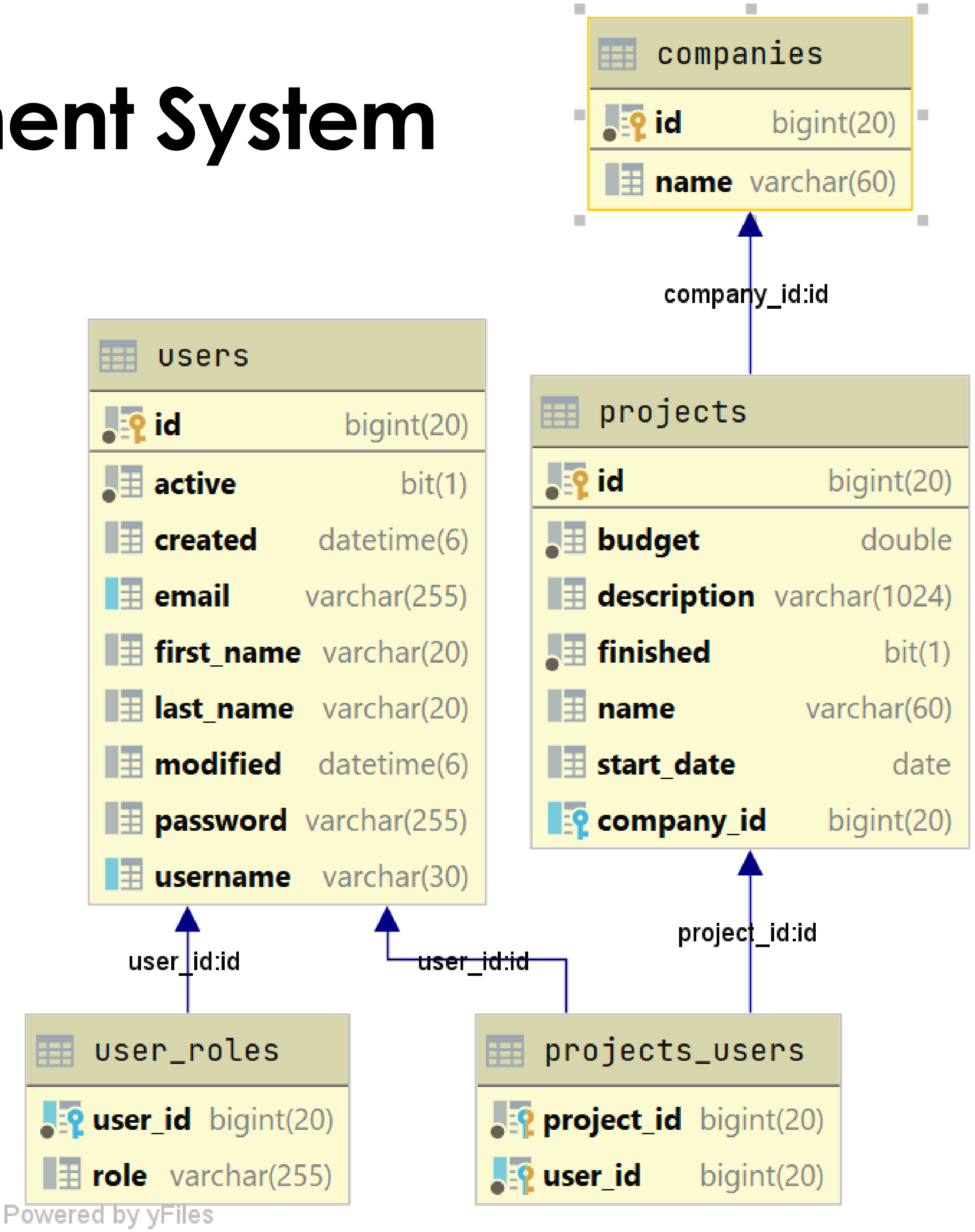
Relational Database Management System (RDBMS)

