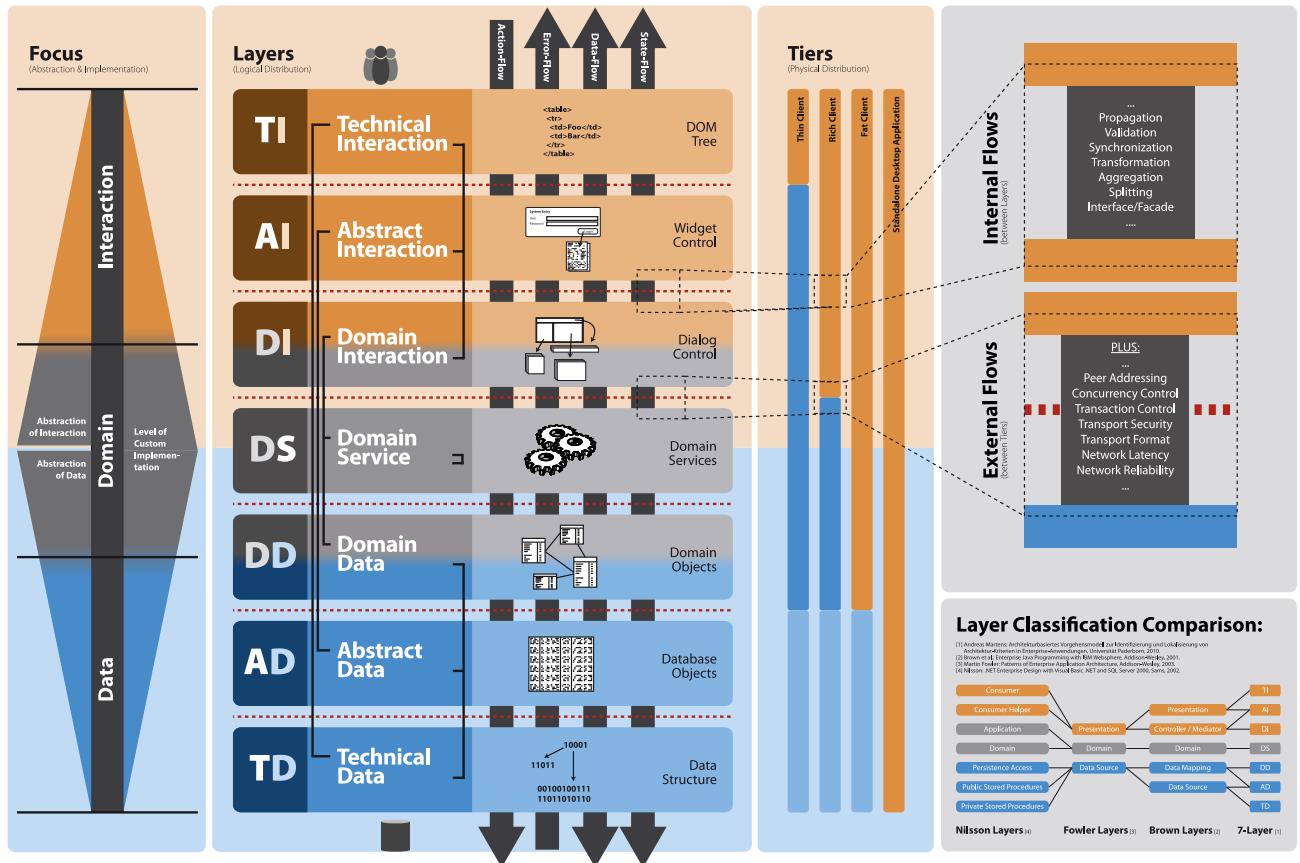




Software Engineering in der industriellen Praxis (SEIP)

Dr. Ralf S. Engelschall



In einer Anwendung kann man 7 logische Layer unterscheiden, die jeweils auf zwei Arten gruppiert sind: einerseits gibt es die drei sequenziellen Layer-Gruppen **Technical/Abstract/Domain Interaction**, **Domain Service** und **Domain/Abstract/Technical Data**, andererseits gibt es die drei geschachtelten Layer-Gruppen **Technical Interaction/Data**, **Abstract Interaction/Data** und **Domain Interaction/Service/Data**.

Zusätzlich kann man in einer Anwendung 4 primäre Flüsse unterscheiden: der **Action Flow** läuft konsequent nur von oben nach unten, da alle Aktionen oben vom Benutzer (oder Nachbarsystemen) abgestoßen werden; der **Error Flow** läuft konsequent nur entsprechend in der Gegenrichtung, also von unten nach oben, da Fehler im Worst-Case dem Benutzer gemeldet werden müssen; der (fachliche) **Data Flow** und der (technische) **State Flow** laufen hingegen in beiden Richtungen, da Daten und Zustände sowohl persistiert als auch angezeigt werden müssen.

