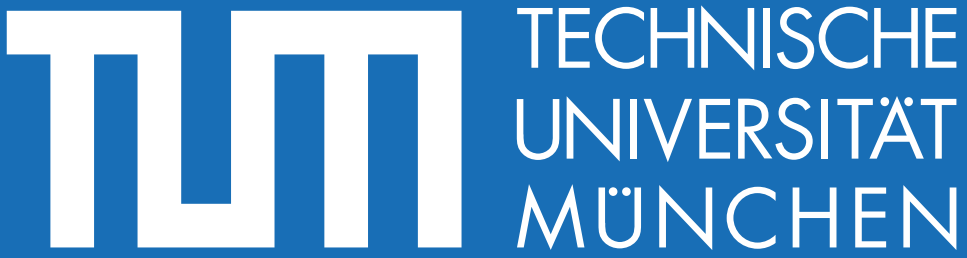


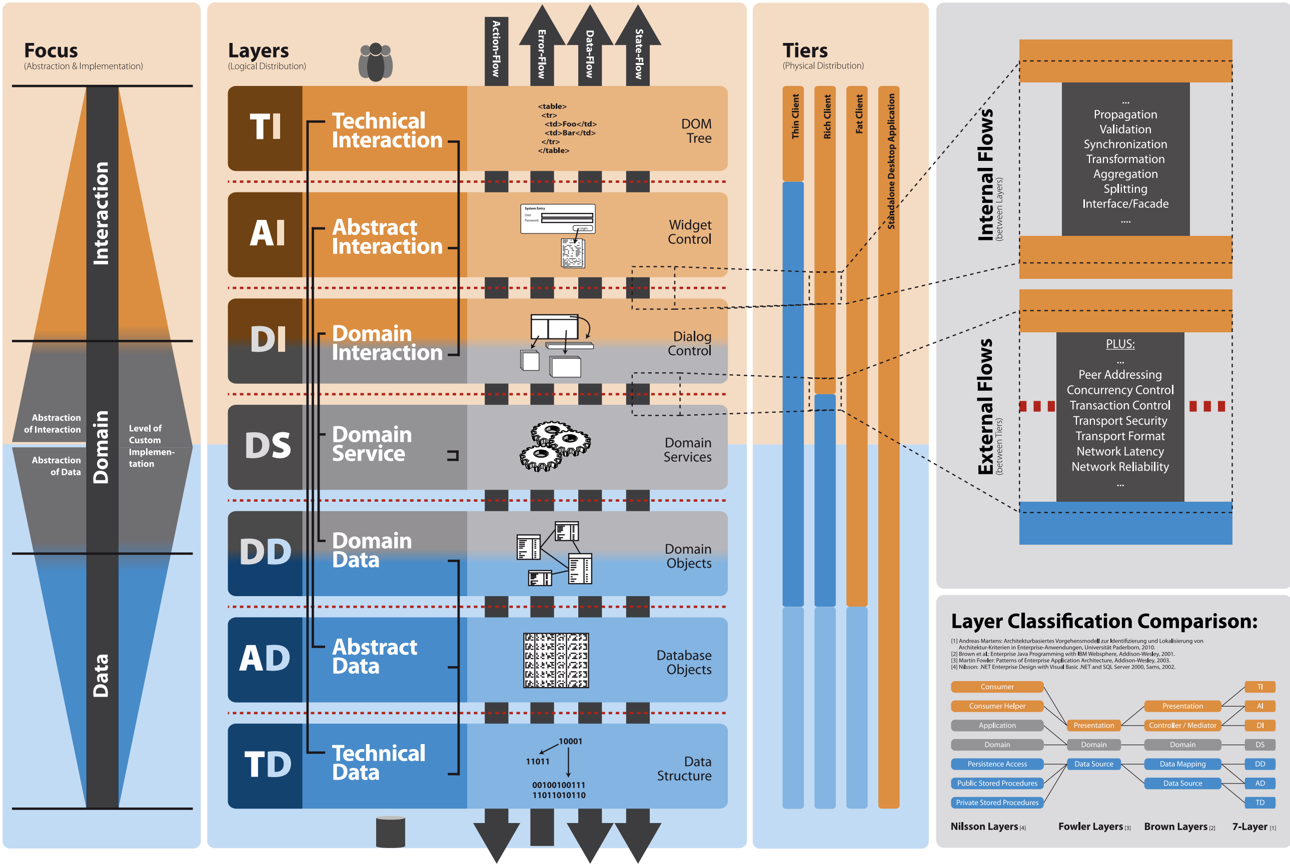
Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall

ARCHITECTURE FUNDAMENTALS

Application Reference Architecture

TECHNISCHE UNIVERSITÄT MÜNCHEN



In an application, it is possible to distinguish 7 logical layers, each grouped in two ways: on the one hand, there are the three sequential layer groups **Technical/ Abstract/Domain Interaction**, **Domain Service** and **Domain/Abstract/Technical Data**, on the other hand, there are the three nested layer groups **Technical Interaction/Data**, **Abstract Interaction/Data** and **Domain Interaction/Service/Data**.