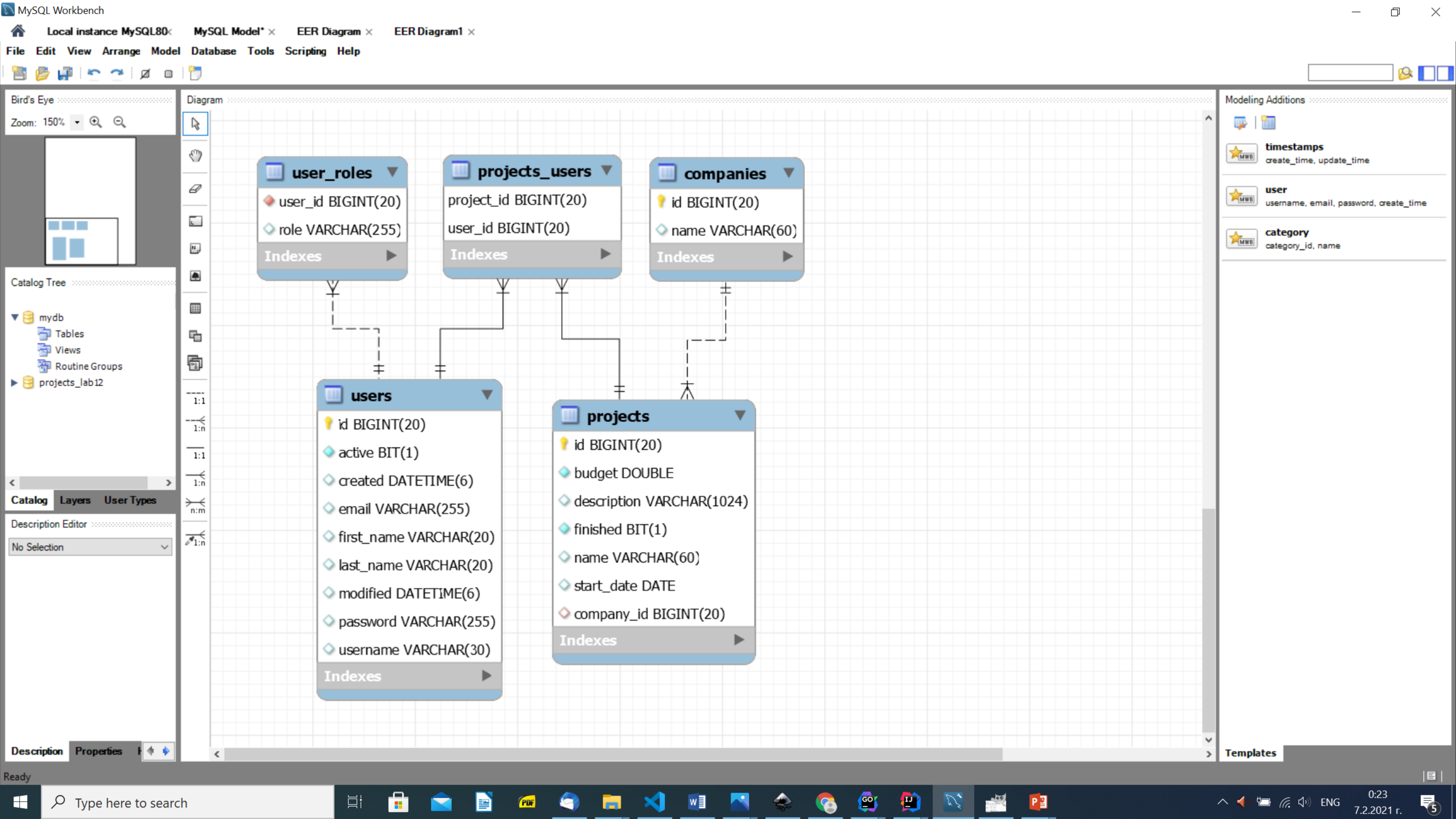


Accessing Relational Databases with Go – database/sql

Simple Project Management System

- Companies can develop multiple projects
- Each project can involve multiple users (employees)
- Users can have multiple roles in the project management system and can work on multiple projects
- Users should have unique username used for logging in the system together with a password