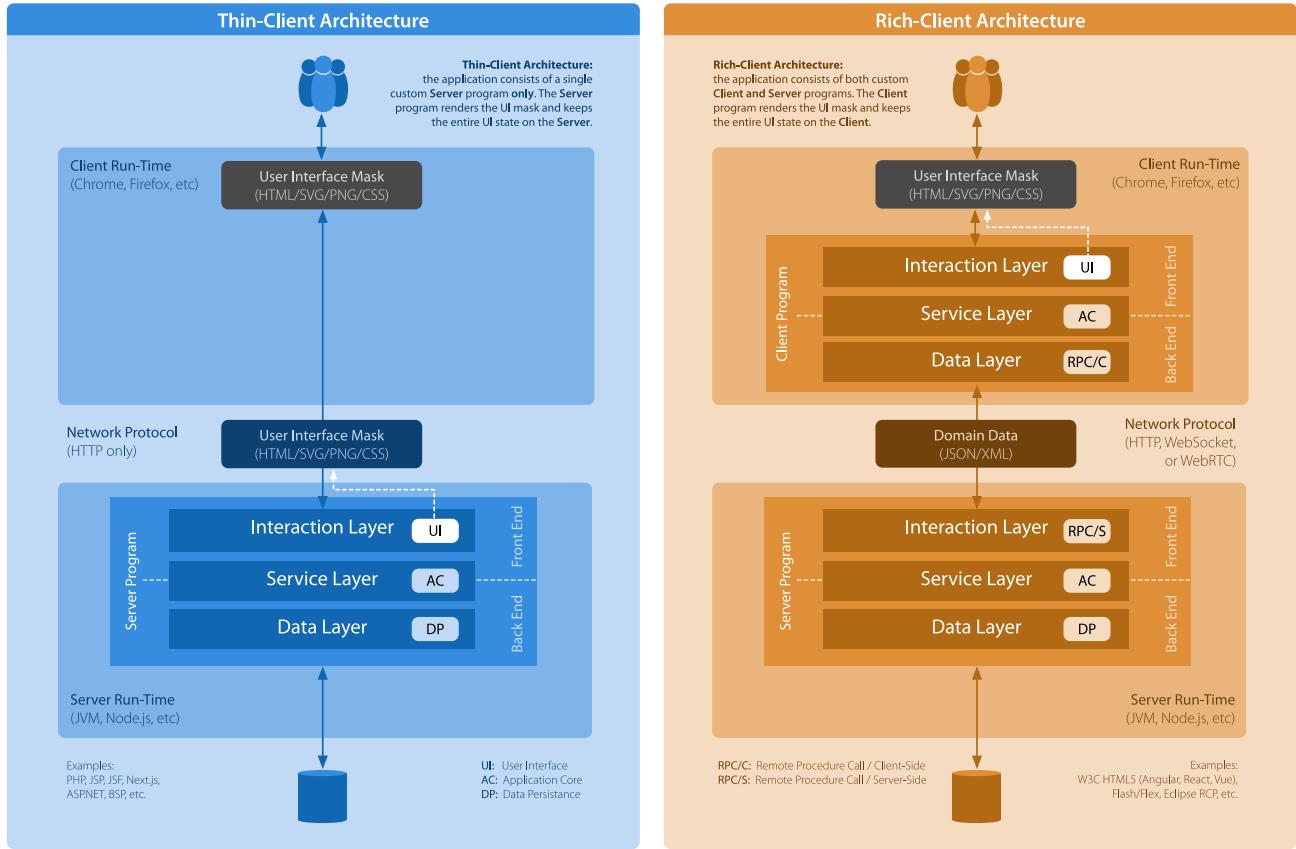
Die Abstraktion bei Interaction/Data in den Layern nimmt von oben/unten zur Mitte hin zu, weshalb auch dort der meiste fachliche Code einer Anwendung geschrieben wird. Für die oberen/unteren Layer wird üblicherweise massiv auf Open Source Libraries/Frameworks gesetzt.

Wenn man statt einem logischen Schnitt (der in einem **Internal Flow** zwischen den Layern resultiert) zwischen zwei Layern einen physikalischen Schnitt macht (der dann in einem **External Flow** resultiert), also die Anwendung in einzelne Programme auf unterschiedlichen Rechnern verteilt, so nennt man die daraus entstehende Architektur nach dem Umfang und der Verantwortung des Clients.

Bei **Thin Client** wird nur die **Technical Interaction** auf den Client ausgelagert, bei **Rich Client** wird die gesamte Benutzeroberfläche (also alle drei Layer **Technical/Abstract/Domain Interaction**) autonom auf den Client ausgelagert (üblicherweise als eine sog. "HTML5 Single-Page-Application"), bei **Fat Client** gibt es gar keinen zugehörigen Server mehr und bei der **Standalone Application** gibt nur noch ein einziges Programm.

Fragen

- ?
- Wie nennt man die Anwendungs-Architektur, bei der die gesamte Benutzeroberfläche autonom auf dem Client läuft, während der Server nur rein fachliche Services liefert?
- ?
- Wie heißen die Web-Anwendungen, welche eine **Rich Client** Architektur umsetzen?



Bei der **Thin-Client-Architektur** besteht die Anwendung aus nur einem einzigen individuellen Server-Programm. Dieses Server-Programm erzeugt die Maske der Benutzeroberfläche und hält den gesamten Zustand der Benutzeroberfläche auf dem Server.

Der Vorteil dieser Architektur ist, daß die Anwendung sehr leicht aktualisiert werden kann. Der Nachteil dieser Architektur ist, daß die Benutzeroberfläche träge reagiert und der Zustand der Benutzeroberflächen aller Clients auf dem Server gehalten werden muss, was den Server zum Flaschenhals werden lassen kann.

