# Software Engineering for Business Applications
# Lecture Notes

Efe Kamasoglu

February 13, 2023

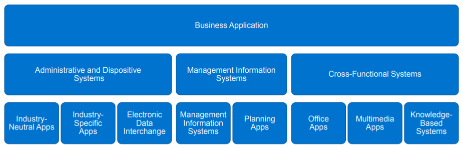
# 1 IT Support for Business Applications

## 1.1 Classification of Business Applications

- **Definition "Business Application":**
  - <u>in narrower sense:</u> totality of all programs, i.e. **application software**, and associated **data** for a concrete business use case
  - <u>in broader sense:</u> additionally **hardware**, **system software** and necessary **communication** facilities required for the use of application software

- **Two roles of Business Applications:**
  - **supporting**, **improving** or **automating** existing operational processes in bookeeping, accounting, etc. (size, speed, correctness...)
  - **enabling** new products and services (e.g. online shopping and banking)

- **Classification of Business Applications by Business Purpose:**



  <u>*Examples of:*</u>

  - **administrative systems:** financial accounting, payroll accounting, administration of stocks
  - **disposition systems:** calculation and cost accounting, material procurement, field service control
  - **management information systems (MIS):** use of internal company data, use of external data, combination of multiple data sources in a flexible form
  - **planning systems:** planning of individual functional areas, integrated planning of several functional areas, corporate planning

- **Cross-Cutting Applications:**
  - independent of company hierarchy and fuctional domains
  - used either directly via user interface or programmatically via administration and disposition systems
  - *Examples:* office suites, groupware, workflow management systems

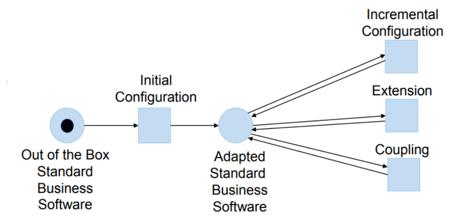
- **Enterprise Resource Planning (ERP): ERP system** is an integrated business application (suite, collection of programs), which supports all essential functions of administration, disposition and management with a common interface and a shared and integrated data management.
  - consists of platform and function-oriented application components that exchange info and events
  - is realized as (customizable) standard software
  - *Examples:* external accounting, controlling, procurement
  - Today's ERP systems support an **extended value chain**[1].

## 1.2 Standard and Custom Software

- **Standard Software vs. Custom Software:**
  - **Standard software** *(e.g. SAP)*
    - developed for specific **market**
    - distributed by a software house
    - can be used by **several companies**
    - implements "standard business processes" at its core
    - maintained by **manufacturer**, adapted to changes
    - must or can be **customized** to company (e.g. authorizations and roles, currencies)
  - **Custom software**
    - specifically developed for **one company**
    - tailored to specific business processes/requirements
    - result of a project for a known client
    - **individually** maintained and adapted to changes

[1] **Value chain** is a business model that describes the full range of activities needed to create a product or service.

- **Adaptation Techniques for Standard Business Software:**
  - Adaptation of operational standard software can be divided into **Configuration**, **Extension** and **Coupling** (= **Customizing**).


Out of the Box Standard Business Software
Initial Configuration
Adapted Standard Business Software
Incremental Configuration
Extension
Coupling

  - **Configuration** describes functionalities and techniques
    - that are obligatory on first deployment
    - that allow to define predefined settings
    - that lead to an individual variation of standard software
  - **Extension** describes functionalities and techniques
    - that are optional for productive use
    - that allow to map requirements not foreseen by manufacturer
    - implemented by manufacturer to expand the range of services
  - **Coupling** refers to functionalities and techniques
    - to connect external systems of other manufacturers
    - to connect external systems of the same type
    - that are predefined in the form of data file formats, APIs, or communication protocols
  - *Example:* mapping the structure of a company to SAP applications via organizational units (can be assigned to single or multiple apps)
- **Configuration: Challenges**
  - A **standard software** must
    - provide all relevant configuration options
    - support a wide range of different corporate structures and processes
    - check dependencies between these many variants
    - provide appropriate documentation about the effects of individual configurations
  - **Consequences:**
    - need for experts who are familiar with configuration options of each release and componant
    - scarcity of such experts

        * expensive training
        * expensive consultancy services

- **Examples for Extensions:**
    - automation of **multi-step business workflows**
    - integration of company-specific calculations/rules/checks
    - connecting customers
- **Coupling Options:**
    - different coupling options depending on the scenario
    - programming language used for coupling
    - available mechanisms to couple
- **Multi Tenancy:** Software multitenancy is a software architecture in which a single instance of software runs on a server and serves multiple tenants (e.g. companies).
    - several companies can be represented in one system
    - distinction between tenant-dependent and -independent data
    - supporting tenant-dependent authorization (e.g. A may only perform transactions in client 002)
    - individual adaptations of tenants (e.g. currency, couplings)
- **Multilingualism:**
    - **Multilingualism of a business information system** makes it possible to
        * store and display texts in different languages in the system
        * assigning graphics and symbols specific to different languages
    - Multilingualism requires
        * that one system can process all relevant character sets at once
        * storage and recognition of words, numbers etc.
        * that a system can assign users to languages or user can choose their own
        * that texts (graphics, symbols) can be assigned to a language
- **Localization (l10n):** Adaptation of a software product to meet the language, culture, and other requirements of each locale (e.g. adaptation of graphics, currencies, date and time)
- **Internationalization (i18n):** Process of preparing a software-based product for localization (to support global markets)

## 1.3 Characteristics of Business Applications

- **Multiple Stakeholders and changing requirements:**
  - **Requirements Elicitation and Requirements Management**
    - many stakeholders, different views and concerns
    - Waterfall: upfront requirements document and/or technical specification => Req. Documentation
    - Issue: changing requirements once IT support is implemented
    - Agile: incremental and iterative => Agile Req. Engineering
    - typically, very large number of requirements
    - need for formalization and early consistency checking => Conceptual Modeling
    - need for cost and time prediction => Software Estimation
  - **Programming Challenges**
    - design, implement and test changes in an existing complex system => Change Mgmt.
    - deliver incremental changes without invalidating existing data => Release Mgmt.
    - parallel development at manufacturer and at customer site => Version Mgmt.
    - automated and quality-controlled assembly of application software => Build Mgmt.
- **Persistent Data and Concurrent Data Modification:**
  - **Data consistency** is a must:
    - many users perform **transactions** simultaneously on central databases
    - data must not be lost even in case of system failures
  - **Programming challenges:**
    - database is managed by an independent application, on a different server / hardware
    - object orientation is not supported by common data bases
    - database concepts must be transferred to the application logic (transactions, rights, primary keys)
- **Distributed Actors and Data Repositories:**
  - **Many users access central data concurrently:**
    - users need data in different locations at different times
    - Client-Server architecture => Layered Architectures
    - web clients => REST protocol
  - **Programming challenges:**

    * software components must be able to found in network => Naming services
    * communication always via a network => Serialization[2] & failed execution
    * authentication and authorization => Security
    * concurrent accesses => Transactions

- **Integeration of Data and Application from (Semi-)Autonomous Sources:**
  - **Separation of applications and data repositories:**
    * multiple apps work on independent or shared data resources
    * multiple apps communicate with each other => RPC, Message Passing
    * business processes involve multiple apps => Workflow Mgmt. Systems
    * application landscapes with lots of interacting applications => Enterprise Architecture Mgmt.
  - **Programming challenges:**
    * integration of multiple languages and databases
    * loose coupling through interfaces to avoid code change propagationi
    * error recovery to avoid runtime failure propagation
- **Scalability:**
  - **Growing number of users and data volume**
    * business apps are used by thousands of employees world-wide around the clock
    * customers and business partners interact directly with business apps and expect real-time sub-second response times
    * volatile load (e.g. online shop in christmas season vs. summer season)
  - **Programming challenges:**
    * delayed execution of resource-intesive operations => Batch processing[3]
    * dynamically increasing/decreasing number of users => Instance pools
    * single server cannot handle the load => Load balancing, Caching

[2]**Serialization** is the process of translating a data structure into a format that can be stored or transmitted and reconstructed later.

[3]**Batch processing** is when a computer processes a number of tasks that it has collected in a group. It is designed to be a completely automated process, without human intervention.

# 2 Requirements Engineering

- **Software requirements** express the needs and constraints placed on a software product.
- **Requirements engineering** is concerned with **elicitation**, **analysis**, **specification** and **validation** of software requirements as well as the management of requirements.
- **Requirements Management** deals with the administration and maintenance of requirements documents, in particular:
    - change requirements (change management)
    - trace and link requirements (requirements tracing)
    - verify requirements

## 2.1 Traditional Requirements Engineering

- **Objectives of Requirements Management:**
    - **Efficient** preparation of **high quality** requirements and system specifications,
        * coordinated with all stakeholders (different objectives and interests)
        * coordinated with all specifications and constraints
        * evaluated according to profitability and feasibility
    - **Specification documents** are basis for:
        * contract negotiation and contractual agreements
        * coordination between the stakeholders (customers, developers)
        * design, realization, integration
        * software acceptance (test specification)
        * future developments, projects
- **Requirement Classification:** Distinction between functional and non-functional requirements and constraints:
    - **Functional requirements** describe interactions between the system and its environment independent of their realization.
    - **Non-functional requirements** describe general properties of the system.
    - **Restrictions (Constraints)** determine the solution space for the realization.
- **Stakeholder Management:** It includes

  - processes required to identify people that could impact or be impacted by the project
  - to analyze stakeholder expectations and their impact on the project
  - to develop appropriate management strategies for effectively engaging stakeholders in project decisions and execution

- **Requirement Specification:**
  - technical result document of requirement identification phase
  - **contains** stakeholder identification, functional and non-functional requirements, constraints, evaluation plan and metrics
  - list of all deliverables and services to be fulfilled by contractor within contract as defined by customer
  - **what** is to expect from the solution (product)
  - formulation of requirements should be as general as possible and as restrictive as necessary
  - enables the contractor to develop optimal solutions