In addition, one can distinguish 4 primary flows in an application: the **Action Flow** consequently runs from top to bottom only because all actions at the top are triggered by the user (or neighboring systems); the **Error Flow** consistently runs only in the opposite direction, i.e., from the bottom to the top, because errors, in the worst case, must be reported to the user; the (domain-specific) **Data Flow** and the (technical) **State Flow** run in both directions because data and states have to be persisted as well as displayed.

The abstraction of Interaction/Data in the layers increases from the top/bottom towards the middle, so most of the functional code of an application is written there. For the upper/lower layers, one usually massively relies on Open Source libraries/frameworks.

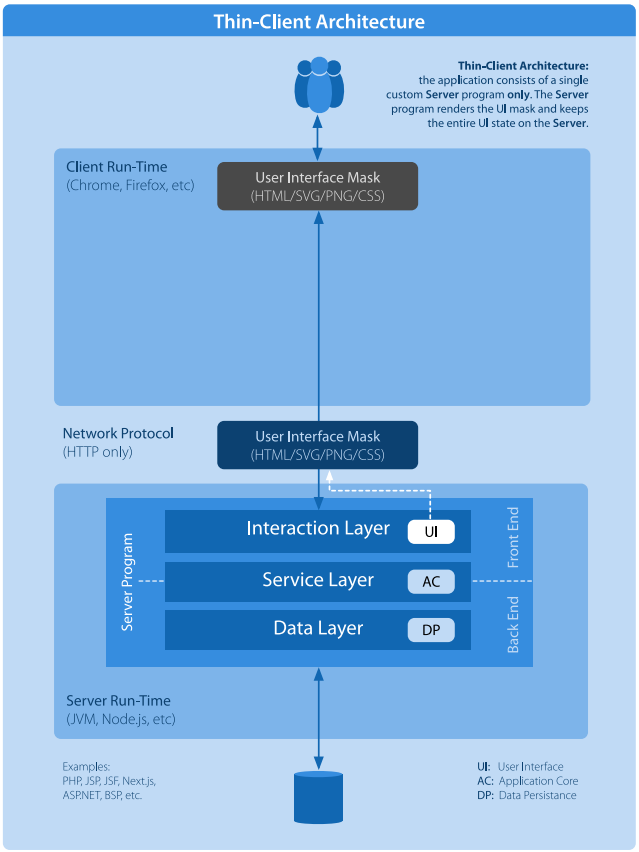
If instead of a logical cut (resulting in an **Internal Flow** between the layers) between two layers, one makes a physical cut (which then results in an **External Flow**), i.e., one distributes the application into single programs on different computers, then the resulting architecture is called according to the scope and responsibility of the client.

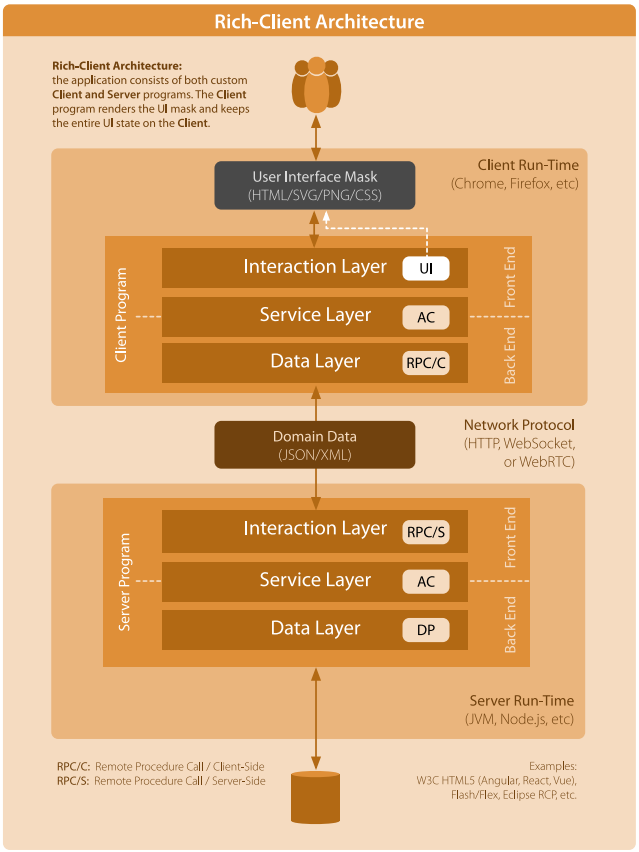
With **Thin Client**, only the **Technical Interaction** is offloaded to the client, while with **Rich Client** the entire user interface (i.e., all three layers **Technical/Abstract/ Domain Interaction**) is autonomously offloaded to the client (usually as a so-called "HTML5 Single-Page-Application"), with **Fat Client** there is no more associated server at all, and with the **Standalone** application, there is only one single program.

Questions

- What is the name of the application architecture in which the entire user interface runs autonomously on the client, while the server only provides purely functional services?
- What are the web applications named that implement a **Rich Client** architecture?