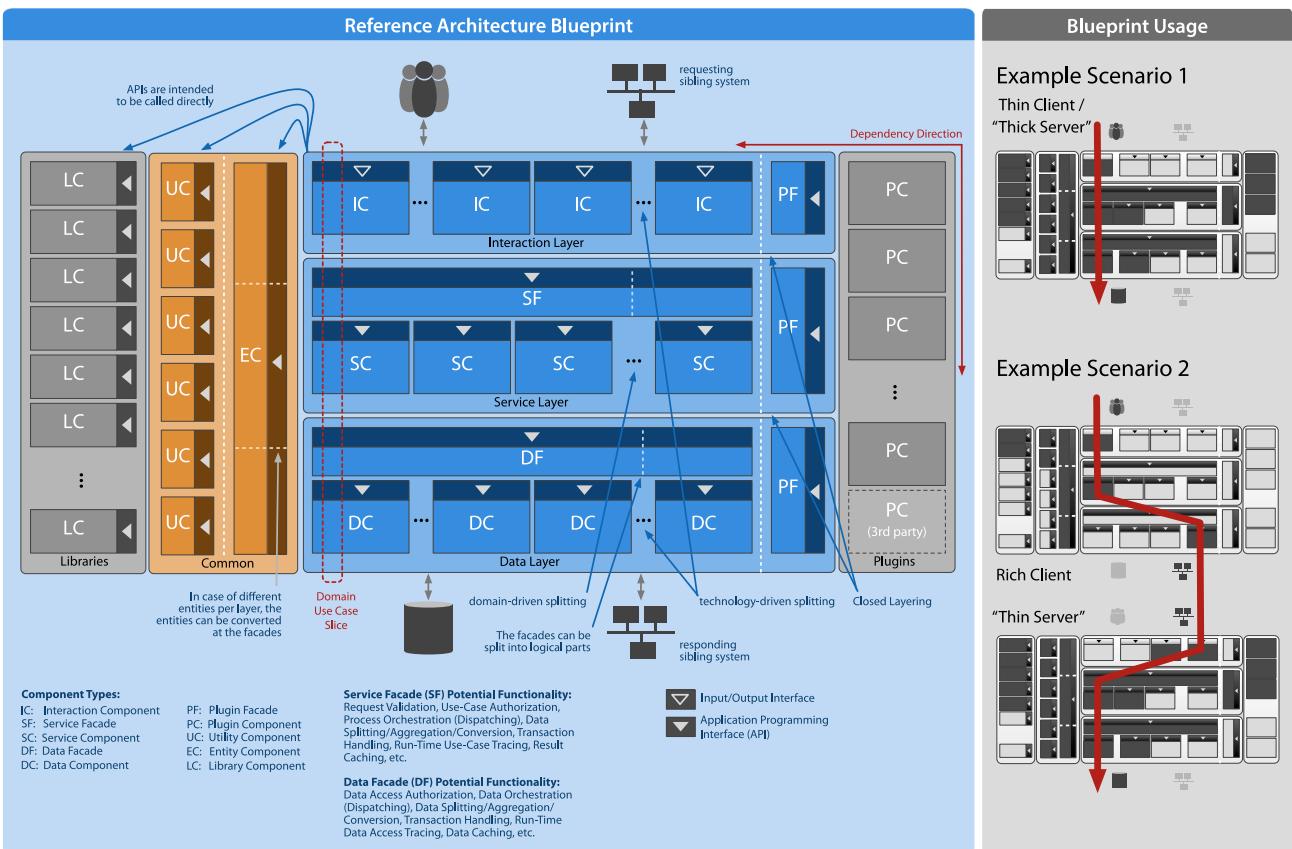
Bei der **Rich-Client-Architektur** besteht die Anwendung sowohl aus einem individuellen Client- als auch einem Server-Programm. Das Client-Programm erzeugt die Maske der Benutzeroberfläche und hält den gesamten Zustand der Benutzeroberfläche auf dem Client.

Der Vorteil dieser Architektur ist, daß die Benutzeroberfläche eine hohe Reaktionsfähigkeit zeigt, nur fachliche Daten zwischen Client und Server ausgetauscht werden müssen und der Server weniger stark zum Flaschenhals wird. Der Nachteil dieser Architektur ist, daß gegebenenfalls der Client explizit über eine Installationsprozedur aktualisiert werden muss.

Fragen

- ?

 - Bei welcher Client-Architektur bietet die Benutzeroberfläche die höhere Reaktionsgeschwindigkeit?



Ein (betriebliches) Informationssystem folgt üblicherweise einer stringenten Komponenten-basierten Referenz-Architektur. Diese wird "full blown" dargestellt und kann beliebig "abgespeckt" werden.

Zuerst besteht diese Referenz-Architektur aus 3 wesentlichen Layern: dem **Interaction Layer** mit den (technisch geschnittenen) Komponenten, welche die I/O-basierten Schnittstellen zum Benutzer (User Interface) und/oder anfragenden Nachbarsystemen (über Network Interface) bereitstellen, dem **Service Layer** mit den (fachlich geschnittenen) Service-Komponenten (auch Anwendungskern genannt) und dem **Data Layer** mit den (technisch geschnittenen) Komponenten, welche die Anbindung an die eigene Datenbank und/oder abzufragenden Nachbarsystemen realisieren.