- **Requirements Validation: Validation**, **Consistency check** (no conflicts), **Completeness check**, **Reality check**, **Verifiability**

- **Functional Specification:**
  - defines the purpose of the system
  - solution proposal created by contractor based on the requirement specification provided by client
  - **contains** target determination, product usage, environment (e.g. hardware), functions, UI, global test cases
  - system description or solution specification, which describes **how** the solutions is to be realized (concrete solution approaches)
  - the **what** from **requirement specification** is detailed

## 2.2 Agile Requirements Engineering

- **Requirements Engineering and Agile Software Development:**
  - **Agile software development** focuses more on **continuous collabration** (workshops, interviews etc.) with stakeholders instead of relying on **specification documents** (*example: SCRUM*)
  - **Traditional requirements engineering**
    * focuses on customer collabration mainly at an early phase of the project (longer change cycles)
    * emphasizes a heavy-weight process with extensive, **static specification documents**

    - **Agile requirements engineering**
        * fosters communication with the customer during the whole development process to **continuously update requirements**
        * focuses less on extensive documentation, but specification documents **might be necessary** because of legal or contracting reasons etc.
        * includes activities and artifacts that are similar to classical requirements engineering activities

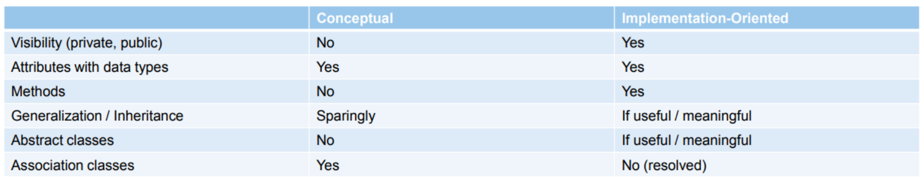
- **Typical Requirement Artifacts in Agile Software Development:**
    - **user story**, **story card**, **use case**, **scenario**, **UML diagram**, **prototype**

- **User Stories:**
    - explanation of a software feature written from the perspective of the end user
    - most frequently used artifact in **agile software development**
    - mnemonic for writing good user stories: INVEST[4]

- **Typical Requirements Engineering Challenges:**
    - different interest groups can raise **conflicting requirements**
    - the people who **pay** for the system are rarely the ones who **use** it
    - the organization and the technical environment may **change** after the system rollout
    - requirements that change during implementation (Change Requests) can lead to additional costs -> project duration/milestones can be affected significantly

## 3 Conceptual Modeling with UML

- **Conceptual Class Diagram vs. Implementation-Oriented Diagram:**

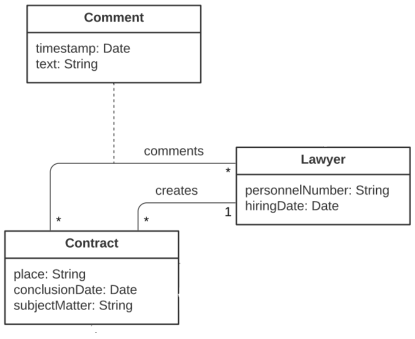


| | Conceptual | Implementation-Oriented |
|---|---|---|
| Visibility (private, public) | No | Yes |
| Attributes with data types | Yes | Yes |
| Methods | No | Yes |
| Generalization / Inheritance | Sparingly | If useful / meaningful |
| Abstract classes | No | If useful / meaningful |
| Association classes | Yes | No (resolved) |

- **Associations between Classes:**

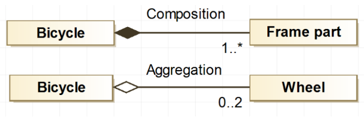
[4]independent, negotiable, valuable, estimable, small, testable

- **Multiplicity:**



*A Lawyer can* <u>*create*</u> ***multiple*** *Contracts, whereas every Contract has a* ***single*** *Lawyer.* -> <u>creates</u> (action) on the side of Lawyer (actor)

- **Aggregation:** implies a relationship where the child can exist independently of the parent (part of the parent)
- **Composition:** implies a relationship where the child <u>cannot exist</u> independent of the parent
- <u>*Example:*</u>



# 4 Software Estimation

## 4.1 Fundamentals of Estimation Methods

- **Software Estimation:**
  - In principle, software estimation relies on **forecasting effort**, from which cost and duration are derived.
  - Regardless of the project and software methodology applied, every initiative requires the definition of a **budget** and a specific **time frame** necessary to deliver a final outcome.
  - These two are obtained during the **early stages** of the project lifecycle through the process of estimation.

  - **Estimation** aims to provide an **approximation** of the amount of recources required to complete project activities and produce a product or service in accordance to specified **functional** and **non-functional characteristics**.
  - **Software estimation conducted in early phases of the project lifecycle:**
    - necessary for contract negotiations
    - predict expected efforts (and derived costs) for a software project before implementation
    - best possible estimation given the available info
  - **Agile estimation:**
    - estimation of individual requirements during project
    - incremental allocation of developers in the most efficient manner
    - cost estimates are made several times during development project with varying degrees of detail

- **Software Estimation: Cone of Uncertainty**
  - At the beginning of the project, not much is known about the product/project -> estimates underly high uncertainty
  - As the project progresses, more information is available -> decrease in uncertainty

- **Software Estimation: Costs**
  - **Cost categories:**
    - **Development costs:** costs to produce a software product
    - **Personnel costs:** major share of development costs for personnel
      - usually low costs for office materials etc. in relation to the personnel costs
      - proportionate allocation of CASE[5] environment costs (including hardware and software) for product development

## 4.2 Traditional Software Estimation

- **Sneed's Devil's Square:**
  - Quantity
  - Quality
  - Development duration
  - Cost

[5]Computer power-assisted software package Engineering

are mutually dependent.

- **Quantity:**
  - size of program code (example basis of assesment: LOC[6])
  - functional and data scope
  - possible additional weighting with complexity
- **Quality:**
  - higher quality requirements => greater effort
  - no **THE quality**, but different quality characteristics
- **Productivity:**
  - influenced by many different factors
  - number of communication links grows **quadratically** with the team size
- **Development time:**
  - need more members to shorten development time
  - more members => more communication effort
  - higher communication => decrease in productivity
- **Methods for Effort Estimation:**
  - **Estimation Strategies:**
    - **Top-Down:** estimation of the total project effort using mathematical algorithms based on the functional requirements
    - **Bottom-Up:** expenses for each expense item are calculated separately and added to calculate the total project effort
  - **Comparison methods:**
    - estimation based on effort analysis of already accomplished similar developments
  - **Algorithmic methods:**
    - effort calculated with algorithmic methods
    - based on statistical models or actual expenditure of already completed projects
  - **Key figure methods:**
    - total cost of the software product determined by estimating the cost of individual units or project phases
  - None of the listed basic methods alone is sufficient.

[6]Lines of Code

- Depending on the point in time and knowledge of effort-relative data, one or the other method should be used.

- **Concrete Procedures for Effort Estimation:**
  - **Goal:** Combine advantages of several effort estimation methods to deliver accurate results. (*example: Function Point Method*)

- **Function Point Method:** It is a combined relation and weighting method.
  1. **Categorization** of each product requirement (input, query, output, database, reference data)i
     - **Input:** by the user
     - **Output:** displaying query results, calculated data
     - **Query:** performed on the **database** of the system, read and write
     - **Reference data:** used to validate input, generate the output or construct the query (read-only)
  2. **Classification** of each product requirement
     - **simple**
     - **medium**
     - **complex**
  3. **Entry** into calculation form
  4. **Evaluation** of influencing factors
     - the **influence factors** refer to the application as a whole and not to individual functions or function points
  5. **Calculation** of the evaluated Function Points (FP)
  6. **Determination** of the personnel expenses based on a FP-PM curve or table
     - significant productivity decreases in large projects (FP-PM[7]: increase in FP => increase in PM) -> non-linear growth
  7. **Update** of empirical data as an estimation basis for follow-up project
     - After completion of a development estimated with the Function Point Method, the new value pair (FP, Actual PM) is used to update the existing curve.

- **Function Point Method: Requirements:**
  - evaluation once the project requirements are known
  - evaluation by employees with sufficient knowledge of requirements
  - product considered from the perspective of client

[7]**FP:** function points, **PM:** person month (= MM: man month)

  - company-specific training, guidelines are needed to minimize the effect of subjective individual estimates during the classification and evaluation of influencing factors
  - actual efforts must be measured for post-calculation

- **Function Point Method: Advantages**
  - product requirements, not LOC as starting point
  - adaptability to different application areas (change of categories)
  - adaptability to new techniques (change of influencing factors, influence evaluation)
  - adaptability to company-specific environments (if, ie and class factors per class)
  - refinement of the estimate according to the development process
  - first estimate is possible at a very early stage (planning phase)
  - good estimation accuracy

- **Function Point Method: Disadvantages**
  - only total effort can be estimated -> conversion to individual phases must be made using a percentage-based method
  - personnel-intensive, not easy to automate
  - too strongly function-oriented
  - influence factors do not clearly separate project and product characteristics

## 4.3 Agile Estimation Methods

- **Estimation in the SCRUM Framework:**
  1. Estimation of **Story Points**[8] for each item in the **Product Backlog**
     - an **ordered list** of everything that is known to be needed in the product
     - **Product Backlog Refinement:** act of adding detail, **estimates**, and order to items in the **Product Backlog**
     - **User Story** is the unit with which software features are **estimated** and developed.
  2. **Time Estimation** (in days) for each item in the **Sprint Backlog**

- **Estimation with the help of Planning Poker:**

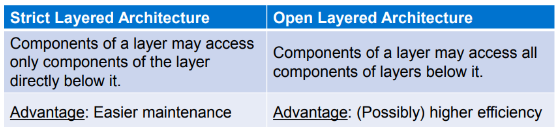
[8]**Story points** are units of measure for expressing an estimate of the overall effort required to fully implement a product backlog item or any other piece of work.