Client-Server Architecture





In the **Thin-Client Architecture**, the application consists of a single custom Server program only. This Server program renders the User Interface Mask and keeps the entire state of the User Interface on the Server.

The advantage of this architecture is that the application can be updated very easily. The disadvantage of this architecture is that the user interface reacts sluggishly, and the state of the user interfaces of all clients must be kept on the server, which can make the server a bottleneck.

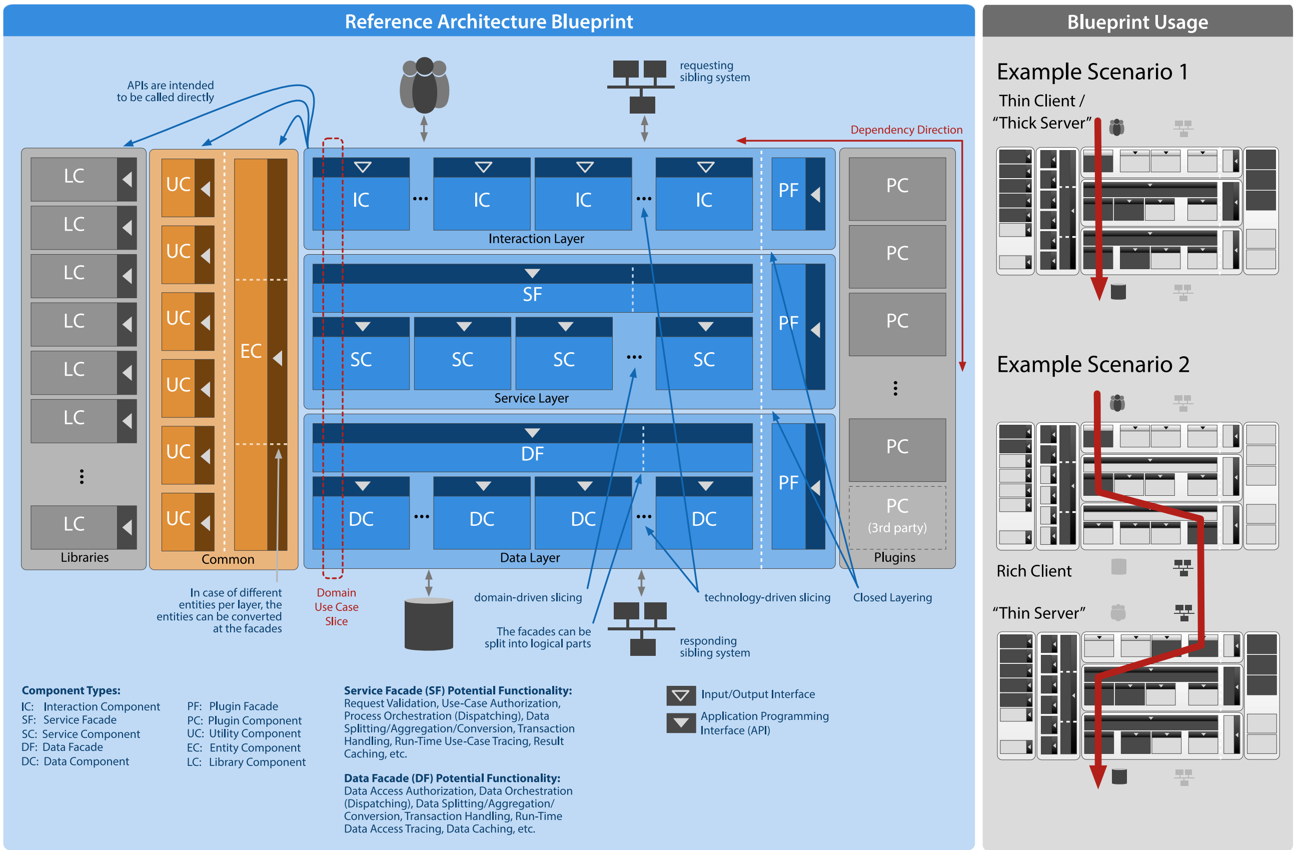
In the **Rich-Client Architecture**, the application consists of both custom Client and Server programs. The Client program renders the User Interface mask and keeps the entire state of the User Interface on the Client.

The advantage of this architecture is that the user interface is highly responsive, only domain-specific data has to be exchanged between the client and the server and the server server becomes less of a bottleneck. The disadvantage of this architecture is that, if necessary, the client has to be updated explicitly via an installation procedure.

Questions

- With which Client Architecture does the User Interface offer the higher responsiveness?

Information System Architecture



A (business) Information System usually follows a stringent component-based reference architecture. This is represented "full blown" and can be arbitrarily "slimmed down."

First, this reference architecture consists of 3 substantial Layers: the **Interaction Layer** with the (technically cut) components, which provide the I/O-based interfaces to the user (User Interface) and/or requesting neighboring systems (via Network interface), the **Service Layer** with the (domain-specifically cut) service components (also called Application Core) and the **Data Layer** with the (technically cut) components that provide the connection to the own database and/or neighboring systems to be queried.