Man beachte, daß die "Andock-Position" eines Nachbarsystems von seinen Rollen abhängt: wenn es anfrägt, dockt es am Interaction Layer an; wenn es abgefragt wird, dockt es am Data Layer an. Wenn es zufällig beide Rollen innehaben sollte, dockt es zweifach an. Die andere Sichtweise ist, daß sowohl der Benutzer als auch die Datenbank als spezielle "Nachbarsysteme" begriffen werden können.

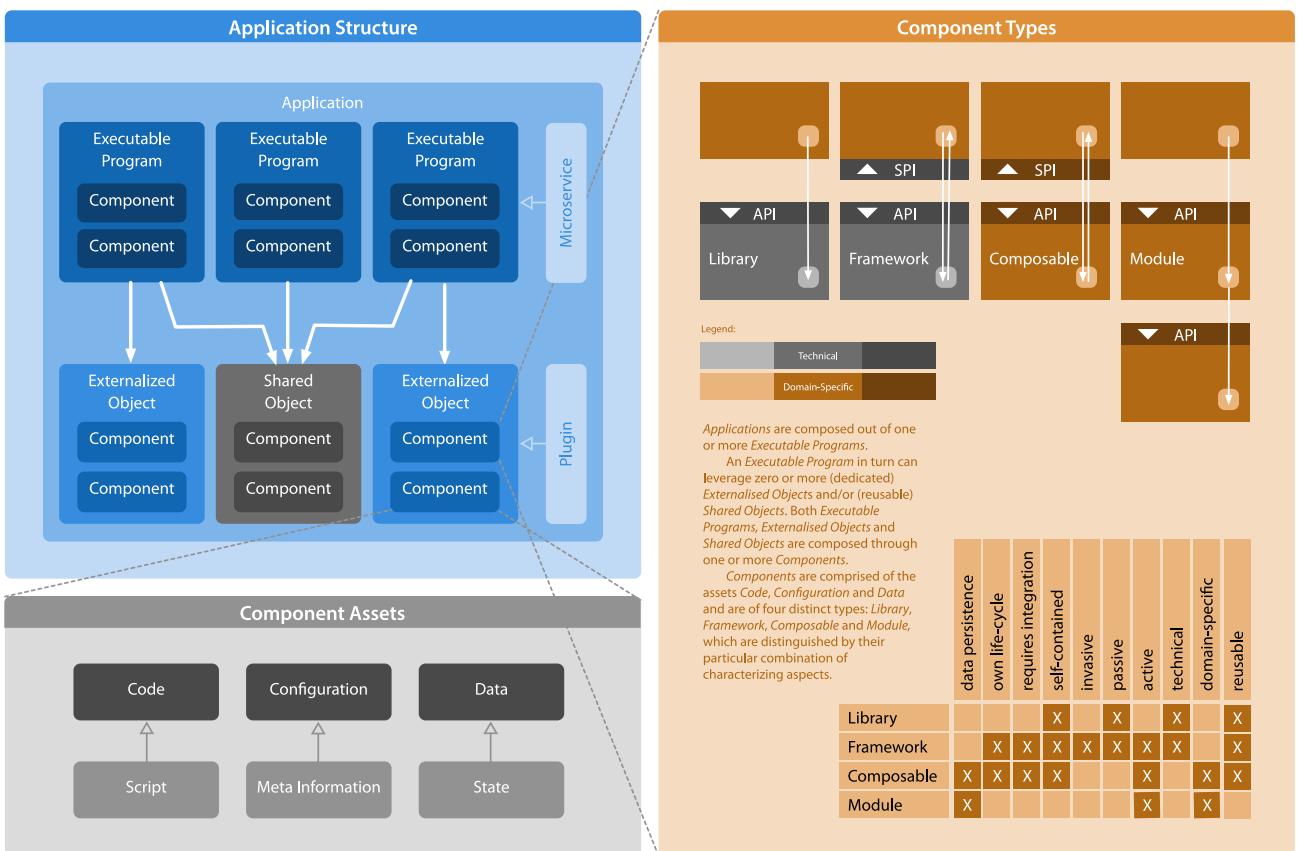
Um die **N Interaction Components** (IC) mit **M Service Components** (SC) zu verbinden, wird üblicherweise eine entkoppelnde **Service Facade** (SF) eingezogen. Aus gleichem Grund gibt es üblicherweise auch eine **Data Facade**.

Das Datenmodell wird in gemeinsame **Entity Components** (EC) ausgelagert. Zusammen mit ggf. gemeinsam genutztem Code lebt beides in einem **Common Slice**. Libraries und Plugins sind ebenfalls in eigene Slices ausgelagert, es gibt aber zwei wesentliche Unterschiede: Libraries sind passiv und bieten der Anwendung ihre Funktionalität über ihre Schnittstellen an. Plugins sind aktiv und steuern die Anwendung, in dem sie sich über Service-Provider Interfaces (SPI) in den **Plugin Facades** in die Anwendung einklinken.

In der Anwendung darf es nur genau eine **Dependency Direction** geben, damit die Anwendung (in der Gegenrichtung der Dependencies) sauber gebaut werden kann. Die Referenz-Architektur wird außerdem üblicherweise zweifach instanziert, um sowohl einen Rich-Client als auch einen zugehörigen "Thin-Server" daraus zu konstruieren.