  - reason to use **planning poker** is to **avoid the influence of the other participants** (group thinking)
  - estimates are **story points** from different members (developers)
  - estimates are revealed simultaneously to assure the indepence between group members
  - estimates are used during **release** and **sprint planning** meetings to create release and sprint plans

# 5 Technical Foundation of Business Information System

## 5.1 Architecture of Business Information Systems

- **Architecture Patterns:**
  - An **architecture pattern** describes a particular recurring design problem that arises in specific **design contexts** and presents a well-proven **generic scheme** for its solution.
  - The solution scheme is specified by describing its constituent **components**, their **responsibilities** and **relationships**, and the ways in which they **collaborate**.
  - *Examples:* Layered Architecture, Tiered Architecture
- **Layered Architectures:**
  - layers define a **logical partitioning** of software components to reduce overall system complexity
  - **Two types of layered architectures:**



| Strict Layered Architecture | Open Layered Architecture |
|---|---|
| Components of a layer may access only components of the layer directly below it. | Components of a layer may access all components of layers below it. |
| Advantage: Easier maintenance | Advantage: (Possibly) higher efficiency |

  - a component of a layer may not access a layer above it
  - **high cohesion** between conponents within a layer, **low coupling** between different layers
  - a layer can have an explicit **interface** that distinguishes public and private components of a layer
- **Tiers in Architectures:**
  - Tiers define a **physical partitioning** of logical software components into different process spaces of a distributed system.

  - a tier identifies an independent **process space** within a distributed application
  - these process spaces can be executed on a single/different computers in a **network**
  - each tier has a particular **responsibility** in the system and addresses a coherent set of **concerns** and **requirements** that may change over time
  - tiers are a concept relevant for structuring software components during **execution**
  - **n-tier architecture** defines how many tiers there are within a distributed application
  - Typical concerns and requirements in an **information system**:
    * **Presentation Tier:** how to interact with the users?
    * **Business Logic Tier:** how to capture and structure business logic and ensure the integrity of data?
    * **Resource Tier:** how to persistently store and efficiently manage data?

- **Client-Server Architecture:**
  - two components: client and server
  - client requests service via network protocol from server, server sends response
  - one server can serve multiple clients
  - a server can be a client of another server

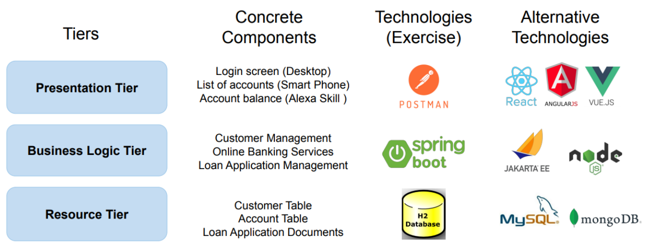
- **Two-Tier Architecture:**
  - client tier and server tier
  - assignment of tasks:
    * Presentation -> Client tier
    * Business logic -> Client tier or server tier, both
    * Resource management -> Server tier
  - Advantages:
    * easy to implement
    * high performance

- **Three-Tier Architecture:**
  - Assignment of tasks:
    * Presentation -> Client tier
    * Business logic -> Middle tier

    * Resource management -> Server tier
  - Standard model for simple **web applications**:
    * Client tier -> HTML/CSS/JavaScript (loaded dynamically by a **browser**)
    * Middle tier -> **web application server**
    * Server tier -> off-the-shelf **database management system**

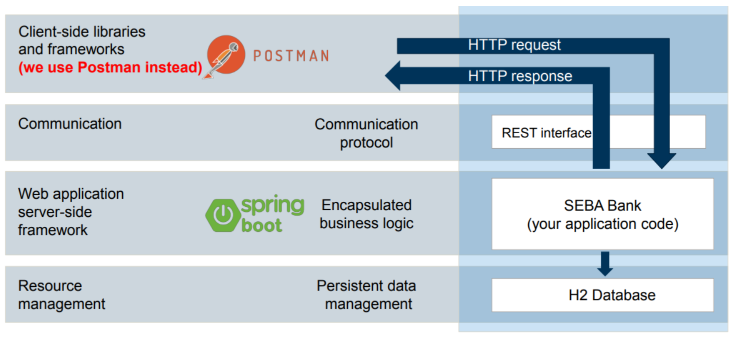
- **Four- and N-Tier Architectures:**
  - extension of three-tier architectures: **business logic** is distributed to several layers
  - Motivation:
    * further complexity reduction of individual tiers
    * improved protection and isolation
    * use of multiple and concurrent application processes

- **Technologies and Tiers Used in the Exercises:**



| Tiers | Concrete Components | Technologies (Exercise) | Alternative Technologies |
|---|---|---|---|
| Presentation Tier | Login screen (Desktop)<br>List of accounts (Smart Phone)<br>Account balance (Alexa Skill ) | POSTMAN | React, ANGULARJS, VUE.JS |
| Business Logic Tier | Customer Management<br>Online Banking Services<br>Loan Application Management | spring boot | JAKARTA EE, node JS |
| Resource Tier | Customer Table<br>Account Table<br>Loan Application Documents | H2 Database | MySQL, mongoDB |

- **Target Architecture (SEBA Bank, SEBA Mobility Services):**


Client-side libraries and frameworks
(we use Postman instead)
POSTMAN
HTTP request
HTTP response
Communication
Communication protocol
REST interface
Web application server-side framework
spring boot
Encapsulated business logic
SEBA Bank (your application code)
Resource management
Persistent data management
H2 Database

- **Web Server:**
  - processes incoming requests over various network protocols (HTTP)

  - provides its clients with static or dynamically generated content (HTML, CSS, files, images)
  - **Additional tasks:**
    * resource management (sockets, static files)
    * access control
    * cookie[9] management
    * script execution
    * caching

- **Application Server:**
  - web servers that execute application code to respond to HTTP requests with HTTP responses
  - enterprise software platforms offer their own application servers: Jakarta EE, SAP Web Application Server
  - **Additional tasks:**
    * authentication
    * authorization
    * session management
    * encapsulation of databases
    * transaction processing
    * asynchronous communication

- **Database Server:**
  - **Database server (software)**
    * software to implement data management, query optimization, concurrency control, access control
    * can belong to different categories: Relational DB etc.
    * provides administration tools
  - **Database server (hardware)**
    * database servers usually run on a separate high-performance machines (disk IO, main memory, number of processes and threads)
    * taking in the role of the **server**
  - **Used in the exercises:**
    * H2 database (relational)
    * no separate database server / tier (embedded, in-memory)
    * not suitable for production

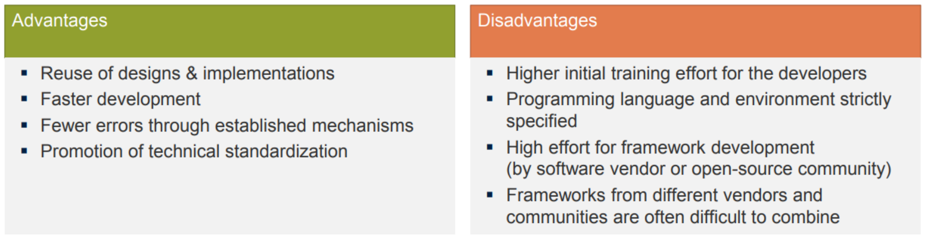
[9] A **cookie** is a small piece of information that a website stores on your computer, and uses it at the time of your iteration on that website.

- **Data Exchange Formats: XML and JSON:**
  - A **Web API** consists of a defined set of HTTP request messages.
  - for each request -> Web API specifies the structure of response messages
  - messages expressed in JSON or XML => **human-readable** data interchange

## 5.2 Libraries and Frameworks

- **Library: reusable software component** that consists of several classes
  - **functions** of the library are called by the code of the users via its **Application Programming Interface (API)**
  - **API:** the order in which the provided functions are called is determined by the user
  - *Examples:* Log4J (logging), JDBC (database access), dom4j (XML parsing)
- **Framework: partially finished software system** (completed code), which consists of a variety of coordinated software components from which an **adapted software system** can be created with relatively little effort
  - **Frameworks offer**
    * a basic architecture for a software system
    * a high degree of reusability
    * a given set of functions that user can / have to extend
    * whereby the general processing logic
  - **Frameworks are tailored for specific purposes**
    * GUIs: Java Swing
    * Web development: Spring Web
    * Unit testing: JUnit
  - *Framework Examples:* JUnit, Spring, Jakarta EE
- **Inversion of Control (IoC): IoC** distinguishes a **framework** from a **library**
  - Since **developer** is in charge of application flow, he decides when to call the **library**.
  - However, when developer uses a framework, **framework** decides when to call the **library**.
  - This shift in control of calling the library from the **application code** to the **framework** is an inversion of control.

- **Advantages and Disadvantages of Frameworks:**



| Advantages | Disadvantages |
|---|---|
| ▪ Reuse of designs & implementations<br>▪ Faster development<br>▪ Fewer errors through established mechanisms<br>▪ Promotion of technical standardization | ▪ Higher initial training effort for the developers<br>▪ Programming language and environment strictly specified<br>▪ High effort for framework development (by software vendor or open-source community)<br>▪ Frameworks from different vendors and communities are often difficult to combine |

- **Jakarta EE (Framework):** a set of specifications for different purposes -> an implementation is needed to use them
- **Spring (Framework):** a framework (configuration model) for building web applications
  - **Features:**
    - IoC container with **dependency injection**
    - data access
    - testing
- **Spring Boot:** a project within the **Spring Framework** that provides a simplified way to configure **based on conventions** and run Spring applications
  - **Motivation:** minimize amount of manual configuration (convention over configuration)
  - **Features:**
    - creation of stand-alone Spring apps
    - embedded web servers
    - use of annotations

## 5.3 Java Annotations

- **Motivation: Aspects and Cross-Cutting Concerns:**
  - Software components, libraries and frameworks define **reusable code**
  - functionality separated from user code through API calls
  - user programs remain **unchanged**
  - desirable to extract **repetitive code elements** that address a certain **aspect** of overall system functionality from user programs
  - **Examples** of such aspects which address **cross-cutting concerns**[10] of whole app:

[10]**Cross-cutting concern** relies on or affects many other aspects within that program.