MySQL Workbench
Local instance MySQL80
MySQL Model*
EER Diagram
EER Diagram1
File Edit View Arrange Model Database Tools Scripting Help
Bird's Eye
Zoom: 150%
Catalog Tree
mydb
Tables
Views
Routine Groups
projects_lab12
Catalog Layers User Types
Description Editor
No Selection
Description Properties
Diagram
user_roles
user_id BIGINT(20)
role VARCHAR(255)
Indexes
projects_users
project_id BIGINT(20)
user_id BIGINT(20)
Indexes
companies
id BIGINT(20)
name VARCHAR(60)
Indexes
users
id BIGINT(20)
active BIT(1)
created DATETIME(6)
email VARCHAR(255)
first_name VARCHAR(20)
last_name VARCHAR(20)
modified DATETIME(6)
password VARCHAR(255)
username VARCHAR(30)
Indexes
projects
id BIGINT(20)
budget DOUBLE
description VARCHAR(1024)
finished BIT(1)
name VARCHAR(60)
start_date DATE
company_id BIGINT(20)
Indexes
Modeling Additions
timestamps
create_time, update_time
user
username, email, password, create_time
category
category_id, name
Templates
Ready
Type here to search
ENG
0:23
7.2.2021 г.

SQL Tutorials and Resources

- W3Schools SQL Tutorial – https://www.w3schools.com/sql/sql_select.asp
- MySQL 8.0 reference Manual – https://dev.mysql.com/doc/refman/8.0/en/
- MySQL Tutorial – https://www.mysqltutorial.org/basic-mysql-tutorial.aspx
- W3Resource MySQL Tutorial – https://www.w3resource.com/mysql/mysql-tutorials.php

database/sql: https://golang.org/pkg/database/sql/

- Create a connections pool to MySQL DB:

```
db, err := sql.Open("mysql", "root:root@/golang_projects?parseTime=true")
if err != nil {
	log.Fatal(err)
}
defer db.Close()
```

- Can add more settings to *sql.DB:

```
db.SetConnMaxLifetime(time.Minute * 5) // ensure connections are closed by the
//driver safely before MySQL server, OS, or other middlewares, helps load ballancing
db.SetMaxOpenConns(10) // maximum size of connection pool
db.SetMaxIdleConns(10) // maximum size of idel connections in the pool
db.SetConnMaxIdleTime(time.Minute * 3) // maximum time connection is kept if idle
```

database/sql: Ping the DB:

```go
ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
defer cancel()

status := "up"
if err := db.PingContext(ctx); err != nil {
	status = "down"
}


log.Println(status)
```

database/sql: Simple SELECT Query

```
func FindAllProjects(db *sql.DB) (projects []entities.Project, err error) {
	rows, err := db.Query("SELECT * FROM projects")
	if err != nil {
		return
	}
	defer rows.Close()
	for rows.Next() {
		p := entities.Project{}
		err = rows.Scan(&p.ID, &p.Name, &p.Description, &p.Budget, &p.Finished, &p.StartDate,&p.CompID)
		if err != nil {
			return
		}
		projects = append(projects, p)
	}
	if err = rows.Err(); err != nil {
		return
	}
	return
}
```