Fragen

- ?
 - Mit welchem Layer-Pattern werden in einem Informationssystem die **Interaction Components** von den **Service Components** entkoppelt?
 - ?
 - In welcher Reihenfolge werden die Komponenten einer Anwendung gebaut?

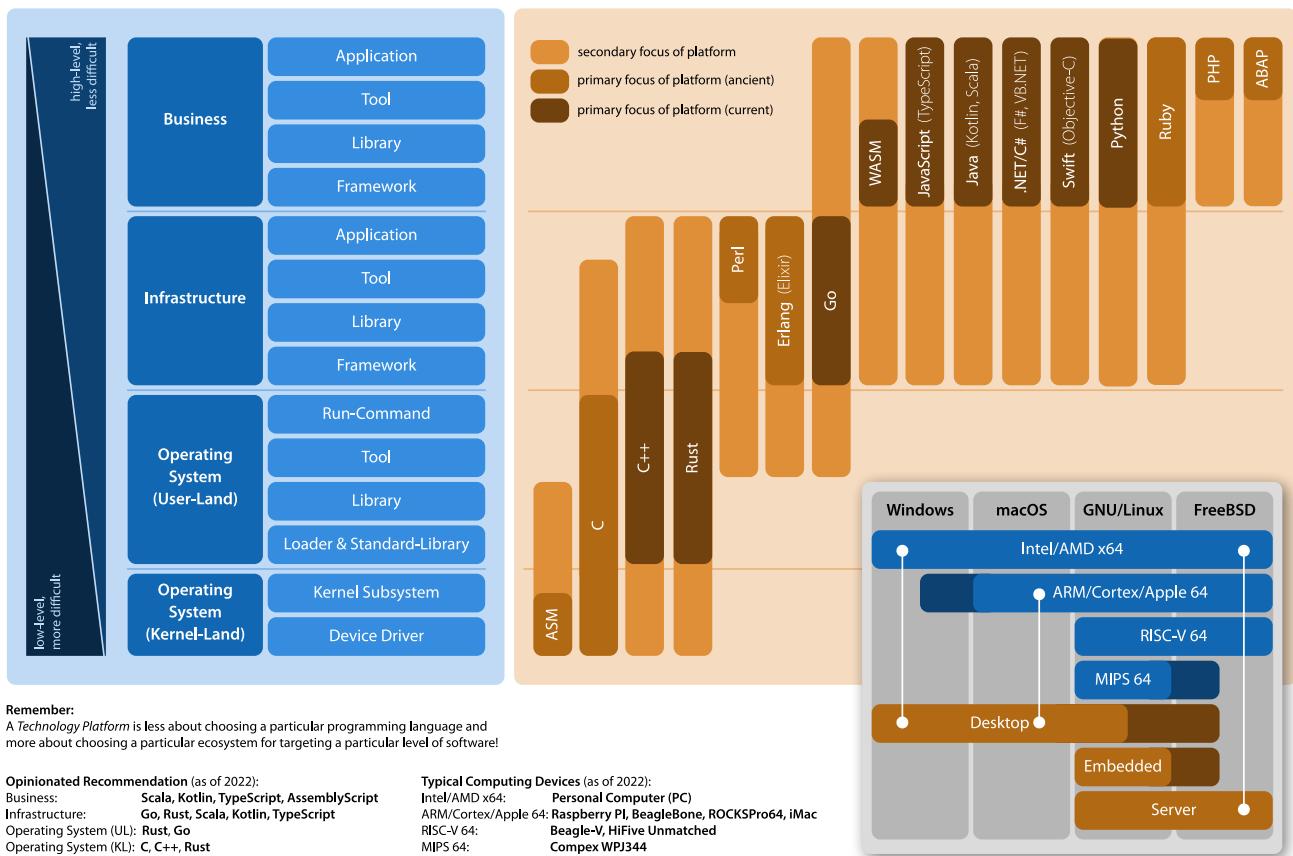


Anwendungen werden aus einem oder mehreren Executable Programs zusammengesetzt. Ein Executable Program kann seinerseits null oder mehrere (dedizierte) Externalized Objects und/oder (wiederverwendbare) Shared Objects nutzen. Sowohl Executable Programs, Externalized Objects und Shared Objects bestehen aus einer oder mehreren Components. In einer Microservice Architecture werden die Executable Programs als Microservices bezeichnet. In einer Plugin Architecture werden die Externalized Objects als Plugins bezeichnet.

Es gibt vier verschiedene Typen von Components: Library (Bibliothek), Framework, Composable und Module (Modul). Sie lassen sich durch ihre besondere Kombination von charakteristischen Aspekten unterscheiden. Am wichtigsten ist, ob sie ein Application Programming Interface (API) für den Konsumenten der Component bereitstellen und/oder ob sie verlangen, dass der Konsument der Component eine Art von Service Provider Interface (SPI) erfüllen muß.

Fragen

- ❓ Was ist der wesentliche Unterschied zwischen einer Library und einem Framework?