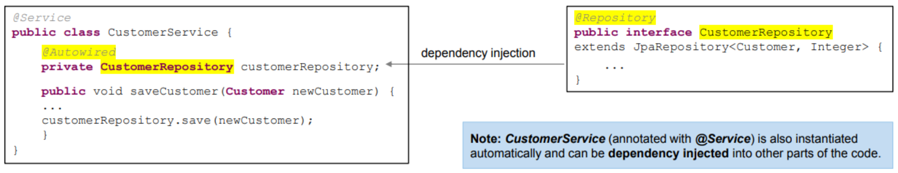
    * component configuration and binding
    * monitoring and logging
    * access control
    * data conversion for data exchange
    * exception and transaction management
  - **combining these aspects freely** which makes it impossible to isolate them from the user code

- **Annotation:** a tag that represents **metadata** i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

- **Dependency Injection (DI):** a design pattern in which an object (client) receives other objects (services or dependencies) that it depends on.

  - code that passes the service to the client is called **injector** -> injector tells client what service to use (rather than allowing client to choose)
  - intent behind DI is to achieve **separation** of concerns of **construction** and **use** of objects

- **Dependency Injection in Spring (Boot):** The **injector code** is part of the framework and triggered by **annotations**.



```
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public void saveCustomer(Customer newCustomer) {
    ...
    customerRepository.save(newCustomer);
    }
}
```

```
@Repository
public interface CustomerRepository
extends JpaRepository<Customer, Integer> {
    ...
}
```

**Note:** ***CustomerService*** (annotated with ***@Service***) is also instantiated automatically and can be **dependency injected** into other parts of the code.

***CustomerService*** *is the* ***client****, whereas* ***CustomerRepository*** *is the* ***service****.* ***@Autowired*** *is the* ***injector****.*

- **Use of Annotations in Jakarta EE:**

  - Application servers or persistence frameworks like **Hibernate** provide aspects that implement the specified functionality.
  - *Examples:* ***@Entity*** => for classes to be persisted into database

- **Use of Annotations in Spring (Boot):**

  - ***@Component***: generic stereotype for any Spring-managed components

  Special cases of ***@Component***:

  - ***@Controller***: for classes at presentation layer

- *@Service*: for classes at service layer
- *@Repository*: for classes at persistence layer

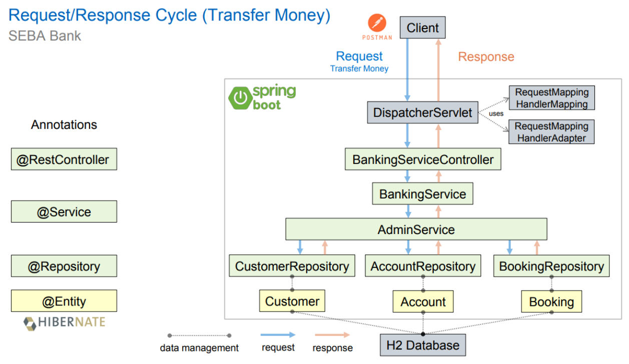
  Other Annotations:

- ***@SpringBootApplication***: for Spring Boot apps
- ***@Bean***: for objects which are created by Spring framework when the app starts (DI)
- ***@Autowired***: for DI (automatic instantiation of specific classes (beans))

- **Reflection:** process in which a program accesses info that belongs to the structure of the program itself
  - allows programs to examine, introspect and modify their own structure and behaviour at **compile-time** or **run-time**
  - *Example:* calling a method of an object by its name

```
public class ReflectionTest {
    public String test() {
      Account a = new Account();
      return Account.class.getMethod("getBalance").invoke(a);
    }
}
```

- **Request/Response Cycle:** *Representation of a Software Application => From* ***Persistence Layer*** *(Buttom) to* ***Presentation Layer*** *(Top)*



# 6 Persistent Data Management

## 6.1 Motivation

- **Management of Persistent Data in Business Applications:**
  - **Persistent Data:** data that is infrequently accessed, not likely to be modified and stored beyond the lifetime of the user session (non-volatile) (e.g. master data, transactional data, historical data)
- **Impedance Mismatch:** a set of conceptual and technical difficulties that are often encountered because objects or class definitions must be mapped to database tables defined by a relational schema -> RDBMS[11] with object-oriented programming
  1. specialized data structures for specific access patterns
  2. relational data storage for storage of bulk data
- **Databse Management Systems (DBMS):** entirety of programs for accessing database, checking consistency and modifying the data is called a database management system
  - **Persistence related functionalities:**
    - persistent data retention
    - modification of stored data
    - parallel data modifications
    - handling of mass data
    - ensuring compliance with integrity conditions
    - recovery in case of error
  - **Variants:**
    - **Relational Database**
    - **NoSQL Database**
- **Transactions (ACID)**:
  - a **transaction** is a single unit of work, often made up of multiple operations
  - transactions adhere to the **ACID** paradigm
  - **ACID:**
    - **Atomicity:** entire transaction takes place at once or does not happen at all
    - **Consistency:** database must be consistent before and after the transaction

[11] relational database management system

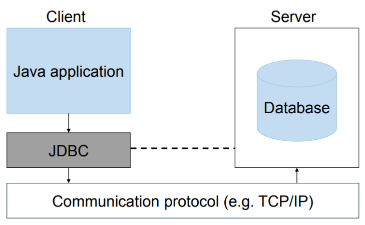
        * **Isolation:** multiple transactions occur independently without interference
        * **Durability:** changes of a successful transaction occurs even if the system fails

## 6.2 Programmatic Access to Relational Databases

- **Basics of Relational Databases:**
    - a **relational database** is a set of named tables
    - number of rows -> cardinality of relation, number of columns -> arity of relation
    - every relation has a **primary key** which can be a single attribute or can consist of several attributes
    - primary key of a table in another table -> **foreign key**
- **H2 Database:** a relational database written in Java
    - **Main Features:**
        * very fast, open-source, JDBC[12] API
        * embedded mode and server mode
        * in-memory database -> volatile, wiped out after the execution of app
        * browser-based console app (supports SQL)
- **Access to Persistent Relational Databases:**
    - **Goal:** business logic source code accesses persistently stored data
    - **Common Scenario:**
        * business logic developed in object-oriented programming language
        * relational database is used as persistent data storage
    - **Requirements:**
        * ACID for transactions
        * business logic independent of data access
    - **Different Implementation Strategies:**
        * **Direct SQL calls:** using an applicable technology from the programming language
        * **Software for Object-Relational Mapping:** automates aspects of access to the persistent data store

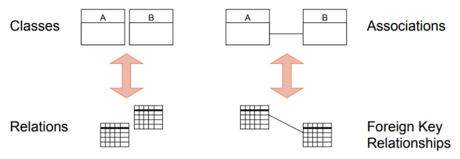
[12]java database connectivity

- **Java Database Connectivity (JDBC): Overview**



- **Advantages and Disadvantages of JDBC:**
  - **Advantages:** direct use of JDBC in applications is appropriate, if
    - stored procedures should be called
    - special queries have to be executed
    - proprietary database functionality is to be accessed
  - **Disadvantages:**
    - during development often error-prune, handling is too complex for developers
    - requires commitment to a certain persistence strategy
  - **JDBC calls should never be integrated directly into business logic code!**
- **Jakarta Persistence API (JPA):** Jakarta Persistence defines a standard API for persistent management of relational data in Java environments.
  - **Main Features:**
    - Object-Relational Mapping (ORM)
    - management of database access
    - JPA itself is only a specification -> part of Jakarta EE
- **Hibernate (Framework):**
  - **Main Features:**
    - **Object-Relational Mapping framework** for Java and RDBMS
    - open-source
    - full support for JPA
    - provides an SQL-like query language: **Hibernate Query Language**
    - serves as **abstraction above JDBC**
    - **easy to integrate** into projects

- **Object-Relational Mapping (ORM):** maps the state of objects to data in relational database to provide transparent persistent data storage and access



- **Software for Object-Relational Mapping:**
  - more complex structured transparently decomposed into flat structures of the RDBMS
  - developer sees only structures of object-oriented programming language -> transparent conversion from object-oriented operations to relational operations
  - features already provided by RDBMS do not need to be realized by the object-oriented application again -> ensuring data integrity
- **Object Serialization and Deserialization:**
  - **Serialization** describes process of converting an object and all its iteratively reachable objects into a **byte or character stream** such that the object can be reconstructed through **deseerialization**.
  - stream contains representations of attribute values, types, associations (links) between objects
  - can be used to store or send complex data structures
  - not all data types can be serialized (e.g. threads, files) -> exception thrown
  - in Java through the interface ***Serializable***

## 6.3 Persistent Entities

- **Basics of Persistent Entities:**
  - Persistent Entities provide object-oriented access to the persistent info in the database (e.g. customer class/table).
  - An **entity** is a persistent domain object annotated with ***@Entity***:
    - Entity classes are mapped to tables in relational database.
    - Each row represents an instance of that class.
  - multiple clients can use entity instances that represent the same data
  - each entity instance has a **unique primary key**

  - entities exit as long as database exists (secure against server failures) or until they are deleted

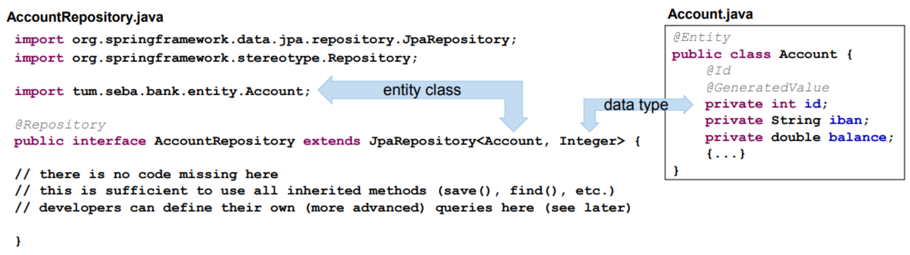
- **Specification/Development of an Entity:**
  - Entities obey JPA Specification:
    - class must be annotated with ***@Entity***
    - class must have a **default constructor** without arguments
    - state of an entity available to clients through **entity's methods**, getters/setters
    - class and methods must **not be final**
  - Entities are annotated for ORM:
    - for entity: ***@Entity***
    - for primary key: ***@Id***
    - for generation of primary keys: ***@GenerateValue(strategy = GenerationType.IDENTITY)***
    - for (re)naming a column (attribute) of table: ***@Column(name = "name")***
    - for (re)naming a table: ***@Table(name = "name")***
    - for enums: ***@Enumerated(EnumType.STRING)***
    - for handling circular references: ***@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")***