Note that the "docking position" of a neighboring system depends on its roles: if it requests, it docks to the Interaction Layer; if it is queried, it docks at the Data Layer. If it happens to have both roles, it docks twice. The other view is that both the user and the database can be understood as special "neighbor systems."

To connect the N **Interaction Components** (IC) with M **Service Components** (SC) a decoupling **Service Facade** (SF) is usually inserted. For the same reason, there is usually also a **Data Facade**.

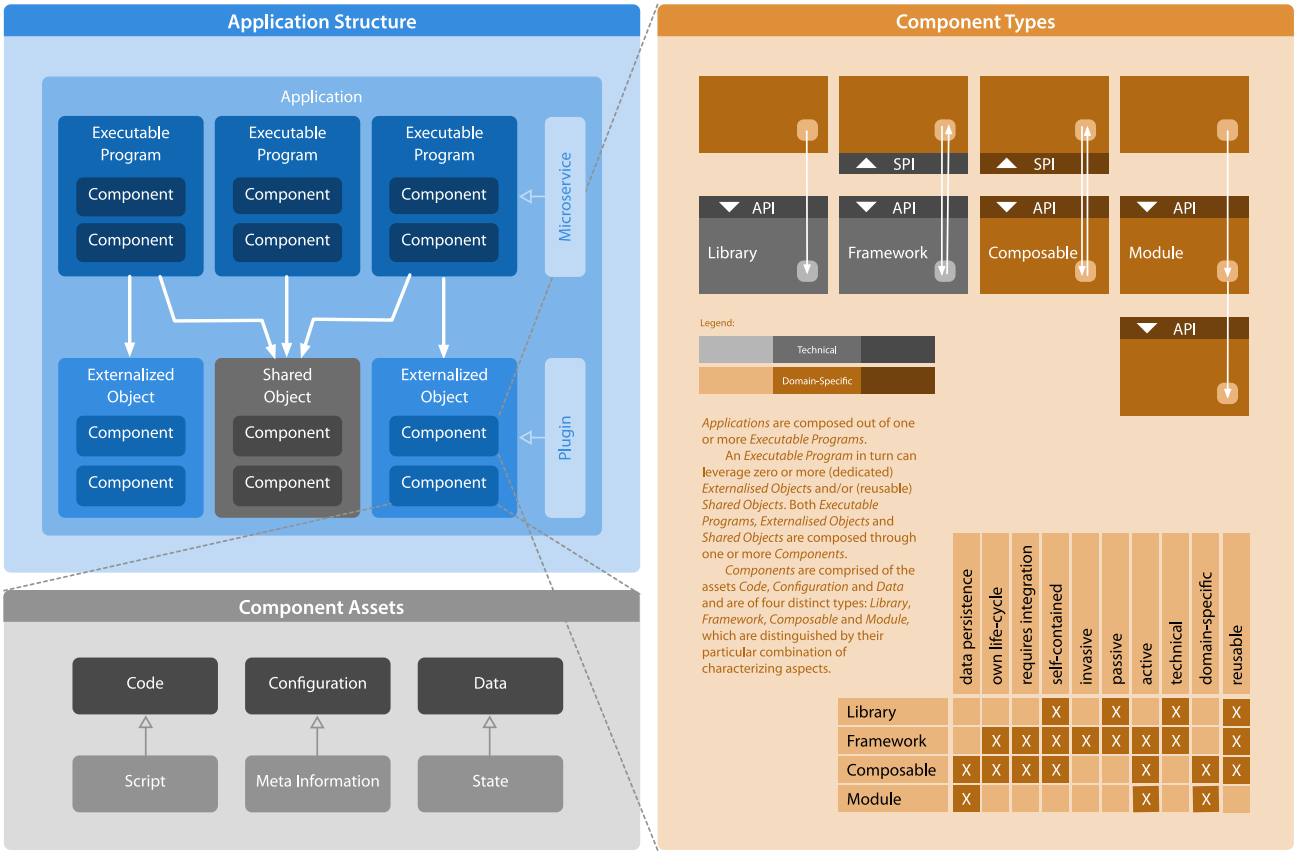
The Data Model is offloaded to common **Entity Components** (EC). Together with possibly shared code, both live in one **Common Slice**. Libraries and Plugins are also offloaded to separate slices, but there are two major differences: Libraries are passive and provide their functionality to the application via their interfaces. Plugins are active and control the application in that they hook into the application via Service Provider Interfaces (SPI) of the **Plugin Facades**.

In the application, there has to be only exactly one **Dependency Direction** so that the application (in the opposite direction of the dependencies) can be built cleanly. The reference architecture is usually also instantiated twice in order to design both a Rich Client and an associated "Thin Server" from it.

Questions

- With which layer pattern can in an Information System the **Interaction Components** be decoupled from the **Service Components**?
- In which order are the components of an application built?