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

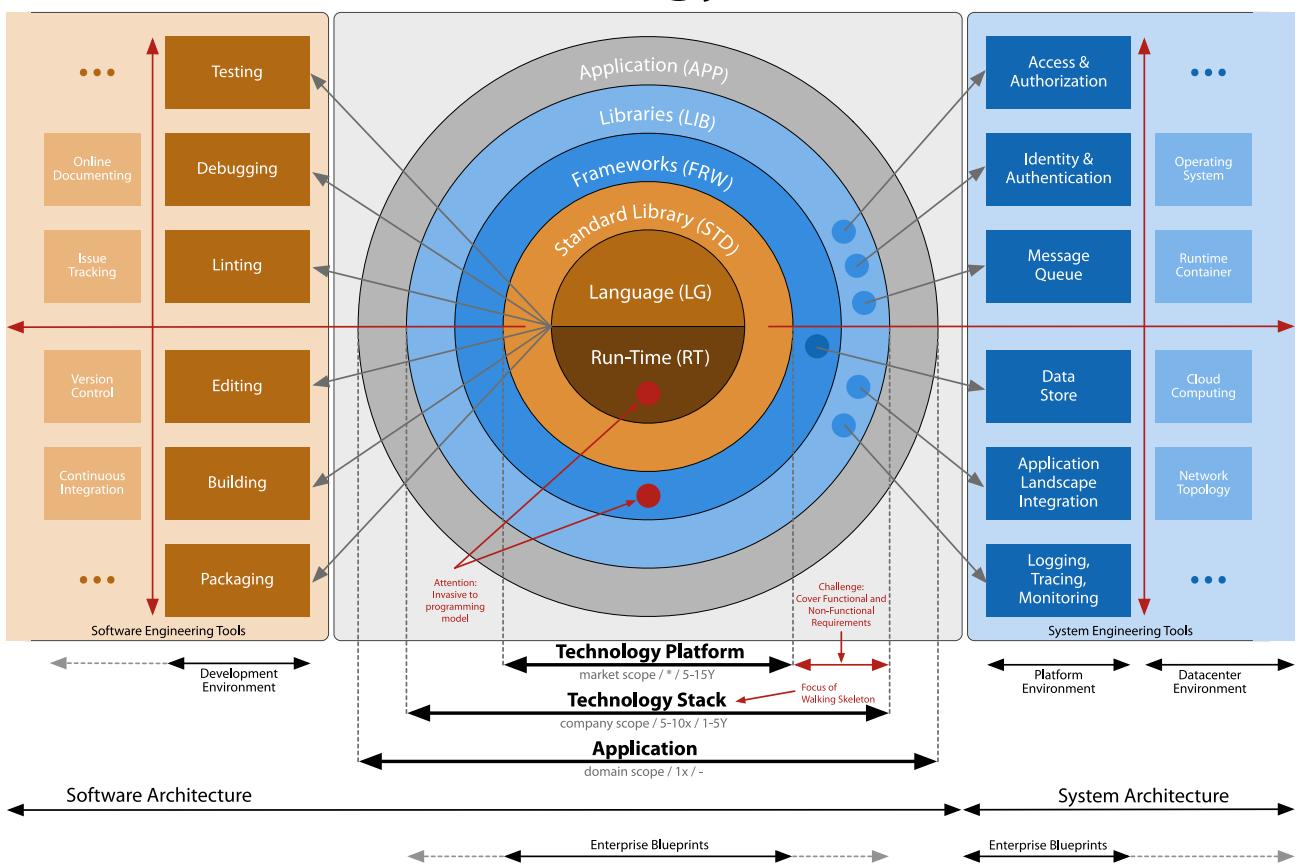
Es gibt verschiedene Ebenen von Software, von "low-level" und schwieriger (auf der Ebene des Betriebssystems) bis "high-level" und weniger schwierig (auf der Ebene von Business-Logik).

Bei einer Technologie-Plattform geht es weniger um die Auswahl einer bestimmten Programmiersprache, und mehr um die Auswahl eines bestimmten Ökosystems, um auf eine bestimmte Art von Software abzuzielen!

Fragen

?

Ist die Technologie-Plattform **Node.js** geeignet, um auf der Ebene des Betriebssystems ein Kernel-Subsystem zu implementieren?



Eine **Technology Platform** besteht aus einer **Language**, einer optionalen **Run-Time-Umgebung** und einer **Standard Library**. Darauf aufsetzend, erweitern **Frameworks** und **Libraries** dies zu einem **Technology Stack**, in dem mit ihnen vor allem die Voraussetzungen zum Erzielen der funktionalen und nicht-funktionalen Anforderungen in der **Application** geschaffen werden.

Es ist zu beachten, daß die **Run-Time** und die **Frameworks** üblicherweise extrem "invasiv" für das Programmiermodell sind und somit fast nie nachträglich ausgewechselt werden können. Deshalb wird beim sogenannten **Walking Skeleton** (deutsch "Technischer Durchstich") der Fokus vor allem auf den zu definierenden und zu integrierenden **Technology Stack** gelegt.

Während die **Application** einen fachlichen Geltungsbereich besitzt und nur ein Mal implementiert wird, wird ein konkreter **Technology Stack** üblicherweise von einer Firma definiert und dann mehrfach über einen Zeitraum von ein paar Jahren wiederverwendet.

Die unterliegende konkrete **Technology Platform** dagegen wird von einem Dritten für den Markt realisiert, wird beliebig oft wiederverwendet und muss für einen recht langen Zeitraum bestehen.

Große Firmen legen deshalb üblicherweise die zu nutzenden **Technology Platforms** und **Technology Stacks** in ihren **Enterprise Blueprints** stringent fest.