- **Developer View of a Persistent Entity:** developer requires methods of two different types:
  1. methods to **create**, **read**, **update** and **delete** (CRUD) <u>instances of entities</u>:
     - **EntityManager** in Jakarta
     - **Hibernate** provides its own EntityManager called Session
  2. methods to **read** and **update** <u>attribute values</u> of an entity instance

- **EntityManager:**
  - allows access to the data store by implementing programming interfaces and lifecycle rules defined by JPA
  - associated with a persistence context, within which entity instances and their lifecycles are managed

- **Spring Data:** Since managing **EntityManager** manually is cumbersome, error-prone and leads to boilerplate code, Spring provides **Spring Data**

- **Spring Data and Spring Data JPA:**
  - **Spring Data:**
    - ∗ focuses on repository abstraction
    - ∗ provides a familiar and consistent Spring-based programming model for data access
    - ∗ reduces the amount of boilerplate code required to implement data access layers
    - ∗ supports Criteria API
  - **Spring Data Jpa:**
    - ∗ part of larger Spring Data family, makes it easy to implement JPA-based repos
    - ∗ not a JPA implementation but an abstraction layer to use
- **Spring Data Repositories:**
  - **Annotation: *@Repository***
  - Spring will detect the annotation during component scanning and **provide an instance** at runtime
  - **CrudRepository:** provides CRUD functions for a given entity class
  - **PagingAndSortingRepository:** extends CrudRepository, pagination, sorting records
  - **JpaRepository:** extends PagingAndSortingRepository, flushing/batch deleting records
- **Crud Repository:**
  - a repository must extend ***JpaRepository<EntityClass, TypeOfId>***
  - **CrudRepository** provides the methods:
    - ∗ ***save()***, ***delete()***, ***findAll()***, ***findById()***, ***count()***...
    - ∗ <u>*Example:*</u>



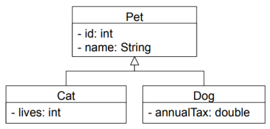
AccountRepository.java

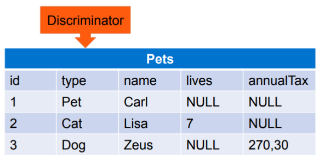
```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import tum.seba.bank.entity.Account;

@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

// there is no code missing here
// this is sufficient to use all inherited methods (save(), find(), etc.)
// developers can define their own (more advanced) queries here (see later)

}
```

Account.java

```
@Entity
public class Account {
    @Id
    @GeneratedValue
    private int id;
    private String iban;
    private double balance;
    {...}
}
```

- **Spring Data Repositories-Advantages and Disadvantages:**
  - **Advantages:**
    - less boilerplate code
    - easy configuration
    - simple queries out-of-the-box
  - **Disadvantages:**
    - code is coupled to library and its specific abstractions
    - complete set of persistence methds are exposed -> loss of control
- **Relational Mapping of Inheritance Hierarchies:**
  - **Single Table Strategy:** ***@Inheritance(strategy = InheritanceType.SINGLE_TABLE)***
    - all classes of inheritance hierarchy mapped to **one (same) table**
    - all attributes mapped to **colums**
    - **Advantages:**
      - easy **primary key** handling
      - good **polymorphic query** performance
    - **Disadvantages:**
      - many **NULL** values
      - **NOT NULL** constraints on subclass entity attributes are not possible

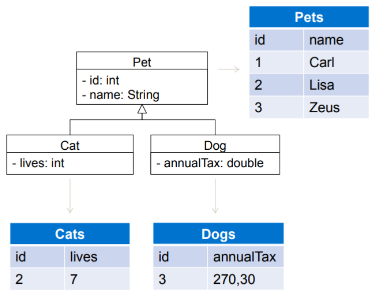



Discriminator
Pets

| id | type | name | lives | annualTax |
|---|---|---|---|---|
| 1 | Pet | Carl | NULL | NULL |
| 2 | Cat | Lisa | 7 | NULL |
| 3 | Dog | Zeus | NULL | 270,30 |

  - **Joined Table Strategy:** ***@Inheritance(strategy = InheritanceType.JOINED)***
    - each class of inheritance hierarchy mapped to **different tables**
    - only common attributes in **parent's table**, subclass-specific attributes, id in **child's table**
    - **Advantages:**
      - **no NULL** values
      - easy **primary key** handling

    - **Disadvantages:**
        - search for instances of Pet with **JOIN**

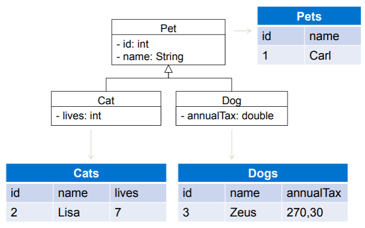


| Pets | |
|---|---|
| id | name |
| 1 | Carl |
| 2 | Lisa |
| 3 | Zeus |

| Cats | |
|---|---|
| id | lives |
| 2 | 7 |

| Dogs | |
|---|---|
| id | annualTax |
| 3 | 270,30 |

- **Table per Class Strategy:** ***@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)***
    - each class of inheritance hierarchy mapped to **different tables**
    - respective attributes in **one table**, instances of child **not in** parent's table (nor the opposite)
    - **Advantages:**
        - **no NULL** values
        - search for instances of Pet **without JOIN**
    - **Disadvantages:**
        - complex **primary key** handling



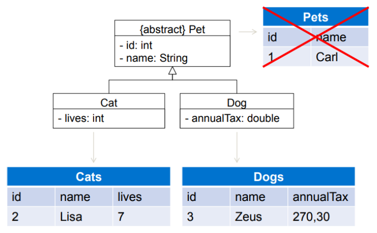
| Pets | |
|---|---|
| id | name |
| 1 | Carl |

| Cats | | |
|---|---|---|
| id | name | lives |
| 2 | Lisa | 7 |

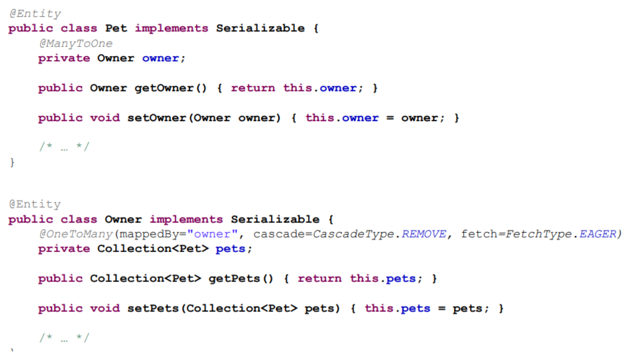
| Dogs | | |
|---|---|---|
| id | name | annualTax |
| 3 | Zeus | 270,30 |

- **Mapped Superclass Strategy:** ***@MappedSuperclass***
    - if parent class is **abstract**, **Mapped Suoerclass Strategy** can be used => no instances of parent
    - each subclass mapped to **different tables**
    - respective attributes in **one table**
    - **Advantages:**
        - **no NULL** values
        - easy **primary key** handling
        - easy way to share **mapping info** between entities
    - **Disadvantages:**

  · **polymorphic queries** not possible
  · **superclass (parent)** cannot contain associations with other entities



- **ORM for Associations Between Objects:**
  - **Associations in JPA:**
    * ***@OneToOne***
    * ***@OneToMany / @ManyToOne***
    * ***@ManyToMany***
  - **Parameters for update/delete propagation:** e.g. ***@OneToOne(cascade = CascadeType.REMOVE)***
  - **Different loading strategies:**
    * referenced objects loaded immediately: ***fetch = FetchType.EAGER***
    * referenced objects loaded later on demand: ***fetch = FetchType.LAZY***
  - *Example in Java:* ***Pet*** *Class has the* ***primary key*** *of* ***Owner*** *Class (mappedBy) as a* ***foreign key*** *in its table.*

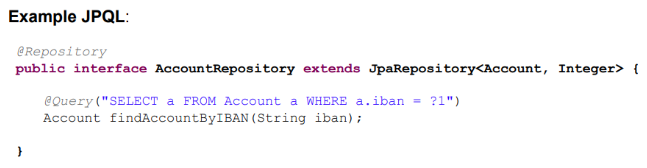


```
@Entity
public class Pet implements Serializable {
    @ManyToOne
    private Owner owner;

    public Owner getOwner() { return this.owner; }

    public void setOwner(Owner owner) { this.owner = owner; }

    /* … */
}


@Entity
public class Owner implements Serializable {
    @OneToMany(mappedBy="owner", cascade=CascadeType.REMOVE, fetch=FetchType.EAGER)
    private Collection<Pet> pets;

    public Collection<Pet> getPets() { return this.pets; }

    public void setPets(Collection<Pet> pets) { this.pets = pets; }

    /* … */
`
```

## 6.4 Query Languages

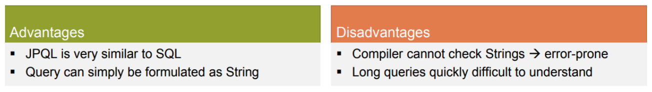
- **Java Persistence Query Language (JPQL):**

  JPQL is not **type-safe**!