database/sql: Better SELECT Query

```
func FindAllProjects(db *sql.DB) (projects []entities.Project, err error) {
	rows, err := db.Query("SELECT * FROM projects")
	if err != nil { return }
	defer rows.Close()
	for rows.Next() {
		p := entities.Project{}
		err = rows.Scan(&p.ID, &p.Name, &p.Description, &p.Budget, &p.Finished, &p.StartDate,&p.CompID)
		if err != nil { return }
		projects = append(projects, p)
	}
	// If the database is being written to ensure to check for Close errors that may be returned from the driver. The
	// query may encounter an auto-commit error and be forced to rollback changes.
	err = rows.Close()
	if err != nil {
		return
	}
	// Rows.Err will report the last error encountered by Rows.Scan.
	if err = rows.Err(); err != nil { return }
	return
}
```

database/sql: Prepared Statement – INSERT

```
stmt, err = db.Prepare(`INSERT INTO projects(name, description , budget, start_date, finished, company_id)
                                        VALUES( ?, ?, ?, ?, ?, ? )`)
if err != nil { log.Fatal(err) }
defer stmt.Close() // Prepared statements take up server resources and should be closed after use.

for i, _ := range projects {
	projects[i].Finished = true
	result, err := stmt.Exec(projects[i].Name, projects[i].Description, projects[i].Budget, projects[i].StartDate,
		projects[i].Finished, projects[i].CompanyId);
	if err != nil { log.Fatal(err) }
	numRows, err := result.RowsAffected()
	if err != nil ||  numRows != 1 { log.Fatal("Error creating new Project", err) }
	insId, err := result.LastInsertId()
	if err != nil { log.Fatal(err) }
	projects[i].Id = insId
}
```

Transactions and Concurrency

- **Transaction** = **Business Event**
- **ACID rules:**
- **Atomicity** – the whole transaction is completed (commit) or no part is completed at all (rollback).
- **Consistency** – transaction should presetve existing integrity constraints
- **Isolation** – two uncompleted transactions can not interact
- **Durability** – successfully completed transactions can not be rolled back

Transaction Isolation Levels

- DEFAULT - use the default isolation level of the underlying datastore
- READ_UNCOMMITTED – dirty reads, non-repeatable reads and phantom reads can occur
- READ_COMMITTED – prevents dirty reads; non-repeatable reads and phantom reads can occur
- REPEATABLE_READ – prevents dirty reads and non-repeatable reads; phantom reads can occur
- SERIALIZABLE – prevents dirty reads, non-repeatable reads and phantom reads