Application Composition



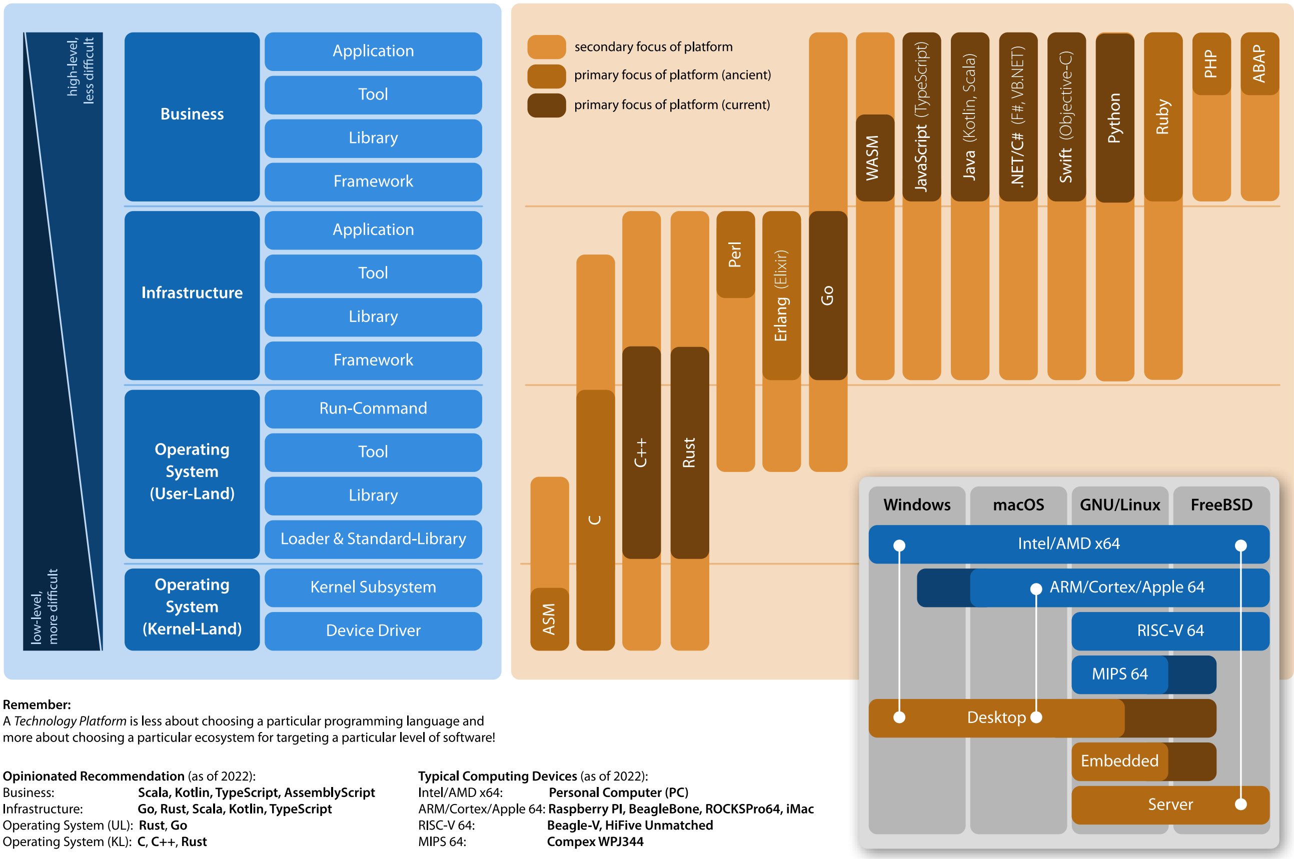
Applications are composed out of one or more Executable Programs. An Executable Program in turn can leverage zero or more (dedicated) Externalised Objects and/or (reusable) Shared Objects. Both Executable Programs, Externalised Objects and Shared Objects are composed through one or more Components. In a Microservice Architecture, the Executable Programs are called Microservices. In a Plugin Architecture, the Externalised Objects are called Plugins.

There are four distinct types of Components: Library, Framework, Composable and Module. They can be distinguished by their particular combination of characterizing aspects. Most prominently, whether they provide an Application Programming Interface (API) to the consumer of the Component and/or whether they require the consumer of the Component to fulfill some sort of Service Provider Interface (SPI).

Questions

- What is the main difference between a Library and a Framework?

Technology Platforms



Remember:
A *Technology Platform* is less about choosing a particular programming language and more about choosing a particular ecosystem for targeting a particular level of software!

Opinionated Recommendation (as of 2022):
Business: **Scala, Kotlin, TypeScript, AssemblyScript**
Infrastructure: **Go, Rust, Scala, Kotlin, TypeScript**
Operating System (UL): **Rust, Go**
Operating System (KL): **C, C++, Rust**

Typical Computing Devices (as of 2022):
Intel/AMD x64: **Personal Computer (PC)**
ARM/Cortex/Apple 64: **Raspberry PI, BeagleBone, ROCKSPro64, iMac**
RISC-V 64: **Beagle-V, HiFive Unmatched**
MIPS 64: **Compex WPJ344**

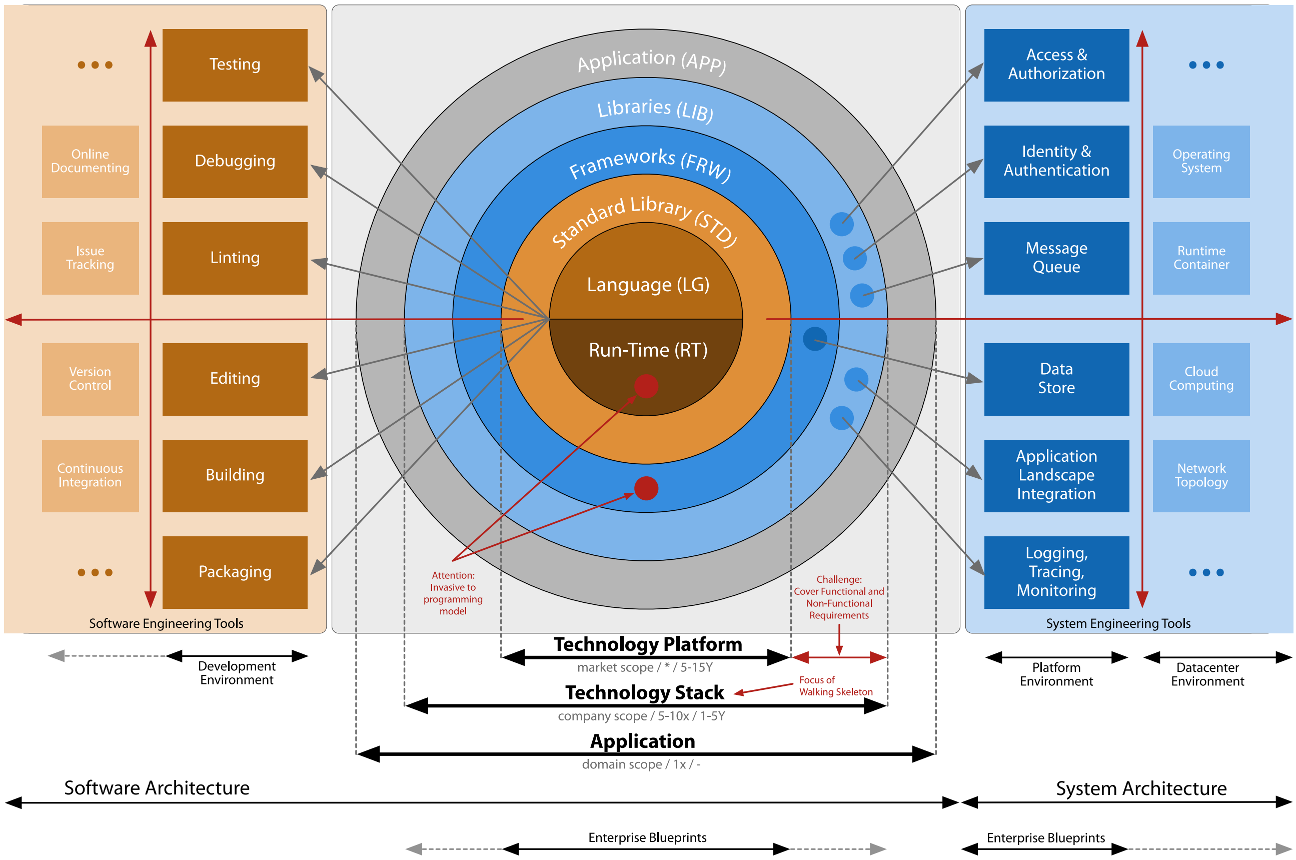
There are various levels of software, ranging from low-level and more difficult (operating system related) to high-level and less difficult (business-logic related).

A Technology Platform is less about choosing a particular programming language and more about choosing a particular ecosystem for targeting a particular kind of software!

Questions

❓ Is the Technology-Platform **Node.js** suitable to implement a Kernel-Subsystem at the level of the operating system?

Technology Stack



A **Technology Platform** consists of a **Language**, an optional **Run-Time** environment and a **Standard Library**. On top of this, **Frameworks** and **Libraries** extend this to a **Technology Stack**, in which with them especially all the prerequisites for the functional and non-functional requirements in the **Application** are achieved.

It has to be noted that the **Run-Time** and the **Frameworks** are usually extremely "invasive" to the programming model and thus can almost never be replaced afterwards. Therefore, with the so-called **Walking Skeleton**, the focus is mainly on the **Technology Stack** to be defined and integrated.

While the **Application** has a functional scope and is implemented only once, a particular **Technology Stack** is usually defined by a company and then reused several times over a period of a few years.

The underlying particular **Technology Platform**, on the other hand, is implemented by a third party for the market, is reused as often as required and must exist for quite a long period of time.

Large companies, therefore, usually stringently define the **Technology Platforms** and **Technology Stacks** in their **Enterprise Blueprints**.

For the **Software Engineering Tools** one should take into account the tools for **Testing**, **Debugging**, **Linting**, **Editing**, **Building** and **Packaging** of the **Development Environment**, because these are usually directly dependent on the particular **Technology Stack**.

The situation is similar for the **System Engineering Tools** of the **Platform Environment**: these require at least the associated **Libraries** in the **Technology Stack**, in order to be addressed during the run-time of the **Application**.

Questions

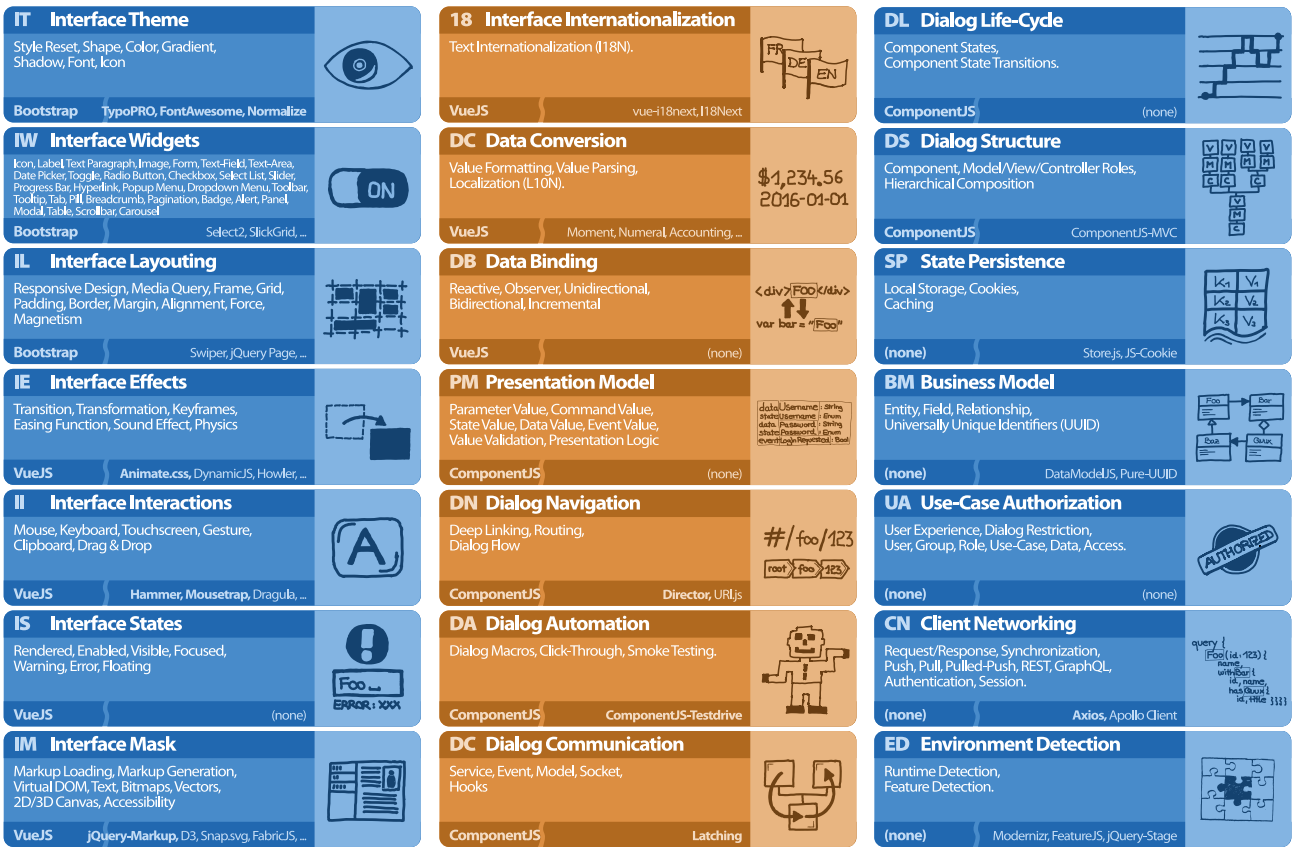
- ? What are the three components of a **Technology Platform**?
- ? Which two additional components make up a **Technology Stack**, compared to the **Technology Platform**?
- ? Which two components of a **Technology Stack** are most "invasive" to the programming model?

ARCHITECTURE FUNDAMENTALS

Rich-Client Aspects

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 12.5



To define a Technology Stack for a **Rich Client**, 21 **Aspects** have to be considered. Each aspect is covered by at least one **Framework** or **Library**. In practice, each aspect is usually covered by one Framework and zero or more Libraries. The goal always is: to achieve the greatest possible coverage of the aspects with a minimum number of Frameworks and Libraries.

It is advisable to use Open Source Software (OSS) for both Frameworks and Libraries and, if possible, to no own custom implementations, as the effort usually is not in proportion to the benefit. Because all aspects are technical — and not functional — aspects of a user interface.

In the case of a Thin-Client Architecture (instead of a Rich-Client Architecture), a few aspects like **Client Networking** and **Environment Detection** are omitted. All other aspects are still valid, even if, in the case of a Thin-Client Architecture, the frontend (and thus the aspects of the user interface) of the application runs on the server.

Two important aspects deal with the data model: the **Business Model** is a data model that comes directly from the server and is exactly the same in slicing and granularity than the business data model of the server. Its data is synchronized with a **Presentation Model**, which in slicing and granularity is more like the (more technical) data model of the user interface (especially via the aspects **Interface Mask** and **Data Binding**).

Questions

- How does one ensure in a **Rich-Client Architecture** that the numerous technical **Aspects** of a user interface are addressed?
- Which two **Aspects** of a **Rich-Client Architecture** hold the data model and take care of the fact that the data supplied by the server cannot be used directly in the user interface?

ARCHITECTURE FUNDAMENTALS

Thin-Server Aspects

TECHNISCHE UNIVERSITÄT MÜNCHEN

AF 12.6

ED Environment Detection
Detect the run-time environment, like underlying operating system, execution platform, network topology, feature toggles, etc.
Node | process, syspath

AP Argument Parsing
Parse options and arguments of the Command-Line Interface (CLI) to bootstrap application parameters.
(none) | yargs

CP Configuration Parsing
Load and parse directives from configuration file to bootstrap application parameters.
(none) | js-YAML

PD Process Daemonizing
Detach from the startup terminal and host process in order to run fully independently.
(none) | daemonize2

PM Process Management
(Pre-)fork child processes and/or threads of execution and monitor and control them during the life-cycle of the application.
(none) | cluster, nodemon

CM Component Management
Structure the code into components, instanciate them under run-time and manage them in a stateful component life-cycle.
Microkernel | (none)

CC Component Communication
Provide inter-component communication mechanisms like events, hooks, registry, etc.
Microkernel | Latching

SN Server Networking
Listen to network sockets, accept connections and manage request/response and message communication.
HAPI | hapi-plugin-websocket, ws

PI Peer Information
Determine unique identification and add-on information about the client peer.
HAPI | hapi-plugin-peer, geoip

SH Session Handling
Manage secured per-connection sessions to keep state between communication requests and/or client sessions.
HAPI | YAR

UA User Authentication
Determine and validate the unique identity of the user communicating over the current network connection.
HAPI | JWT, Passport

RV Request Validation
Validate the syntactical and semantical compliance of the requests and sanitize the requests.
HAPI | Joi, DuckyJS

RP Request Processing
Process the request by dispatching execution according to the provided request and determined context information.
HAPI | GraphQL.js

RA Role Authorization
Determine whether the role of the current user is allowed to execute the current request.
(none) | GraphQL-Tools-Sequelize

CN Client Networking
Provide mechanisms to connect to peers over the network and perform request/response and/or publish/subscribe communication.
(none) | Axios, MQTT.js, ws

TS Task Scheduling
Schedule and execute recurring tasks independent of regular I/O operations.
(none) | node-scheduler

ET Execution Tracing
Provide mechanisms for tracing the execution by logging event and measurement information at certain points of interest.
Microkernel | Winston

DA Database Access
Map in-memory domain entities onto data store dependent persistent data structure.
Sequelize | GraphQL-Tools-Sequelize

DC Database Connectivity
Locally or remotely connect the database access layer to the underlying data store.
Sequelize | sqlite3, pg

DS Database Schema
Create, update or downgrade the data schema inside the underlying data store.
Sequelize | (none)

DB Database Bootstrapping
Create, update or downgrade both mandatory bootstrapping and optional domain-specific data inside the underlying data store.
Sequelize | ini

To define a Technology Stack for a **(Thin-)Server**, 21 **Aspects** have to be considered. Each Aspect is covered by at least one **Framework** or **Library**. In practice, each Aspect is usually covered by one Framework and zero or more Libraries. The goal always is: to achieve the greatest possible coverage of the Aspects with a minimum number of Frameworks and Libraries.

It is advisable to use Open Source Software (OSS) for both Frameworks and Libraries and, if possible, to no own custom implementations, as the effort usually is not in proportion to the benefit. Because all Aspects are technical — and not functional — Aspects of a user interface.

It is to be noted that a server usually does not only have the Aspect **Server Networking** (for the connection of the Rich Clients), but also the Aspect **Client Networking**, in order to be able to query other servers.

In addition, it is to be noted that, above all, two important Aspects address security issues: the Aspect **User Authentication** identifies and authenticates the user ("Is the user the one?"). The Aspect **Role Authorization**, on the other hand, before all business processes, checks whether the authenticated user really is authorized to initiate the processes due to his role(s) ("Is the user allowed to do this?").

Questions

- Why does a **(Thin-)Server** usually have, besides the obvious aspect **Server Networking**, also the aspect **Client Networking**?
- Which Aspect of a **(Thin-)Server** takes care of the Question "Is the user the one"?
- Which aspect of a **(Thin-)Server** takes care of the question "Is the user allowed to do this"?