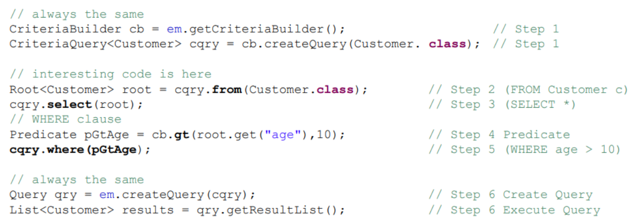
**Example JPQL**:

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {

    @Query("SELECT a FROM Account a WHERE a.iban = ?1")
    Account findAccountByIBAN(String iban);

}
```



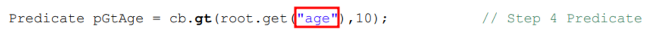
| Advantages | Disadvantages |
|---|---|
| ▪ JPQL is very similar to SQL | ▪ Compiler cannot check Strings → error-prone |
| ▪ Query can simply be formulated as String | ▪ Long queries quickly difficult to understand |

- **Criteria API:**
  - **Motivation:**
    - Neither attributes nor classes are **type-safe** in **JPQL**.
    - Also, compiler cannot check the JPQL keywords (SELECT, FROM).
  - *Example:*



```
// always the same
CriteriaBuilder cb = em.getCriteriaBuilder();                       // Step 1
CriteriaQuery<Customer> cqry = cb.createQuery(Customer. class);   // Step 1

// interesting code is here
Root<Customer> root = cqry.from(Customer.class);           // Step 2 (FROM Customer c)
cqry.select(root);                                         // Step 3 (SELECT *)
// WHERE clause
Predicate pGtAge = cb.gt(root.get("age"),10);              // Step 4 Predicate
cqry.where(pGtAge);                                        // Step 5 (WHERE age > 10)

// always the same
Query qry = em.createQuery(cqry);                          // Step 6 Create Query
List<Customer> results = qry.getResultList();              // Step 6 Execute Query
```

    - **Methods for predicate definition:**
      - ***cb.gt()***: greater than
      - ***cb.equal()***: equal
      - ***cb.between()***: between
      - ***cb.and(Predicate a, Predicate b)***: WHERE a AND b
  - **Type Safety:**
    - **Type safety on attributes not yet ensured:** Reference to attribute with its name as String -> error prune