Source: https://docs.spring.io/spring-framework/docs/

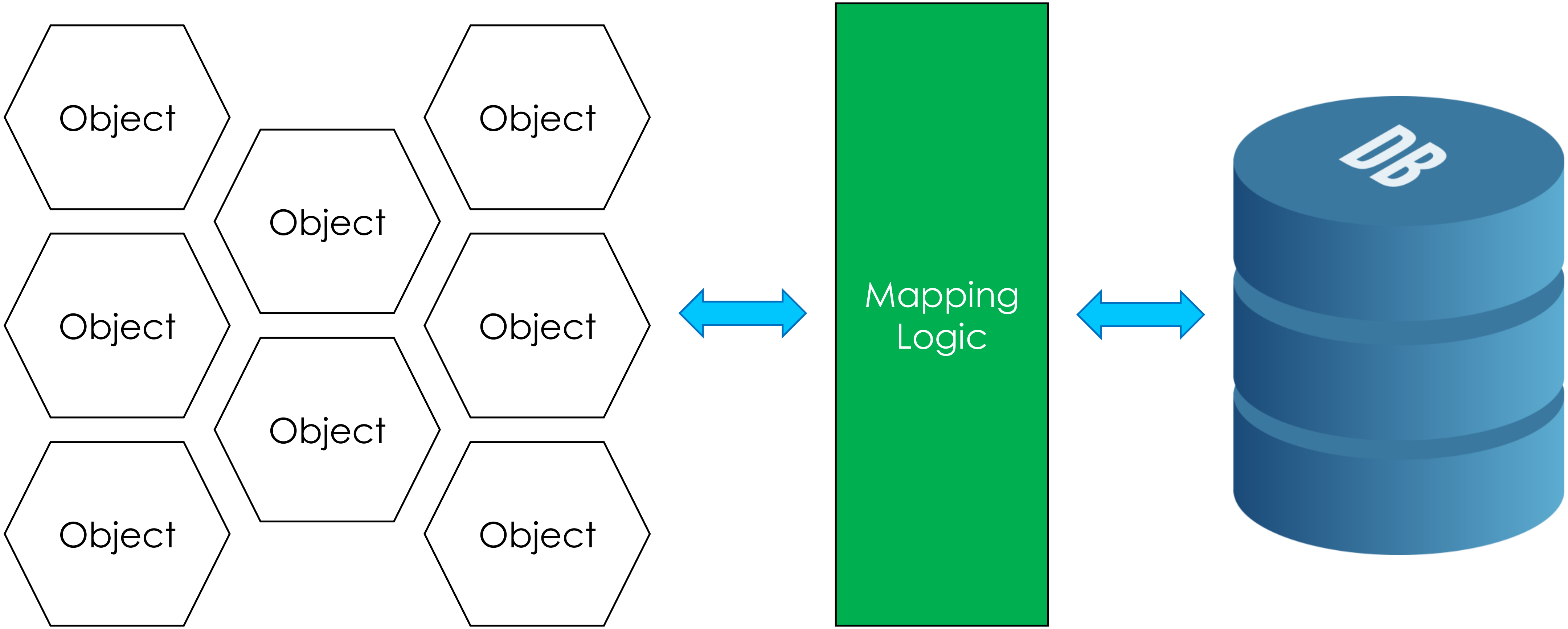
database/sql: Update Project Budgets in Transaction

```
tx, err := db.BeginTx(ctx, &sql.TxOptions{Isolation: sql.LevelSerializable})
if err != nil { log.Fatal(err) }
result, execErr := tx.ExecContext(ctx, `UPDATE projects SET budget = ROUND(budget * 1.2)
                                        WHERE start_date > ?;`, startDate)
if execErr != nil {
        if rollbackErr := tx.Rollback(); rollbackErr != nil {
                log.Fatalf("update failed: %v, unable to rollback: %v\n", execErr, rollbackErr)
        }
        log.Fatalf("update failed: %v", execErr)
}
rows, err := result.RowsAffected()
if err != nil { log.Fatal(err) }
log.Printf("Total budgets updated: %d\n", rows)

if err := tx.Commit(); err != nil {
        log.Fatal(err)
}
```

Go Object to Relation Mapping with GORM

Object To Relational Mapping (ORM)



gorm.io/gorm: https://gorm.io/

```
dsn := "root:root@tcp(127.0.0.1:3306)/gorm_projects?charset=utf8mb4&parseTime=True&loc=Local“
db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
if err != nil { log.Fatal(err) }
user := entities.User{FirstName: "Rob", LastName: "Pike2", Email: "pike2@golang.com", Username: "rob2",
		Password: "rob", Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}}

result := db.Create(&user) // pass pointer of data to Create

if result.Error != nil {
		log.Fatal(result.Error) // returns error
}
fmt.Printf("New user created with ID: %d -> %+v\nRows afffected: %d\n",
		user.ID, // returns inserted data's primary key
		user,
		result.RowsAffected, // returns inserted records count
)
```

gorm.io/gorm: Detiled Configuration

```
db, err := gorm.Open(mysql.New(mysql.Config{
        DSN: "root:root@tcp(127.0.0.1:3306)/golang_projects?charset=utf8&parseTime=True&loc=Local",
        DefaultStringSize: 256, // default size for string fields
        DisableDatetimePrecision: true, // disable datetime precision, which not supported before MySQL 5.6
        DontSupportRenameIndex: true, // drop & create when rename index (not supported before MySQL 5.7)
        DontSupportRenameColumn: true, // `change` when rename column (not supported before MySQL 8)
        SkipInitializeWithVersion: false, // auto configure based on currently MySQL version
}), &gorm.Config{})
```