Bei den **Software Engineering Tools** sollte man bei der Definition eines Technology Stacks auch die Werkzeuge für **Testing, Debugging, Linting, Editing, Building und Packaging des Development Environments** mit berücksichtigen, denn diese sind üblicherweise direkt vom konkreten Technology Stack abhängig.

Ähnlich ist es bei den **System Engineering Tools** des **Platform Environments**: diese benötigen im **Technology Stack** mindestens zugehörige **Libraries**, um zu Laufzeit der **Application** angesprochen werden zu können.

Fragen

- ② Aus welchen drei Bestandteilen besteht eine **Technology Platform**?
 - ② Aus welchen zwei zusätzlichen Bestandteilen besteht ein **Technology Stack** gegenüber der **Technology Platform**?
 - ② Welche zwei Bestandteile eines **Technology Stack** sind am "invasiystem" für das Programmiermodell?

IT Interface Theme Style Reset, Shape, Color, Gradient, Shadow, Font, icon		I8 Interface Internationalization Text Internationalization (I18N). vue-i18next, i18nNext	DL Dialog Life-Cycle Component States, Component State Transitions.
IW Interface Widgets Icon, Label, Text, Paragraph, Image, Form, Text-Field, Text-Area, Date Picker, Toggle, Radio Button, Checkbox, Select List, Slider, Progress Bar, Hyperlink, Popup Menu, Dropdown Menu, Toolbar, Tooltip, Tab, Pill, Breadcrumb, Pagination, Badge, Alert, Panel, Modal, Table, Scrollbar, Carousel		DC Data Conversion Value Formatting, Value Parsing, Localization (L10N). \$1,234.56 2016-01-01	DS Dialog Structure Component, Model/View/Controller Roles, Hierarchical Composition
Bootstrap Bootstrap, TypoPRO, FontAwesome, Normalize		VueJS Moment, Numeral, Accounting, ...	ComponentJS (none)
IL Interface Layouting Responsive Design, Media Query, Frame, Grid, Padding, Border, Margin, Alignment, Force, Magnetism		DB Data Binding Reactive, Observer, Unidirectional, Bidirectional, Incremental <div>>FOO</div> var bar = "FOO";	SP State Persistence Local Storage, Cookies, Caching (none) Store.js, JS-Cookie
IE Interface Effects Transition, Transformation, Keyframes, Easing Function, Sound Effect, Physics		PM Presentation Model Parameter Value, Command Value, State Value, Data Value, Event Value, Value Validation, Presentation Logic data.username = "foo"; state.username = "bar"; data.password = "string"; state.password = "string"; event.click = "callback";	BM Business Model Entity, Field, Relationship, Universally Unique Identifiers (UUID) (none) DataModelJS, Pure-UUID
II Interface Interactions Mouse, Keyboard, Touchscreen, Gesture, Clipboard, Drag & Drop		DN Dialog Navigation Deep Linking, Routing, Dialog Flow #!/foo/123 rest > Foo > 123	UA Use-Case Authorization User Experience, Dialog Restriction, User, Group, Role, Use-Case, Data, Access. (none) (none)
IS Interface States Rendered, Enabled, Visible, Focused, Warning, Error, Floating		DA Dialog Automation Dialog Macros, Click-Through, Smoke Testing. 	CN Client Networking Request/Response, Synchronization, Push, Pull, Pulled-Push, REST, GraphQL, Authentication, Session. query { Foo(id: 123) { name, withoutId, withId, hasBlock { id, title } } }
IM Interface Mask Markup Loading, Markup Generation, Virtual DOM, Text, Bitmaps, Vectors, 2D/3D Canvas, Accessibility		DC Dialog Communication Service, Event, Model, Socket, Hooks 	ED Environment Detection Runtime Detection, Feature Detection. (none) Modernizr, FeatureJS, jQuery-Stage
VueJS Animate.css, DynamicJS, Howler, ...		ComponentJS Latching	
VueJS Hammer, Mousetrap, Dragula, ...		ComponentJS ComponentJS-Testdrive	

Um einen Technology Stack für einen **Rich-Client** zu definieren, müssen **21 Aspects** berücksichtigt werden. Jeder Aspect wird dabei mit mindestens einem **Framework** oder einer **Library** abgedeckt. In der Praxis wird üblicherweise jeder Aspect von einem Framework und null oder mehreren Libraries abgedeckt. Das Ziel ist immer: mit einer minimalen Anzahl an Frameworks und Libraries eine möglichst große Abdeckung der Aspects zu erzielen.

Es ist ratsam für sowohl Frameworks als auch Libraries Open Source Software (OSS) zu verwenden und nach Möglichkeit keinerlei Eigenimplementierungen, da üblicherweise sonst der Aufwand nicht im Verhältnis zum Nutzen steht. Denn bei allen Aspects handelt es sich um technische — und nicht fachliche — Aspects einer Benutzeroberfläche.

Im Falle einer Thin-Client Architecture (statt einer Rich-Client Architecture) fallen ein paar Aspects wie **Client Networking** und **Environment Detection** weg. Alle anderen Aspects sind aber weiterhin gültig, auch wenn im Fall einer Thin-Client Architecture das Frontend (und damit die Aspekte der Benutzerschnittstelle) der Application auf dem Server läuft.

Zwei wichtige Aspects behandeln das Datenmodell: das **Business Model** ist ein Datenmodell, welches direkt vom Server kommt und vom Schnitt und der Granularität exakt dem fachlichen Datenmodell des Servers entspricht. Dessen Daten werden mit einem **Presentation Model** synchronisiert, welches vom Schnitt und der Granularität exakt dem (eher technischen) Datenmodell der Benutzeroberfläche (vor allem über die Aspects **Interface Mask** und **Data Binding**) entspricht.

Fragen

- ?
- Wie sorgt man in einer **Rich-Client Architecture** dafür, daß die zahlreichen technischen **Aspects** einer Benutzeroberfläche adressiert werden?
- ?
- Welche zwei **Aspects** einer **Rich-Client Architecture** halten das Datenmodell und kümmern sich um die Tatsache, daß man die fachlichen Daten, wie sie vom Server geliefert werden, nicht direkt in der Benutzeroberfläche verwenden kann?

ED Environment Detection Detect the run-time environment, like underlying operating system, execution platform, network topology, feature toggles, etc.	SN Server Networking Listen to network sockets, accept connections and manage request/response and message communication.	CN Client Networking Provide mechanisms to connect to peers over the network and perform request/response and/or publish/subscribe communication.
Node process, syspath	HAPI hapi-plugin-websocket, ws	(none) Axios, MQTT, js, ws
AP Argument Parsing Parse options and arguments of the Command-Line Interface (CLI) to bootstrap application parameters.	PI Peer Information Determine unique identification and add-on information about the client peer.	TS Task Scheduling Schedule and execute recurring tasks independent of regular I/O operations.
(none) yargs	HAPI hapi-plugin-peer, geoip	(none) node-scheduler
CP Configuration Parsing Load and parse directives from configuration file to bootstrap application parameters.	SH Session Handling Manage secured per-connection sessions to keep state between communication requests and/or client sessions.	ET Execution Tracing Provide mechanisms for tracing the execution by logging event and measurement information at certain points of interest.
(none) js-YAML	HAPI YAR	Microkernel Winston
PD Process Daemonizing Detach from the startup terminal and host process in order to run fully independently.	UA User Authentication Determine and validate the unique identity of the user communicating over the current network connection.	DA Database Access Map in-memory domain entities onto data store dependent persistent data structure.
(none) daemonize2	HAPI JWT, Passport	Sequelize GraphQL-Tools-Sequelize
PM Process Management (Pre-)fork child processes and/or threads of execution and monitor and control them during the life-cycle of the application.	RV Request Validation Validate the syntactical and semantical compliance of the requests and sanitize the requests.	DC Database Connectivity Locally or remotely connect the database access layer to the underlying data store.
(none) cluster, nodemon	HAPI Joi, DuckyJS	Sequelize sqlite3, pg
CM Component Management Structure the code into components, instantiate them under run-time and manage them in a stateful component life-cycle.	RP Request Processing Process the request by dispatching execution according to the provided request and determined context information.	DS Database Schema Create, update or downgrade the data schema inside the underlying data store.
Microkernel (none)	HAPI GraphQL.js	Sequelize (none)
CC Component Communication Provide inter-component communication mechanisms like events, hooks, registry, etc.	RA Role Authorization Determine whether the role of the current user is allowed to execute the current request.	DB Database Bootstrapping Create, update or downgrade both mandatory bootstrapping and optional domain-specific data inside the underlying data store.
Microkernel Latching	(none) GraphQL-Tools-Sequelize	Sequelize ini

Um einen Technology Stack für einen (**Thin-Server**) zu definieren, müssen **21 Aspects** berücksichtigt werden. Jeder Aspect wird dabei mit mindestens einem **Framework** oder einer **Library** abgedeckt. In der Praxis wird üblicherweise jeder Aspect von einem Framework und null oder mehreren Libraries abgedeckt. Das Ziel ist immer: mit einer minimalen Anzahl an Frameworks und Libraries eine möglichst große Abdeckung der Aspects zu erzielen.

Es ist ratsam für sowohl Frameworks als auch Libraries Open Source Software (OSS) zu verwenden und nach Möglichkeit keinerlei Eigenimplementierungen, da üblicherweise sonst der Aufwand nicht im Verhältnis zum Nutzen steht. Denn bei allen Aspects handelt es sich um technische — und nicht fachliche — Aspects eines Servers.

Es ist zu beachten, daß ein Server üblicherweise nicht nur den Aspect **Server Networking** (für die Anbindung der Rich-Clients) besitzt, sondern auch den Aspect **Client Networking**, um selbst andere Server abfragen zu können.

Außerdem ist zu beachten, daß vor allem zwei wichtige Aspects den Themenkomplex Security adressieren: der Aspect **User Authentication** identifiziert und authentifiziert den Benutzer ("Ist der Benutzer derjenige?"). Der Aspect **Role Authorization** dagegen überprüft vor allen fachlichen Vorgängen, ob der authentizierte Benutzer aufgrund seiner Rolle(n) auch wirklich autorisiert ist, die Vorgänge auszulösen ("Darf der Benutzer das?").

Fragen

- ?
- Wieso besitzt ein (**Thin-Server**) üblicherweise neben dem offensichtlichen Aspect **Server Networking** auch den Aspect **Client Networking**?
- ?
- Welcher Aspect eines (**Thin-Server**) kümmert sich um die Frage "Ist der Benutzer derjenige"?
- ?
- Welcher Aspect eines (**Thin-Server**) kümmert sich um die Frage "Darf der Benutzer das"?