```
Predicate pGtAge = cb.gt(root.get("age"),10);           // Step 4 Predicate
```

- **Querydsl:** a framework that enables construction of statically typed SQL-like queries
  - *Example:*

    ```
    List<Person> persons = queryFactory.selectFrom(person)
    .where(
        person.firstName.eq("John"),
        person.lastName.eq("Doe"))
    .fetch();
    ```

  - **Advantages:**
    - more human-readable than Criteria API
    - open-source
    - statically typed = type-safe queries

## 6.5 Alternatives for Persistent and Bulk Data Management

- **NoSQL Databases:** a new generation of database systems which considers following points:
  - associated data model **not relational**
  - systems designed for **distributed and horizontal scalability**
  - **schemaless** or weaker schema restrictions
  - due to distributed architecture, **easy data replication**
  - **simple API**
  - **eventually consistent**, but **not ACID**
- **When to use NoSQL instead of Relational Databases:**
  - performance problems in relational databases:
    - indexing of large amount of documents
  - Relational databases are efficient if they are **optimized for frequent but small transactions** or for **large batch transactions with rare write accesses**.
  - Relational databases are unable to handle **high data requirements** and **frequent data changes** at the same time.
  - NoSQL databases can handle many **write** and **read** requests (e.g. Facebook, Amazon).

# 7 Architecture of Distributed Information Systems

## 7.1 Characteristics of Distributed Systems

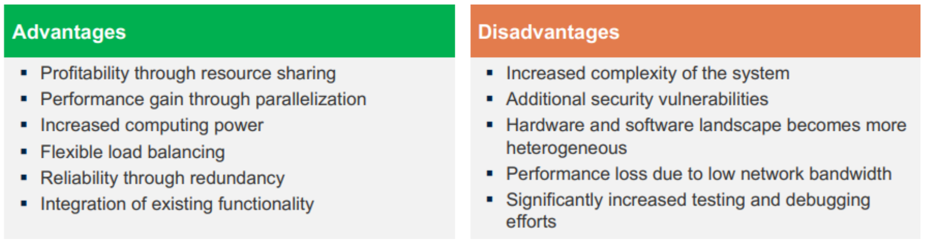
- **Foundations:**
  - **Distributed system:** a system that is comprised of several physically disjoint compute resources interconnected by a network
  - **Distributed Application:** an application consisting of severeal processes that run distributed in several process spaces
  - **Characteristics of a distributed system:**
    - resource sharing
    - openness (communication protocols and interfaces)
    - concurrency
    - scalability
    - failure tolerance
    - distribution transparency
- **A Centralized Information System Architecture:**
  - *Example:* Central mainframe
    - multi-user operation
    - optimized for large amounts of data
    - high reliability
    - minimal logic on the terminal, only display of data
- **A Local Area Network of Distributed Clients and Servers:**
  - distributed system within a company, department
    - multi-user operation
    - distributed services and information
  - different functionalities supported by different servers
  - automation of business processes
  - requirement of integration effort and leads to **productivity paradox**:
    - more IT investments do not lead to higher productivity
    - limited realization of compound effects
- **Resource Sharing:**
  - **client-server model** -> based on service-oriented achitecture for resource sharing

    - **server** processes provide **resource managers**, they provide shared resources (e.g. data, code, hardware, processes)
    - **client** processes issue requests to use these remote resources
    - client initiates the communication

- **Opennes and Concurrency:**
    - **Openness**
        * extensibility of the system at the
            · **hardware level:** new peripherals, memory, etc.
            · **software level:** new resource services, communication protocols
        * specifications for **interfaces** of the system are disclosed and documented
    - **Concurrency:**
        * multiple users issue independent and concurrent requests via client processes
        * server processes run concurrently
        * multiple processes may exist for each resource type to improve scalability
- **Scalability and Failure Tolerance:**
    - **Scalability**
        * adapting the system for larger data volumes, workloads, faster throughput
        * increase in complexity
        * ideally without changing application architecture
        * **Goal:** constant performance with increasing load
    - **Failure tolerance**
        * ability of distributed system to provide functionalities even if a number of defective subsystems (servers/clients) exist
        * implementation via redundant subsystems, a certain number of which can be defective
        * guarantee of high availability
- **Transparency:**
    - **distributed system apppears as a single computer system** -> structure hidden from programmer
    - **Forms of transparency**
        * **Access transparency:** local and remote objects are treated equally

    * **Location transparency:** objects can be accessed without knowing their location
    * **Concurrency transparency:** multiple processes can interact concurrently with shared objects without interference
    * **Replication transparency:** transparent use of multiple instances of an object
    * **Migration transparency:** location change of an object is transparent

- **Misconceptions About Distributed Systems:**
  - network is reliable, secure, homogeneous
  - latency[13] and transport cost is zero
- **Distributed Systems vs. Centralized Information Systems:**



| Advantages | Disadvantages |
|---|---|
| ▪ Profitability through resource sharing<br>▪ Performance gain through parallelization<br>▪ Increased computing power<br>▪ Flexible load balancing<br>▪ Reliability through redundancy<br>▪ Integration of existing functionality | ▪ Increased complexity of the system<br>▪ Additional security vulnerabilities<br>▪ Hardware and software landscape becomes more heterogeneous<br>▪ Performance loss due to low network bandwidth<br>▪ Significantly increased testing and debugging efforts |

## 7.2 Synchronous and Asynchronous Communication

- **Communication in Distributed Systems:**
  - It is a **fundamental architectural decision** whether the communication of distributed components is implemented **synchronously** or **asynchronously** since it leads to **different forms of process coupling**
  - **Asynchronous communication:** sender process does not wait for receiver process
  - **Synchronous communication:** sender waits for receiver process to accept/perform a request
- **Synchronous Communication via Remote Procedure Calls:**

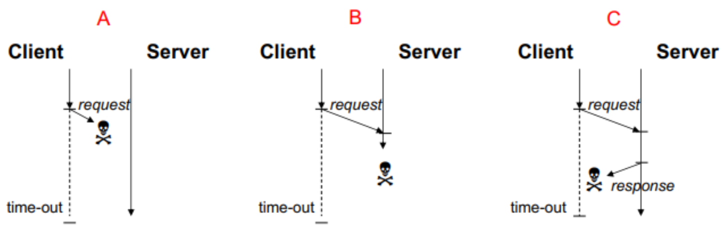
  **Remote Procedure Call (RPC)**

  - clients call **remote methods**

[13]**Network latency** is the delay in network communication. It shows the time that data takes to transfer across the network.

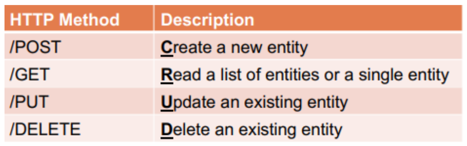
  - **Marshalling:** transformation of structured argument values or results into linear messages (memory representation -> data format suitable for storage or transmisson)
  - **Unmarshalling:** transformation of linear messages into structured argument values or results
  - **Proxy** and **skeleton** are provided by a middleware and encode and decode the messages

- **General Challenge of Distributed Systems:**
  - client and server can crash independently
  - messages can be lost
  - difficulty to reach a consensus on the distributed world
  - *Example:* client does not receive answer to a **synchronous** request in a remote procedure call -> request times out, but what happened (**A**, **B** or **C**)?



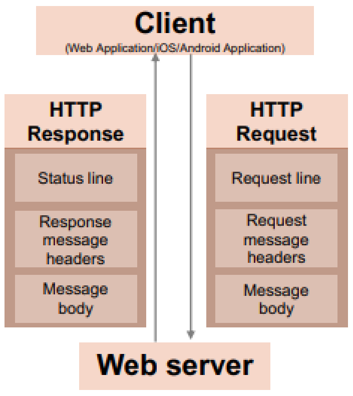
- **Alternatives for Handling Time-Outs in a RPC:**
  - The programmer has to decide how to deal with a **time-out** based on the **semantics** of the executed operation and the specific **business context**.
  - **Maybe semantics:** The server procedure is executed **once** or **not at all**. In case of an error, the client cannot know whether the procedure. -> in case of time-out, client **does not repeat** its request
  - **At-least-once semantics:** The server procedure is executed **at least once**. -> in case of time-out, client **repeats** its request
  - **At-most-once semantics:** The server procedure is executed **at most once**. -> in case of time-out, client **repeats** its request and **marks** it as a repeat
  - It is practically impossible to achieve the semantics of a **local** procedure call:
  - **Exactly-once:** The procedure is executed **exactly once**.

- **Hypertext Transfer Protocol (HTTP):** is an application-layer protocol for transmitting hypermedia documents, such as HTML
  - designed for **synchronous communication** between **web browsers** and **web servers**
  - follows **client-server model**, with client opening a **connection**
  - **stateless** => every HTTP request is independent and server does not keep any record of previous client requests
- **Client-Server Communication with HTTP:**
  - client sends **request** to server
  - server receives request, carries out the operation defined in request and sends a **response** to client
  - to represent **structured data** in the request and response as hypertext, XML or JSON are used
- **Create, Read, Update, Delete (CRUD), HTTP Request Methods:** HTTP defines a fixed set of **request methods** to indicate the desired action for a given resource.

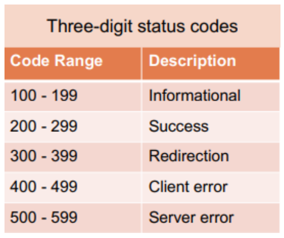


| HTTP Method | Description |
|---|---|
| /POST | Create a new entity |
| /GET | Read a list of entities or a single entity |
| /PUT | Update an existing entity |
| /DELETE | Delete an existing entity |

- **HTTP Messages-Request and Responses:**



- **HTTP Status Code:**
  - **status code** is explained with a **textual phrase** that is sent along as part of the status line

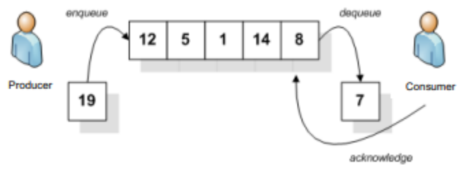


Three-digit status codes

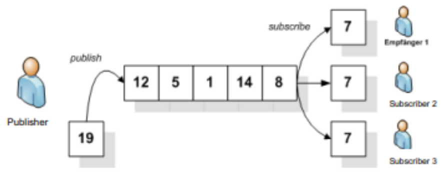
| Code Range | Description |
|---|---|
| 100 - 199 | Informational |
| 200 - 299 | Success |
| 300 - 399 | Redirection |
| 400 - 499 | Client error |
| 500 - 599 | Server error |

- **Web API:** provides services to be consumed by application code
  - consists of a defined set of HTTP request messages
  - for each request, web API specifies the structure of response
  - response is typically expressed in JSON or XML
- **Developer Support for Testing Web APIs:**
  - HTTP requests are normally sent from application code (e.g. Java). For debugging and development, following tools can be used:
    - **Postman** is a collabration platform for API development. It has an API Client to send REST requests directly within the app.
- **Messaging and Message-Oriented-Middleware:**
  - **Message-Oriented Communication**
    - **asynchronous** distributed communication through the exchange of **transient or persistent messages** between software components
    - introduction of a queue as an "intermediate component" between sender and receiver allows an **indirect** communication
    - *Example:* E-Mail Services (unstructured messages)
  - **Message-Oriented Middleware**
    - provides generic services for **reliable** message-oriented communication through persistent and transient queues
    - allows to implement **reliable** message delivery (exactly once semantics), but also lower levels of reliability
    - *Examples:* JBoss Messaging

- **Messaging Models:**
  - **Point-to-Point Communication**
    - **producer** (sender: creates message/request) and **consumer** (receiver)
    - producer knows the destination of message and places it in the queue
    - asynchronous communication between producer and consumer
    - each message received by the consumer is confirmed by him



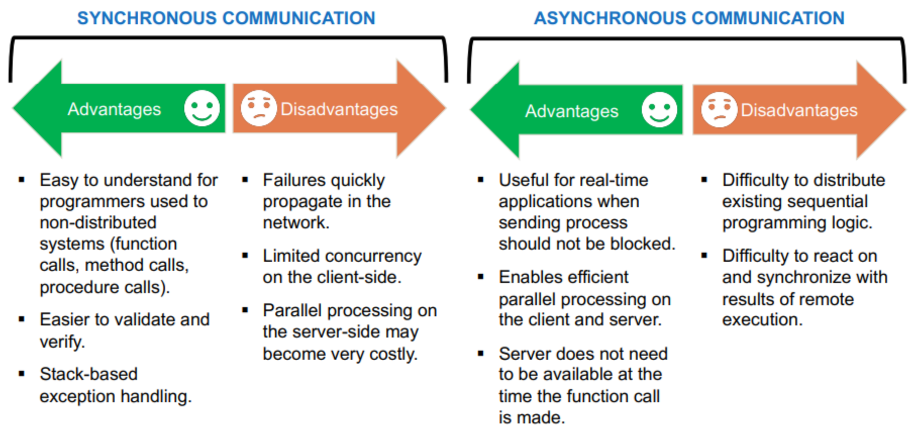
  - **Publish-Subscribe**
    - **publisher** (sender) and **subscriber** (receiver) are not known to each other
    - subscriber subscribes to receive messages on a **topic**
    - publisher knows the destination of message and places it into a topic
    - multiple subscriber receive identical messages
    - asynchronous messaging



- **Indirect Asynchronous Messaging:**
  - **Advantages of indirect asynchronous messaging**
    - setup of sender and receiver component **independent** of each other -> allows easy exchange of components
    - sender need **no info** about receiver, vice versa
    - sending and subsequent processing by sender

  This leads to a **loose coupling** of software components.

- **Asynchronous vs. Synchronous Communication:**



SYNCHRONOUS COMMUNICATION

Advantages

- Easy to understand for programmers used to non-distributed systems (function calls, method calls, procedure calls).
- Easier to validate and verify.
- Stack-based exception handling.

Disadvantages

- Failures quickly propagate in the network.
- Limited concurrency on the client-side.
- Parallel processing on the server-side may become very costly.

ASYNCHRONOUS COMMUNICATION

Advantages

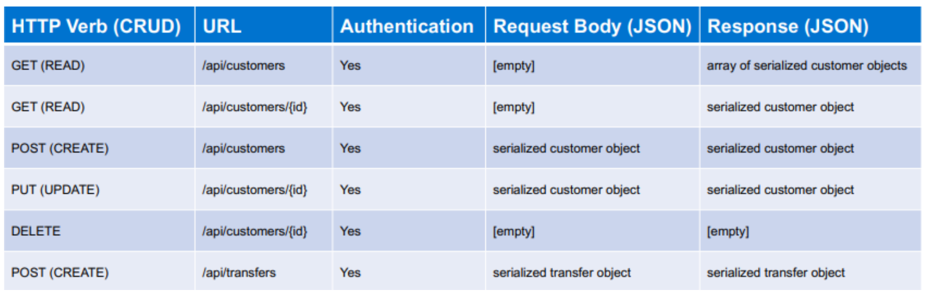
- Useful for real-time applications when sending process should not be blocked.
- Enables efficient parallel processing on the client and server.
- Server does not need to be available at the time the function call is made.

Disadvantages

- Difficulty to distribute existing sequential programming logic.
- Difficulty to react on and synchronize with results of remote execution.

## 7.3 Implementing REST API

- **Representational State Transfer (REST):** a software architectural style for distributed hypermedia systems like the world wide web
  - provides a set of **architectural constraints** that, when applied as a whole, emphasizes:
    - ∗ scalability of component interactions
    - ∗ generality of interfaces
    - ∗ independent deployment of components
    - ∗ intermediary components to reduce interaction latency, enforce security and encapsulate legacy systems
- **An Extract API for the SEBA Banking App:**



| HTTP Verb (CRUD) | URL | Authentication | Request Body (JSON) | Response (JSON) |
|---|---|---|---|---|
| GET (READ) | /api/customers | Yes | [empty] | array of serialized customer objects |
| GET (READ) | /api/customers/{id} | Yes | [empty] | serialized customer object |
| POST (CREATE) | /api/customers | Yes | serialized customer object | serialized customer object |
| PUT (UPDATE) | /api/customers/{id} | Yes | serialized customer object | serialized customer object |
| DELETE | /api/customers/{id} | Yes | [empty] | [empty] |
| POST (CREATE) | /api/transfers | Yes | serialized transfer object | serialized transfer object |

- **Spring RESTful Service Controllers:**
  - **RESTful controllers** populate and return domain objects serialized as JSON via HTTP whereas Spring MVC controllers return views.
- **Annotations and Types for REST API:**
  - ***@RestController*** indicates that the data returned by each method will be written straight into the response body -> for **classes**
  - ***@RequestMapping*** maps HTTP requests to handler methods of REST controllers. -> for **classes**
  - ***@GetMapping*** maps GET request onto specific methods of the controller class -> for **methods**
  - ***@PostMapping***, ***@PutMapping***, ***@DeleteMapping*** -> for **methods**
  - ***ResponseEntity<?>*** represents an HTTP response -> for **return values of methods**
  - ***@PathVariable*** indicates that a method parameter should be bound to a URI template variable[14]
  - ***@RequestBody*** is used to access HTTP request body, body content is converted to the method argument ((un)marshalling of Java objects to/from JSON, XML) -> for **parameters of methods**
  - ***@Valid*** triggers the **validation** of the annotated method parameter and puts the result into ***BindingResult*** => **Request Validation**, if incorrect request body, HTTP (error) status code as reponse (e.g. Status 400: Bad Request) -> for **parameters of methods**
  - ***@ControllerAdvice*** allows developers to handle exceptions across the whole application -> for **classes**
  - ***ResponseEntityExceptionHandler*** handles common exceptions out-of-the-box (e.g. ***TypeMismatchException***) => **Exception Handling** -> for **extending classes**
  - ***@ExceptionHandler*** configures the advice to only respond if a ***EntityNotFoundException*** is thrown -> for **methods**

## 8 Security of REST-based Business Applications

### 8.1 Introduction and Fundamentals

- **Security in Business Applications:**
  - data processed in business applications often contains **sensitive information**:

[14]A **URI path template** has one or more variables, with each variable name surrounded by curly braces.

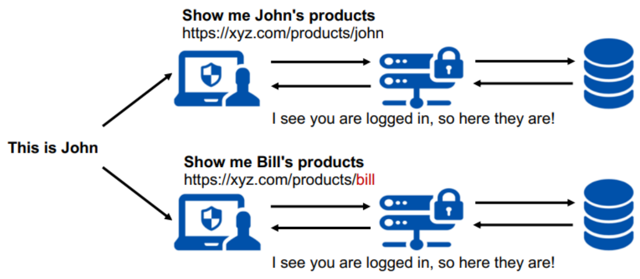
        * personal data
        * confidential business information
        * classified information
    - Therefore, only **a specific user or trusted user** should be able to access this data. => data must be **protected** against attackers, otherwise **financial** and **reputational** damages

- **Definitions and Delimitation:**
    - **Information Security:** processes and methodologies for preservation of confidentiality, integrity and availability of information -> not only about **technology**, but also **physical security** (protection against natural disasters), **human factor** (protection against social engineering)
    - **IT Security:** refers to **technical aspects of Information Security** -> not only **software**, but also **physical hardware** and **IT networks**
    - **Security Engineering: Information Security + Software Engineering** => focusing on **secure implementation** of a web-based **API** with the help of **Spring Security** framework, building on top of **Spring Boot**
- **IT-Security Layers:** A key aspect in the engineering of secure software systems is to apply **security measures in multiple layers**.
    - to **properly secure** business app, **infrastructure, that business app is runing on,** must be secured
    - focusing on **securing application layer** (top layer) with the help of **Spring Security** framework
- **Core Information Security Properties (CIA):**
    - **Confidentiality:** Can **non-authorized** parties see data?
    - **Integrity:** Has data been **altered** (and should I know this)?
    - **Availability:** Is data **always accessible**?
- **Extension of CIA:**
    - **Non-repudiation:** impossibility to inappropriately deny a transaction or having sent a message
    - **Auditability:** ability to reconstruct earlier states of a system
    - **Usability:** ability to easily use and apply the security measures in place

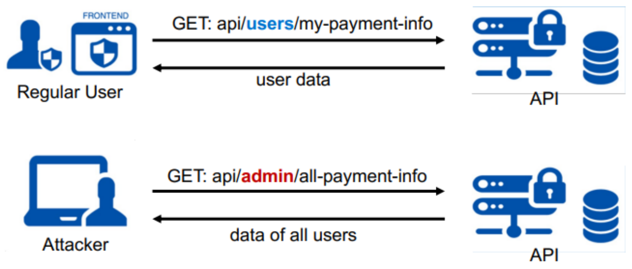
- **Core Information Security Principles:**
  - There is **no guarantee** in security:
    - total security is unrealistic
    - **risk assessments** are important
    - analyzing impact and probability of occurrence
    - **Goal: balancing trade-off between risk, cost and usability**
  - how to know if **enough security**? => difficult to measure, depends on risk appetite of the company

## 8.2 Common Risks in Web-Based APIs

- **Broken Object Level Authorization:**
  - **lack of proper authorization checks** allows access to a resource object that is not meant for current authenticated user
  - simply by replacing the identifier in the URL


Show me John's products
https://xyz.com/products/john
I see you are logged in, so here they are!
This is John
Show me Bill's products
https://xyz.com/products/bill
I see you are logged in, so here they are!

- **Broken Function Level Authorization:**
  - some administration functions are **exposed as endpoints of the API**, but the user role is not checked on the server side
  - API relies on the client application to use the proper endpoint of the respective role of the user


FRONTEND
Regular User
GET: api/users/my-payment-info
user data
API
Attacker
GET: api/admin/all-payment-info
data of all users
API

## 8.3 Important API Security Measures and Spring Security

- **Managing Users and Access:**
  - management of users and their **access rights**
  - identification of user -> **Authentication**
  - management of users and their **credentials**, login/logout, register, importing users from existing sources
  - deactivate/delete account by user/admin
  - based on user identity, user have access to certain resources or functionality -> **Authorization**
- **Authentication:** <u>how the identity</u> of who is trying to access a particular resource is <u>verified</u>
  - **Common authentication types:**
    - username and password
    - certificate authentication
    - multi-factor authentication
  - **Authentication input:** How are the credentials passed to the server?
    - form-based authentication -> in request body
    - basic authentication -> in request header
  - **Storage mechanisms:**
    - in-memory authentication
    - JDBC authentication
    - LDAP[15] storage
- **Dealing with Passwords:**
  - **never** storing passwords in **plain-text**
  - using **hashing algorithms**
  - **Spring Security** provides the ***PasswordEncoder*** interface and multiple implementations
  - **Encoding:** process of converting data from one form to another => reversible (decoding), for efficient transmission or storage
  - **Encryption:** process of transforming data in order to keep it secret from others, only specific individuals can reverse the transformation (e.g. with a private key)
  - **Hashing:** process of generating a deterministic output from a given input => one-way, irreversible, for validation of integrity of data

[15]lightweight directory access protocol

- **Basic Authentication:**
  - **Spring Security** uses **HTTP Basic Authentication**
  - basic auth only requires the client to send username and password through **HTTP Authorization header**
  - *Base64*[16] is only an encoding, not an ecryption or hashing method
  - **encrypted transit** with HTTPS
  - in value of header: *Authorization: Basic <Base64 Encoding of Credentials>*
  - **limitation of basic auth:** user has to send credentials with each request
- **Authorization:** how to check permission or role of the user, if the user is allowed that access (access control) or not

  **Common authorization methods:**
  - role-based access control (RBAC): access decisions based on roles
  - attribute-based access control (ABAC): access decisions based on user attributes -> more fine-grained, also more complex
- **Security for Backend Frontend Separation:**
  - **client/frontend** communicates with **server/backend** (via REST endpoints)
  - *HTTP Basic Authentication* implies sending credentials with each call -> not well suited for many real-world apps
    - user must retype credentials with every request
    - credentials should not be stored on the client side (difficult to protect)
  - **server-side sessions** and **session cookies** solve this by storing a **session id** on the client, which is used to identify the client and keep him logged in without having to resend credentials
  - server-side sessions conflict with **statelessness** on the server-side -> in a proper REST API we usually aim to **avoid them**
  - **limitation of basic auth:** credentials cannot be managed by a separate system -> but in many cases, **delegation of authentication and authorization to different app**
- **Tokens:** As a modern alternative, **token-based authentication and authorization** can be used. Instead of retyping credentials or a session cookie, **a token is sent with every request**.

[16]**Base64** is used to encode binary data as printable text.

  - fit well to the **statelessness** of a RESTful web sevice -> no session context stored on the server-side, session state entirely in the client, sent with every request to the server => can be done with **token**
  - has a **fixed lifetime**, when expires generate new

- **JWT Tokens:** defines a compact and self-contained way for securely transmitting information between parties as a JSON object
  - consists of **header**, **payload (data)** and **signature**
  - **header:** describes the type of the token as well as the hashing algorithm used for the signature
  - **payload:** contains multiple claims, e.g. username, expiration date
  - **signature:** is generated by the hash of the header, payload and a secret held by the server, it ensures the integrity of the token.

- **OAuth:** a protocol for authorization (**not** an authentication protocol)
  - Defines 4 main roles:
    * **Resource Owner:** capable of granting access to protected resource = end-user
    * **Resource Server:** server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens
    * **Client:** an application making protected resource requests on behalf of the resource owner and with its authorization.
    * **Authorization Server:** server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization
  - **Advantage:** enables user to allow a client app to trigger a certain action on the server, without the user having to share his credentials with the app

- **Validate User Input:**
  - user input should never be trusted
  - any data that reaches API could be altered, if it is not cryptographically signed
  - therefore, validate input for unexpected input, reject the request if necessary:
    * using **allowlist** (only specific input possible, reject everything else)
    * using **blocklist** (all inputs possible, except for blocked input)

  - avoid building SQL queries from user input, instead **bind parameterized data**[17]
  - **input validation** can be done at **multiple layers** (filters, controllers, services or persistence layer)

- **Protect Data in Transit:**
  - never send unencrypted sensitive data over the internet -> traffic can be intercepted easily and all of plaintext data in the transferred packets can be read
  - *TLS (Transport Layer Security)* is an encryption protocol that is commonly used with the *HTTP*-protocol to encrypt data iver a network -> *HTTP* runs through *TLS* = ***HTTPS***
  - **Spring Security** helps to ensure that clients only speak with your API via *HTTPS* by enabling *HSTS* (*HTTP Strict Transport Security*)

- **Log and Monitor Application Activity:**
  - to be able to respond to security incidents within the application, it is essential to detect those first => proper logging and monitoring of application enables detection and response
  - **Spring Security** does not provide specific loggin functionality
  - in **Spring Boot** *Logback*

- **Spring Security:** a framework that focuses mainly on **authentication**, **authorization** and **protection against common exploits**

- **Implementing Spring Security:**
  - to prevent unauthenticated access to the application, Spring Security applies **authentication before** the request enters the ***Controller***
  - **authorization** at 3 different layers:
    - ∗ endpoint/controller layer
    - ∗ service layer
    - ∗ repository layer
  - if user not authorized for a specific endpoint or method, (by default) a 403 Forbidden HTTP error will be returned

- **Restricting Access at the Endpoint Layer:**
  - we can use the configure-method of Spring Security to enforce authentication and authorization for our application

[17]**Binding parameters** an alternative way to pass data to the database. Instead of putting the values directly into the SQL statement, you just use a placeholder and provide the actual values using a separate API call.

  - in most cases, it is necessary to apply different rules for different requests
  - matcher methods can be used to choose the endpoints. (e.g. ***mvcMatchers()***)

- **Managing Users and their Privileges:**
  - as part of user management, Spring uses ***UserDetailsService*** and ***UserDetailsManager*** interfaces:
    * ***UserDetailsService*** is only responsible for retrieving the user by username
    * ***UserDetailsManager*** adds behavior that refers adding, modifying or deleting the user
  - for actual user class, Spring Security offers ***UserDetails*** interface:
    * for authorization purposes, **a user has a set of privileges**, which are the actions user is allowed, or **role that grants** the user these permissions
    * Spring Security represents these with ***GrantedAuthority*** interface
    * ***SecurityContext*** keeps details about the authenticated entity after successful authentication, for the duration of processing the request