gorm.io/gorm: Batch User Creation

```
db.AutoMigrate(&entities.User{})        // Automatically create database schema – all the tables
db.AutoMigrate(&entities.Company{})
db.AutoMigrate(&entities.Project{})
users := []entities.User{
    {FirstName: "Linus", LastName: "Torvalds", Email: "linus@linux.com", Username: "linus", Password: "linus",
                Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}},
    {FirstName: "Rob", LastName: "Pike", Email: "pike@golang.com", Username: "rob", Password: "rob",
                Active: true, Model: gorm.Model{CreatedAt: time.Now(), UpdatedAt: time.Now()}},
}
result := db.Create(&users) // pass pointer of data to Create
//db.CreateInBatches(users, 100) // batch size 100
if result.Error != nil { log.Fatal(result.Error) } // returns error
fmt.Printf("New users created with IDs: ")
for _, user := range users { fmt.Printf("%d, ", user.ID) }
```

gorm.io/gorm: ... and Query

```go
result = db.Find(&users)     // Get all users - SELECT * FROM users;

if result.Error != nil {          // if returns error

        log.Fatal(result.Error)
}

fmt.Printf("Number of users: %d\n", result.RowsAffected) // returns found records count, equals `len(users)`

utils.PrintUsers(users)
```

gorm.io/gorm: Preloading (Prefetching) Associations

```
result = db.Preload(clause.Associations).Find(&companies) // SELECT * FROM companies fetching projects
if result.Error != nil {
		log.Fatal(result.Error) // returns error
}
fmt.Printf("Number of companies: %d\n", result.RowsAffected)// returns found records count, equals `len(users)`
utils.PrintCompanies(companies)
```

```
result = db.Preload(clause.Associations).Find(&projects)      // SELECT * FROM users with associations
if result.Error != nil { log.Fatal(result.Error) } // returns error
err = db.Model(&(projects[0])).Association("Users").Find(&users) // Association mode
if err != nil { log.Fatal(result.Error) } // returns error
fmt.Printf("Users in Project '%s': %v\n", projects[0].Name, users)
```

```
db.Session(&gorm.Session{FullSaveAssociations: true}).Updates(&user) // Saving all associations
```

Common Pitfalls when Using RDBs with Go - I

- Deferring rows.Close() inside a loop → memory and connections
- Opening many dbobjects (sql.Db) → many TCP connections in TIME_WAIT
- Not doing rows.Close()when done → Run rows.Close() as soon as possible, you can run it later again (no problem). Chain db.QueryRow() & .Scan()
- Unnecessaey use of prepared statements → if concurrency is high, consider whether prepared statements are necessary → re-prepared on busy connections → should be used only if executed many times
- Too much strconv or casts → let conversions to .Scan()
- Custom error-handling and retry → database/sql should handle connection pooling, reconnecting, and retrys

Source: Common Pitfalls When Using database/sql in Go:
https://orangematter.solarwinds.com/2017/03/23/common-pitfalls-when-using-database-sql-in-go/

Common Pitfalls when Using RDBs with Go - II

- Don't forgetting to check errors after **rows.Next()** → rows.Next() can exit with error
- Using **db.Query()** for **non-SELECT** queries → iterating over a result set when there is no one, leaking connections.
- Don't assuming subsequent statements are executed on same connection → Two statements can run on different connections → to solve the problem execute all statements on a single transaction (**sql.Tx**).
- Don't mix **db** access while using a **tx** → **sql.Tx** is bound to transaction, **db** not
- Unexpected **NULL** → to scan for NULL use one of the **NullXXX** types provided by the database/sql package – e.g. NullString

Source: *Common Pitfalls When Using database/sql in Go:* *https://orangematter.solarwinds.com/2017/03/23/common-pitfalls-when-using-database-sql-in-go/*

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD