In this course unit [CM-W-WEB] a structured software development process of web applications is illustrated. The application development makes use of current software development concepts, such as Behavior-Driven Development (BDD), Domain-Driven Design (DDD), microservice architectures including a systematic design of the web Application Programming Interfaces (API) of the microservices.

(1) An overview of the relevant development concepts is given and the application development process based on these concepts is introduced. The core concepts and technologies are illustrated with the example of a web application.

(2) The development of advanced web application requires a complex tool environment. The organization of software development projects is supported by a specific set of project management and version control tools which are described in more detail in this chapter.

(3) This chapter deals with the analysis, the first phase of the development process, in which the requirements are specified. The approach uses Gherkin features which are the central artifact of the BDD concept.

(4) The domain model is the central artifact of the subsequent design phase. In the domain model which is built according to DDD principles, the (structural and functional) knowledge needed to implement the software system is formally captured. API specifications are derived from the domain model and the application architecture.

(5) In the implementation phase the functionality as is was specified in the analysis and design phase is coded based on a mircroservice architecture. The domain model is systematically transformed into code and the code is extended by annotations by which the microservice API is specified as a part of the code.

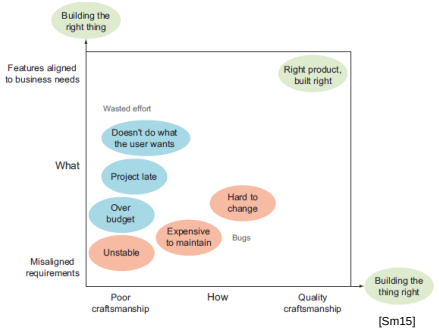
(6) The implemented web application is deployed in a container-virtualized infrastructure. Leading technologies used to build and run this infrastructure are Docker and Kubernetes.

C&M	Cooperation & Management
KIT	Karlsruhe Institute of Technology

[CM-W-WEB] Cooperation & Management: WEB APPLICATION DEVELOPMENT. WASA Course Unit. *https://team.kit.edu/sites/cm-tm/Mitglieder/2-2.WASA_Lecture*

OVERVIEW: (I) GENERAL CONCEPTS – Challenges in Software Development

(1) Software projects often fail
  (1) Main reasons: late delivery, running out of budget, missing features
  (2) More than 20 percent of the projects are entirely cancelled
(2) Development of features that
  (1) The user really needs
  (2) Are well-designed and well-implemented
(3) Behavior-Driven Development (BDD) provides methods and techniques to build the right software and to build the software right



[Sm15]

1 20.04.2021 WEB APPLICATION DEVELOPMENT: OVERVIEW Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

The motivation behind a clear analysis and design is to build and deliver better software systems [Sm15].

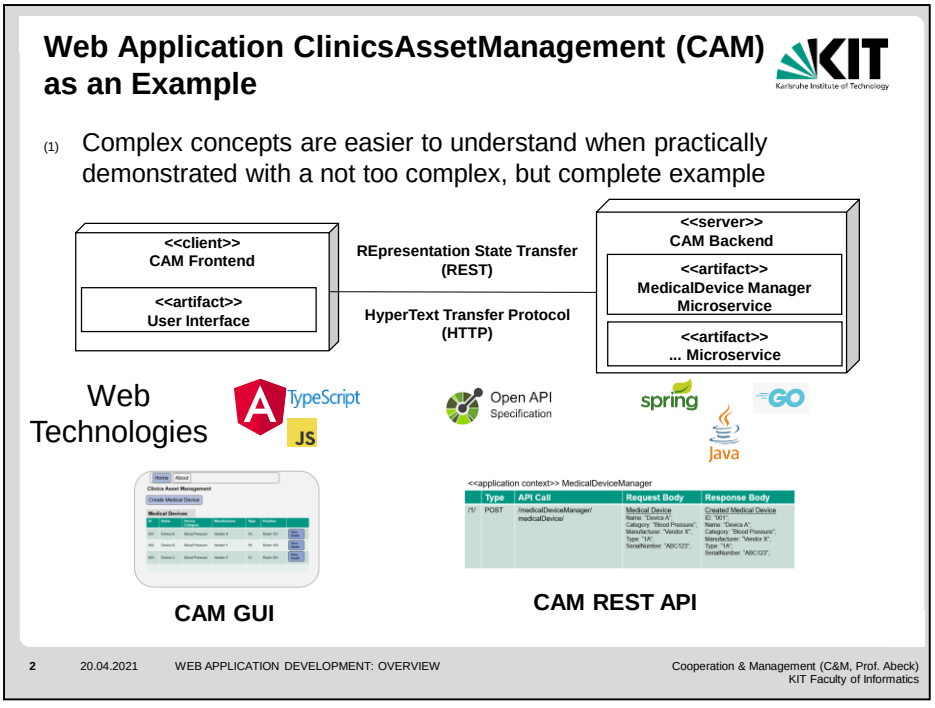
(1) According to a number of studies, nearly half of all software projects fail to deliver in some significant way.
(1.1) In a CHAOS Report published by the Standish Group, in 42 % of the software projects one or more of these problems occurred.
(1.2) This results in billions of dollars in wasted effort.

(2) The two goals center around the questions of WHAT to develop and HOW to develop.
(2.1) This goal describing the WHAT is shown by the vertical (y) axis "Building the right thing" in the figure.
(2.2) This goal describing the HOW is shown by the horizontal (x) axis "Building the thing right" in the figure.

(3) BDD encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand.

BDD Behavior-Driven Development

[Sm15] John Ferguson Smart: BDD in Action – Behavior-Driven Development for the whole software lifecycle. Manning Publications, 2015. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literatur/SoftwareEngineering



Throughout this course unit, the ClinicsAssetManagement (CAM) web application by which a user can manage medical devices in clinics, is used to demonstrate the technologies and methods of web application development. Further technical details of the application can be found in [CM-W-CAM]. All relevant (analysis, design, implementation and test, deployment and operations) artifacts related to the application can be found in the C&M GitLab [CM-G-0Doc-CAM]

(1) To be able to develop complex web applications, it is not only necessary to (theoretically) understand the concepts of microservice engineering, but also to (practically) apply them. By introducing the CAM example, the practical use of the concepts is demonstrated.

(<<client>> CAM Frontend, HTTP, <<server>> CAM Backend) Since CAM is a web application, it consists of a web client and a web server which communicate via the Internet application protocol HyperText Transfer Protocol (HTTP). The diagram describes the physical architecture of the CAM application and uses modeling elements specified by the Unified Modeling Language (UML). It is a so-called deployment diagram by which the physical systems (in this case, CAM Frontend and CAM Backend) are modeled as specific UML symbol called nodes.
Remark: In this specific example, all microservices from the CAM application are deploying (i.e. distributed) on one backend system. In general, each microservice can be deployed on a separate system which leads to many backend systems interconnected by a network.

(Web Technologies) Technologies play an important role in web application development. According to the physical architecture of a web application, frontend and backend technologies as well as technologies concerning the interface (Application Programming Interface, API) can be distinguished)
(A, TypeScript, JS) The "A" logo stands for Angular which is a leading frontend technology. Angular uses the script language TypeScript which is based on JavaScript (see JS logo).
(CAM GUI) This is a mockup of the CAM Graphical User Interface (GUI) which is implemented in Angular.
(Spring, Java, Go) These are examples of leading backend technologies used by C&M for the engineering of the microservices.
(OpenAPI Specification) The OpenAPI Specification is a manufacturer-independent defacto standard for the definition of REST APIs.

(CAM REST API) REST stands for Representational State Transfer which is a widely used concept for the specification of web APIs.
(POST /medicalDeviceManager/medicalDevice) This is an excerpt of the REST API which is provided by the MedicalDeviceManager microservice. Each REST operation is a specific HTTP method, such as GET or POST. For example, the REST operation "POST /medicalDeviceManager/medicalDevice" has a parameter which describes the medical device to be created.

API	Application Programming Interface
CAM	ClinicsAssetManagement
HTTP	HyperText Transfer Protocol
JS	JavaScript
REST	Representational State Transfer
UML	Unified Modeling Language

[CM-G-0Doc-CAM] Cooperation & Management: ClinicsAssetManagement, GitLab Repository. https://git.scc.kit.edu/cm-tm/cm-team/healthcare/clinicsassetmanagement/doc_cam/-/blob/master/README.md
[CM-W-CAM] Cooperation & Management: CAM DEVELOPMENT WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-3.WASA_Guidelines

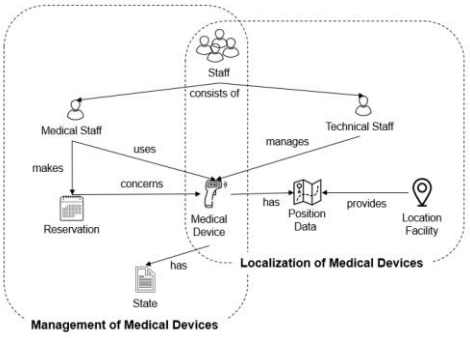
CAM Application Sketch

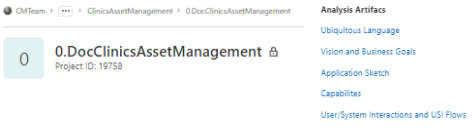
(1) An application sketch is an artifact created in the beginning of C&M's development process

(2) The CAM application sketch introduces the most relevant objects an subjects as well as their relationship

(3) All terms are part of a ubiquitous language

(4) All analysis and design artifacts of a software system developed by C&M are stored in a GitLab documentation repository





3 20.04.2021 WEB APPLICATION DEVELOPMENT: OVERVIEW Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

(1) The first phase of the C&M microservice engineering process is the analysis phase. Therefore, the application sketch is an analysis artifact.

(Staff Icon, Medical Device Icon, ...) C&M's icon collection builds the foundation of the icons appearing in an application sketch. According to the process description documented in [CM-T-ICO], every new icon is added to this collection.

(2) The functionality of the application is expressed by the relationships of the subjects and objects from the application.

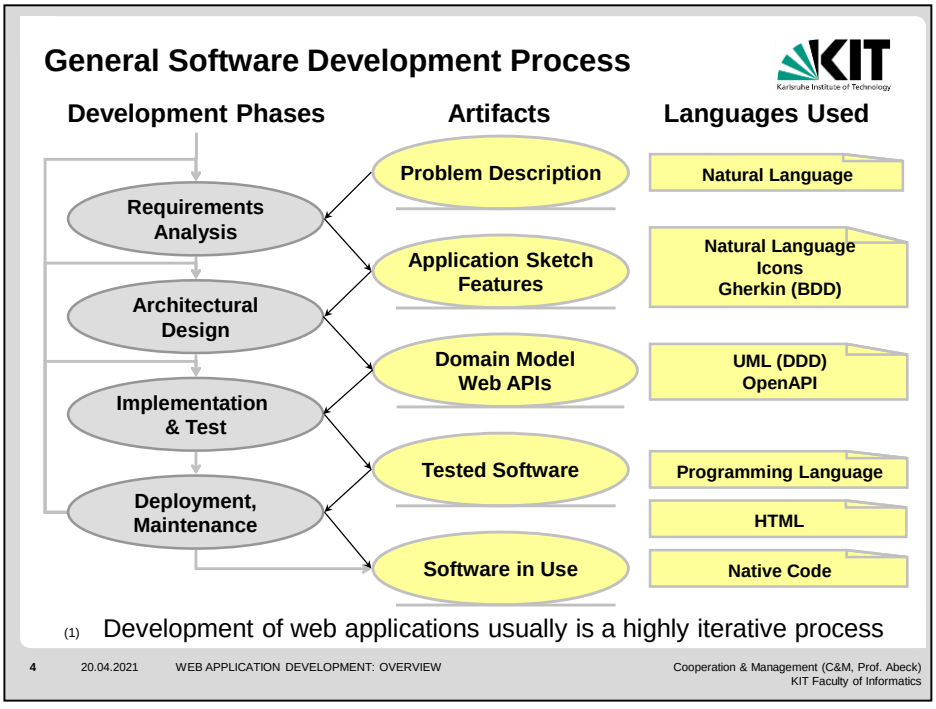
(Medical Staff, Technical Staff) They are part of a clinic. The technical staff manages the medical devices by capturing the description data of the medical devices which are used by the medical staff.
(Reservation) Before a medical device can be used, a reservation is necessary.
(Position Data, Location Facility)) A specific part of the description data of the medical devices is their current position which is provided by the IoT data management. For this, a location facility is needed which is able to locate the medical devices.

(3) All central terms appearing in the application sketch are part of a ubiquitous language. As will be explained later in detail, separate ubiquitous languages exist for each domain and each application.

(4) GitLab is a powerful tool which uses Git as version control system and organizes all files generated during the software development in so-called repositories. Besides the different types of repositories used for the code, a specific documentation repository exists for each software developed by C&M.

(0.ClinicsAssetManagement, Analysis Artifacts) The screen dumps are taken from the documentation repository of the CAM application [CM-G-0Doc-CAM]. Besides the application sketch, other analysis artifacts (e.g. ubiquitous language, vision and business goals, capabilities, ...) and all other artifacts (e.g. design) can be found in this repository.

[CM-G-0Doc-CAM] Cooperation & Management: ClinicsAssetManagement, GitLab Repository. https://git.scc.kit.edu/cm-tm/cm-team/healthcare/clinicsassetmanagement/doc_cam/-/blob/master/README.md
[CM-T-ICO] Cooperation & Management: C&M ICON Collection, C&M Team Document. https://team.kit.edu/sites/cm-tm/Mitglieder/3.FORSCHUNG_PROJEKTE/3.Icon_Collection



The figure introduces the different phases that have to be carried out to develop software systems, esp. web applications. Besides the programming language, natural language and several description languages are needed to develop a web application in a sound and systematic way.

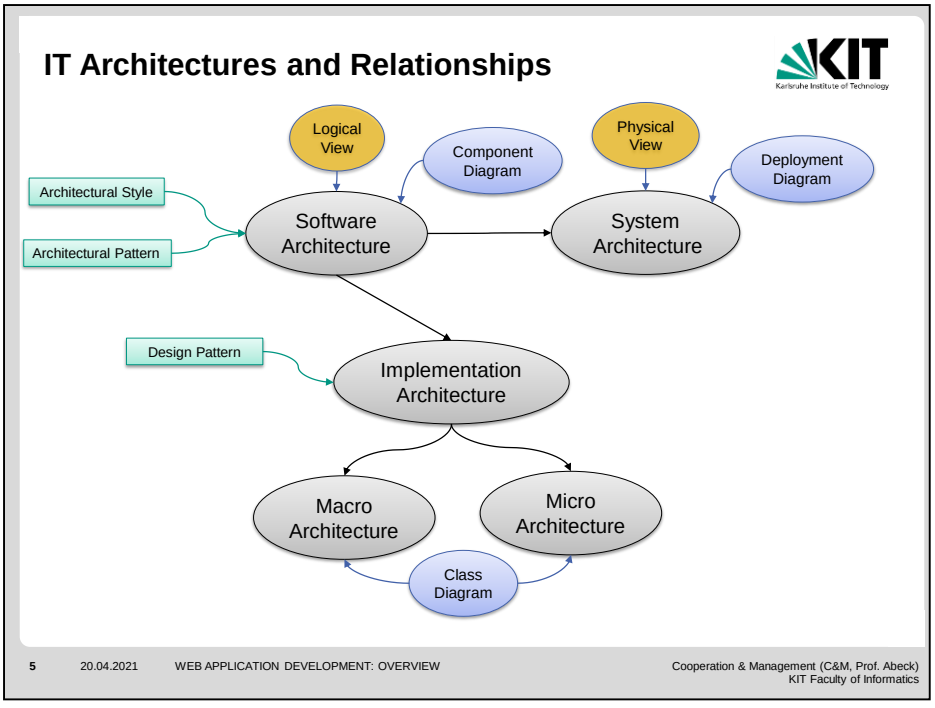
(1) Iterative in this context means that the phases are not completely passed in a sequential order but jumps to a the preceding phase are the usual case.

(Feature, Domain Model) Features are the main artifacts from Behavior-Driven Development whereas the domain model is the main artifact introduced by the Domain-Driven Design (DDD).

(Unified Modeling Language, eXtensible Markup Language, Hypertext Markup Language) are widely used description languages in the development process of web applications. Gherkin is a language which is used in the BDD approach.

(OpenAPI) This is a standardized language by which an API (Application Programming Interface) of a microservice based on REST (REpresentational State Transfer) can be specified.

API	Application Programming Interface
BDD	Behavior-Driven Development
DDD	Domain-Driven Design
REST	REpresentational State Transfer
UML	Unified Modeling Language
XML	eXtensible Markup Language



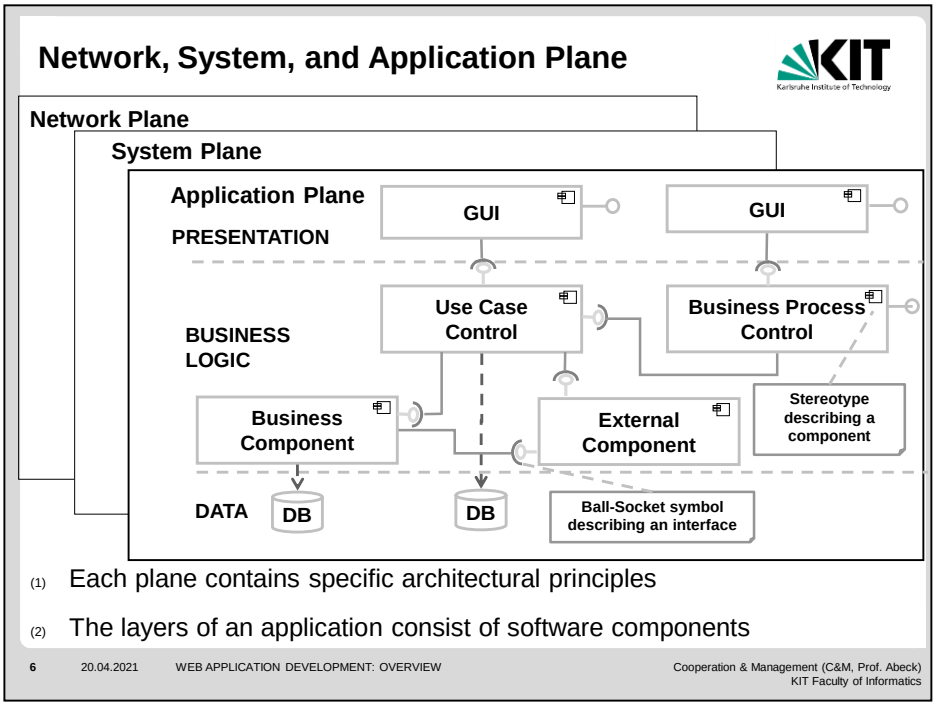
This figure illustrates the various architectural concepts and their relationships. It should be noted that no standardized definitions exist. For example, [CMU-SA] lists more than 30 definitions for software architecture.

(Software Architecture) The software architecture takes a logical view on the system by dividing it into logical components.
(System Architecture) The system architecture takes a physical view on a system and describes its structure which consists of network and hardware components. Furthermore, their properties and relationships to each other as well as to their environment and other systems are shown. A possible modeling type is the deployment diagram.
(Implementation Architecture) The implementation architecture describes the structure of the system from a technical point of view. This includes packages, libraries and frameworks.
(Macro Architecture, Micro Architecture) The macro architecture is used to describe the principles of the subsystems, e. g. micoservices or modules, on a black box level. The micro architecture deals with the internal structure of a single subsystem.

(Architectural Style) Architectural styles are application-independent solution principles that are used throughout the project. These can be assigned to the following categories: communication (e.g. message bus oriented), deployment (e.g. N-tier architecture) and structure (e.g. layer architecture).
(Design Pattern) Design patterns provide program code-specific patterns (e.g. visitor pattern) for recurring problems.
(Architectural Pattern) Architectural patterns are solutions to recurring problems which, unlike design patterns, affect several architectural elements.

(Component / Deployment / Class Diagram) Deployment View) The Unified Modeling Language (UML) provides different types of diagrams to specify the different types of architecture as models.

[CMU-SA] CMU: Software Architecture Getting Started Glossary Modern Software Architecture Definitions. *https://www.sei.cmu.edu/architecture/start/glossary/moderndefs.cfm*



(1) The fundamental principles of each plane (e.g. layering) are described in the following.

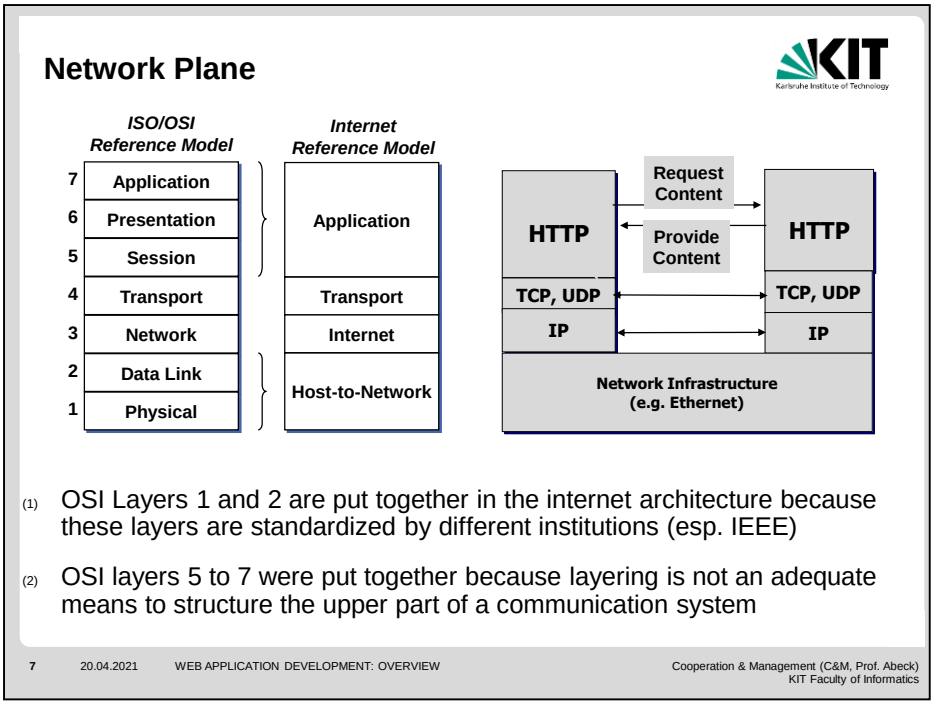
(Network Plane) Web applications run on several independent systems which heavily use Internet technologies for communication purposes. The network plane consists of communication layers. In each such layer the communication is organized by standardized protocols (e.g. IP, HTTP).

(System Plane) Internet protocols (such as the web protocol HTTP) are based on a client-server principle. The system plane is characterized by different operating systems leading to heterogeneous IT infrastructures.

(Application Plane) Web-based applications are located on the application plane. The term "web-based" especially means that the presentation layer is implemented by a standard web browser.
(DATA, BUSINESS LOGIC, PRESENTATION) The logical separation concerns the different (logical, conceptual, architectural) aspects a distributed application has to cover. There are three aspects – data, function (or business) and presentation – which can be found in every application. This architecture is called a three-layer architecture.

(2) The Unified Modeling Language (UML) is an adequate language to graphically describe the architecture of a software application. Modeling is necessary in order to develop the application in a structured and systematic way.
(Stereotype describing a component) A component in UML is modeled as a rectangle with a specific icon in the right upper corner. This icon is a graphical representation of a stereotype which is further defined in the UML meta model. An equivalent textual representation of this stereotype defining a component is <<component>>.
(Ball-Socket symbol describing an interface) The ball represents the provided part and the socket represents the required part of the component interface.

DB	DataBase
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
UML	Unified Modeling Language



There are so-called communication reference models that define the functionality of the layers that build the network plane. Two important models are shown on this page [KR05, Ta06].

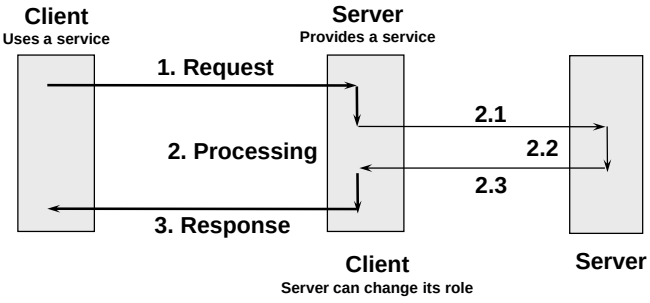
(HTTP, Request Content, Provide Content) The HyperText Transfer Protocol (HTTP) is an Internet application protocol which allows a HTTP client to request and also change content residing at an HTTP server by offering different HTTP operation, such as GET or POST.

(1) (2) The merging of the two lower layers in the Internet architecture has "organizational" reasons whereas the missing layering in the application system is conceptually motivated.

HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISO	International Standards Organization
OSI	Open Systems Interconnection
TCP	Transmission Control Protocol

[KR05] James F. Kurose, Keith W. Ross: Computer Networking – A Top-Down Approach Featuring the Internet, Pearson Education, 2005.
[Ta06] Andrew .S. Tanenbaum: Computer Networks, Prentice Hall, 2006.

System Plane



(1) "Client" and "server" are terms for roles the systems take in the process of distributed processing
  (1) During the processing of a request received from a client system a server might request one or more other server systems

(2) Many different types of servers do exist
  (1) Examples: file, database, print, application

On the system plane of a web application the involved systems take the roles of clients and servers.

(1) Processing is usually organized according to a client-server principle: The client asks the server to do some processing.

(2) Central tasks the server systems listed in the slide have to provide, are:
- File server: Upload and download of files
- Database server: SQL-based search queries
- Print server: offers functionality specific to the print service, such as queuing or prioritizing of print orders
- Application server: contains executable code specific to one or more applications

SQL Standard Query Language

EXERCISES GENERAL CONCEPTS

(1) What are the two main goals of software development?

(2) Of which physical systems a web application is built of and via which protocol do these systems communicate?

(3) Which web technologies are applied in CAM?

(4) What is the artifact name of
  (1) the graphical description of the main terms and their relationships?
  (2) the collection in which these terms are defined?

(5) Which views do the software architecture and the system architecture take on a software system and which diagram types does UML offer to describe these two vies on the architecture of a software system?

(6) Which are the planes of a distributed IT systems and what do they contain?

(1) ++Challenges in Software Development++
- Goal 1: To develop the right software (WHAT should be developed?)
- Goal 2: To develop the software right (HOW should the software be developed?)

(2) ++Web Application ClinicsAssetManagement (CAM) as an Example++
- Physical systems: Web client (frontend), web server (backend)
- HTTP (HyperText Transfer Protocol) which is an Internet application protocol.

(3) ++Web Application ClinicsAssetManagement (CAM) as an Example++
- Angular: A leading frontend technology
- TypeScript: Used by Angular and based on Java Script
- Spring, Java: Spring is one of the leading Java-based backend technologies.
- OpenAPI Specification: A manufacturer-independent de facto standard for the definition of REST APIs.

(4) ++CAM Application Sketch++
(4.1) Application sketch
(4.2) Ubiquitous language

(5) ++Types of Architectures and Relationships++
- Software architecture: logical view, UML component diagram
- System architecture: physical view, UML deployment diagram

(6) ++Network, System and Application Plane++
- Three planes: Network, System, Application
- Network: Internet communication protocols
- System: Client-server principle
- Application: Three layer application architecture (data, business logic, presentation)

OVERVIEW: (II) MICROSERVICE ARCHITECTURE – Chronological Overview

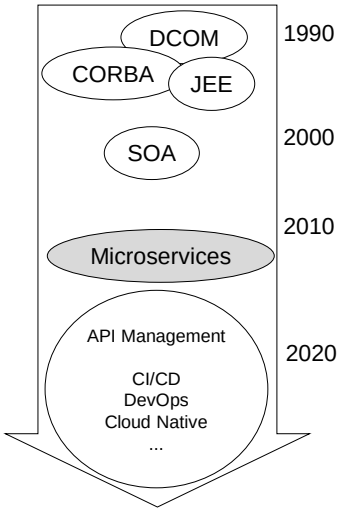
(1) The microservice architecture is located on the application plane

(2) The evolution started with distributed technologies (e.g. JEE) leading to the traditional three-layer architecture

(3) Followed by an evolution towards service orientation and the SOA (Service-Oriented Architecture)

(4) Microservices brought the breakthrough of the service-oriented architecture

(5) APIs and their professional management are going to change the trading of software



On the application plane the architecture of the distributed application is described.

(1) The microservice architecture today is the most relevant architecture to build advanced and maintainable web applications.

(2) In the decade from 1990 to 2000 technologies such as CORBA (from OMG), DCOM (from Microsoft) and JEE (from Sun) came up to support the development of distributed software systems.

(3) These technologies were followed by an evolution of the underlying software architecture which added a service layer as a sublayer of the business logic layer. This resulted in the Service-Oriented Architecture (SOA).

(4) Especially the web APIs based on the REST architectural principle have lead to many APIs that were provided by Internet companies, such as Facebook, Google, Twitter.

(5) APIs more and more become a business case where companies make money by providing (selling) them on a software market place. API management supports the publishing, optimization and control of APIs. An App developer is able to define and document the APIs being part of the developed software. API economy describes the way APIs can positively affect an organization's profitability. Today, products are available (e.g. Apigee) that allow to store and manage lightweight service description (e.g. based on the OpenAPI specification).

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
J2EE	Java 2 Enterprise Edition
REST	Representational State Transfer
SOA	Service-Oriented Architecture

The Who-is-Who in the Field of Microservices

(1) Work on what we today call microservices started about 2010 in IT industry, not in the academic world
  (1) Adrian Cockcroft: "A microservices architecture is a service-oriented architecture composed of loosely coupled elements that have bounded contexts" (-> Eric Evans, DDD)

(2) James Lewis was among the first who used the term in a publication
  (1) 2012: Micro services - Java, the Unix Way

(3) Martin Fowler and James Lewis were the first researchers who summarized the main characteristics of microservices
  (1) 2014: Microservices – a definition of this new architectural term

(4) Sam Newman published the first widely accepted and often cited textbook on microservices
  (1) 2015: Building Microservices: Designing Fine-grained Systems

(5) Observation: Microservices and related topics are highly relevant in the IT industry, but not yet a main topic in the academic world

(1) Netflix is an American media-services provider offering online streaming of a library of films and television programs.
Other companies include Amazon and Google.
(1.1) The citation is taken from [Ma15]. Bounded context is a central term of the Domain-Driven Design (DDD) from Eric Evans.

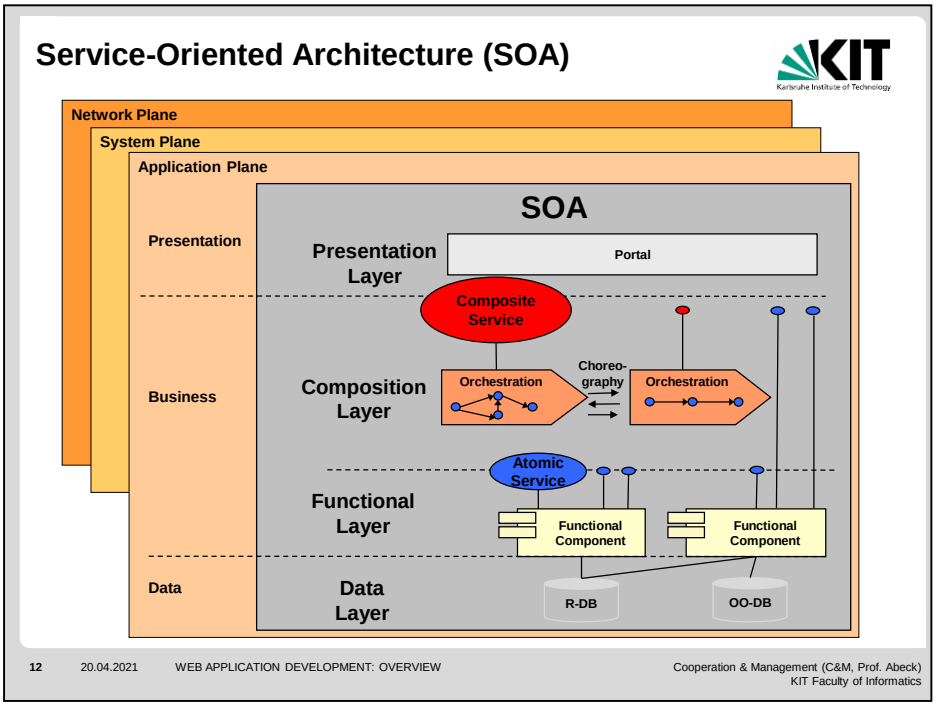
(2) Slides of the presentation [Le12] are available in the web.

(3) The article [LF15] is available in the web.

(4) Sam Newman is a technical consultant at ThoughtWork since 2004. His book [Ne15] was published by O'Reilly.

(5) Microservice engineering takes place in real projects.

[Le12] James Lewis: Micro services - Java, the Unix Way, March 2012, URL: http://2012.33degree.org/talk/show/67, [retrieved: 2019-04-09].
[LF15] James Lewis, Martin Fowler: Microservices, March 2014. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/2.MicroservicesAPIs
[Ma15] Tony Mauro: Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, [retrieved: 2019-04-13].
[Ne15] Sam Newman, Building Microservices: Designing Fine-grained Systems. O'Reilly Media, 2015. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/2.MicroservicesAPIs



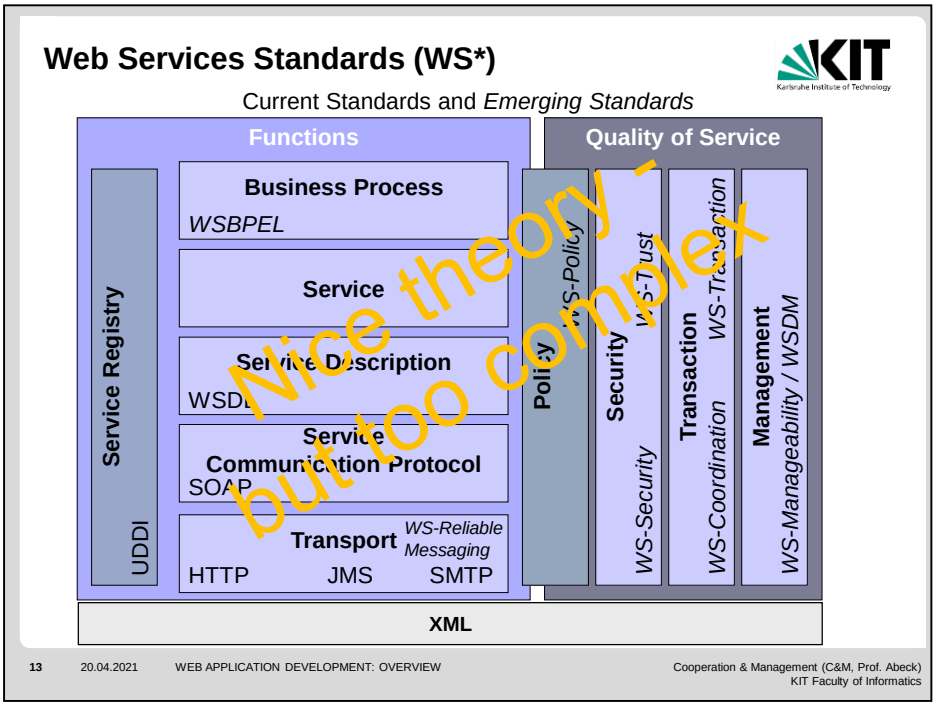
In an SOA four logical layers can be identified which are further described in the following.

(Data Layer) This layer includes the different types of databases, such as file systems or relational databases to store data permanently.

(Functional Layer) This layer is strongly related to the business layer of the traditional 3-layer application architecture. There is one important difference compared to the traditional business layer: The functional layer is implementing only business logic that should not be flexibly adapted according to changing business processes. The adaptable business logic is shifted from the "Programming in the Small" to the "Programming in the Large" as part of the Process Layer.

(Composition Layer) This layer can be seen as the innovative part of a SOA. Its main task is to compose atomic services derived from different components to build a composite service. If the involved components in the general case are running on different systems this layer fulfills an integration task.
A so-called orchestration is comprised of atomic services and directly supports a business process. Choreography on the other hand refers to the observable (public) interactions of (business) Web services. In other words: The choreography specifies how a client (user or machine) has to interact with the (composite or orchestrated) service to get a valid result. Furthermore, a choreography description can be used to define a multi-party contract that describes from a global point of view the externally observable behavior across multiple clients (which are generally Web services). External observable behavior is thereby represented by the messages that are exchanged between the multiple clients.

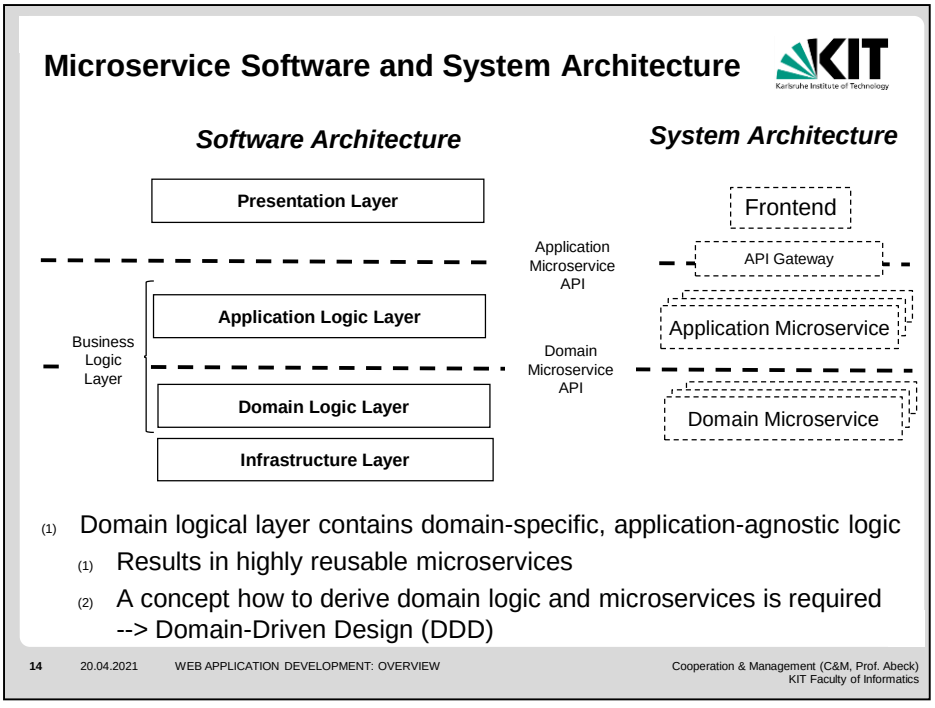
(Presentation Layer) This layer implements the user interface which in a service-oriented application is usually realized through a role-based Web portal.



Web Services is a particular set of standardized technologies – SOAP, WSDL, and others – that can be used to create a Service Oriented Architecture. Note that SOA can be implemented using Web Services technologies, but it does not necessarily require the use of Web Services technologies.

SOAP, WSDL, and other emerging standards from W3C and OASIS provide interoperability, platform-neutral security, reliability, business process description, transactionality, and more. The specifications can be used as required to implement requirements of the service.

(WS-ReliableMessaging) A protocol that allows messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures.
(WS-Policy) A base set of constructs that can be used and extended by other Web services specifications to describe a broad range of requirements, preferences, and capabilities of service interfaces
(WS-Security) describes how to attach signature and encryption headers to SOAP messages. In addition, it describes how to attach security tokens, including binary security tokens such as X.509 certificates and Kerberos tickets, to messages.
(WS-Trust) describes a framework for trust models that enables Web services to securely interoperate.
(WS-Privacy) introduces a model for how Web services and requesters state privacy preferences and organizational privacy practice statements.
(WS-Coordination, WS-Transaction) A set of Web service interface definitions and protocols that support participant control and agreement on the outcome of distributed, multi-party interactions.
(WS-Manageability) is defined as a set of capabilities for discovering the existence, availability, health, performance, and usage, as well as the control and configuration of a Web service within the Web services architecture. It is developed by the WSDM (Web Services Distributed Management) Technical Committee.



A microservice architecture is located on the application plane as it is introduced in ++Network, System, and Application Plane++. While the software architecture is described by the logical layers specified by a specific DDD pattern LAYERED ARCHITECTURE, the system architecture introduces several subystems (domain microservices, application micoservices, API gateway. Two types of application programming interfaces (domain microservice API, application microservice API) are separating the logical layers on the software architecture side and the microservices on the system architecture side.

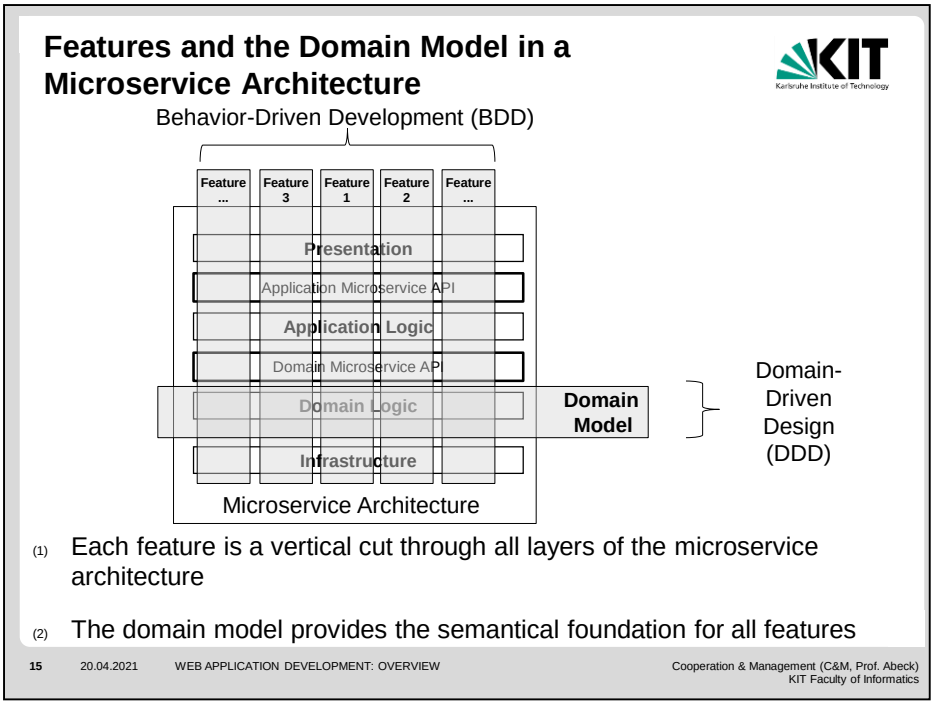
(Presentation Layer) This layer renders the UI elements in the browser. Technologies that support the implementation are Angular and Bootstrap. The presentation includes a logic which controls the interaction with the application microservice API. An optional API gateway is often used to provide cross-cutting concerns, such as load balancing or security aspects.

(Application Logic Layer, Application Microservice) This layer realizes the orchestration of domain microservices in order to provide the application logic to fulfill the requirements made to the application. A technology that supports the implementation of this functionality is Spring.

(Domain Logic Layer, Domain Microservice) This layer implement the domain microservices which mainly are CRUD operations on the domain objects.

(1) In contrast to a traditional three-layer application architecture, the business logic layer in a microservice architecture is split into two layers, the domain logic layer and the application logic layer. The reason for that is to promote the reuse of business logic functionality by distinguishing between application-agnostic (= domain logic) and application-specific (= application logic) functionality.

API	Application Programming Interface
CRUD	Create, Read, Update, Delete

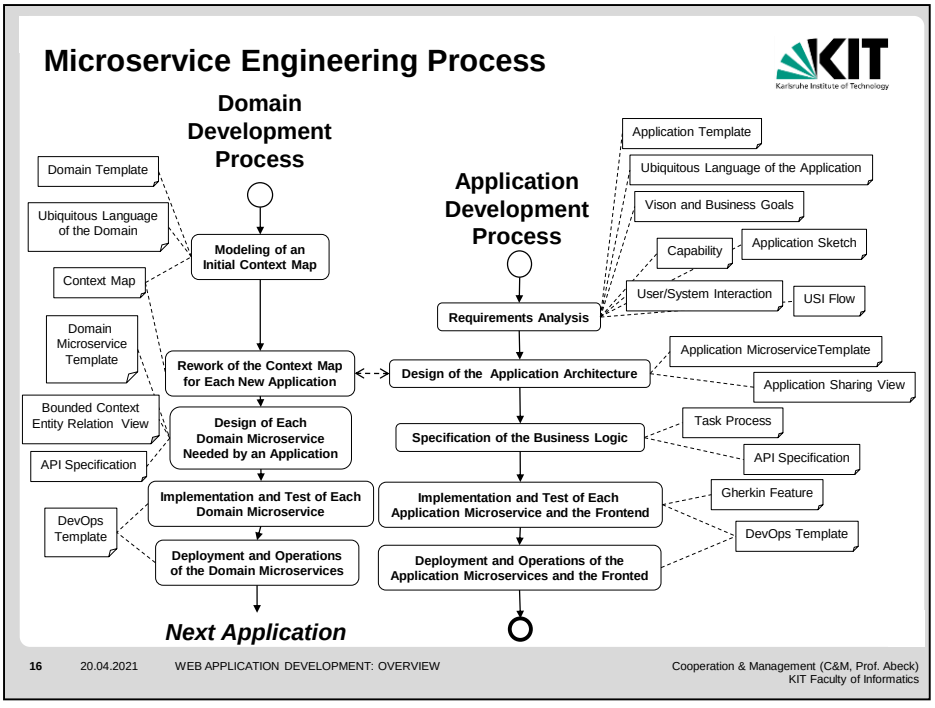


The software development process applied by C&M combines the concepts of Behavior-Driven Development (BDD) an Domain-Driven Design (DDD). Both concepts provide complementary contributions to the layered microservice architecture as the figure illustrates.

(1) An implemented feature can be seen as a deployable increment of the software system. (Feature1, Feature 2, ...) The ordering of the features in the figure implies that the first feature should cover the core functionality of the software system.

(2) The domain model makes sure that the static and dynamic domain knowledge is consistently used by each feature. This ensures that the features build a consistent whole although each feature is developed and deployed independently from other features.

BDD	Behavior-Driven Development
DDD	Domain-Driven Design



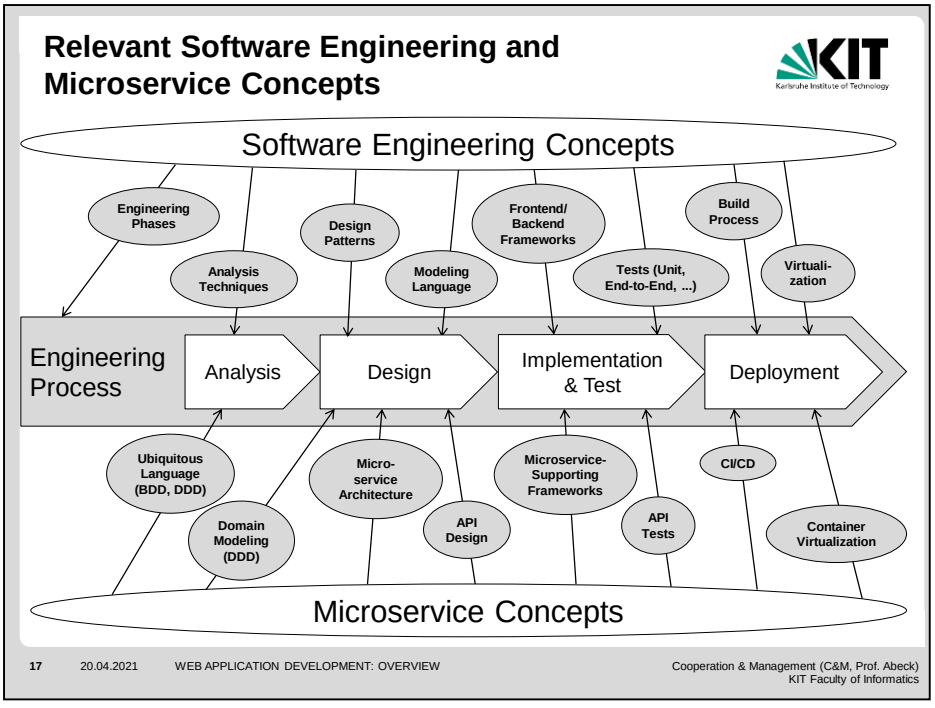
The activity diagram describes two (sub-) processes of the microservice engineering process: (i) the domain development process by which the application-agnostic domain knowledge is modeled as a context map and by which each bounded context being part of the context map is implemented as a domain microservice, (ii) the application development by which the microservice-based application is developed based on the domain's context map and domain microservices.

(Modeling of an Initial Context Map) The domain development process starts earlier than the application development process since the context map and also the ubiquitous language of the domain are essential for the analysis and design of the application.
(Ubiquitous Language of the Domain) (Ubiquitous Language Of the Application) Since the ubiquitous language of the application makes references to the the ubiquitous language of domain, the ubiquitous language of the domain must exist before the application development process can start.
(Context Map) The main artifact of the domain development process is the context map which contains the application-agnostic domain logic of the domain.
(Design of Each Domain Microservice Needed by an Application) Each bounded context is a candidate for a domain microservice. In the C&M approach, a domain microservice is implemented only when an application needs this domain microservice.

(Capability, User/System Interaction, USI Flow) A capability defines a self-contained and coherent functionality of the application from the business point of view. From each capability a number (about 3 to 6) system/user interactions are derived. Each user/system interaction is described in a step-by-step format where each step defines an action carried out by the user or by the system. A USI flow is a visual representation of a system/user interaction. They have the purpose to clarify the sequence of steps and the UI elements (e.g. text fields, input fields, buttons) which are needed for the user/system interaction. USI flows are optional artifacts and they should only be modeled when the cost-benefit ratio is justified.
(Application Sharing View) This view contains for each application context those bounded contexts from the domain which provide the domain-specific and application-agnostic logic of the application. Each capability results in an application context.
(Task Process) A task process describes the service orchestration of an application context. Each service call is modeled as task (in analogy to the Business Process Execution Language BPEL).
(API Specification) The API of a domain microservice is derived from the domain entity view, whereas the API of an application microservices is derived from the task processes. The APIs of both types of microservices are described by using the OpenAPI standard.
(Gherkin Feature) The Gherkin features are only used for testing purposes. Additionally, they may support a BDD/test-driven development of the application microservices.

(Domain / Domain Microservice / Application / Application Microservice /DevOps Template) These templates are made available in the C&M GitLab in the subgroup CMTeam > 1.DOCUMENTATION > 1.Templates. The templates make sure that that each development project run by C&M provide all artifacts in the same GitLab structure.

API	Application Programming Interface
BDD	Behavior-Driven Development
BPEL	Business Process Execution Language
DevOps	Development and Operations
UI	User Interface
USI	User/System Interaction



The figure shows that concepts specific to the microservice field do not replace, but complement, existing software engineering concepts.

(Analysis Techniques) In the first phase of the engineering process, the analysis phase, different technique to elicit and specify the requirements exist.

(Ubiquitous Language (BDD, DDD)) In the process we apply for the development of microservice-based software, the requirements are specified by Gherkin features which are part of Behavior-Driven Development (BDD). There is a match between BDD and mircroservices since BDD provides a ubiquitous language which is needed to design the APIs of a microservice.

(Domain Modeling (DDD)) BDD fits well to the the Domain-Driven Design (DDD) which we estimate to be one of the most relevant concept for the design of microservices.
(Design Patterns) DDD is a set of design patterns which are a well-known engineering concept to capture the knowledge of the structure (architecture, code) of a software system.
(Microservice Architecture) One of these DDD patterns, called Layered Architecture, provides the foundation of the microservice architecture. By the Layered Architecture pattern, the functional layer from the well-known three-layer application architecture is divided into a domain logic layer and an application logic layer. The functionality captured in the domain logic layer has to be application agnostic. This is exactly the functionality a microservice API should provide. By using the DDD patterns, a domain model of the domain, the software system belongs to, is built.

(API Design) The domain model supports both the implementation of the microservice and the design of the web API.

(Frontend/Backend Frameworks, Microservice-Supporting Frameworks) For the implementation of a micro-service-based web application, powerful frameworks for the implementation of the frontend and the backend are available. Certain frontends provide a specific support for the development of microservices, especially related to the web API via which the functionality of the microservice can be accessed.

(CI/CD, Container Virtualization) One of the driving forces of microservices is the continuous integration (CI) and continuous deployment (CD) of implemented software systems. In contrast to monolithic architectures, microservices are loosely coupled. Changes in the software system relate to single micro-services which can be tested and deployed independently from the other part of the software system. In practice, each microservice is deployed in a so-called container which provided by a container-virtualized computing infrastructure to run the microservice.

EXERCISES MICROSERVICE ARCHITECTURE

(1) What are the predecessors of the microservice concept?

(2) Who used the term microservice (in fact, micro service) for the first time in a publication and when was it?

(3) What are the specifics of a microservice architecture compared to the traditional three-layer architecture?

(4) What are the contributions of BDD and DDD to the microservice architecture?

(5) In which phases of the software development process BDD and DDD are applied?

(1) ++Application Plane++
1990: Distributed technologies (e.g. Corba, DCE, JEE) leading to the traditional three-layer architecture
2000: SOA (Service-Oriented Architecture)
2010: Microservices

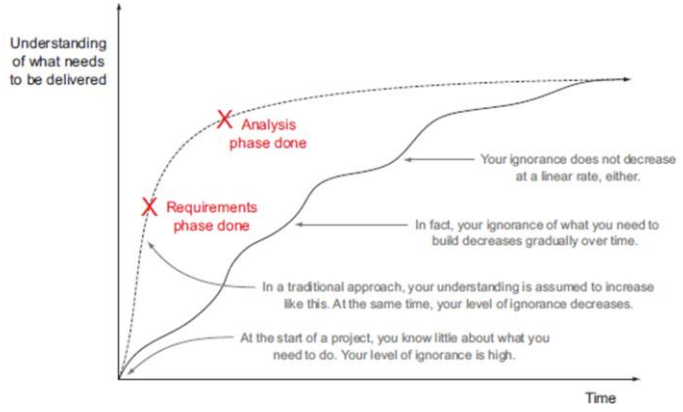
(2) ++The Who-is-Who in the Field of Microservices++
- James Lewis
- 2012: Micro services - Java, the Unix Way

(3) ++Planes and Three-Layer Architecture++, ++Microservice Architecture++
- Separation of the business logic layer (middle layer) into two layers, the application layer and the domain layer
- Explicit introduction of two APIs, the Backend Microservice API and the Backend for Frontend Microservice API

(4) ++Features and the Domain Model in a Microservice Architecture++
- BDD: Each feature is a vertical cut through all layers of the microservice architecture
- DDD: The domain model is implemented in the domain logic layer and provides the semantical foundation for all features

(5) ++BDD/DDD-Based Software Development++
- BDD: Concerns the analysis and implementation and test phase. At C&M, BDD is only used to define the end-to-end tests since the experiences have shown that a full specification based on BDD features becomes too complex.
- DDD: The domain model is the central design artifact of the development process.

ANALYSIS: (I) BEHAVIOR-DRIVEN DEVELOPMENT – Motivation



[Sm15]

(1) When the project starts the requirements the software must fulfill are not known

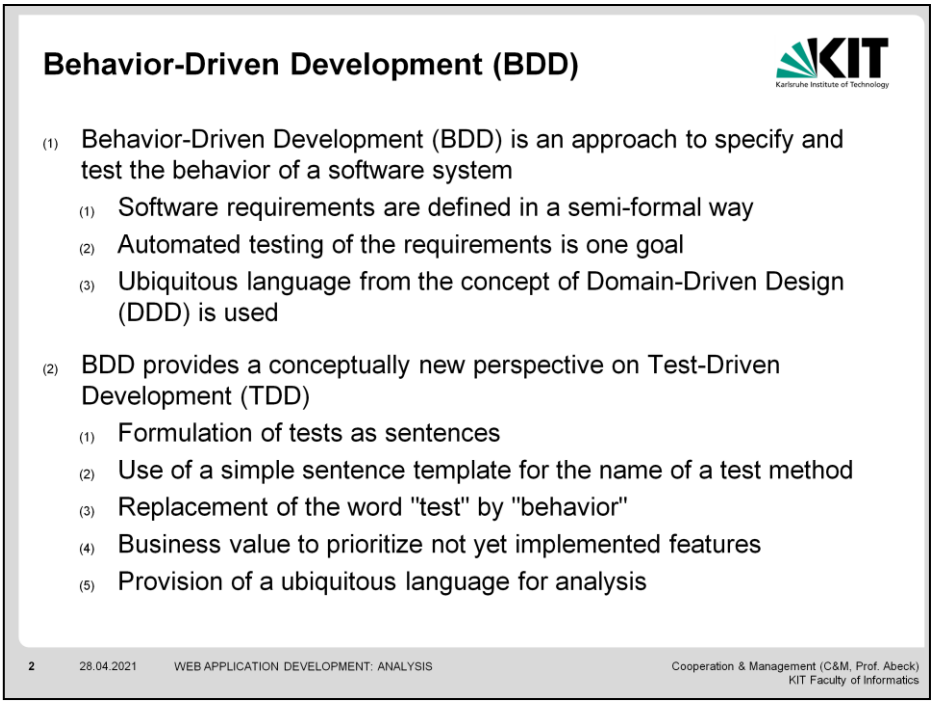
(2) Uncertainty will always remain in a project

The "right" software can only be built when the requirements the software must fulfill are precisely elicited. Only in this case the problem the software should solve is well understood.

(1) At project start the level of ignorance is at its maximum. The learning path is neither linear nor predictable.

(2) It is hard to predict what the development team will learn as the project progresses. It is not advisable trying to force reality to fit into your plan according to a Swiss Army proverb: "When the terrain disagrees with the map, trust the terrain."

[Sm15] John Ferguson Smart: BDD in Action – Behavior-Driven Development for the whole software lifecycle, Manning Publications, 2015. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering

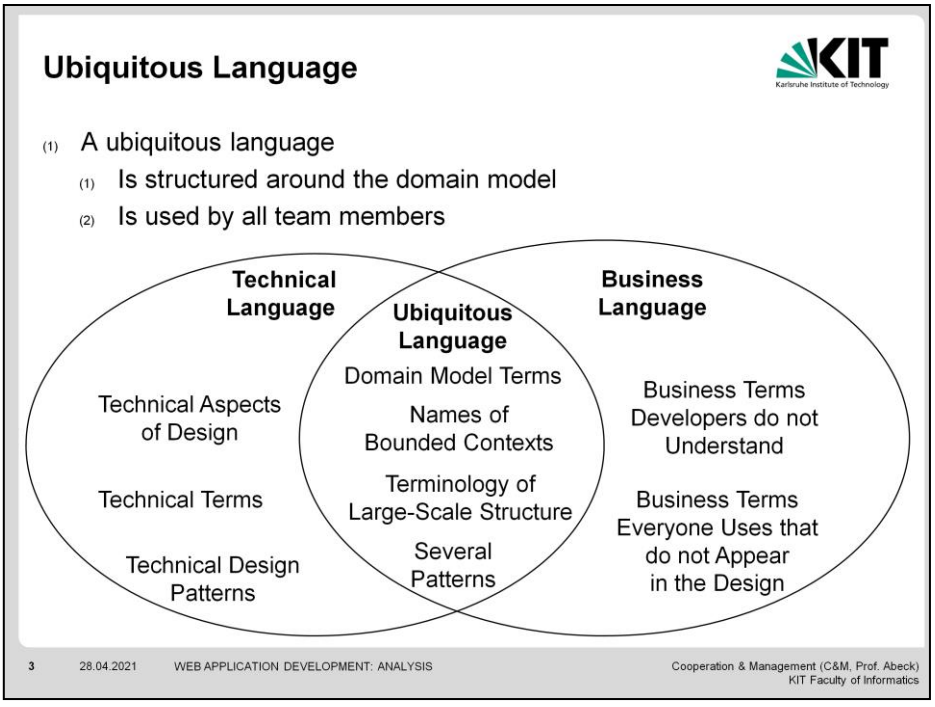


(1) Behavior-Driven Development (BDD) can be seen as an agile software development technique. Since the specified behavior is the basis of BDD it is also called Specification-Driven Development.
(1.1) The requirements comprise the tasks, goals and expected results of the software.
(1.2) The founder of BDD, Dan North, started from the concept of Test-Driven Development (TDD) which pursues the same goal of an automated and consistent testing of the software.
(1.3) The domain model constructed according to the concept of Domain-Driven Design provides the syntax and semantics (behavior) of the terms that are used to describe the tests according to the BDD concept.

(2) Dan North developed the ideas of BDD from deficiencies he saw in the TDD concept.
(2.1) The idea of formulating Junit test classes as sentences stems from a utility called agiledox from Chris Stevenson.
Developers were motivated to use sentences for documentation purposes.
(2.2) The sentence template used by Dan North was: "The class should do something".
(2.3) By this replacement it became clear what test means and how to proceed. The sentence describes the next behavior one is interested in.
(2.4) The question behind the determination of the business value is: What is the next most important thing the system does not do?
(2.5) The behavior-driven thinking was applied to defining requirements based on a consistent vocabulary for analysts, testers, developers, and the business. This idea of a ubiquitous language Dan North (together with the business analyst Chris Matts) had at about the same time when the DDD textbook [Ev03] from Eric Evans was published.

BDD	Behavior-Driven Development
DDD	Domain-Driven Design
TDD	Test-Driven Development

[Dan-BDD] Dan North & Associates: Introducing BDD. Better Software Magazine, March 2006. https://dannorth.net/introducing-bdd/
[Ev03] Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering



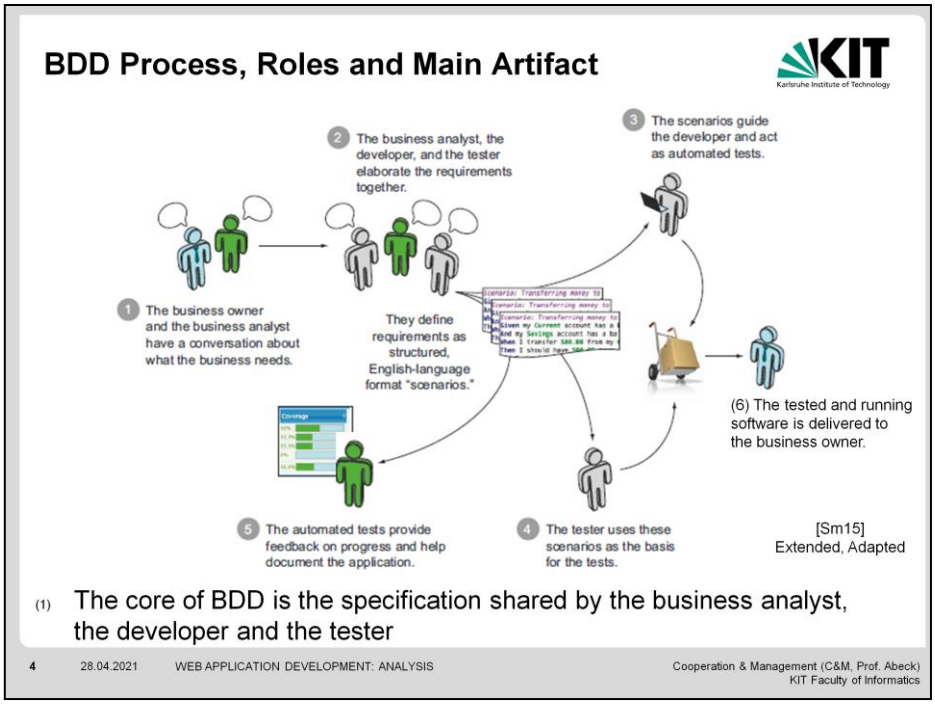
(1) This is the definition of a ubiquitous language in the context of domain-driven design (DDD).
(1.1) In the DDD approach the domain model becomes the core of a common language for a software project.
(1.2) The language is expressed by the domain model which should not only be understood by the software developer but also by the domain expert. The language connects all the activities of the team with the software

(Figure) The ubiquitous language is defined by the overlap of the technical language and the business language.

(Domain Model Terms) The domain model terms are the terms that are understood by both the domain experts and the developer. Besides these terms there are technical terms and business terms which are used solely by developers and domain experts.

(Bounded Contexts) Bounded contexts give team members a clear and shared understanding of what has to be consistent and what can be developed independently.

(Large-Scale Structure) A set of high-level concepts, rules, or both that establishes a pattern of design for an entire system. A language that allows the system to be discussed and understood on a broad and high level.



A main characteristic of the process behind the Behavior-Driven Development (BDD) is the collaboration of three central roles (business analyst, developer, tester) in the development process. A fourth role is the business owner using the software.

(1 The business owner ... business needs.) The first step is similar to the traditional development process.

(2 The business analyst ... requirements together.) Two aspects in BDD are new and different compared to the traditional process:
(i) The collaboration of the business analyst, developer and tester.
(ii) The structured specification all three roles agree on.

(3 The scenarios guide the developer ...) For the developer the specification defines what should be implemented.

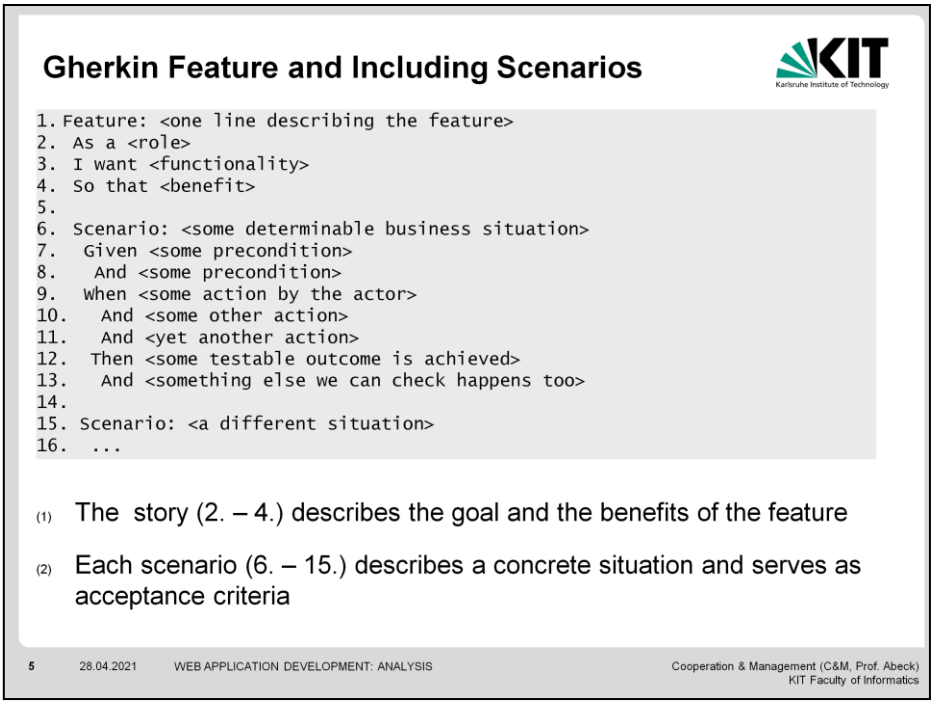
(4 The tester ... for the tests) For the tester the specification defines what should be tested.

(5 The automated tests ... document the application) For the business analyst (and the two other roles) the specification and the outcome of the tests document the software and its current state of implementation. The scenarios can be seen as a low-level technical documentation and provide up-to-date examples of how the system works.

(6) The structured collaboration of the roles taking part in the development process leads to a software which does the right thing and which does the thing right.

(1) The advantage of BDD compared to the traditional development process is the common specification which prevents misunderstanding and miscommunication, especially between the business analyst and the developer. This specification which is structured according to a defined grammar and keywords is the core of BDD. It consists of a so-called story and scenarios described in natural language.

(Extended, Adapted) The text "6 The tested ..." was extended. The role of the business analyst (i.e., a green instead of a grey stickman) was adapted in step 5.

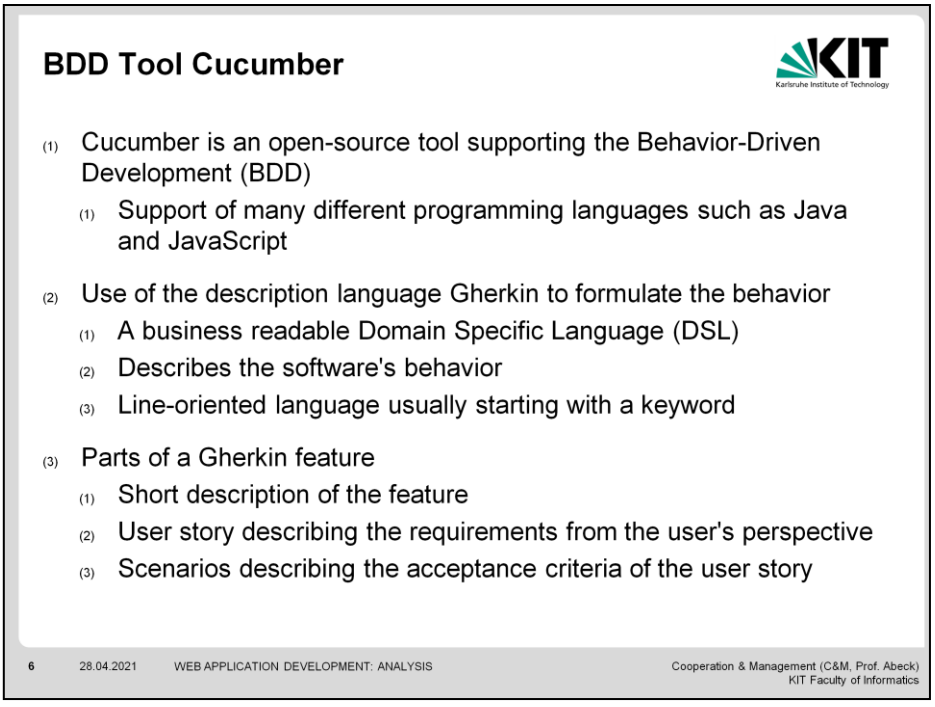


BDD provides a language Gherkin consisting of specific keywords used to define the features (and the scenarios to test these features) which the software system must fulfill. The proposed template is essentially taken from two sources: The upper feature description part is taken from [DNA-Story] and the lower scenario description part is taken from [WH12].

(1) In the description the business value of the feature should be recognizable.
Although Gherkin does not prescribe any structure and use of specific keyword in this part, it is recommended to use the structure "As a – I want – So that" proposed by Dan North.

(2) Behind BDD's Given-When-Then concept a finite state machine can be seen.
The structure and the keywords "Given – When – Then" are prescribed by Cucumber.

[DNA-Story] Dan North & Associates: What's in a Story. https://dannorth.net/whats-in-a-story/
[WH12] Matt Wynne, Aslak Hellesoy: The Cucumber Book. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering



Cucumber was created by Aslak Hellesoy [WH12]. The company behind the software is called Cucumber Ltd [Cuc-Web]. Cucumber is available as an open-source version and as a professional version (Cucumber pro)

(1) Cucumber allows to formulate requirements on a software as a textual specification and to automatically check the implementation if the specification is fulfilled.

(1.1) Originally, Cucumber was developed in Ruby to apply BDD to Ruby applications.

(2) Gherkin is based on a natural language and defines specific keywords which have specific meanings.

(2.1) It brings together business people and developers and serves as a ubiquitous language.

(2.2) But there are no details how the behavior is implemented.

(2.3) Examples of keywords are: Feature, Scenario, Given, When, Then, And

Like Python and YAML (YAML Ain't Markup Language, Yet Another Markup Language), Gherkin uses indentation to define structure. Line endings terminate statements.

(3) The template provided by Cucumber does only prescribe the structure of the scenarios, but not of the story.

(3.1) Usually, the description consists of only one short sentence.

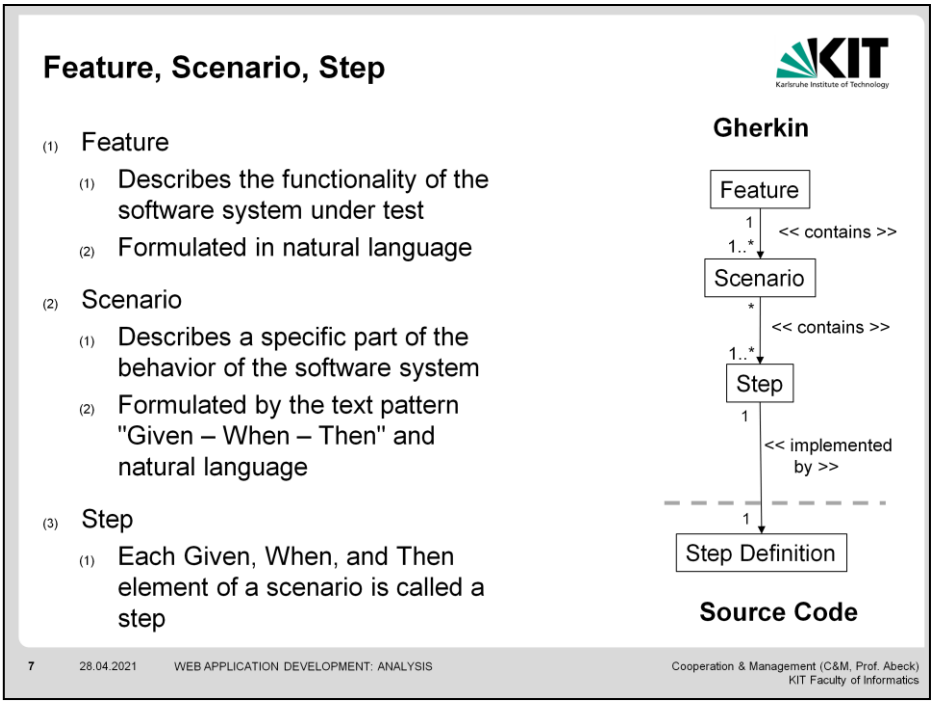
(3.2) A user story is structured according to "As a [role] I want [feature] So that [benefit]". These are not Gherkin keywords.

(3.3) The structure of a scenario is prescribed by the Gherkin keywords "Given – When – Then".

BDD	Behavior-Driven Development
DSL	Domain-Specific Language
YAML	YAML Ain't Markup Language, Yet Another Markup Language

[Cuc-Web] Cucumber: Website. https://cucumber.io/

[WH12] Matt Wynne, Aslak Hellesoy: The Cucumber Book. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering

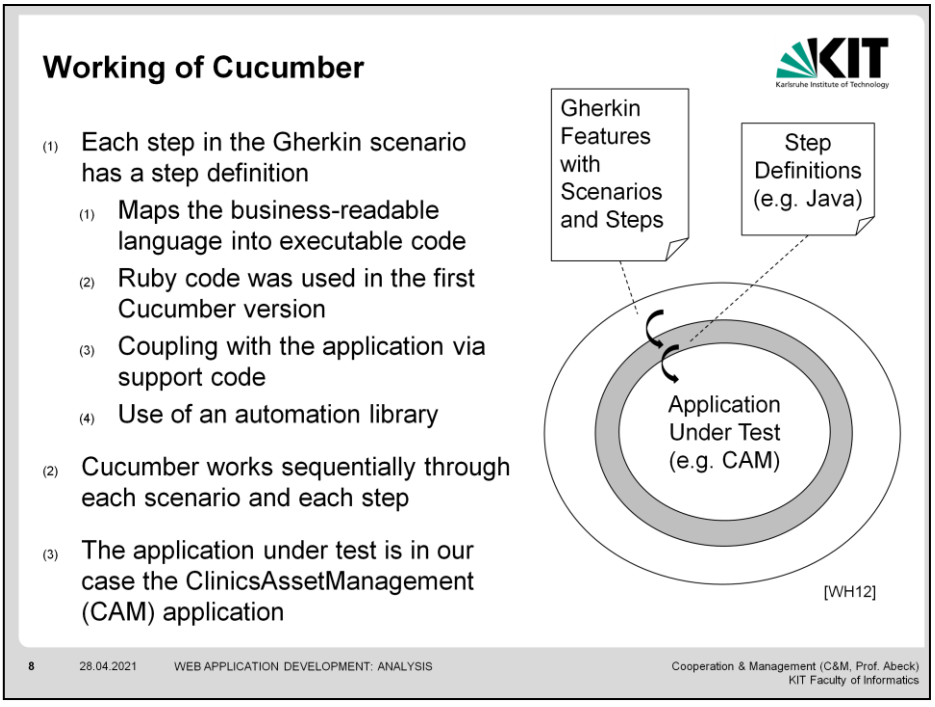


BDD introduces specific concepts and terms to describe the behavior of a software system in a systematic and uniform way.

(1) Each feature is described in a separate file.
(1.1) A feature describes a functionality of the software system to be tested. The test specified by a feature is a user acceptance test, also called ent-to-end test.
Example: A user can log in to the system.
(1.2) Features (and also scenarios and steps) cannot be interpreted by a system but serve the developer and customer to group the scenarios.

(2) A feature can contain several scenarios which are part of the feature file.
(2.1) Both success and failure usage procedures are described.
(2.2) "Given" lists the preconditions, "When" names the actions of the user or the events from the system and "Then" specifies the outcomes that must be testable.

(3) Scenarios consist of steps. In Cucumber, steps are not part of the feature file.



The main building blocks of Cucumber are introduced and their interaction is described on a high level and by using an illustration.

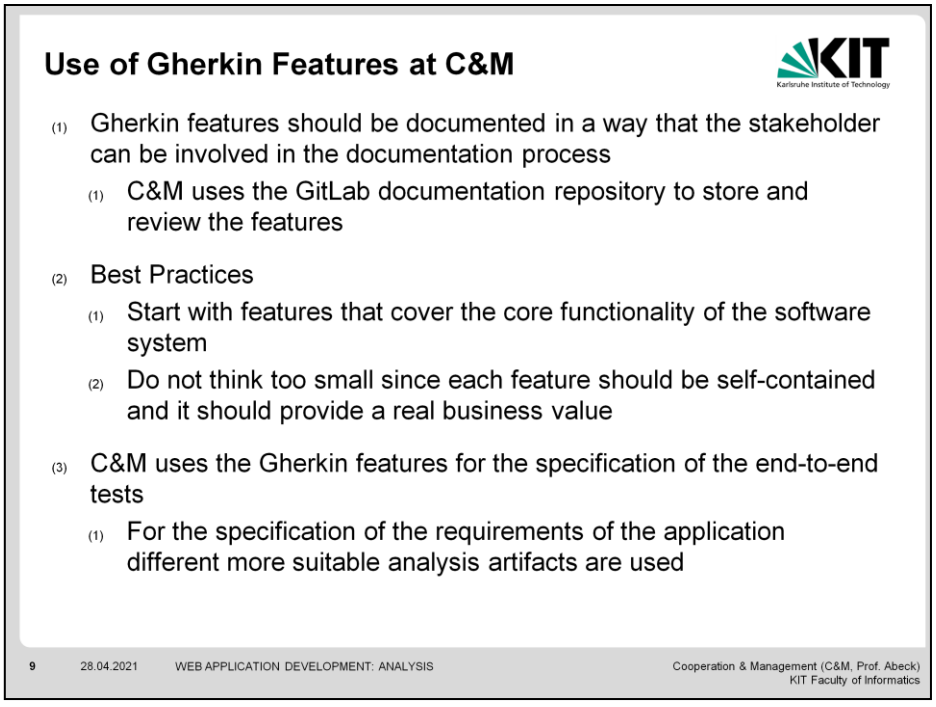
(Gherkin Features with Scenarios and Steps) The features written in the Gherkin language are read in by Cucumber which is a command-line tool. The text in such ".feature" files must follow some basic rules prescribed by Gherkin (such as use of the specific keywords).

(1) The step is the part in a Gherkin feature which can be executed. To do so, to each step a step definition must be added by which the (operational) semantics of this step is expressed.
(1.1) (Step Definitions) The step definition bridges the gap between the informal, text-based description of how the software system should behave and the testing of this behavior.
(1.2) There exist Cucumber versions which provide other programming languages (e.g. Java, JavaScript) to implement step definitions. In a mature test suite, the step definition itself will probably just be one or two lines of Ruby.
(1.3) The step definition code delegates to a support code, specific to the domain of the application under test, that knows how to carry out common tasks on the system. For each step the developer responsible for the BDD tests must provide the support code.
(1.4) An example is a web driver (e.g. Selenium, Protractor) which allows to automate the input via a web browser in order to test a web application.

(2) If an error occurs in one step Cucumber marks the scenario as failed and moves on to the next scenario. If it gets to the end of the scenario without any of the steps raising an error, it marks the scenario as having passed.

(3) The Gherkin features and step definitions to run the user acceptance tests are stored in the GitLab repository of the appropriate application microservice.

CAM ClinicsAssetManagement

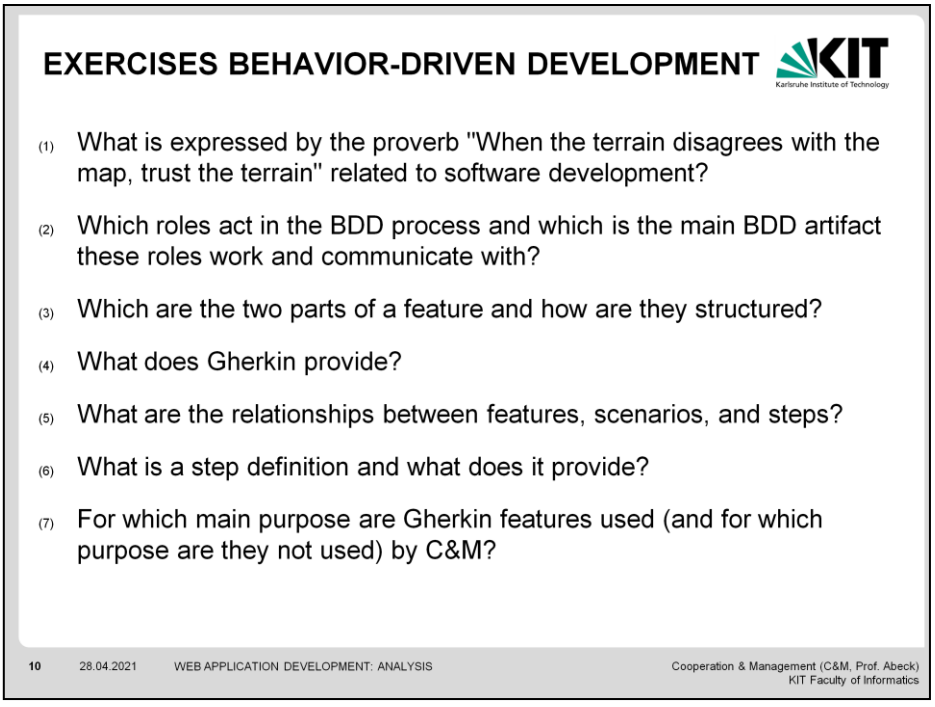


The artifact which describes the Gherkin features according to the BDD is the main outcome of the requirements analysis. Since all further development steps, esp. the modeling of the domain, depend on this artifact it should be documented in an adequate and defined way.

(1) The Gherkin features provide a requirements specification which is an integral artifact of the software development process. Therefore, developers must have access to the features in their IDE (Integrated Development Environment) since they run through the features in order to carry out user acceptance tests of their software implementation. Nevertheless, features are not only relevant for the developers, but also for the stakeholders for which the software is developed. Therefore, an adequate way of documentation of the feature collection must be made available not only for the developers, but also for the stakeholders.
(1.1) At C&M, the feature are stored in the GitLab documentation repository. Stakeholders who have no access to the GitLab can be involved by generating PDF documents which are sent to these stakeholders so that they can make comments.

(2) The specification of features is a creative process which makes it impossible to state complete rules how to go through this process. Nevertheless, best practices can be formulated which give some valuable hints for the process.
(2.1) The first (one to three) features should specify the functionality of the MVP.
(2.2) A business value will usually not be provided by one single functionality but by a set of coherent functions by which a user can fulfill a specific task.

(3) (3.1) As will be shown in the next chapter, the main analysis artifacts used by C&M to specify the requirements are capabilities, user/system interactions (U/SI) and U/SI flows.



(1) ++Motivation++
The proverb concerns the analysis of the requirements the software should fulfill. It means that the plan must always be adapted to reality – and not the other way round.

(2) ++BDD Process, Roles and Main Artifact++
- Three central roles: business analyst, developer, tester
- Scenario description: This specification which is structured according to a defined grammar and keywords is the core of BDD.
-- Hint: Feature description including stories and scenarios would be the correct term.

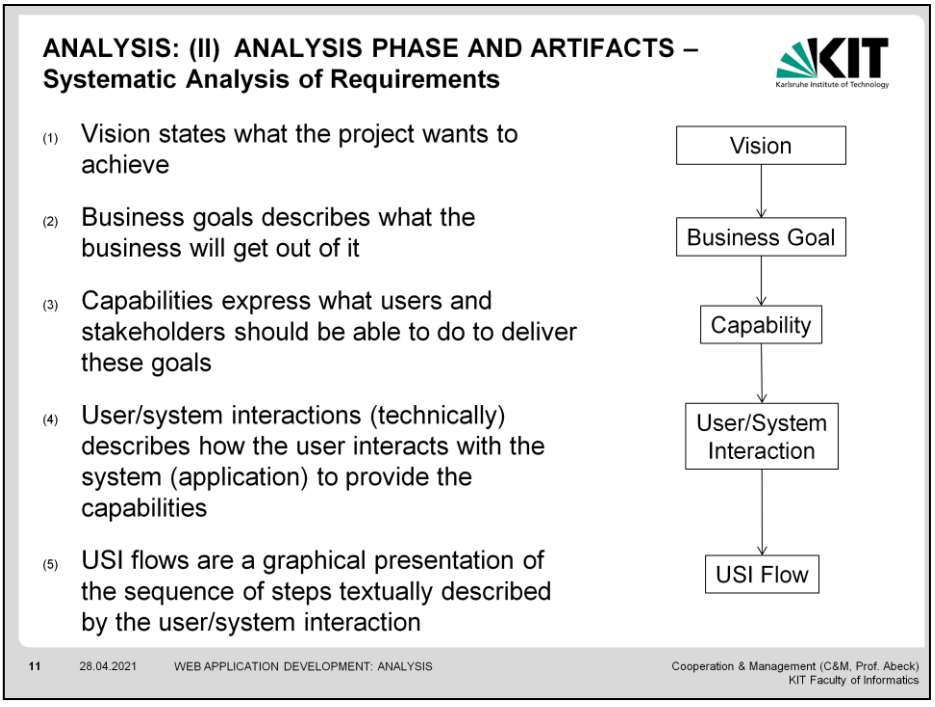
(3) ++Gherkin Feature and Including Scenarios++
A story consists of two parts
- Narrative description of a requirement structured according to "As a – I want – So that"
- A set of acceptance criteria presented as scenarios structured according to "Given – When – Then"

(4) ++BDD Tool Cucumber++
- A domain-specific language (DSL)
- Describes the software's behavior
- Line-oriented language usually starting with a keyword

(5) ++Feature, Scenario, Step++
- A scenario is part of a feature
- A scenario describes the acceptance criteria of the user story which is another part of the feature
- A scenario consists of steps.

(6) ++Working of Cucumber++
- Implemented in a programming language (e.g. Ruby)
- Map the business-readable language into executable code
- Coupling with the application via a library of support code
- Use of an automation library

(7) ++Use of Gherkin Features at C&M++
- Main purpose: specification of the end-to-end tests
- Not a main purpose: specification of the requirements the application must fulfill



Developers use lots of different words to describe what they want to build. The following terminology partly taken from [Sm15] is used to provide a clear understanding in this confusing part of software development.

(1) The project vision provides a high-level guiding direction for the project. The problem must be clarified to understand which software system is needed by which users to solve the problem.

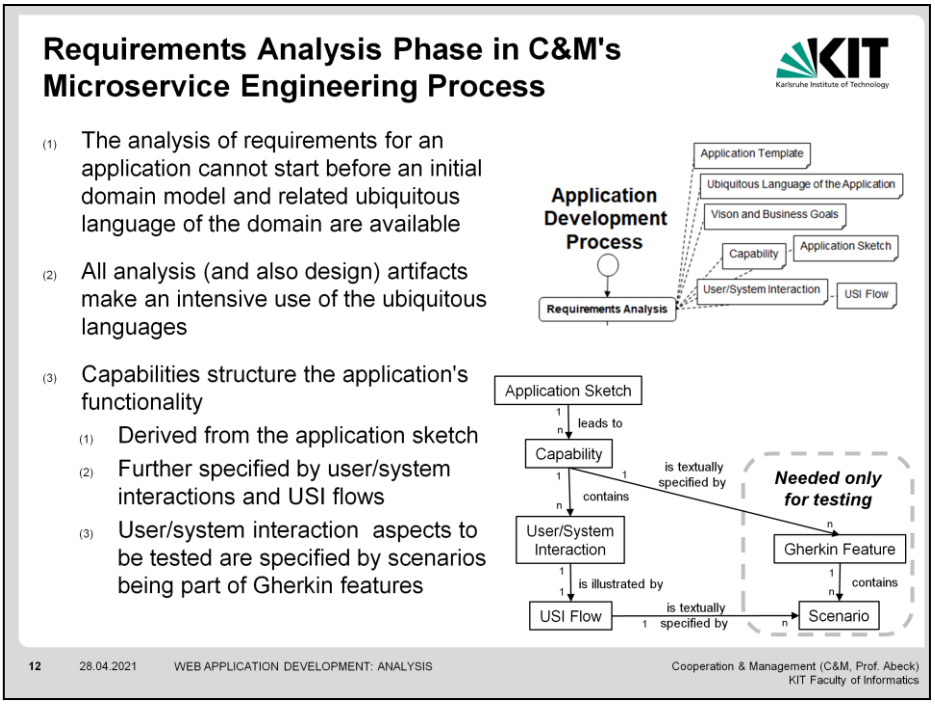
(2) The software system must have a measureable, positive impact on the business of the customer the software is built for.

(3) Capabilities give users or stakeholders the ability to realize some business goal or perform some useful task. A capability represents the ability to do something; it does not depend on a particular implementation.

(4) At this point of the systematic requirements analysis the focus changes from a business perspective to a more technical perspective, since a user/system interaction concentrates on the behavior of the system (i.e. application) with respect to the actions of a user.

(5) The purpose of the USI flows is to increase the understanding of a user/system interaction by illustrating the steps of the interaction in a graphical way.

[Sm15] John Ferguson Smart: BDD in Action – Behavior-Driven Development for the whole software lifecycle, Manning Publications, 2015. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering



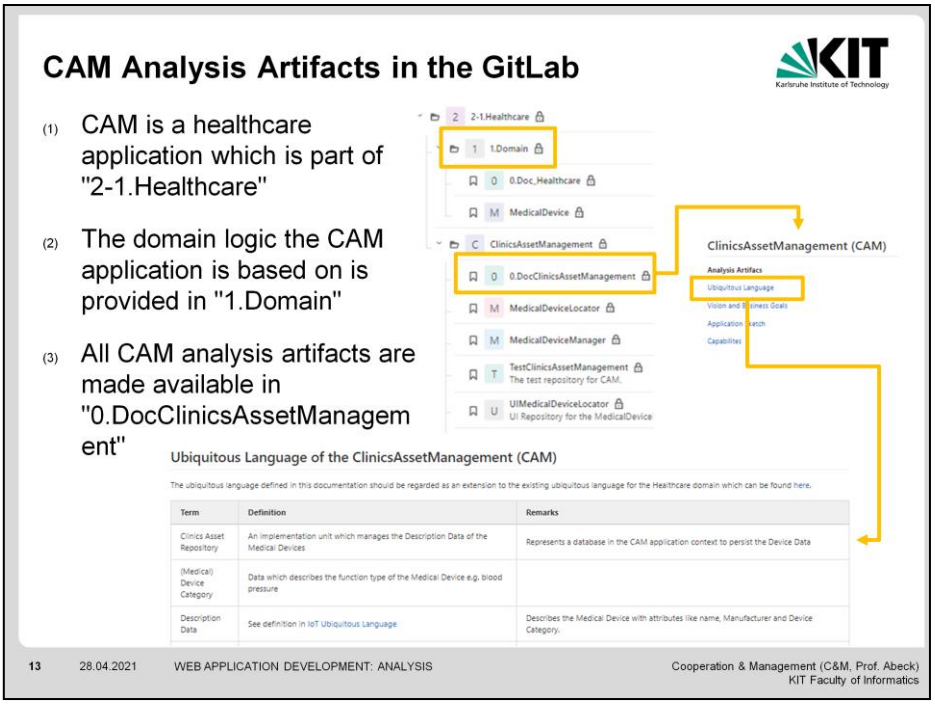
The whole microservice engineering process is introduced in a previous chapter INTRODUCTION of this course unit.

(1) Since C&M's engineering approach is domain-driven, there must be a sufficiently clear understanding of the domain(s) the application is based on.

(2) The ubiquitous language of the application contains references to those terms of the ubiquitous languages of the involved domains which are relevant for the application.

(3) In the requirement analysis approach followed by the engineering process, the capabilities are the central analysis artifact by which the functionality of the application to be developed is structured from the business perspective. A capability should be described by a short text from which user/system interactions can be derived which describe how the capability can be fulfilled by using the application.
(3.1) The application's functionality is informally described by the relations of the objects appearing in the application sketch. The capabilities can be systematically derived from the application sketch by grouping the objects and their relations.
(3.2) Each capability is further refined by a number of user/system interactions (USI). US/I flows show to the future user of the application how he/she should work with the software application in order to put this capability into practice.
(3.3) Gherkin features are only specified in the context of tests which should be carried out. A capability is mapped to one or more Gherkin feature and a user/system interaction is mapped to one or more scenarios depending on the aspects which should be tested. A scenario specification comprises (i) the state of the application before the user/system interaction is carried out (Given), (ii) the user/system interaction itself (When), and (iii) the resulting state of the application (Then).

USI User/System Interaction



All artifacts of the analysis phase and the subsequent phases of the C&M microservice engineering process are made available in a systematic GitLab structure in order to allow a collaborative work on these artifacts in the team.

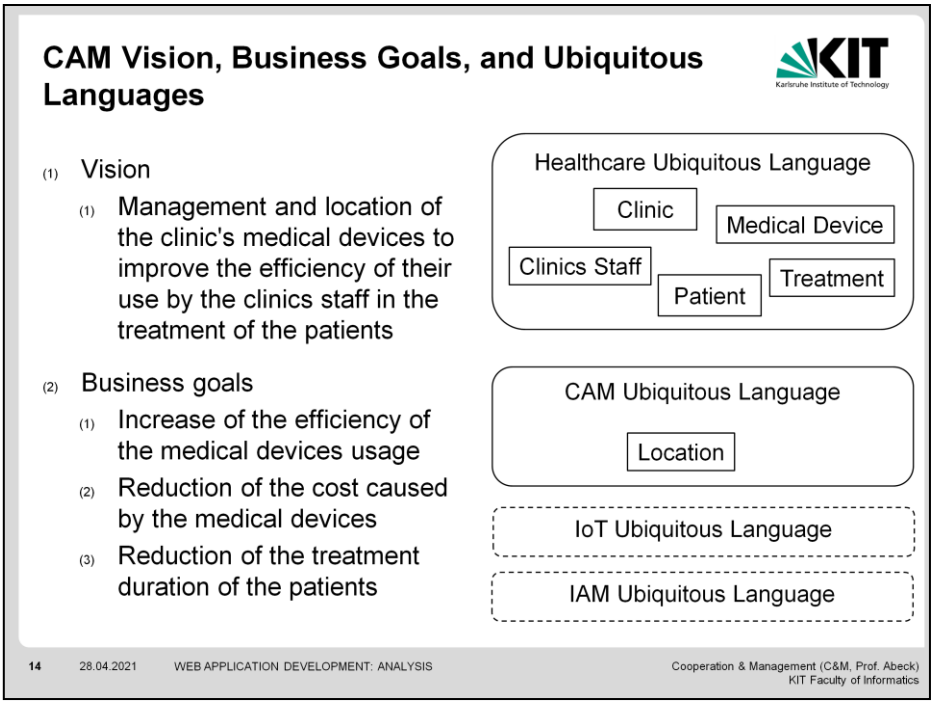
(1) "2-6.Healthcare" is a subgroup [CM-G-Hea] which belongs to the DOMAINS&APPLICATIONS part of the C&M GitLab. Examples of other domains investigated by C&M are ConnectedCar, Environmental, and ServiceEnvironment.

(2) "1.Domain" is a subgroup which consists of a documentation repository (0.Doc_Healthcare) and repositories for each domain microservice (MedicalDevice).

(3) "0.DocClinicsAssetManagement" is the documentation repository of the CAM application. It contains documentations to all CAM analysis artifacts, such as the ubiquitous language or the application sketch.

(Ubiquitous Language of the ...) As an example, the analysis artifact "Ubiquitous Language" is shown. The ubiquitous language of the CAM application is defined according to the C&M artifact guidelines [CM-G-Gui] which exist for this and all other artifacts constructed during the C&M microservice engineering process.

[CM-G-Gui] Cooperation & Management: Guidelines for the Construction of Artifacts. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation/3.artifactguidelines
[CM-G-Hea] Cooperation and Management: Healthcare, GitLab. https://git.scc.kit.edu/cm-tm/cm-team/healthcare



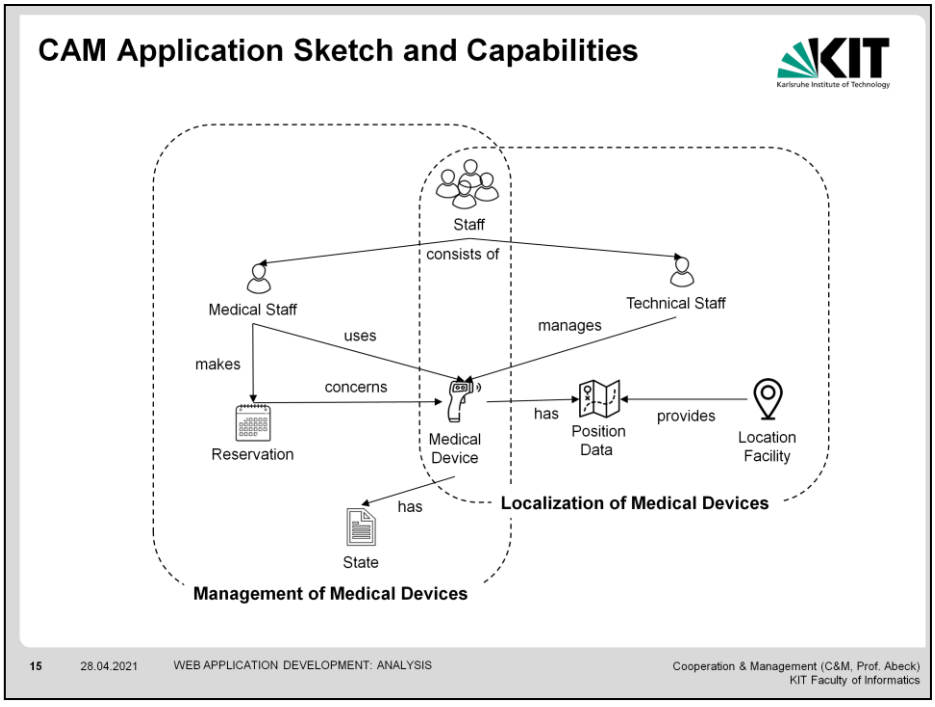
By defining the vision and the business goals, the software system to be developed is described from the business perspective. Since this description is part of the CAM documentation, it is stored in the documentation repository [CM-G-CAM].

(Healthcare Ubiquitous Language) When the vision and the business goals of an application (which in this case is the ClinicsAssetManagement CAM) are formulated, the terms from the underlying business domain(s) are to be used. In the case of CAM this is the healthcare domain.
(CAM Ubiquitous Language) All terms which have a central meaning for the application, but which are not part of the healthcare domain, become a part of the CAM ubiquitous language.
(IoT Ubiquitous Language, IAM Ubiquitous Language) These are two ubiquitous languages which will also be relevant for the CAM. The dotted lines express that these ubiquitous languages are not "visible" on the business level.

(1) The vision should make clear for which purpose the software system is used. In the case of CAM the location information of the medical devices is used to improve the efficiency of the usage of these devices.

(2) A business goal expresses the measureable, positive impact on the business of the customer the software is built for. The CAM application has a positive impact on (i) the medical device's usage efficiency, (ii) the medical device's costs, (iii) the treatment duration.

[CM-G-CAM] Cooperation and Management: ClinicsAssetManagement, Git Respository. https://git.scc.kit.edu/cm-tm/cm-team/healthcare/clinicsassetmanagement/doc_cam



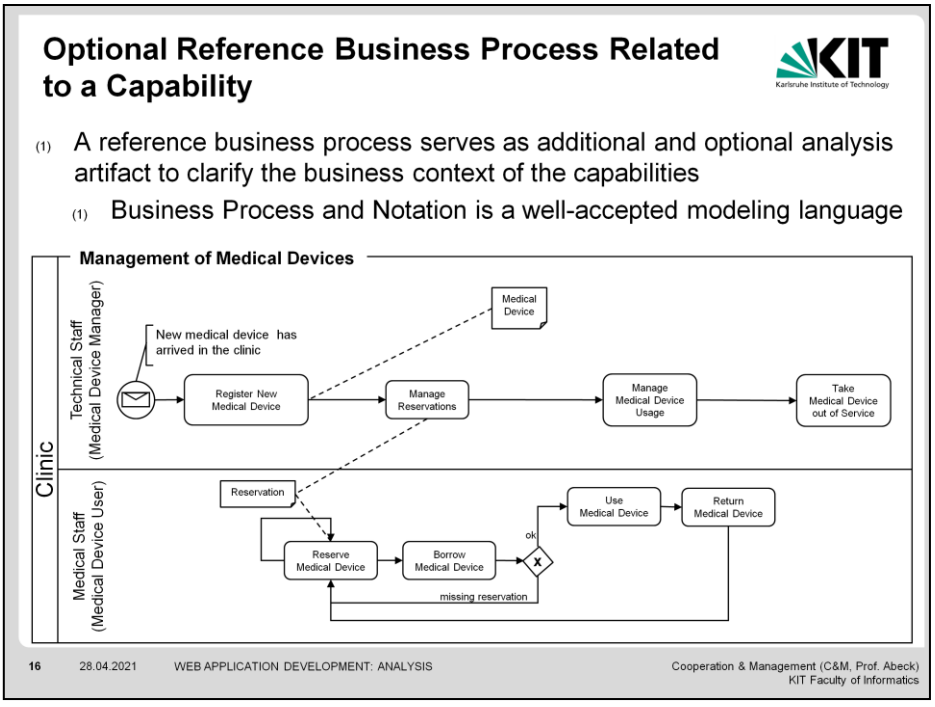
The CAM application sketch [CM-G-CAM-App] introduces the main subjects and objects appearing in the CAM application. The terms used in the sketch are defined in one of the ubiquitous languages which are relevant for the CAM application and which are shown on the page ++ClinicsAssetManagement (CAM) Vision and Business Goals and Their Relation to the Ubiquitous Languages++.

The relationships specified in the application sketch define the business functionality the application should provide. A grouping of coherent functionality leads to the capabilities of the application. In the case of CAM two capabilities can be identified which are informally described by an informal text. This text is written from the perspective of a business process in which this capability is needed.

(Management of Medical Devices) The management of medical devices is necessary to keep an overview about all medical devices in the clinic.
Therefore, the various device information is collected so that information about the medical device can be viewed at any time.
When new medical devices arrive in the clinic, the medical devices are registered.
Before the medical device can be used, a reservation is necessary.
In a regular interval, the medical devices need to be inspected. Old or defect medical devices are outsourced and removed from the inventory list.

(Localization of Medical Devices) The current location of the medical equipment allows the medical staff to find it easily and save time. For this purpose, the position of the medical device must be known. If a device is moved to another location, the current position in the list must be adjusted.

[CM-G-CAM-App] Cooperation & Management: CAM Apploication Sketch, GitLab Documentation Repository. https://git.scc.kit.edu/cm-tm/cm-team/healthcare/clinicsassetmanagement/doc_cam/-/blob/master/Pages/ApplicationSketch.md



Capabilities describe the functionality which should be provided by the application from a business perspective. An optional reference business process can be used to deepen the understanding of the business context.

(1) Reference in this specific context means that the business process provides an abstraction from a concrete business process in a concrete clinic.

(2) The diagram does not necessarily conform to the BPMN standard in all details. This does not cause any problems in this specific context since the diagram serves only for an overall (informal) understanding of the process.

(Clinic) The BPMN diagram models the reference business process for the capability "Management of a Medical Device" of the CAM application.

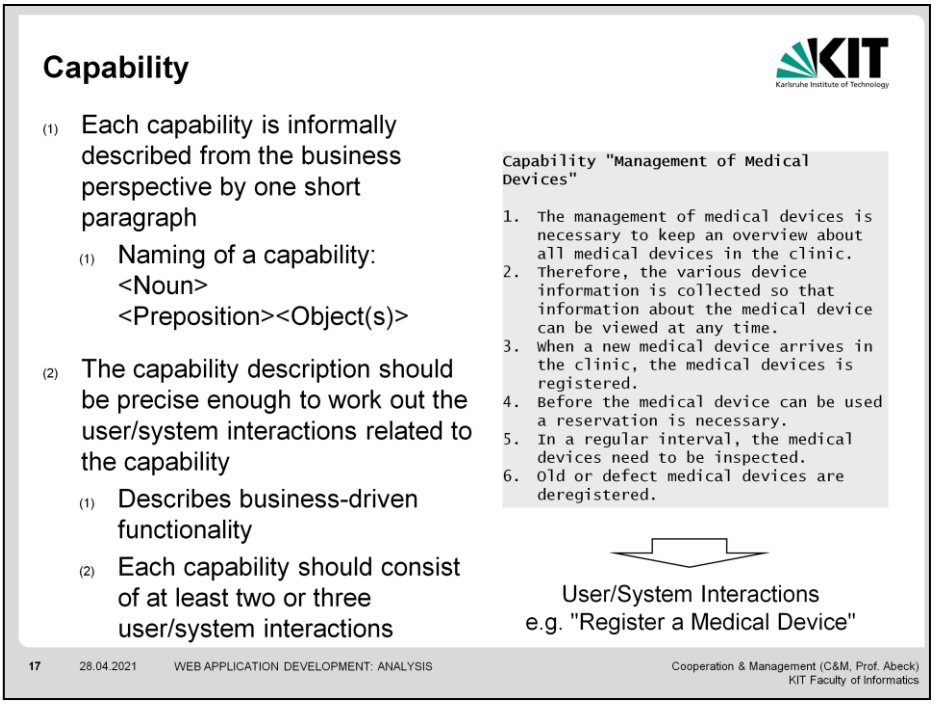
(Medical Device Manager, Medical Device User) The lanes and associated "roles" are introduced to ease the understanding of the process. They are put in brackets to express that these terms are not part of the ubiquitous language.
There are different ways, how the role of a medical device manager (one person, a group of persons, completely decentralized) is implemented in a concrete clinic depending on the organization of the management of the medical devices.

(Reservation) The reference process assumes that a medical device must be reserved before it can be borrowed and used. Each reservation is defined by a start and end time. A medical device user must be allowed to make a reservation (not yet modeled in the process!) and he/she can make more than one reservation. Reservation times should not overlap. Possibly, a preparation time between two reservations must be foreseen.

(Borrow Medical Device, ok) When a medical device user wants to borrow a medical device it is checked if a valid reservation exists for this user.

BPMN Business Process Model and Notation

[OMG-BPMN] Object Management Group: Business Process Model and Notation, Version 2.0, Tech. rep., URL http://www.omg.org/spec/BPMN/2.0, 2010. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering

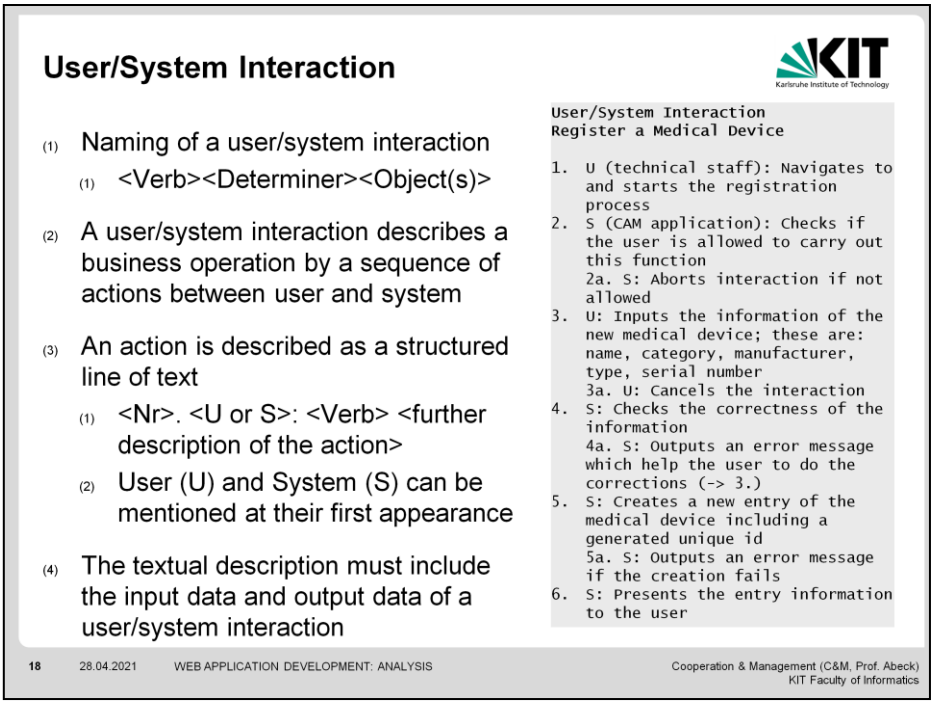


Each capability identified in the application sketch will be informally described and the user/system interactions are derived from this description. Afterwards the user/system interactions will be picked up in the USI Flows.

(1) It must be ensured that the textual description fits to the graphical description of the application sketch with respect to the content and to the use of the terms.

(1.1) The noun usually expresses an activity carried out on an object.
(Management of Medical Devices) In the example, the noun is "Management" and the objects are "Medical Devices".

(2) The description of the capability should split the main capability into business operations and describe them in a concise way. Besides the analysis of the capability's text description the relations appearing in the application sketch provide a valuable input for the definition of the user/system interactions. User/system interactions are business operations carried out by a user with the application which takes the role of the system.
(2.1) In the capability descriptions the core functions of the application are already mentioned. The application delivers the needed functionality.
("…to keep an overview about all medical devices…") In the example, this part of the description of a capability mentions that the application needs the functionality to show all medical devices to the user.
(2.2) Each capability contains usually multiple user/system interaction candidates. The user/system interactions are derived from the business functionality which is mentioned in the capability.



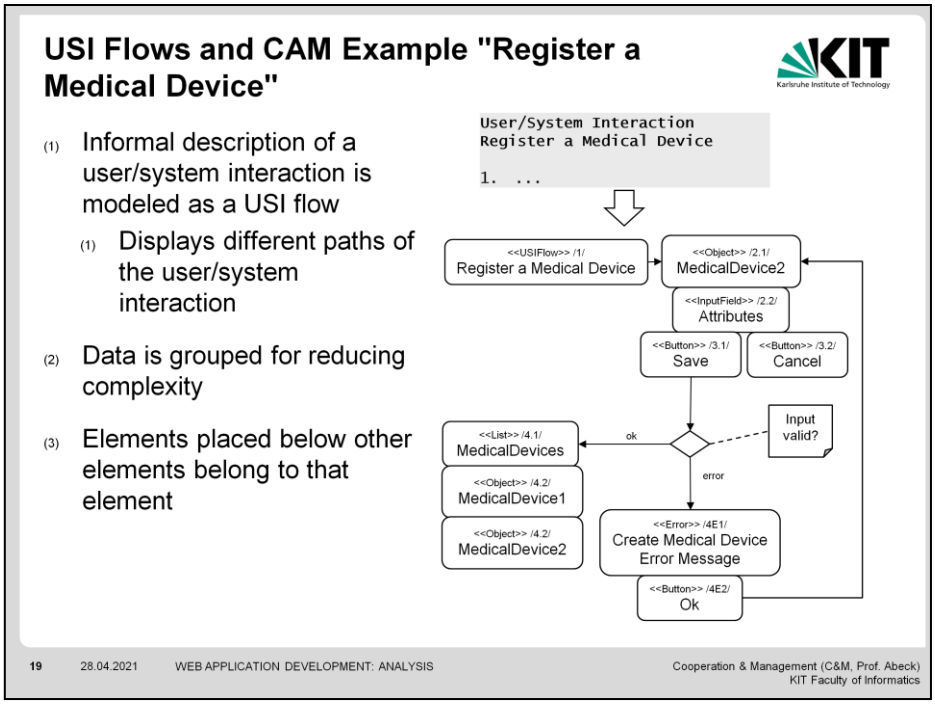
The user/system interactions are derived from the capabilities. They describe on a detailed level how this part of the capability's functionality is carried out by a flow actions between the user and the application. The application in this context takes the system role.

(1) All user/system interactions starts with a verb followed by a determiner and the object related to the user/system interaction.
("Create a Medical Device") is an example for a user/system interactions. Here "Create" is the verb, "a" is the determiner and "Medical Device" the object.

(2) A user/system interaction describes the business operation from the user's perspective. It describes what a user requires to input and what he receives from the system as output of the interaction.

(3) Each line describes an action carried out by the either the user or the system.
(3.1) The user/system interaction "Register Medical Device" on the right hand side shows examples of this structure.
(3.2) (1. U ...) (2. S ...) This is don in the example in line 2 and line 3.

(4) It must be possible to identify the input data and output data since this information is needed to derive the microservice API in the design phase
(3. U: ... name, category, ...) In this line the input data is described. In this case, the input data are the attributes which describe a medical device.
(5. S: Creates ...) The output data is the unique id generated by the system.



The USI flow is an optional analysis artifact by which the behavior of the application to be developed is further illustrated. The goal of a USI flow is not to give the future user an impression of the "look and feel" of the application which is the goal of mockups. The USI flows provide a valuable input for mockups.

(1) The informal description provides the information for creating the corresponding USI flow. The USI flow models this flow.
(1.1) The USI flow uses UI elements and transitions to model the user/system interaction. A USI flow usually considers multiple possible paths e.g. success, abort, error.

(2) Instead of modeling each attribute as a separate input field, it is recommended to group the data. In the example, the input field "DescriptionData" groups different attributes (such as name, category, manufacturer) which are required as user inputs to reduce the modeling complexity.

(3) UI elements which belong to another element are placed below another element. In the example, a MedicalDevice /2.1/ requires the user to enter the DescriptionData /2.2/ into input fields.

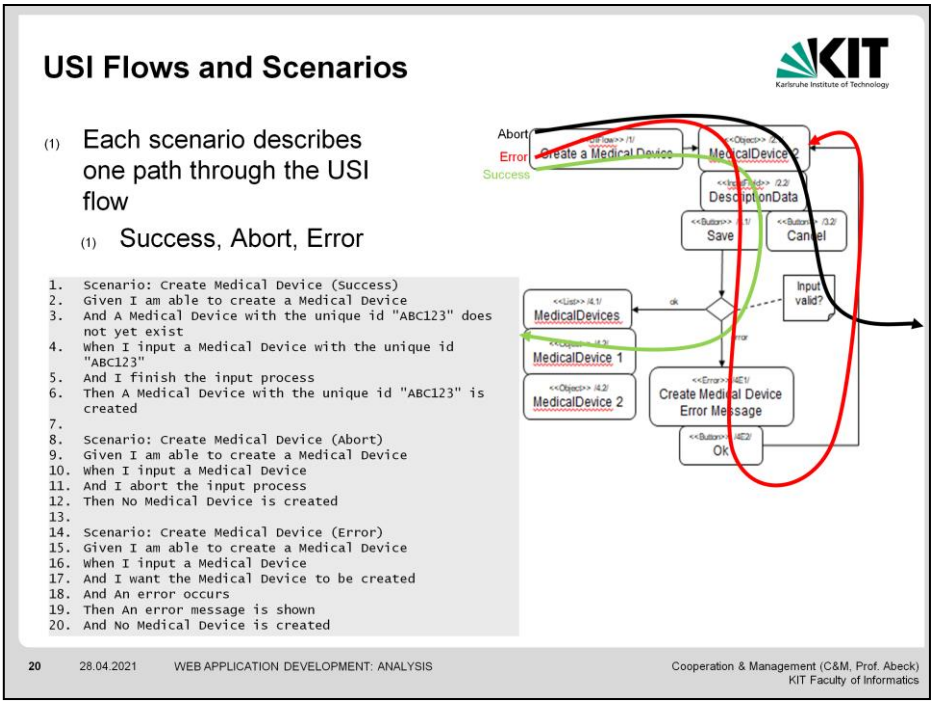
(<<USI Flow>> /1/) Each USI flow starts with an USI flow element which follows the naming convention of the user/system interaction. The number of this element is always /1/.

(<<Object>> /2.1/) The user gets a list with the attributes of DescriptionData which he can insert to create a new object MedicalDevice2.
(<<InputField>> /2.2/) A user can insert the into the input fields. The attributes of DescriptionData are specified during the user/system interaction derivation process. In order to keep the modeling of the USI flow simple, not all attributes are modeled, but are grouped according to the term DescriptionData in the application sketch (see ++Derivation of the User Interaction and Related UI Elements++).
(<<Button>> /3.1/ Save) A user can save the inputs by clicking this button. Afterwards the next step depends on the validity of the inputs. If the inputs are valid, the USI flow continues with step /4.1/, else step is /4E1/ is executed and an error is shown.
(<<Button>> /3.2/ Cancel) A user can abort the creation of a medical process.

(<<List>> /4.1/) The user receives a list which contains all MedicalDevices.
Note: It is assumed that the object MedicalDevice1 is already in the list when the user/system interaction is started.

(<<Object>> /4.2/) The object contains the attributes defined in the DescriptionData. The values of the attributes are displayed to the user.

(<<Error>> /4E1/) If the inputs of the user are invalid, an error message is shown to the user.
(<<Button>> /4E2/) The error message is closed and the user/system interaction restarts at step /2.1/.



The USI flows are the most relevant input for the specification of scenarios which describe a specific part of the behavior of the application to be developed.

(1) Different types of paths, esp. success, abort, error, must be distinguished.

(Create Medical Device) All three types appear in this exemplary USI flow of the CAM application.

(1. 8. 13.) The titles of the scenarios are built of two parts: (i) the title of the user/system interaction (ii) the type of path/scenario put into brackets

(2. 9. 14.) This precondition expresses that the user (i.e. the staff) is able to carry out this specific user/system interaction (i.e. Create Medical Device).
Remark: Since several user/system interactions are described as scenarios in the feature this step cannot be described in a background section of a feature.

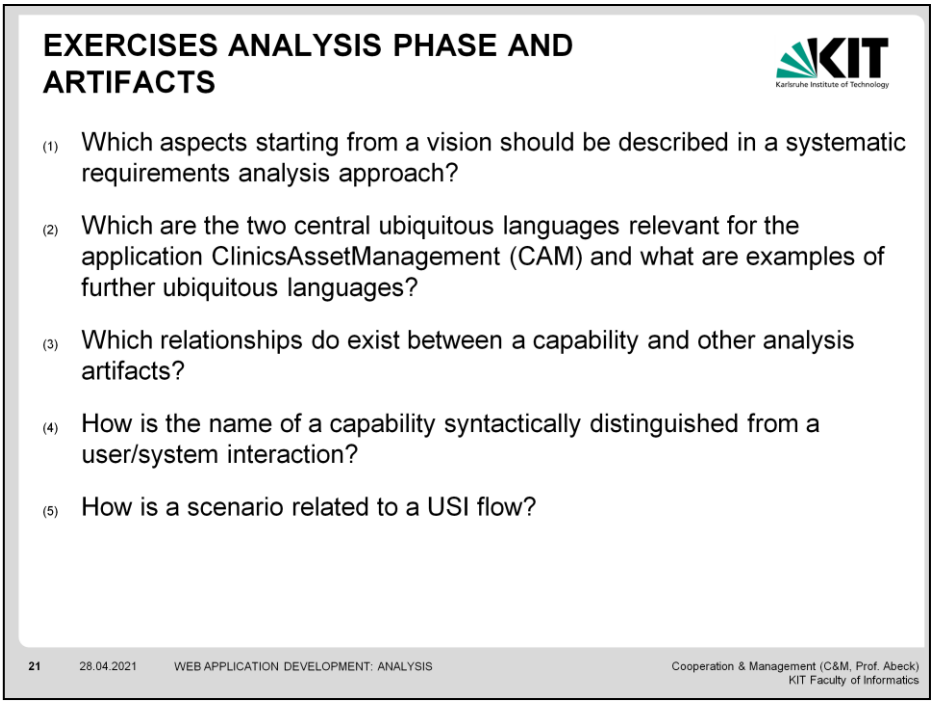
(3.) In the scenario, so far only one possible error case (duplicate medical devices) is covered.

(4. 5.) The input of the other attribute values of a medical device is not covered by the scenario.

(5.) (11.) These actions corresponds to the click on the OK button or the CANCEL button.

(17.) In the current draft of the error scenario, the different types of errors which can occur, are not distinguished. In a further refinement step, an error scenario for each type of error can be introduced:
(i) Scenario: Create Medical Device (Error: Duplicate Medical Device)
(ii) Scenario: Create Medical Device (Error: Incomplete input)
(iii) ...

General remark: It is recommended to restrict the feature and including scenario description to the core part and express additional aspects (in this case the different input fields of a medical device or the errors which can occur) as additional informal information. Otherwise, the number of scenarios would explode which produces a high effort and makes the specification hard to handle.



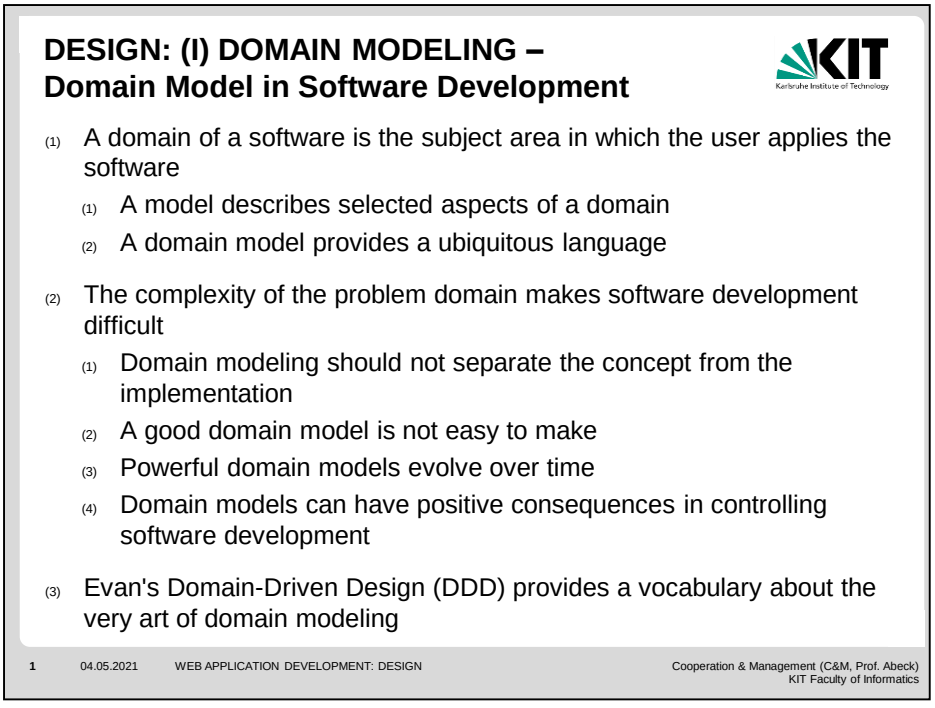
(1) ++Systematic Analysis of Requirements++
- Vision: states what the project wants to achieve
- Business goal: describes what the business will get out of it
- Capability: expresses what users and stakeholders should be able to do to deliver these goals
- User/system interaction: (technically) describes how the user interacts with the system (application) to provide the capabilities
- USI flows: are a graphical presentation of the sequence of steps textually described by the user/system interaction

(2) ++ClinicsAssetManagement (CAM) Vision and Business Goals and Ubiquitous Languages++
- CAM and Healthcare ubiquitous language
- IoT and IAM ubiquitous languages

(3) ++Requirements Analysis Phase in C&M's Microservice Engineering Process++
- Derived from the application sketch
- Contains one or more user/system interactions
- Is textually described by one Gherkin feature

(4) ++Capability++ ++User/System Interaction++
- Capability: <Noun> <Preposition><Object(s)>
- User/system interaction: <Verb><Determiner><Object(s)>

(5) ++USI Flows and Scenarios++
A scenario is a path through a USI flow.



The motivation of Domain-Driven Design (DDD) is taken from Martin Fowler's foreword of Eric Evans's book "Domain-Driven Design – Tackling Complexity in the Heart of Software" [Ev03].

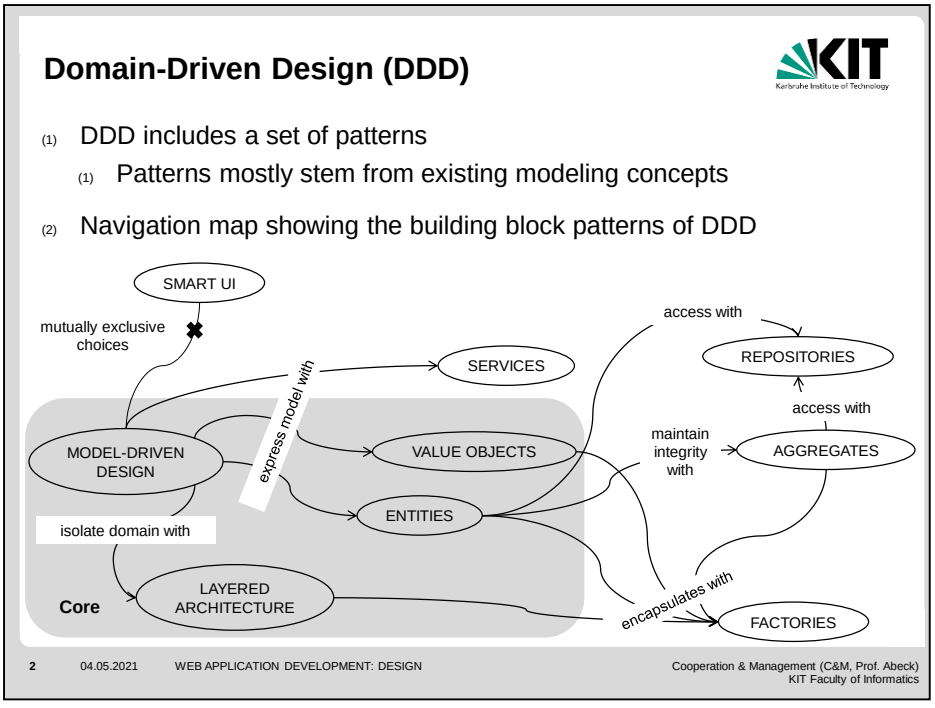
(1) A general definition of a domain is: A sphere of knowledge, influence, or activity.
(1.1) A model itself can be defined as a system of abstractions.
(1.2) By this language domain experts and technologists are tight together.

(2) There are many things that make software development complex. A deep understanding of the problem domain is in the heart of this complexity.
(2.1) An effective domain modeler can both use the whiteboard during his talk with the customer and work together with a programmer on a Java program.
(2.2) Only few people can do it well – and the competence to build good domain models is not easy to teach.
(2.3) Domain models are not first modelled and then implemented. Instead, an experienced modeler gets splendid ideas from earlier releases of the domain model.
(2.4) DDD teaches a lot of people how to apply modeling concepts to add structure and cohesion to the software development.

(3) Based on this vocabulary the activity and the hard-to-learn skill of domain modelling can be explained.

DDD Domain-Driven Design

[Ev03] Eric Evans: Domain-Driven Design – Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.
https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering



The figure shows a navigation map of the building blocks of Domain-Driven Design (DDD) [Ev04]. Each building block can be seen as a design pattern. Besides the pattern group of "Building Blocks" there exist further pattern groups related to "Supple Design", "Context", "Distillation", and "Large-Scale Structure".

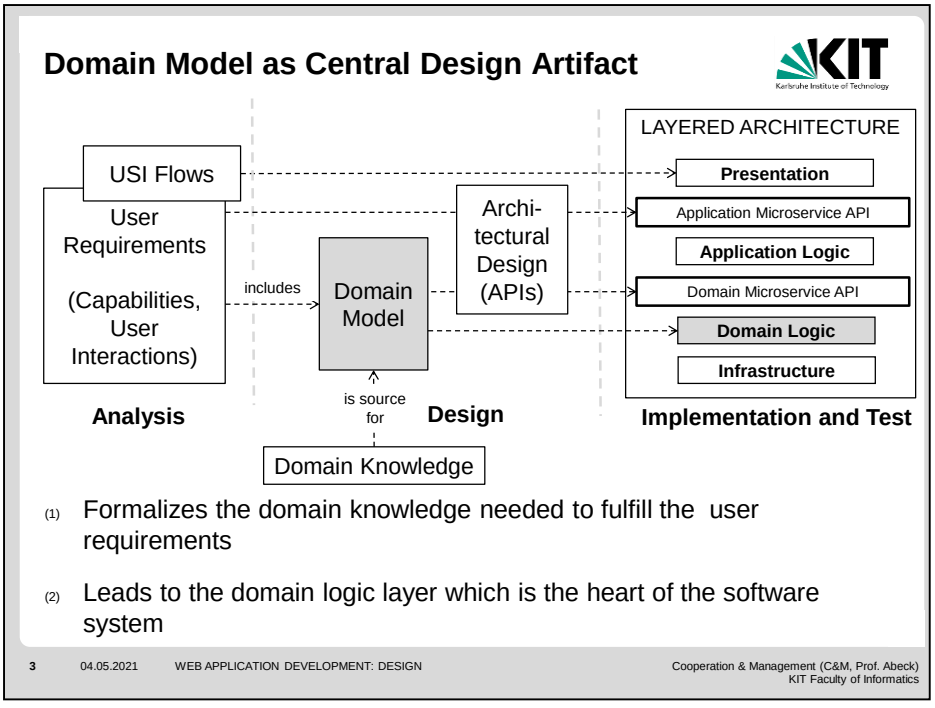
(1) (1.1) Most of the DDD patterns are not new, but a summary of existing modeling concepts such as object-oriented design (as e.g. described by Larman); responsibility-driven design (by Wirfs-Brock); contract by design (described by Meyer).

(2) The navigation map shows the most relevant relations between the patterns starting from overall DDD pattern.
The following explanations are mainly taken from the glossary.

(DOMAIN-DRIVEN DESIGN) A design in which some subset of software elements correspond closely to elements of a model. Also, a process of co-developing a model and an implementation that stay aligned with each other.
(LAYERED ARCHITECTURE) A technique for separating the concerns of a software system, isolating a domain layer, among other things.
(ENTITY) An object fundamentally defined not by its attributes, but by a thread of continuity and identity.
(SMART UI) A systematic approach for connecting the user interface (UI) to the application and domain layers (the most common pattern is the "Model View Controller", MVC). The Smart UI Pattern is an alternative, mutually exclusive fork in the road, incompatible with the approach of domain-driven design (since in a Smart UI the presentation layer (UI) and the business layer are explicitly not separated).
(SERVICE) An operation offered as an interface that stands alone in the model, with no encapsulated state.
(VALUE OBJECT) An object that describes some characteristics or attributes but carries no concept of identity.
(REPOSITORY) A mechanism for encapsulating storage, retrieval, and search behavior which emulates a set of objects.
(AGGREGATE) A cluster of associated objects that are treated as a unit for the purpose of data changes.
(FACTORY) A mechanism for encapsulating complex creation logic and abstracting the type of a created object for the sake of a client.

(Core) The following patterns besides MODEL-DRIVEN DESIGN can be seen as the core of DDD:
LAYERED ARCHITECTURE illustrates where in the software architecture the domain model is located.
ENTITIES and VALUE OBJECTS can be seen as the fundamental modeling elements of a domain model.

MVC	Model View Controller
UI	User Interface



To understand how domain modeling can be practically applied it is important to understand where it can be located in the overall software development process. The domain model is located in the design phase of a software development process. Since it is used to clarify specific aspects of the domain with a domain expert (not with the user) it also contributes to the analysis phase.

(LAYERED ARCHITECTURE) The logical layers of the architecture which includes the domain logic layer are defined by a DDD pattern introduced on the page ++Domain-Driven Design (DDD)++.

(Domain Knowledge, User Requirements) It is important to notice that the domain model has its origin in the domain and the knowledge contained in the domain. The domain model should not be derived from the user requirements which an application residing in the domain must fulfill. The domain model is included in the user requirements in the way that the application's business logic includes the application-agnostic domain logic (besides the application-specific) application logic.

(USI Flows, Presentation) On the way from the USI flows to the presentation logic, mockups can be designed in order to define the layout and UI/UX (User Interface / User eXperience) aspects of the user interface to be implemented.

(Architectural Design (APIs)) The domain model provides relevant contributions to the Domain Microservice APIs (Application Programming Interface) whereas the application microservice APIs are derived from the user/system interactions being part of the user requirements.

(1) The domain model concentrates on the domain knowledge that is needed to provide the functionality of the software system. A precise and semi-formal description as a domain model is the main challenge of the design of the software system.

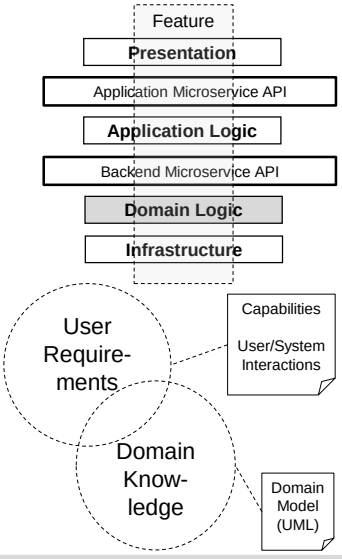
(2) Only one layer of the overall software architecture, the so-called domain logic layer is covered by the domain model. Since this layer can be seen as the heart of a software system it provides the kernel of the system. A major goal of domain modeling is to keep the model (of the domain) and implementation (of the domain logic layer) close together.

API	Application Programming Interface
UI/UX	User Interface / User eXperience

Logical Layers and Domain Knowledge

(1) Features specify the business logic of the software system
  (1) One part of the business logic is the domain logic which is application agnostic

(2) The following questions should be answered to derive the domain knowledge contained in a feature
  (1) What is known about the "relevant" domain objects or activities?
  (2) Is this knowledge domain-specific or is it application-specific?
  (3) Which (types of) relationships between the identified domain objects exist?



According to the overall BDD/DDD-based development process the domain logic contained in features should be taken from the domain model.

(1) A capability and including user/system interactions describs a specific part of the business logic from the viewpoint of a user who applies the functionality in a certain role and in certain situations.
(1.1) The second type of logic is the application logic. It is important to notice that in the DDD approach the business logic is divided into the domain logic and the application logic. The domain logic is application-agnostic which means that this logic is relevant for other applications of this domain. Therefore, the domain logic is the re-usable part of the business logic appearing in a feature.

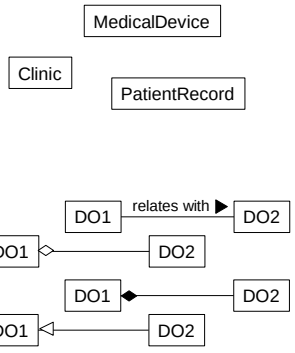
(2) Like the process of feature specification the derivation of domain knowledge from the features is a highly creative process. Nevertheless, some best practices do exist.
(2.1) By answering this question those terms of the feature should be identified that are relevant for the understanding of the user requirement.
(2.2) Only those aspects identified by Question 1 should be added to the domain model which are not specific to the regarded application.
(2.3) Three types of relationships can be distinguished: general associations, containment (composition, aggregation), inheritance

Modeling of Domain Elements

(1) Each bounded context is tactically modeled by domain objects

(2) DDD modeling elements are extended UML classes
  (1) UpperCamelCase is used as notation

(3) Types of relationships between domain objects
  (1) Association
  (2) Aggregation
  (3) Composition
  (4) Generalization



The context map is the main artifact of strategic modeling. It introduces the bounded contexts and their relationships. The tactical modeling starts from one bounded context by showing its structure and content.

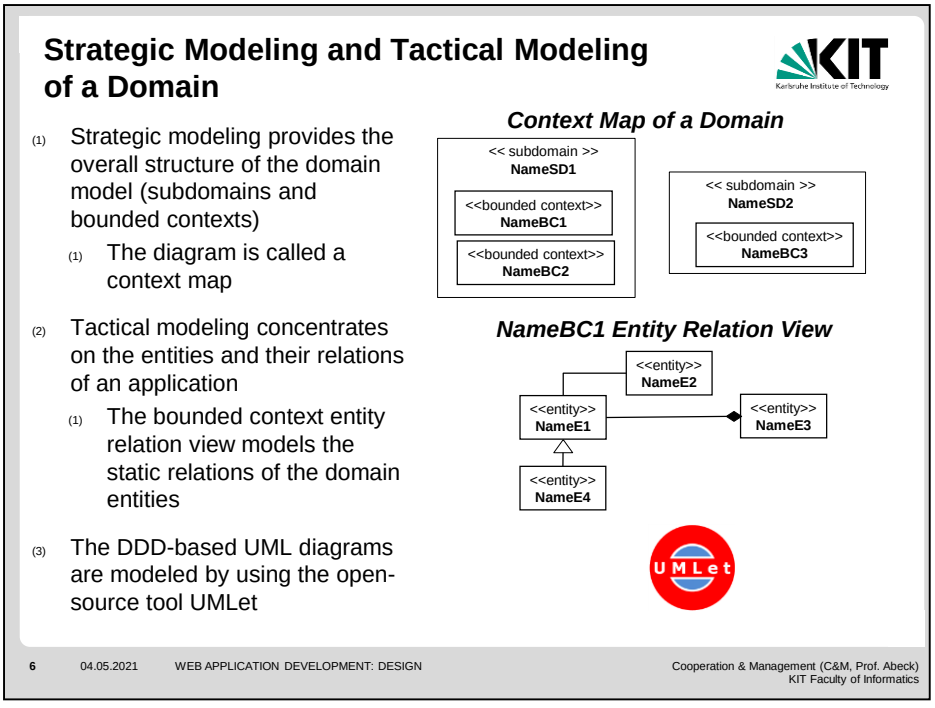
(1) In DDD, entities and value objects are the fundamental domain objects (see also the page ++Domain-Driven Design++).

(2) The graphical notation is a rectangle which expresses the stereotype <<class>>.
(2.1) This notation is also used for all other modeling elements, such as the structuring element of a package that has been introduced before.
(Clinic, MedicalDevice) These are examples of domain objects from the CAM domain.

(3) The following types of relationships are standardized by UML which can be used in a class diagram.
(3.1) The association is the most general relationship between two domain objects DO1 and DO2. The association can be named (e.g. defines, is responsible for, creates). An arrow indicates the direction of the association.
(3.2) The aggregation is a stronger version of association and it is used to indicate that a domain object DO1 contains another domain object DO2 [MH06:86]. The lifetimes of the containing domain object (DO1) and the contained domain object (DO2) do not depend on each other. An aggregation is modeled by using an empty diamond arrowhead.
(3.3) The composition is as the aggregation a containment relationship, i.e. domain object DO1 contains another domain object DO2 [MH06:86]. In contrast to an aggregation, the lifetime of the containing domain object (DO1) depends on the lifetimes of the contained domain object (DO2). This means that the contained domain object (DO2) cannot exist without the containing domain object (DO1). A composition is modeled by using a filled diamond arrowhead.
(3.4) Generalization which is otherwise known as inheritance is used to describe that a domain object DO1 is a generalization of a domain object DO2. Therefore, the sub domain object DO2 inherits all properties from the super domain object DO1. The generalization relationship can be denoted as "is a" or "is a type of". It is modeled by an empty generalization arrow.

DO Domain Object

[MH06] Russ Miles, Kim Hamilton: Learning UML2.0, O'Reilly, 2006.



The domain model emerges from two types of modeling, the strategic modeling and the tactical modeling.

(1) The strategic modeling is the real new part of domain modeling and the result, a so-called context map, provides the link to the microservice architecture.
(2.1) The structuring of the domain knowledge is carried out in two steps: The whole domain is structured into subdomains and the subdomains are structured into so-called bounded contexts. The term "bounded context" is derived from the concept of Domain-Driven Design (DDD). From the architectural perspective a bounded context can be seen as a microservice candidate. This is the reason why a domain model serves as a blueprint for the microservice architecture which implements the model in the domain logic layer of the layered architecture.
(2.2) DDD does not prescribe a concrete representation of the model. At C&M, the modeling elements from the Unified Modeling Language (UML) are used to model the results from the strategic and also from the tactical modeling.
(<<subdomain>> NameSD1) The subdomain is modeled as a standard UML package. "NameSD1" stands for the name of the subdomain formulated in UpperCamelCase.
(<<bounded context>> NameBC1) To be able to model a bounded context in UML a new stereotype <<bounded context>> is introduced. The UML modeling symbol of a package is used as graphical representation.

(2) The tactical modeling has similarities with the well-known proceeding of modeling knowledge as conceptual class diagrams.
(2.1) It consists of a diagram called the bounded context entity relation view by which the entities of a bounded context and their relationships are modeled.

(3) UMLet [UMLet] is a UML drawing tool which is easy to learn. The UML diagrams are stored in the UML eXchange (UFX) format. For the cooperation and exchange of the diagrams the corresponding subfolders of "5.SOFTWARE_DEVELOPMENT" on the C&M Teamserver are used. Further details can be found in [CM-T-UML]

UFX UML eXchange

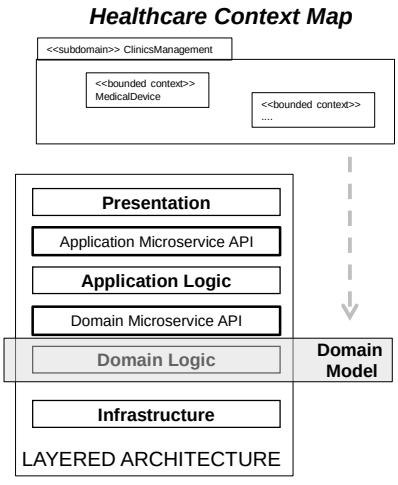
[CM-T-UML] Cooperation & Management: UMLet, Technical Documentation. https://team.kit.edu/sites/cm-tm/Mitglieder/2-3.WASA_Guidelines
[UMLet] Free UML Tool for Fast UML Diagrams. https://www.umlet.com/

Context Map Construction

(1) A context map prepares the domain knowledge in a way that it can be applied in a microservice architecture

(2) Domain knowledge is stored in a variety of sources
  (1) Goal: Subdomains and bounded contexts

(3) Evan's concept of domain-driven design does not support a formalization of the domain model which is necessary for its systematic use in an engineering approach

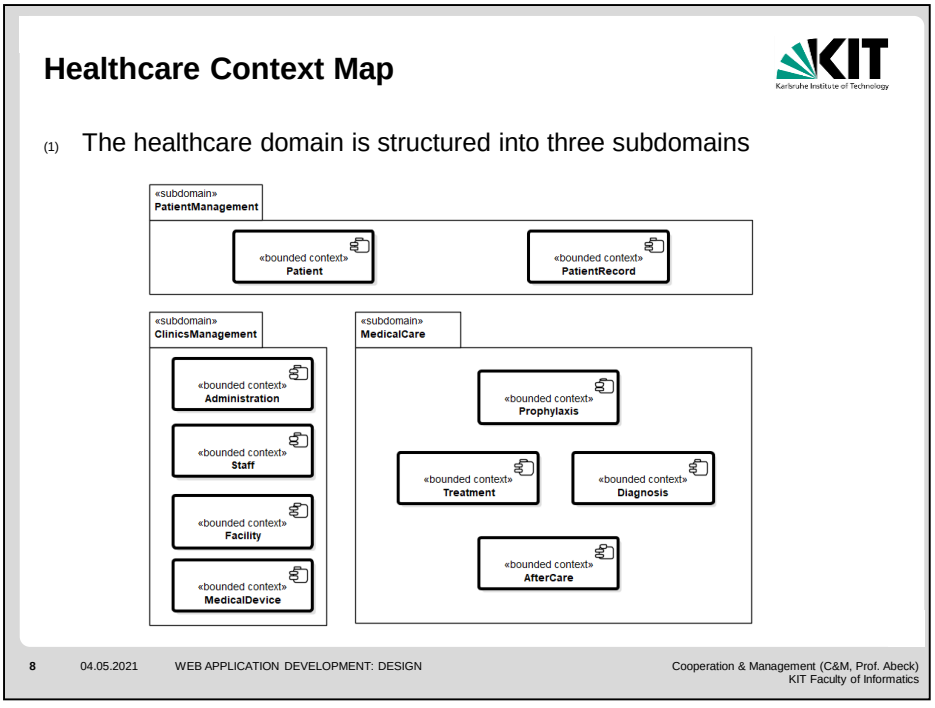


The context map is the central design artifact by which the approach of Domain-Driven Design (DDD) is applied in the microservice engineering process.

(1) The major goal of a context map is to capture the knowledge of a domain in a way that the domain logic layer is shaped according to the domain's structure.

(2) The main sources are: (i) domain experts, (ii) published sources, such as standards, white papers, publications.
(2.1) Subdomains allow a further structuring of the domain. The domain logic contained in each subdomain is expressed by bounded contexts. A bounded context can be seen as a candidate of a domain microservice in the microservice architecture.

(3) A formalization of the DDD approach and its design artifacts is absolutely necessary to be able to apply this approach in a systematic engineering process.



(1) The subdomains and their bounded contexts are described in the following.

(Patient Mangement) The Patient is the core of the healthcare activities, the central goal of the healthcare provider is to cure the Patient. This subdomain contains two bounded contexts Patient and PatientRecord. The bounded context Patient is the core entity of the subdomain and depict the Patient and his data itself such as name, health insurance data, age or sex. The bounded context PatientRecord contains historical medical treatments and the medical status of the patient.

(ClinicsManagement) One important part of the domain is the management of the components of the healthcare provider, e.g. staff or medical device. For example a healthcare provider is a hospital. Therefore, this subdomain covers administrative activities and important administrative entities in the healthcare domain. The bounded contexts Staff, Facility and MedicalDevice cover entities. The bounded context Administration covers administrative activities.

(MedicalCare) This subdomain can be seen as the center of the domain. It contains bounded contexts that provide medical activities to the patient in the healthcare domain. MedicalCare covers all medical activities: (i) The bounded context Prophylaxis includes processes to avoid an illness or injuries of patients, e.g. with medicaments. (ii) To get the right treatment plan the bounded context Diagnosis provides a diagnosis of the illness or injury of the patient. (iii) The bounded context Treatment could be seen as the core task of the healthcare provider. A patient needs a treatment to become cured from his illness or injury. After the treatment the patient needs to be checked and monitored to be sure that the treatment is successful which is covered by the bounded context AfterCare.

[CM-G-0Doc-Todo] Cooperation & Management: 0Doc_Healthcare, Git Respository. https://git.scc.kit.edu/cm-tm/cm-team/2-6.healthcare/0.doc_healthcare/-/blob/master/pages/ContextMap.md

EXERCISES DOMAIN MODELING

(1) What does DDD mean and what does it define?

(2) What is the relationship between the domain model and the layered architecture?

(3) How is a composition and an aggregation modeled and what is the difference of these two types of containment associations?

(4) What does strategic modeling mean and how is it related to microservice architectures?

(5) What is missing in the domain-driven design approach with respect to modeling?

(6) What is an example of a subdomain and an including bounded context of the healthcare context map?

(1) ++Domain-Driven Design++
- Domain-Driven Design
- Patterns, such as Layered Architecture, Entity, Shared Object

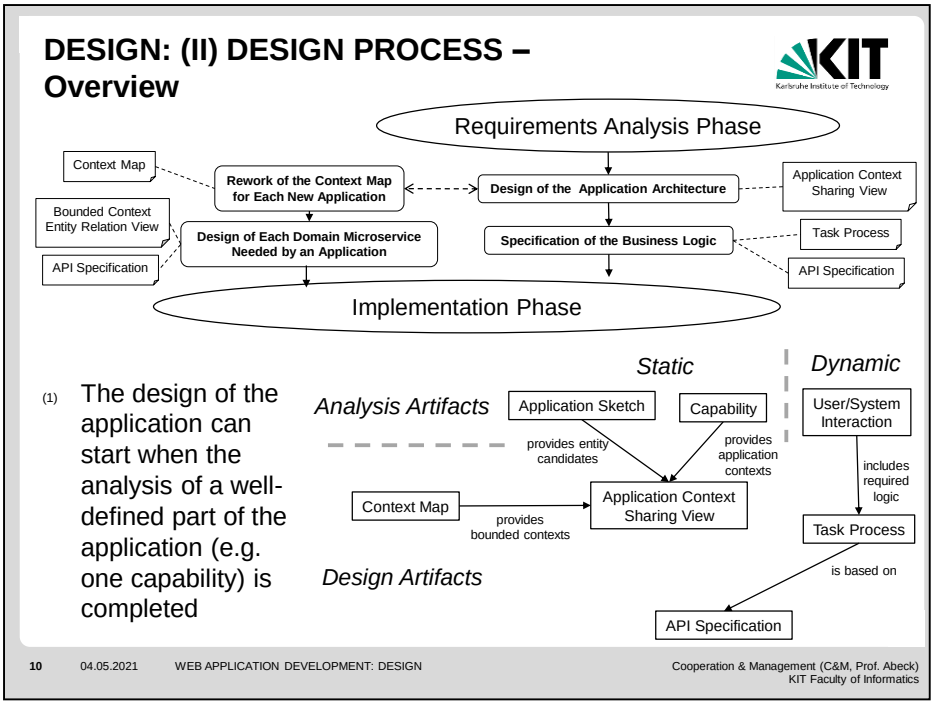
(2) ++Domain Model as Central Design Artifact++
- The domain model is located in the domain logic layer of the layered architecture.

(3) ++Modeling of Domain Elements++
- Composition: filled diamond arrowhead
- Aggregation: white diamond arrowhead
- Difference: In contrast to an aggregation, the lifetime of the containing domain object (DO1) depends on the lifetimes of the contained domain object (DO2)

(4) ++Tactical and Strategic Modeling++
- Strategic modeling provides the overall structure of the domain model by defining subdomains and bounded contexts.
- Relationship: Bounded contexts are the candidates for microservices.

(5) ++Context Map Construction++
- A formalization of the DDD artifacts (e.g. the context map)
- The formalization can be carried out by the modeling language UML and the introduction of UML profiles for the DDD modeling elements

(6) ++Healthcare Context Map++
(i) PatientManagement: Patient, PatientRecord
(ii) ClinicsManagement: Administration, ClinicsStaff, ClinicsFacility, MedicalDevice
(iii) MedicalCare: Prophylaxis, Treatment, Diagnosis, AfterCare



The whole microservice engineering process is introduced in a previous chapter INTRODUCTION of this course unit.

(1) The design phase is the second phase following after the requirements analysis phase. Since the microservice engineering process supports an agile approach, the analysis artifacts (ubiquitous language of the application, application sketch, capabilities, user/system interactions, USI flows) of a specific part of the application must be specified. After the design phase the implementation phase follows.

(Design of Each Domain Microservice Needed by an Application) In this approach, the domain model and the resulting domain microservices are not completely implemented before the development of applications. Instead, a domain microservice is designed and implemented as soon as a bounded context is part of the application sharing view of an application under development. This means that a "domain microservices on demand" approach is supported.

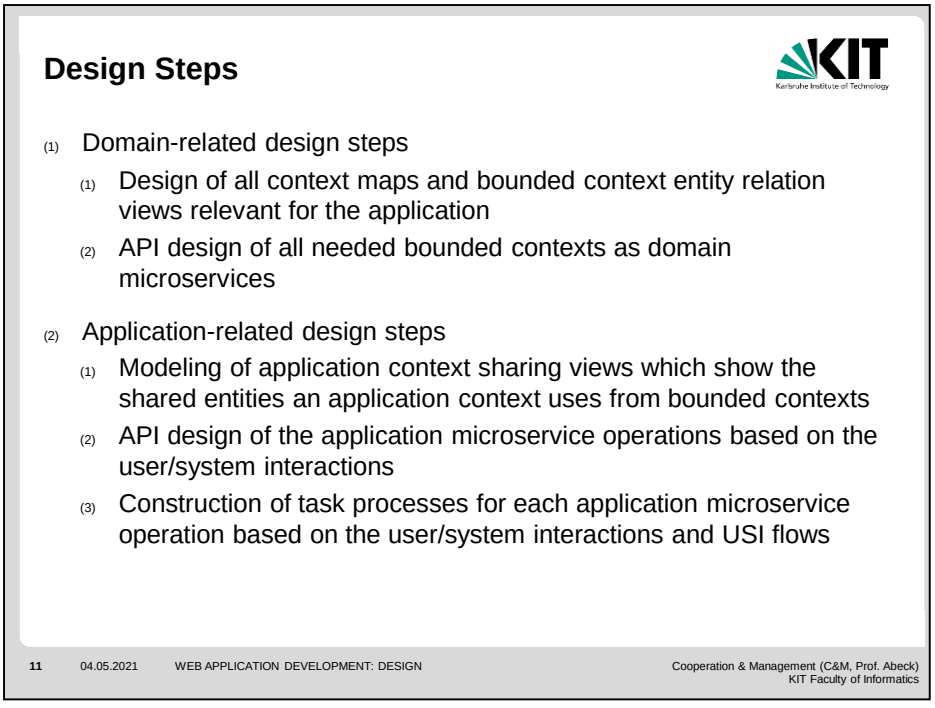
(Application Sketch, Capability, User/System Interaction) These are analysis artifacts built in the requirements analysis phase. Further details can be found in the chapter ANALYSIS of this course unit.

(Application Sketch, Application Context Entity Relation View) By the application context entity relation view the application entities and domain entities and their relation are modeled. The application sketch provides a valuable input since it contains these entities.

(Context Map, Application Sharing View) The application sharing view contains the application contexts and bounded contexts as well as the entities shared between them. The bounded contexts are derived from the context map and the application relation view. Each feature leads to an application context. A bounded context usually provides a domain entity to one or more application contexts.
An application sharing view models the static aspects of the application.

(User/System Interaction, Task Process, API Specification) By a task process the logic as it is textually specified in a user/system interaction is expressed by a sequence of tasks. A task in this context can be seen as a service request/response to/from a bounded context or application context. For this reason, a task description provides relevant input for the specification of the involved microservice APIs of the application.
A task process models the dynamic aspects of the application. The term "process" stems from the concept of the Business Process Execution Language (BPEL), a standard by which the orchestration of web services is defined.

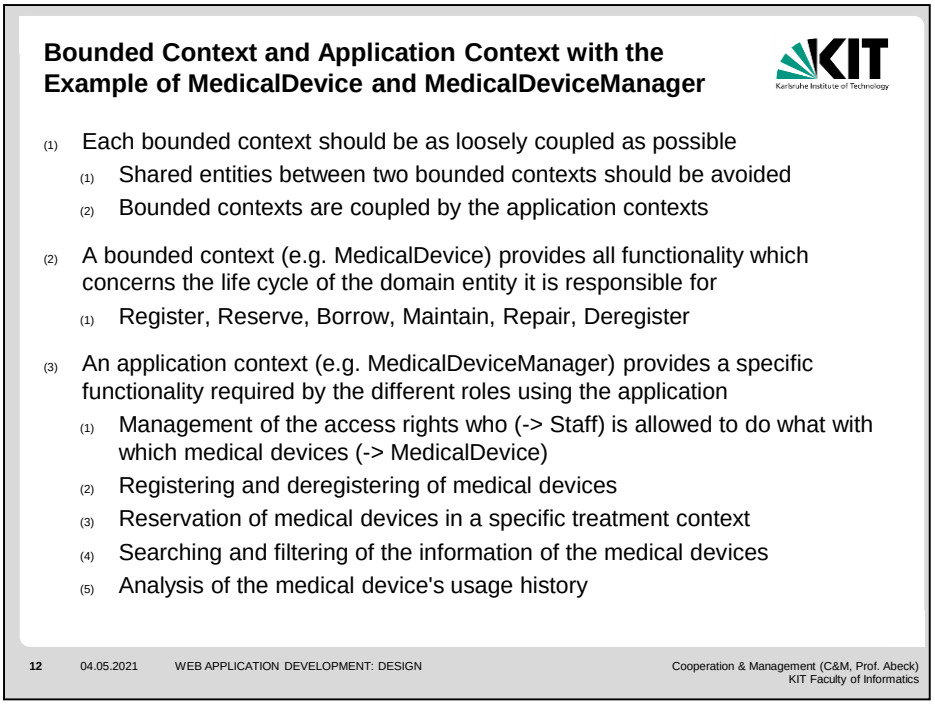
BPEL Business Process Execution Language



This page gives an overview of the sequence of the design steps of the microservice engineering process. In each step, specific design artifact are constructed. The design process can be divided into two parts, a domain-related and an application-related part.

(1) The domain-related design steps concern the context map and the bounded contexts (rsp. domain microservices) which are relevant for the application.
(1.1) The context map construction can be seen as a necessary preparing design step which must be carried out before the application development process can start. Only those context maps and bounded context entity relation views must be designed which are not yet available.
(1.2) This step of the domain development process realizes the idea of the "domain microservices on demand" approach which means that a bounded context being part of a context map is designed and implemented as a domain microservice only when an application under development needs this domain microservice. If the needed domain microservice is already available the application reuses it (if necessary, the domain microservice must be adapted to the application's needs).

(2) The application-related design steps especially concern the application microservices and their operations which are provided based on the underlying domain microservices.
(2.1) Each capability leads to an application context. The application context contains these application contexts and shows the relationship with the bounded contexts via the shared entities used by the application contexts.
(2.2) The API design is the most relevant design artifact for the following implementation phase, in which the application contexts are implemented as application microservices and the user/system interactions are implemented as application microservie operations.
(2.3) A task process specifies the application logic of one application microservice operation as an orchestration of domain microservices and application microservices. If necessary, a class diagram by which application-specific objects are modeled, can be introduced.



An overview of lessons learnt and best practices of the modeling of bounded and application contexts is presented with the example of the bounded context MedicalDevice and the application context MedicalDeviceManager..

(1) Loose coupling is directly related to the degree of reusability.
(1.1) As a consequence, a medical device will know nothing about its potential users who reserve and borrow the device.
(1.2) In the case of the medical device, the coupling of the staff and the medical devices should be transferred into the application context MedicalDeviceManager.

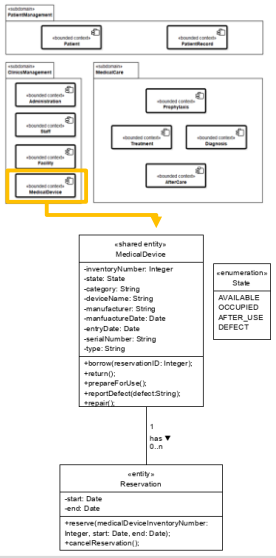
(2) (2.1) This functionality must be restricted to the entity itself. For example, a medical device is able to check if it is already reserved in a specific time period or not. But it is not able to check, if the requestor is allowed to do the reservation without loss of loose coupling.

(3) It is important to understand that the requirements on a bounded context and an application context are derived from completely different sources.
(3.1) The authorization aspect can be seen as a core aspect of the medical device management.
(3.3) The reservation of a medical device includes the authorization aspect. Therefore, even for the reservation of a single device, this functionality is more than the reservation functionality provided by the medical device itself.

Healthcare Context Map and MedicalDevice Entity Relation View

(1) The healthcare context map describes the central business domain of the CAM application
  (1) For CAM, the central bounded context is MedicalDevice

(2) By the bounded context entity relation view, the relationships of the entities and value objects being part of the bounded context are modeled
  (1) MedicalDevice as the central entity of this bounded context
  (2) Relationship with other bounded contexts should be omitted due to loose coupling



13 04.05.2021 WEB APPLICATION DEVELOPMENT: DESIGN Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

In the first step of the design phase, it must be checked if the relevant context maps and entity relation views of the bounded contexts used in the application are available. If this is not the case, they must be designed based on the DDD-based modeling approach followed by C&M.

(1) Since the CAM application is a healthcare application, the healthcare context map provides the domain logic for this application. An example of a cross-cutting domain is IoT (Internet of Things) which is further refined in another course unit [CM-W-Int].
(<<subdomain>> Patient Management ...) The context map is modeled with UML which is extended by the stereotypes <<subdomain>> and <<bounded context>> appearing in the context map (++Tool-Supported Domain Modeling++). The current version of the healthcare context map can be found in [CM-G-Hea_Con].
(2.1) The bounded context MedicalDevice is part of the subdomain ClinicsManagement. Another relevant bounded context is Staff since the clinic staff manages (i.e. create, reserve, use) the medical devices.

(2) While the context map is the result of the strategic modeling, the bounded context entity relation view is a design artifact of the tactical modeling.
(2.1) Usually, a bounded context includes an entity which has the same name as the central entity of this bounded context. Although the modeling elements have the same name (in this case, bounded context MedicalDevice and entity MedicalDevice) they must be well distinguished.
(2.2) In a former draft there was a relationship with the bounded context Staff (by introducing the <<shared entity>> Staff in the bounded context entity relation view). This was changed to decrease the coupling and increase the reuse of the bounded context's functionality.

(Reservation) It was decided that a reservation of a medical device should be part of the bounded context MedicalDevice. An alternative would have been to separate this functionality by introducing another bounded context. This was dismissed since the reservation functionality is restricted to medical devices which excludes a reuse of such a bounded context.

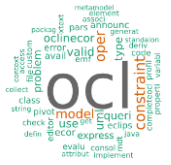
[CM-G-Hea_Con] Cooperation & Management: Healthcare Context Map, C&M GitLab, https://git.scc.kit.edu/cm-tm/cm-team/2-6.healthcare/1.domain/0.doc_healthcare/-/blob/master/pages/context_map.md
[CM-W-INT] Cooperation & Management: INTERNET OF THINGS, WASA Course Unit. https://team.kit.edu/sites/cm-tm/Mitglieder/2-2.WASA_Lecture

Domain Logic Specified by Constraints

(1) Constraints specify invariants of the attributes of the entities and pre/post conditions of operations as OCL expressions

(2) Invariants
  (1) Specify restrictions to attribute values
  (2) Example: Attribute "inventoryNumber" should not be empty and it should be unique.

(3) Pre/post conditions
  (1) Specify restrictions to the input-output relationship of a method
  (2) Example: Borrowing of a medical device




```
1.context MedicalDevice inv:
2.self.inventoryNumber -> notEmpty() implies
 not(exists(md:MedicalDevice | md.inventoryNumber
 = self.inventoryNumber))
3.self.state -> notEmpty()
4.self.manufactureDate -> notEmpty and
 self.entryDate -> notEmpty implies
 self.entryDate.after(self.manufactureDate)
```

```
1.context MedicalDevice::borrow(reservationID):
2.pre: self.state = AVAILABLE
3.     exists(r:Reservation | r.reservationID =
 reservationID and r.start.before(now()) and
 r.end.after(now()))
4.post: self.state = OCCUPIED
```


A bounded context entity relation view introduces all relevant domain entities and their relationships related to that bounded context. This provides a central part of the (static) domain logic which in the design process is extended by constraints. The constraints add to the entity relation view further both static and dynamic domain logic.

(1) The Object Constraint Language (OCL) [OMG-OCL] allows to specify constraints applying to UML models in a declarative way. The UML model in this case is the entity relation view. The constraints concern the attributes and the operations of the entities being part of this view.
The following examples of constraints concern the MedicalDevice bounded context. The latest versions of the OCL expressions can be found in the C&M GitLab [CM-G-Med].

(2) (2.1) An attribute invariant may refer to one or more or more attributes.
(2.2) (2. self.inventoryNumber -> ...) This line contains the OCL expression by which this invariant is expressed.
(4. self.manufactureDate) This is an example of an invariant in which two attributes, manufactureDate and entryDate, are involved.

(3) (3.1) The operation's pre-/post conditions can be seen as the core of the domain logic since they express the operational semantics of the entities' methods from the bounded context. The entities' methods will be implemented as (CRUD) operations of the domain microservice corresponding to this microservice.
(3.2) A medical device can only be borrowed if it is available (line 2.) and if the time of borrowing (which is given by now()) lies in the time of reservation. After the execution of this method the medical device is occupied (line 4.).

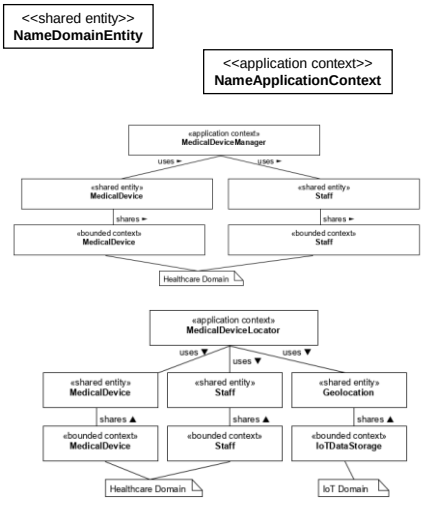
[CM-G-Med] Cooperation & Management: MedicalDevice Domain Constraints, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/2-6.healthcare/1.domain/medicaldevice/-/blob/master/pages/domain_constraints.md
[OMG-OCL] Object Management Group: Object Constraint Language, Tech. rep., URL https://www.omg.org/spec/OCL/2.4 https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/1.SoftwareEngineering

Application Context Sharing Views

(1) Application contexts are based on the domain entities shared by bounded contexts

(2) Sharing views a modeled for each application context

(3) The CAM application consists of two sharing views
  (1) MedicalDeviceManager is concerend with the entities MedicalDevice and Staff
  (2) MedicalDeviceLocator additionally requires an entity Geolocation provided by the IoT domain



15 04.05.2021 WEB APPLICATION DEVELOPMENT: DESIGN Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

In the second part of the design process the application contexts providing the (non-application-agnostic, i.e. application-specific) application logic are designed.

(1) An application contexts uses shared entities and the methods on these entities. These methods provide the domain logic which is (re-)used by application residing in this domain.

(2) From the sharing views of all application contexts of an application an overall application sharing view can be transferred. From the modeling viewpoint, a sharing view is a UML component diagram extended by specific DDD elements, esp. <<bounded context>>, <<shared entity>>, and <<application context>>.

(3) For each capability and application context, a sharing view has to be designed.
(3.1) The central task of the CAM application context MedicalDeviceManager is to support the staff to manage the medical devices. Therefore, this application context combines (or orchestrates) the two bounded contexts MedicalDevice and Staff from the Healthcare domain.
(3.2) The application context covers the specific aspect of the location of the medical devices. Since the functionality by which the devices are located is provided by the IoT domain, a bounded context IoTDataStorage from this domain appears in the sharing view of the MedicalDeviceLocator.

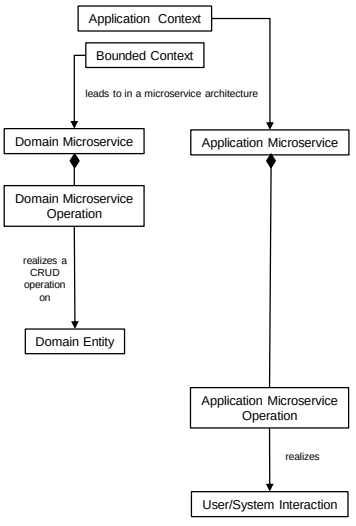
IoT Internet of Things

API Specification Construction

(1) A bounded context is realized by a domain microservice and an application context is realized by an application microservice

(2) The API of a domain microservice is defined independently from any application
  (1) Microservice operations usually are Create, Read, Update, Delete (CRUD) on the domain entity

(3) The API of an application microservice is derived from the application's analysis and design artifacts
  (1) Each user/system interaction leads to a microservice operation



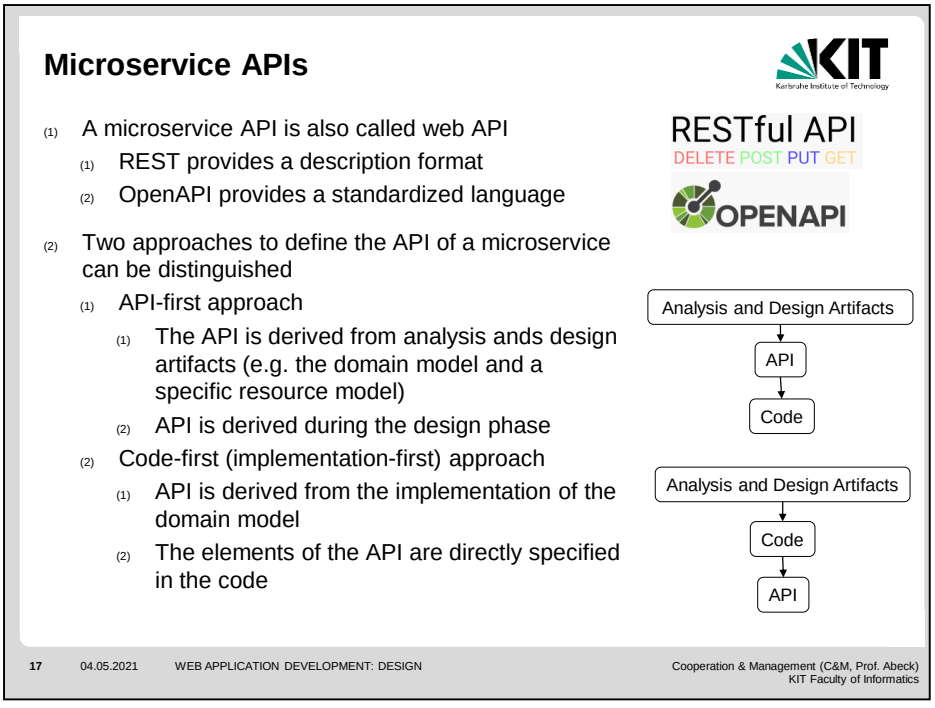
At this point of the design process, all artifacts are available to derive and design the API specifications.

(1) Although the microservice architecture is the best fit to the DDD-based approach, the bounded and application contexts might also be mapped to a different software architecture concept.

(2) The design of the domain microservice only considers aspects of the domain which makes it independent from and agnostic of any application.
(2.1) This means that domain microservices usually are data-centric.

(3) (3.1) The most relevant artifact for the construction of the application microservices are the capabilities and the including user/system interactions which define the application microservice operations.

CRUD Create, Read, Update, Delete



(1) API is an acronym of Application Programming Interface. This term already existed before the microservice era. An API, in general, describes an interface that a software application provides in a way that it can be integrated into another software program. In the the context of microservices the API is accessible via the web. Therefore, the term web API is often used instead of microservice API.
(1.1) (1.2) The REpresentational State Transfer (REST) and OpenAPI are two important concepts used for the definition and description of web API that re used in todays web applications. Both will be introduced on the succeeding pages.

(2.1) By following the API-first approach, the API is defined manually using an API editor (e.g. Swagger) during modeling. This is especially useful to get a grasp on how to map the domain to the API and to establish a contract between the teams, on which they can rely during implementation. Due to the limited insight of the domain before the actual implementation, several changes to the API will be necessary, but can be coordinated through updated API files. After the domain model is created, the API is specified and can be seen as contract between the frontend and backend teams.
(2.1.1) A systematic derivation process of a resource model from the domain model is presented in [Gi18].
(2.1.2) In the API-first approach, no implementation is carried out before the API is specified. Therefore, all actions of this approach take place in the design phase.
(2.2) The code-first approach or Domain-Implementation-First approach requires that first the domain model must be implemented before the API is specified.
(2.2.1) Therefore, the API specification is part of the implementation phase. Nevertheless, the API specification itself is still a design artifact.
(2.2.2) The specification of the API is done via annotations by which the different parts of an API, such as requests, responses and status codes are defined and integrated into the code.

REST REpresentational State Transfer

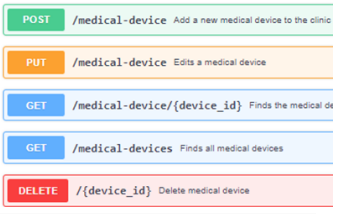
[Gi18] Pascal Giessler: Domänengetriebener Entwurf von ressourcenorientierten Microservices, PhD Thesis, Karlsruhe Institute of Technology, Germany, 2018. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/2.MicroservicesAPIs

REST (REpresentational State Transfer)

(1) REST is a lightweight alternative to the web service standards (esp. WSDL)
- (1) Addressable resources
- (2) Unified restricted service interface
- (3) Stateless communication
- (4) **RE**presentation-oriented
- (5) Format-driven **S**tate **T**ransfer -> hypermedia concept

(2) REST Application Programming Interface (API)
- (1) A contract between a service user and the service
- (2) Enables the access to data or functions

Roy T. Fielding: Architectural Styles and the Design of Network-based Software Architectures, PhD Thesis, **2000**.



The origin of the architectural principle called the REpresentational State Transfer (REST) is the PhD thesis [Fi00] from Roy Fielding.

(1) The core of REST is to provide a concept how a microservice API can be structured. REST replaces the Web Service Description Language (WSDL) which was used to specify web services. In contrast to WSDL, REST is a lightweight approach and, therefore, much easier to apply for software developers.
(1.1) Resources are the main concept and the fundamental abstraction of information in REST. Each resource is addressed by a Uniform Resource Locator (URL).
(1.2) Only a small set of well-defined method to manipulate the resources is made available.
(1.3) Stateless means that the server does not hold any session information. Therefore the client must transfer all information in the request that the server needs to process this request which leads to a loss of performance. Advantage of stateless communication are an increase of scalability and robustness.
(1.4) The representation principle is strongly related to the resources. Each resource is represented by different formats according to the platform on which the resource is stored. Examples of formats and corresponding platforms are: HTML-Browser; JSON – JavaScript application; XML – Java application. Different formats used in Internet protocols are defined by the Multipurpose Internet Mail Extensions (MIME) standard.
(1.5) The state transitions are driven by the format of the resources. This principle is related to hypermedia and the so-called "Hypermedia as the engine of application state" (HATEOAS). In this context the concept of hypermedia (especially the hypermedia links) is used to control the state transitions of an application.

(2) Although REST is defined independently from HTTP it is usually based on this Internet application protocol.
(2.1) (2.2) HTTP operations (GET, POST, HEAD, PUT, DELETE, ...) and the URL which identifies the resource are the main parts of a REST API.

(GET /MedicalDevice ...) This is an example of a REST API of the backend microservice being part of the CAM application.

HATEOAS	Hypermedia As The Engine Of Application State
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
URL	Uniform Resource Locator
WSDL	Web Service Description Language

(Fi00] Roy T. Fielding.: Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, Dissertation, 2000. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/2.MicroservicesAPIs

OpenAPI

(1) OpenAPI is a specification language for REST APIs
  (1) Worked out and published by the OpenAPI Initiative
  (2) Strong support by many IT / Internet companies
  (3) Originated from the Swagger specification

(2) OpenAPI 2.0 specifications are JSON or YAML documents
  (1) Meta information in "info"
  (2) Endpoints and methods in "paths"
  (3) Reuse of parts of the specifications in "definitions"

```
1.  openapi: 3.0.0
2.  info:
3.     title: ...
4.     contact:
5.        name: ...
6.        ...
7.  paths:
8.    /buildings:
9.      get:
10.        description
11.   ...
12.   responses: ...
13.   ...
14. definitions: ...
```

19 04.05.2021 WEB APPLICATION DEVELOPMENT: DESIGN Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

OpenAPI is one of the most broadly accepted API description formats.

(1) The OpenAPI Specification is a manufacturer-independent defacto standard.
Examples of other description formats are API Blueprint or RESTful API Modeling Language (RAML).
(1.1) The OpenAPI Initiative was founded in 2015 as an open source project under the Linux Foundation [OAI-FAQ]. A major driver was SmartBear which donated the Swagger Specification to the OpenAPI Initiative
(1.2) Examples of such companies are Google, Microsoft, IBM, Apigee.

(2) JSON and YAML are both markup language which are based on the same concept (lists and scalars). Each JSON document is a valid YAML document. The first line of the document (i.e. swagger: '2.0') defines that this is an OpenAPI 2.0 specification.
(2.1) In the "info" JSON/YAML object, metadata of the OpenAPI specification can be found, such as the "title" or the "contact" person of the service. All information is described by a cascaded list.
(2.2) In "paths" the (relative) URLs of the endpoints and the HTTP methods are indicated. In "responses" the results including the HTTP status codes are described.
(2.3) Reusable parts are objects that consist of a set of properties. To be able to reuse the object a name is defined which is use as a reference.

JSON	JavaScript Object Notation
RAML	RESTful API Modeling Language
YAML	YAML Ain’t Markup Language

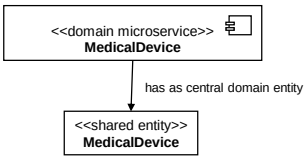
[OAI-FAQ] Open API Initiative: FAQ. https://team.kit.edu/sites/cm-tm/Mitglieder/3-4.Literature/2.MicroservicesAPIs

API Specification of Domain Microservices

(1) The API of a domain microservice specifies the CRUD operations of its central domain entity

(2) Domain entities can appear in multiple instances
   (1) Read operations (GET) for all entities and one specific entity
   (2) Specific entity is identified by /{id}

(3) The API is specified by using the Swagger tool and the OpenAPI standard



Endpunkt	Operation	Input	Output
/medical-device	GET	-	Medical Device List
/medical-device	POST	Medical Device	-
/medical-device/{id}	GET	ID	Medical Device
/medical-device/{id}	PUT	ID	-
/medical-device/{id}	DELETE	ID	-

swagger

20 04.05.2021 WEB APPLICATION DEVELOPMENT: DESIGN Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

Since the APIs of a domain microservice and an application microservice are different, they are separately introduced starting with the domain microservices.

(1) According to the design principle applied for domain microservices, each such microservice centers around a central domain entity. Usually, the domain microservice and its central domain entity have the same name, as this is the case in the example of a MedicalDevice.

(2) It must be distinguished if the domain entity can only appear once (i.e. the cardinality is "0...1") or several times ("0...n"). In the example of medical devices, multiple instances can appear.

(3) The API is based on the REST principles by which each microservice operation is specified as an HTTP operation (e.g. GET, POST) and the resource is addressed by a URL (Unified Resource Locator), e.g. /MedicalDevice or /MedicalDevice/{id}.

URL Uniform Resource Locator

MedicalDevice OpenAPI Specification

```
1.    openapi: 3.0.0
2.    info:
3.      description: "REST API for the MedicalDevice DMS"
4.      title: "MedicalDevice Microservice API"
5.      ...
6.    paths:
7.    /medical-device:
8.      post:
9.      summary: "Add a new medical device to the clinic "
10.     description: " The deviceId of the medical device
                      is set by the server."
11.     operationId: " addMedicalDevice "
12.     requestBody:
13.      description: "Medical device object to be added."
14.     required: true
15.     content:
16.       application/json:
17.         schema:
18.           $ref: "#/components/schemas/MedicalDevice "
19.     responses:
20.       '200':
21.         description: "Successful operation, returns
                        medical device with deviceId "
22.         content:
23.           application/json:
24.             schema:
25.               $ref '#/components/schemas/MedicalDevice'
26.       '400':
27.         description: "Invalid medical device values"
```

(1) "info" provides general information on the API

(2) "paths" describe the resources (endpoints) of the API

(3) "post" is an operation provided at the path

(4) "operationId" identifies an operation uniquely

(5) "requestBody" and "responses" describe the operation

(6) "$ref" defines input and output types which are (re-) used in the operations

The YAML document of the MedicalDevice OpenAPI specification is based on Version 3.0.0.

(1) (2. info) The "info" part contains general information (such as title, description, contact) that describe the MedicalDevice API.

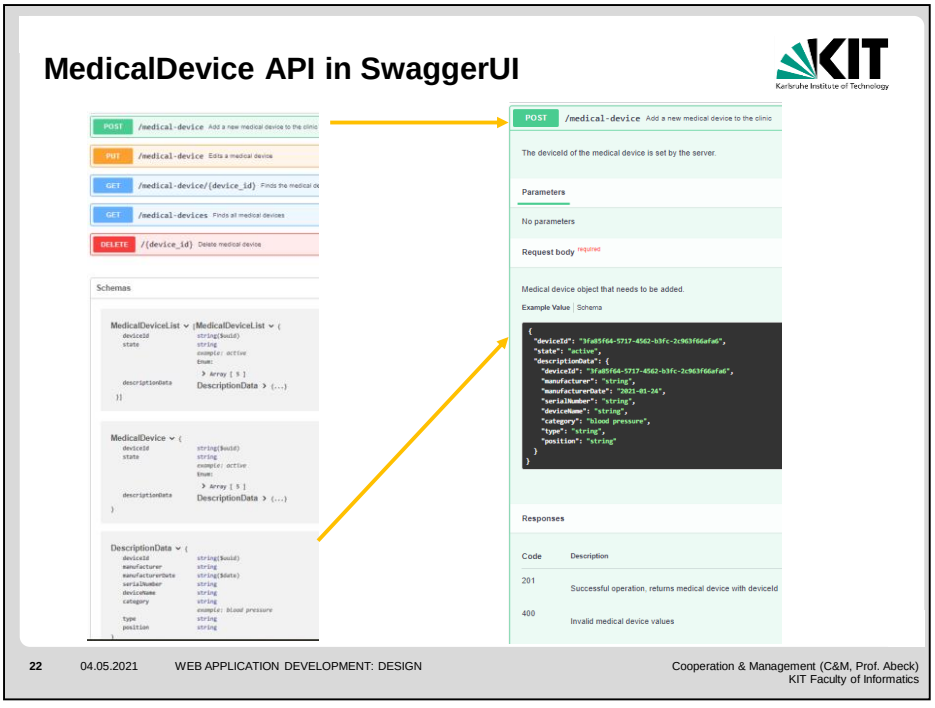
(2) (6. paths:) An endpoint corresponds to a resource that the API exposes.
(8.) In the example, the endpoint of the MedicalDevice API is "/medical-device". This endpoint is a relative URL which is appended to a base URL.

(3) (8. post:) A single path can support multiple operations. In the example, the operation POST /medical-device is described.

(4) (11. operationId: addMedicalDevice) This is the names of the corresponding method in the backend code. The (optional) operationIds can be automatically generated, i.e. when the OpenAPI specification is generated from the code.

(5) (12. requestBody) (19. responses) In these parts of the "paths" object, the request and the different (success and error) responses of the POST operation are described. In the example a successful (20. '200') and an error (26. '400') response are defined.

(6) "$ref" is a reference to the "definitions" part of the OpenAPI specification where the different data structure related to the domain objects (in this case, MedicalDevice) are defined.
(18. 25. $ref: : "#/components/schemas/MedicalDevice') The MedicalDevice schema is used both in the request and the succesful response of the POST operation.



For the specification of the REST-based API of the domain microservice MedicalDevice the tool Swagger was used. The page shows presentations from the tool SwaggerUI.

(POST /medical-device) This HTTP POST operation of the API creates a medical device. When clicking on a list element representing an operation, details of this operation is presented. The screen dump on the right hand side shows the details of this API operation.

(The deviceID of ...) An alternative would be to pass the deviceId as a parameter in the request body of the POST operation.

(Request body) In the request the parameter values of the medical device to be added are defined. The set of parameters and their structure is prescribed by the schemas shown on the left side of the slide.

(deviceId, manufacturer, ..., type, position) By these parameters a medical device is described.

(Responses) Two response codes are defined:
(201) When this code is returned the medical device was successfully created.
(400) A problem occurred and no medical-device was added.

REST API Specification of Application Microservices

(1) Application microservice APIs are derived from the user/system interactions

(2) Parts of the API specification
  (1) The application microservice has the same name as the application context
  (2) Resource named by the user/system interaction (USI)
  (3) HTTP operation POST by which the user/system interaction is called
  (4) Parameters derived from the user/system interaction

(3) The mapping of USI to REST is critical because of a missing resource-orientation
  (1) gRPC provides the better alternative

Register a Medical Device
A new medical devices is entered in the system. Therefore, the medical staff registers **the name, category, manufacturer, type, serial number)of the new medical device**.

/medical-device-manager
/register-a-medical-device
/POST

```
paths:
  /medicalDeviceManager/create-a-medical-device:
    post:
      summary: Create a new Medical Device
      parameters:
        - in: "MedicalDevice"
          description: "Medical Device object that needs to be added"
          required: true
          schema:
            $ref: '#/components/schemas/MedicalDevice'
```

RESTful API
DELETE POST PUT GET

gRPC

Application microservices contain the application logic which is defined by the user/system interactions. An application microservice provides the functionality needed to carry out a user/system interaction. Therefore, the API specifies the input and output of the user/system interactions which are defined by the application context the application microservice is derived from. The systematic derivation of API specifications of application microservices will be demonstrated with the example of the CAM application.

(1) User/system interactions are analysis artifacts which are derived from a capability (which is another analysis artifact). The systematic approach of defining capabilities and deriving user/system interactions from these capabilities is described in the chapter ANALYSIS (esp. page "Capability Description and Derived User Interactions").

(2) One important goal with respect to the API specification of application microservices is to adopt the structure introduced by the existing analysis and design artifacts.
(2.1) In the example, the application context is MedicalDeviceManager since the user/system interaction "Create a Medical Device" is part of the capability "Management of Medical Device". This leads to the microservice name medical-device-manager.
(2.2) The resource introduced for each user/system interaction can be seen as a control resource. Accordingly, in the example the resource is named create-a-medical-device.
(2.3) Control resources do not provide CRUD operations as the resources representing entities do. They only provide the operation PUT by which the functionality related to the user/system interaction is invoked.
(2.4) In the example, the parameters can be found in the user/system interaction description: ... enters the description data (name, category, manufacturer, type, serial number) of the medical device ...

(3) The resulting operation from the user/system interaction is no a CRUD operation on one entity, but a function with parameters and a result.
(3.1) The function-orientation of the application microservice operations have the characteristic of a Remote Procedure Call (RPC). This makes gRPC the better alternative for the specification of these operations.

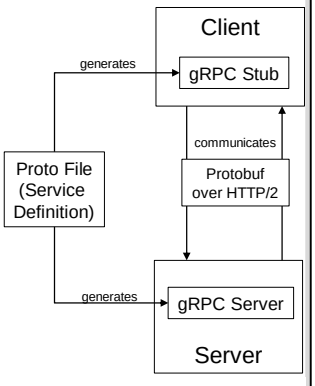
CRUD	Create, Read, Update, Delete
gRPC	gRPC Remote Procedure Call

gRPC (gRPC Remote Procedure Call)

(1) An open source RPC framework published by Google in 2015
  (1) Procedure/method/function-oriented
  (2) Opposed to REST not resource-oriented

(2) Protocol buffer (protobuf) is used as the technology to define the service interface and the structure of payload messages
  (1) Requires HTTP/2
  (2) Described in a proto file

(3) Client- and server-side code are generated from the proto file
  (1) gRPC stub supports the RPCs calls
  (2) gRPC server provides a skeleton to be extended with functionality



24 04.05.2021 WASA Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

gRPC (gRPC Remote Procedure Call, i.e. a recursive acronym) is an alternative to REST to design and implement APIs of microservices.

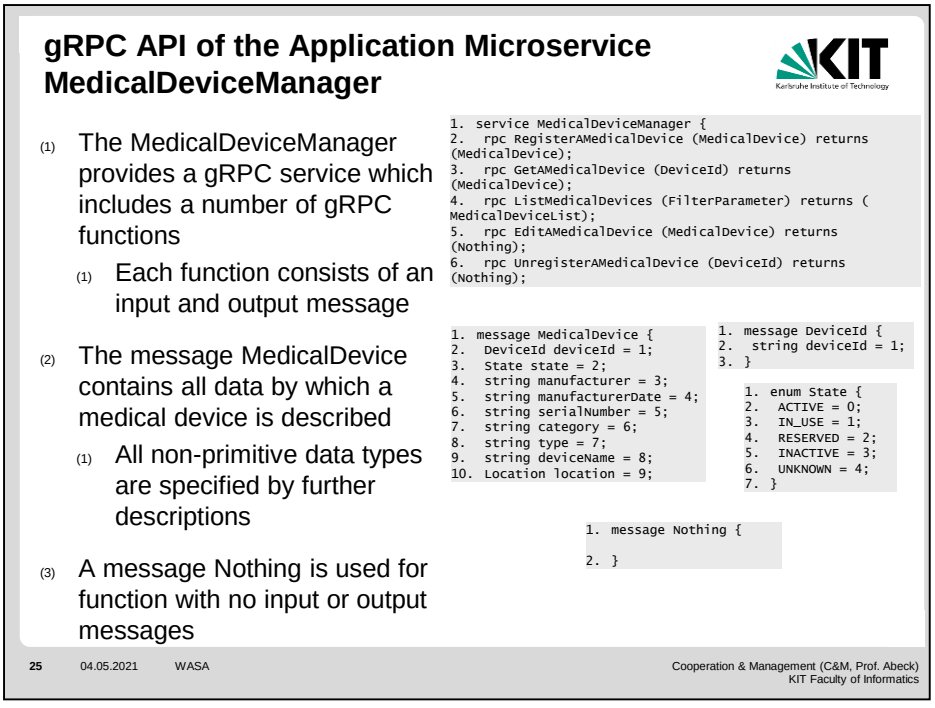
(1) About 15 years after Roy Fielding has published in his PhD thesis the REST paradigm to design APIs, Google developed an alternative which is called gRPC.
(1.1) gRPC is based on the idea of defining a service, specifying the methods (or procedures, functions) that can be called remotely with their parameters and return types [LNX-gRPC-Con].

(2) Protocol buffer, or protobuf, is a programming language-independent, platform-neutral, and extensible mechanism for the serialization of structured messages. In gRPC, protobuf serves as Interface Definition Language (IDL) for the description of both the service interface and the structure of the payload messages. In comparison to JSON, protobuf is much more efficient and faster.
(2.1) A main reason why gRPC requires HTTP/2 is the streaming nature of this standardized protocol.
(2.2) The proto file contains the provided functions and the related input and output messages as logical records, each of which contains name-value pairs.

(3) Protobuf libraries for many different languages (e.g. Go, Java, C++, Python) exist to auto-generate source code which can be used in a program to create structured data and reading and writing it to input and output streams. The generated code consists of strongly typed objects for the messages exchanged between client and server.
(3.1) To call a procedure remotely, the client gRPC stub provides local function calls written in the used programming language.
(3.2) The skeleton is strongly typed base class which a developer extends by the actual implementation.

gRPC	gRPC Remote Procedure Call
IDL	Interface Definition Language

[LNX-gRPC-Con] The Linux Foundation: Core concepts, architecture and lifecycle. https://grpc.io/docs/what-is-grpc/core-concepts/



The gRPC approach is shown with the example of an application microservice from the CAM application. In concrete, the gRPC API of the application microservice MedicalDeviceManager described by a proto file is illustrated.

(1) The name of the gRPC service is "MedicalDeviceManager" (line 1.). The five functions follow in lines 2 to 6.
(1.1) The structure of a gRPC function is as follows:
rpc (<Input Message>) returns (<Output Message>)
The input and output messages consist of data types which are specified in the proto file using the protobuf format.

(2) In the example, a medical device description contains nine "attributes" which are called fields in gRPC. Each field is declared as follows:
<type declaration> <field name> = <field number>;
Field numbers stick with the field and they ensure backward compatibility if a field number used in a released stub is never reused [Con-Int].
In gRPC, primitive data types (e.g. string or int32) are provided.
(2.1) Non-primitive data types are DeviceId (line 2), State (line 3), and Location (line 10).
(1. message DeviceId {) For the device id of the medical device, a separate message object was introduced to increase the readability.
(1. enum State {) The state of a medical device is described by an enumeration "enum" which restricts the attribute value to one of the five states from ACTIVE (line 2.) to UNKNOWN (line 6.). An enum is not a message.

(3) Each gRPC function must consist of an input and output message. Therefore, the Nothing message is a workaround for functions which have no input or no output message.

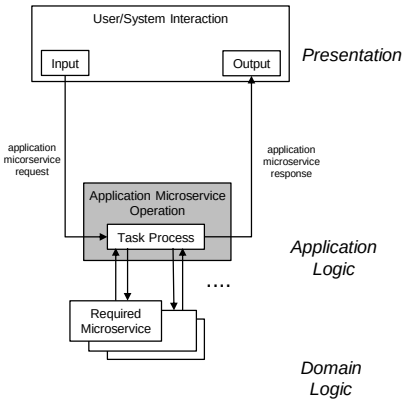
[Sm17] Jason Smith: Introduction to gRPC, Container Solutions, 2017. https://blog.container-solutions.com/introduction-to-grpc

Task Process

(1) Task processes define the business logic of the application

(2) A task process belongs to one application microservice operation and defines the application logic related to one user/system interaction

(3) The task process itself consists of a chain of service calls and in between lying processing steps
   (1) Inspired by the concepts BPEL and SoaML



The task process is a design artifact which is concerned with the specification of the dynamic aspects of the application.

(1) According to the DDD pattern LAYERED ARCHITECTURE, the business logic is separated into two parts, the application logic and the domain logic.

(2) By this relation, the task process is assigned a defined place in the application's architecture. A task process specifies the business logic the application must have in order to provide the functionality of a user/system interaction.

(3) The service calls may concern both application contexts and bounded contexts.

(4) The task process diagram is based on the concepts of the Business Process Execution Language (BPEL) [Ju06] and the Service-oriented architecture Modeling Language (SoaML) [EB+11].

(application microservice request / response) The start and the result of the service chain of the task process are derived from the user/system interaction.
Remark: This only holds for user/system interactions with a defined input and output.

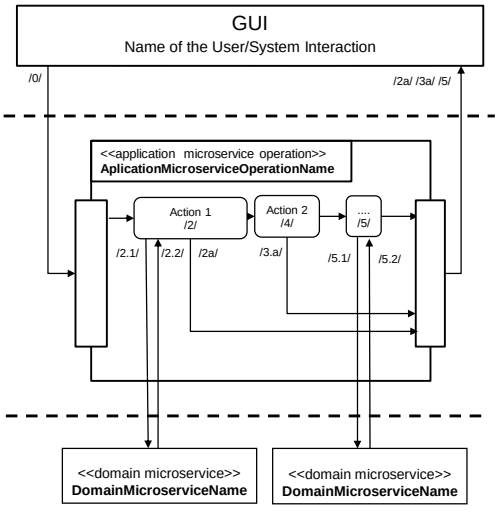
BPEL	Business Process Execution Language
SoaML	Service-oriented architecture Modeling Language

[EB+11] Elvesæter, A.-J. Berre, and A. Sadovykh, “Specifying services using the service oriented architecture modeling language (soaml) - a baseline for specification of cloud-based services,” in CLOSER, 2011, pp. 276–285.
[Ju06] M. B. Juric, “A hands-on introduction to bpel,” Oracle (white paper), p. 21, 2006.

Modeling of a Task Process

(1) A task process describes the flow of actions between the frontend and the involved domain microservices to carry out a user/system interaction

(2) Relevant existing artifacts
- (1) Textual description of the user/system interaction
- (2) API specification of the application microservice operation
- (3) API specifications of the domain microservices needed to carry out the tasks



On this page, the diagram that describes a task process is introduced.

(1) The task process diagram is a dynamic diagram which is based on the logical microservice architecture and its components (GUI and microservices). The tasks are internal processing steps which can make use of domain microservice calls to carry out this task.

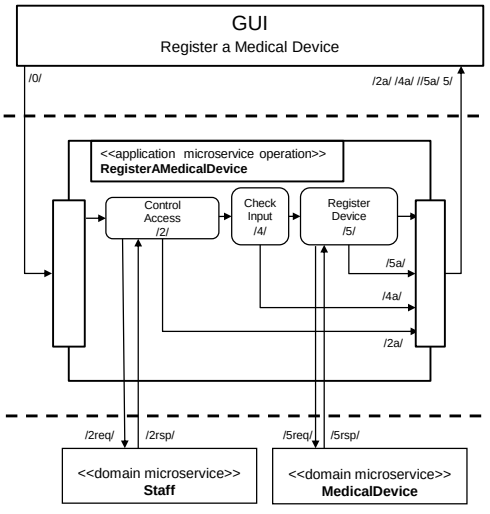
(2) A systematic and consistent derivation of the task processes from the existing artifacts is of high importance.
(2.1) The textual description of a user/system interaction is the main source of the task process. The numbering used to express the sequential flow of the actions inside the task process is taken from the numbering used in the user/system interaction.
Remark: This is the reason why certain numbers might be omitted. In the example on the right hand side these are the numbers /1/ and /4/.
(2.2) The request (which in our approach is a gRPC input message) is always marked with the number /0/ in a task process.
(2.3) A task process can only be constructed when all API specifications of the domain microservices (which in our approach are REST operations) used by the application microservice operation.

GUI	Graphical User Interface
gRPC	gRPC Remote Procedure Call
REST	REpresentational State Transfer
S	System
U	User

Example of the CAM Task Process "Register a Medical Device"

User/System Interaction
Register a Medical Device

1. U (technical staff): Navigates to and starts the registration process
2. S (CAM application): Checks if the user is allowed to carry out this function
   2a. S: Aborts interaction if not allowed
3. U: Inputs the information of the new medical device; these are: name, category, manufacturer, type, serial number
   3a. U: Cancels the interaction
4. S: Checks the correctness of the information
   4a. S: Outputs an error message which help the user to do the corrections (-> 3.)
5. S: Creates a new entry of the medical device including a generated unique id
   5a. S: Outputs an error message if the creation fails
6. S: Presents the entry information to the user



28 04.05.2021 WEB APPLICATION DEVELOPMENT: DESIGN Cooperation & Management (C&M, Prof. Abeck) KIT Faculty of Informatics

The diagram describing a task process is explained with the example of the user/system interaction "Register a Medical Device" as it was introduced in the chapter ANALYSIS.

(GUI Register a Medical Device) The Graphical User Interface (GUI) contains all interface elements that are needed to carry out the user/system interaction. In addition, presentation logic provides the interworking of these elements and the communication with the application microservice via its API.

(/0/) The application microservice operation is called by the gRPC function RegisterAMedicalDevice which was introduced on the page ++gRPC API of the Application Microservice MedicalDeviceManager++.

(1. U ...) This first step of the user/system interaction has no effect on the task process. Therefore, a number /1/ is missing in the task process.

(2. S ...) This step results in the task Control Access /2/.
/2.1/ /2.2/ REST operation provided by the domain microservice Staff
/2a/ Error message (access not allowed) of the gRPC function

(4. S: ...) The task Check Input /4/ makes checks which are not included in the checks carried out by the domain microservice.
/4a/ Error message (error in medical device information) of the gRPC function

(5. S: ...) A successful call /5req/ /5rsp/ of the task Register Device to the domain microservice MedicalDevice leads to a successful output message /5/ of the application microservice operation RegisterAMedicalDevice. In the case of an error an error output message /5a/ is the result.

To be discussed:
(1) So far, the interaction between a task process and the GUI is restricted to the start and the end (which is a characteristic of the underlying RPC concept). But, often an interaction with the GUI as part of an action of the process is needed. In the example, this is the case for the task Check Input: When an error is detected the user should be informed to make corrections. In the current solution, the whole task process stops and an error message is sent to the presentation layer.
(2) The task Check Access has a strong relationship with IAM which is not yet considered in the current draft. An IAM-aware solution must consider externalized authorization based on policies and API management/gateways.

req	request
rsp	response

EXERCISES DESIGN PROCESS

(1) How are OCL constraints used to specify domain logic?

(2) To which type of microservice is (i) a bounded context, (ii) an application context mapped?

(3) How is (i) a domain microservice operation, (ii) an application microservice operation realized?

(4) Which two approaches for the API derivation of a microservice exist and what are main differences of the approaches?

(5) Which are the main concepts of REST?

(6) How are REST and OpenAPI related?

(7) How are the REST properties which concern the resources and the service interface (or operations) are expressed in OpenAPI?

(8) What is an example of a gRPC function?

(9) What does a task process describe and how is it related to domain microservices?

(1) ++Domain Logic Specified by Constraints++
- Two types of constraints are used: (i) Invariants: specify restrictions to attribute values, (ii) Pre-/post conditions: specify restrictions to the input- output relationship of a method

(2) ++API Specification Construction++
(i) A bounded context is mapped to a domain microservice
(ii) An application context is mapped to an application microservice

(3) ++API Specification Construction++
(5.1) A domain microservice operation realizes a CRUD operation on a domain entity.
(5.2) An application microservice operation realizes a user/system interaction.

(4) ++Definition of a Microservice API++
- API-First approach and Domain-Implementation-First (or Code-First) approach
- Main differences:
-- Sequence of API specification and implementation (API-First: specification before implementation; Code-First: specification after implementation)
-- Assignment of the API artifact to a development phase (API-First: design phase; Code-First: implementation phase)

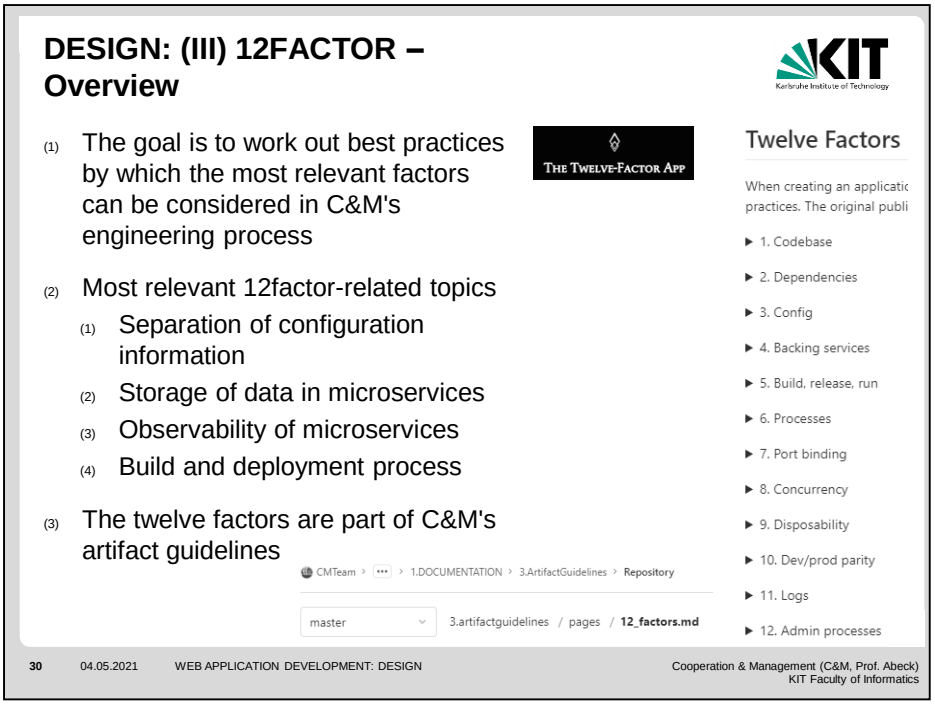
(5) ++REST (REpresentational State Transfer)++
(i) Addressable resources, (ii) Unified restricted service interface, (iii) Stateless communication, (iv) REpresentation-oriented, (v) Format-driven State Transfer -> hypermedia concept

(6) ++OpenAPI++
- OpenAPI is a specification language for APIs defined based on the REST approach

(7) ++MedicalDevice OpenAPI Specification++
- REST property resources: /paths/medical-device
- REST property operations: get: responses: ; post parameters: responses:

(8) ++gRPC API of the Application Microservice MedicalDeviceManager++
- A gRPC service is defined as a function with an input message and an output message.
- Example: rpc RegisterAMedicalDevice (MedicalDevice) returns (MedicalDevice);
-- Function: RegisterAMedicalDevice
-- Input message: MedicalDevice
-- Output message: MedicalDevice

(9) ++Task Process Construction++
- The business logic of an application microservice operation
- An operation of a domain microservice can be called as part of a task



The Twelve-Factor App (12factor) methodology provides an excellent basis for developing and operating (DevOps) applications as a service. A challenge is to defined and prepare the factors in a way that a DevOps-aware software engineer can apply them in his/her engineering process. The factors have an influence on the implementation, deployment, and operations process which are part of the C&M engineering process.

(1) The best practices should make clear to the C&M members how to fulfill the most relevant factors. In this concrete context best practice means: Which artifacts should be created or adapted in which state of the C&M engineering process by using which DevOps tools?

(2) This list of topics is to be discussed.
(3.1) – (3.4) For each topic it is important to work out concrete best-practices (as defined in (1)) which can be applied in the C&M microservice engineering process.

(2) In the C&M guidelines the twelfe factors are classified into the implementation and test phase [CM-G-Twe], Though the 12factor methodology mainly concerns the implementation, deployment, and operation, there are also strong relationships with the design phase and the design artifacts.

12factor Twelve-Factor App

[CM-G-Twe] Cooperation & Management: Twelve Factors. C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation/3.artifactguidelines/-/blob/master/pages/12_factors.md

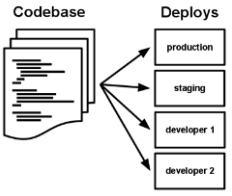
Factors 1 to 4 of The Twelve-Factor App

(1) Codebase: One codebase
  (1) Each application part (backend microservices, BFF microservices, frontend) has exactly one own repository

(2) Dependencies: Explicit declaration and isolation
  (1) In the Java/Spring context the dependencies are declared in a Maven "pom.xml" file

(3) Config: Store configuration in environment variables
  (1) Future use of Helm charts

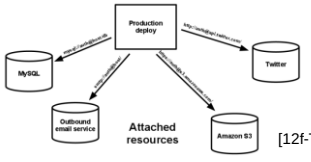
(4) Backing services: Treat backing services as attached resources
  (1) Access of a specific service via a URL



[12f-Twe]

	Sample File	Profiles	Spring Config Server	Helm Chart
Versioned	No	Yes	Yes	Yes
Secure	No	No	Yes (using external secrets backend)	Yes
Framework-agnostic	No	No	No	Yes
Online Reconfiguration	No	No	Yes	No
Kubernetes-Native	No	No	No	Yes

[Qu20]



[12f-Twe]

There are some recommendations for implementing, designing and running microservices which are collected under the term "Twelve-Factor App" [12f-Twe]. In the following, the term "application" is equivalent to a Software-as-a-Service (SaaS) application.Each factor is shortly introduced and, in the sub-item (i.e. (1.1), (1.2), ...), applied to the C&M engineering process [Qu20].

(1) There is a one-to-one relationship between a codebase and an application. Multiple code bases represent a distributed system. Each component of a distributed system should comply with the twelve-factor rules. If an application shares the same code, then it is a violation. The shared source code should be extracted into libraries and then included via dependencies.
(1.1) This factor can be considered as applied. Furthermore, at C&M, there is at the current time only one deployment per service.

(2) The required dependencies should be declared explicitly via a declaration manifest. The dependencies should not be included as binary from the same codebase.
(2.1) At C&M, the microservices are developed using the Java programming language and the Spring framework. Their dependencies are explicitly declared in a "pom.xml" file, which is processed by the package manager Maven.

(3) Configuration is everything that can be changed between different deployments. For instance, configurations could be database configurations, credentials, connection settings to external systems. Instead of bundling into an application, it should be set via environment variables.
(3.1) In fomer development of microservice-based systems, configuration information (e.g. the URL of some database server) was hard coded into the microservice code. As further elaborated in [Qu20], external services should be set by an external configuration based on e.g. environment variables. Helm Charts are a suited concept which make use of environment variables. This allow multiple deployments of a single codebase.
(Table) The table shows the different alternatives of storing configuration information (columns: Sample File, ..., Helm Chart) and criteria (rows: Versioned, ..., Kubernetes-Native) which are fulfilled or not by these alternatives. Since Helm charts are the best choice to obtain a cloud-native solution, it is applied in the C&M microservice engineering process. In short, Helm charts consist of templates which define the skeleton of the application deployment for Kubernetes with placeholder fields. These placeholder fields are the filled using the defined values.

(4) By treating backing services as attached resources, loose coupling is increased. Each attached resource can be swapped at any time without any code changes. Only the corresponding configurations need to be changed via environment variables.
(4.1) The access of a specific service (e.g., a service provided by a database server) mainly appears in the application microservices. There is a strong relationship with the (3rd) factor Config since this factor is only fulfilled when the URL is not hard-coded, but part of the configuration.

SaaS Software-as-a-Service

[Qu20] Mohamad Qattan: Development and Scaling of a Kubernetes-Based Infrastructure Using Machine Learning Methods, Master Thesis, Karlsruhe Institute of Technology (KIT), C&M (Prof. Abeck), 2020.
[12f-Twe] 12factor: Twelve-Factor App. *https://12factor.net/*

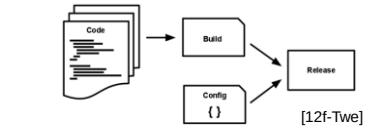
Factors 5 to 8 of The Twelve-Factor App

(5) Build, release, run: Strict separation of stages
  (1) Experimenting with different versioning and release creation approaches

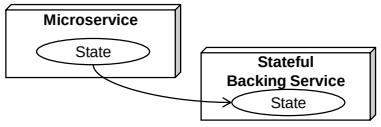
(6) Processes: Execute the app as one or more stateless processes
  (1) Embedded in-memory database must be replaced by a standalone database

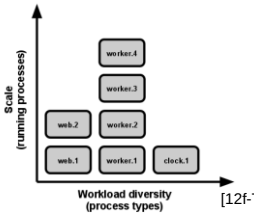
(7) Port binding: Export services via binding
  (1) Used technology can be easily configured to support port binding

(8) Concurrency: Scale out via the process model
  (1) Each mircroservice developed by C&M should cover only one aspect



[12f-Twe]





[12f-Twe]

(5) With the build stage, the source code will be transformed into an executable artifact known as a build. In the following release stage, the executable artificat is linked to the corresponding deployment configuration. Finally, in the run stage a process for starting the application is executed.
(5.1) The separation of the (3rd) factor Config is a necessary prerequisite to separate the build and release stages. At C&M, the build and release stages currently are not separated. In order to fulfill this factor, two problems must be solved: (i) the versioning of the built images and (ii) the merging of the build artifacts and the configuration specified by Helm charts.

(6) The requirement of this factor is that no data gets lost when the process crashes. Any information that needs to be saved must be stored in a stateful backing service, such as a database. No disk space or memory is shared between running applications (only via attached resources).
(6.1) In former microservice applications, C&M took the approach to persist the data in an embedded in-memory database. This violates the (6th) factor Processes since all data is lost when the process crashes.

(7) Each application is self-contained and, therefore, is shipped with a web server that can handle HTTP requests. Only the exposed port needs to be bound with a port on the host system so that a connection can be established.
(7.1) Examples of such technology are the Spring Boot framework and the Tomcat server which can be configured to expose the service using a specific port (e.g. port 8080).

(8) By this factor an application can be easily horizontally scaled. The share-nothing nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation. By supporting this factor, the application is able to handle various workloads. For long-running processes, for example, a dedicated worker process should be used while for handling HTTP requests a web process should be used. This means that the developer architects their app to handle diverse workloads by assigning each type of work to a process type.
(8.1) This factor is not yet fulfilled by all microservices developed by C&M. An example is a charging microservice which does not only provide the functionality of charging an e-car at a charging stations, but also payment aspects.

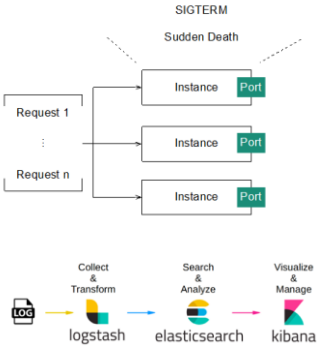
Factors 9 to 12 of The Twelve-Factor App

(9) Disposability: Maximize robustness with fast startup and graceful shutdown
- (9) Assumes that the (6th) factor Processes is fulfilled

(10) Dev/prod parity: Keep stages as similar as possible
- (1) Lack of multiple environments

(11) Logs: Treat logs as event streams
- (1) Logs are part of the observability interface used by applications

(12) Admin processes: Run admin/management tasks as one-off processes
- (1) No admin processes yet to be done



[Ela-ABr]

(9) The application processes should require a minimum of startup time so that in the case of a crash the application is available again as quickly as possible. Furthermore, the application should be able to react to signals of the process manager, so that for example no new requests are accepted. Also, graceful shutdowns should be handled accordingly so that it does not result in further issues.
(Figure) The basic concept can be mapped on distinct instances (processes) running on one machine or on different machines. When a SIGTERM (i.e. a signal to terminate the process) appears, the process should terminate in a way that it does not leave any lost requests.
(9.1) In order to make use of disposability the processes need to operate in stateless manner.

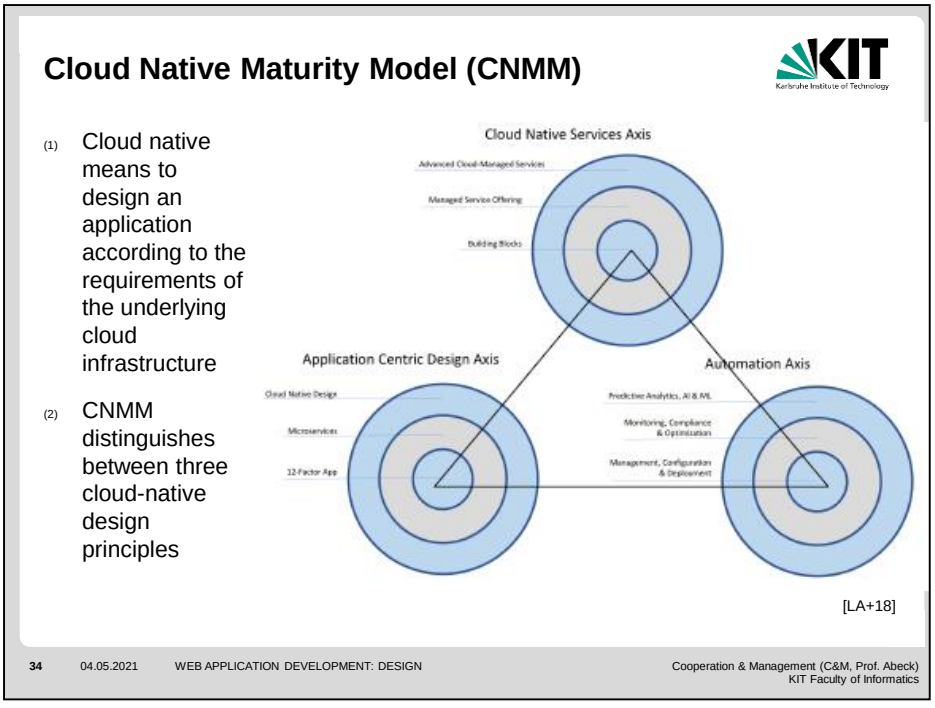
(10) Each deployment stage should be as similar as possible to minimize the gap between development and production. Furthermore, it reduces the risk of unexpected behavior between the different stages.
(10.1) This factor will become relevant as soon as solutions developed in the C&M development environment should be transferred to the production environment of a cooperation partner.

(11) The application should never be responsible to store any logs or manage log files. Instead of this, it should be forwarded to a dedicated output stream for further processing.
(Figure) Popular elastic stack (ELK stack) that is often used for central log management.
(11.1) C&M's approach is to use an observability interface by which observability functionality consisting of monitoring, logging, and tracing are separated from the business logic.

(12) Administration tasks needed for the application's operation (such as database migrations or backing up) should be designed as a one-off process. Properties of such one-off admin processes are:
(i) Started and executed only one time by the developer
(ii) Run in the same environment as the application process
(iii) No dependencies with the system environment
(iv) Same tooling as used alongside the application development
(12.1) In the C&M environment, this factor can only be handled after the other factors are implemented.

ELK Elasticsearch, Logstash, Kibana

[Ela-ABr] Elastic Stack—A Brief Introduction. *https://hackernoon.com/elastic-stack-a-brief-introduction-794bc7ff7d4f*



The goal of the Cloud Native Maturity Model (CNMM) [LA+18] is to evaluate exiting software systems and their underlying architecture according to their maturity to be cloud native. It consists of three axes each divided into three maturity levels.

(1) Cloud native is strongly related with the term of cloud computing which, according to Amazon Web Services (AWS), is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with a pay-as-you-go-service.

(2) In CNMM, the design principles are called axes. According to the model (see figure on the right-hand side) three axes are distinguished by the model.

(Cloud Native Service Axis) Covers the components of the cloud native architecture, especially the building blocks (e.g. storage, compute, networking, monitoring) and the cloud services (e.g. databases, directory services, load balancers, data warehouses). A good understanding of these services is necessary since a cloud-native application uses the services intensively.

(Application Centric Design Axis) Is about how the application itself will be designed and architected. This includes twelve-factor app design patterns, microservices, and native design patterns (e.g. instrumentation, security, parallelization, resiliency, event-driven).

(Automation Axis) Concerns the operational automation which is often referred to as infrastructure as code. By applying this concept, operations teams can focus on the application-specific design and rely on the cloud vendor to handle the deployment of resources. Depending on the degree of automation, artificial intelligence and machine learning techniques can be applied to self-heal self-adapt (e.g. by autoscaling) the infrastructure.

AWS	Amazon Web Services
CNMM	Cloud Native Maturity Model

[LA+18] Erik Farr Tom Laszewski, Kamal Arora, Piyum Zonooz: Cloud Native Architectures: Design High-Availability and Cost-Effective Applications for the Cloud, Packt Publishing, ISBN 1787280543, 2018.

https://books.google.de/books?id=QshsDwAAQBAJ&printsec=frontcover&hl=de#v=onepage&q&f=false

EXERCISES 12FACTOR

(1) Name one typical topic covered by The Twelve-Factor App and explain its relationship with the DevOps approach

(2) What are examples of configuration information and how should this information be stored?

(3) Which property must a microservice (running as a process) have to fulfill the 6th factor "Processes"?

(4) How should an application treat logs?

(5) What is the goal of the Cloud Native Maturity Model (CNMM) and how is it structured?

(6) To which axis of the CNMM does the Twelve Factors App belong?

(1) ++DevOps and The Twelve-Factor App ++
(i) Separation of configuration information -> Separation of build and release stage
(ii) Storage of data in microservices -> Statelessness as a prerequisite for robustness / disposability
(iii) Observability of microservices -> Needed for elasticity (scaling), stability, and other Ops goals
(iv) Build and deployment process -> Bridge between Dev and Ops

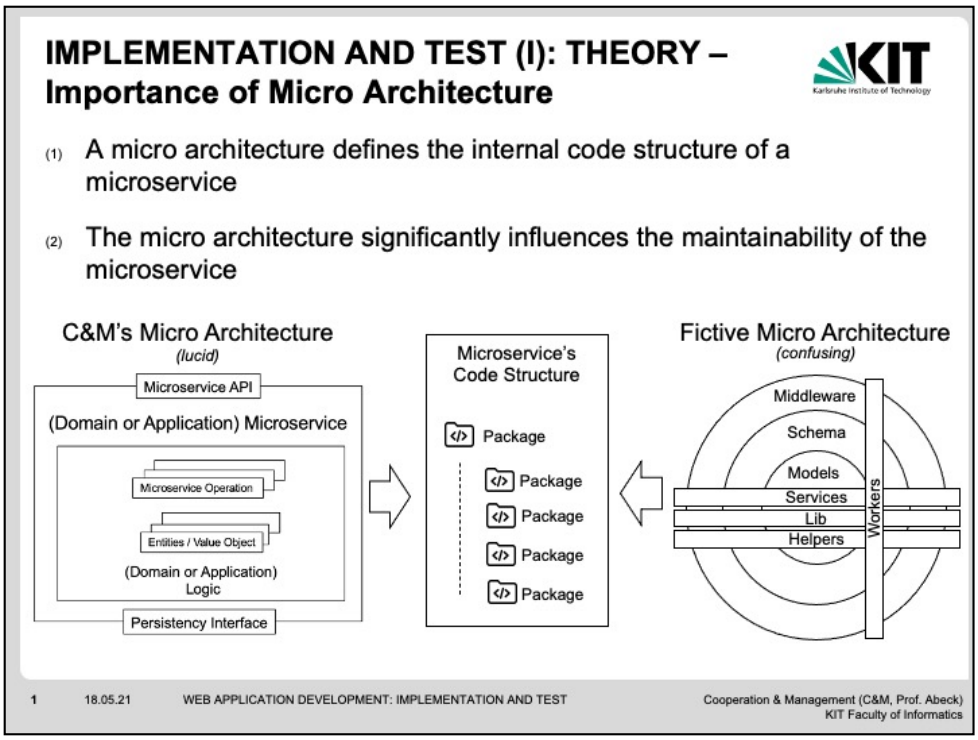
(2) ++Factors 1 to 4 of The Twelve-Factor App++
- Examples of configuration information is: database configurations, credentials, connection settings to external systems
- Config info should be stored externally in environment variables
- Solutions are: Sample files, profiles, Helm charts (consist of templates which define the skeleton of the deployment for K8s with placeholder fields)

(3) ++Factors 5 to 8 of The Twelve-Factor App++
- The requirement of this factor is that no data gets lost when the process crashes.
- To fulfill this the microservice (rsp. the process which executes the microservice) must be stateless.

(4) ++Factors 9 to 12 of The Twelve-Factor App++
- The application should never be responsible to store any logs or manage log files. Instead of this, it should be forwarded to a dedicated output stream for further processing.

(5) ++Cloud Native Maturity Model (CNMM)++
- The goal of the Cloud Native Maturity Model (CNMM) is to evaluate exiting software systems and underlying architecture according to their maturity to be cloud native
- It is structured into three types of design principles which are called axes: (i) Cloud Native Service Axis, (ii) Application Centric Design Axis, (iii) Automation Axis

(6) ++Cloud Native Maturity Model (CNMM)++
- Application Centric Design Axis
- The twelve-factor app is one design pattern besides microservices and patterns of good native design.



(1) Implementation focuses on converting the artifacts of the analysis and design phase into source code. The implementation offers the possibility to implement these artifacts in different micro architectures, which specifies the internal code structure of the microservice. As soon as one relies on the functional and object-oriented programming language paradigms, the micro architecture is mapped in the form of packages.

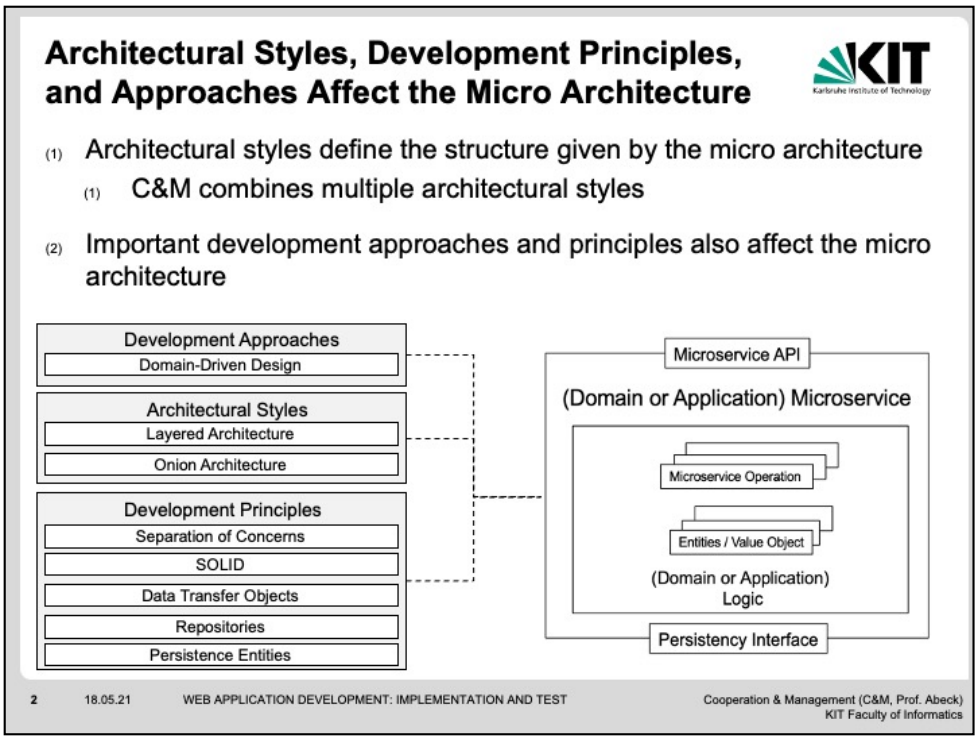
(2) Depending on the complexity of a micro architecture, the further development and maintenance of a microservice is easier or more difficult for software developers. This is where the fundamental principle of SoC comes into play. This principle states that the microservice should be broken down into individual components that take care of exactly one concern. Components here are packages, classes, attributes and functions. If there is a micro architecture that provides for a strict separation of these concerns, the software developer is guided in the implementation.

(C&M's Micro Architecture) The micro architecture designed by C&M provides for a clear separation of concerns. For example, domain logic and application logic are not mixed. These are even divided into their own microservices. The few layers of the micro architecture make it clear and easy to understand.

(Fictive Micro Architecture) The depicted fictitious micro architecture exhibits a high degree of complexity in that it has many different layers that cannot be directly assigned to one concern. For example, "Schema" and "Models" define data structures for the microservice.

(Microservice's Code Structure) Based on the chosen micro architecture, the microservice includes the various packages. A complex micro architecture results in a correspondingly large number of different packages. Consequently, a simple and lucid micro architecture leads to a simple code structure.

SoC Separation of Concerns



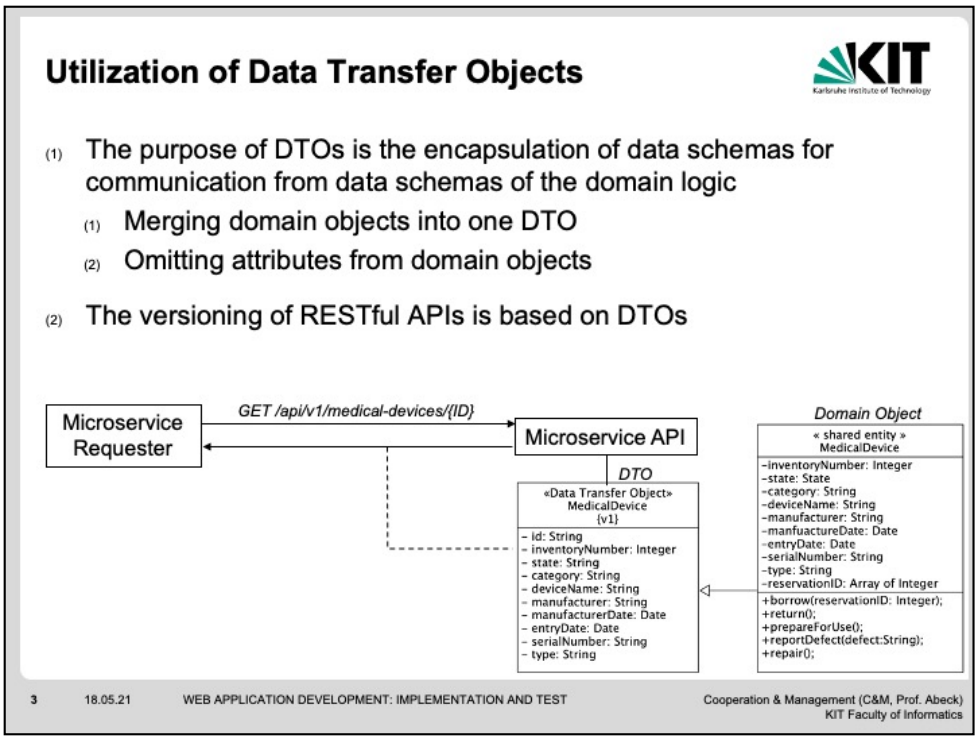
(1) Architectural styles are a collection of choices for recurring problems. Microservice architecture itself is an architectural style. However, the microservice architecture concerns the macro architecture. A variety of architectural styles also exist within the context of micro architecture. Usually, an architectural style can be applied to both macro and micro architecture. This is the case, for example, with DDD's layered architecture [Ev03].
(1.1) C&M's micro architecture represents a distinct architectural style. The origin of this style is composed of already existing architectural styles. Significant for this are the onion architecture and the layered architecture, both of which provide for a clear separation of application logic and domain logic. In the C&M micro architecture, however, the API and persistence still come to the fore.

(2) In addition to architectural styles, development approaches and fundamental development principles also influenced the C&M's micro architecture. For example, DDD leads to domain objects being implemented in the form of classes. Another example are the repositories that map the entire lifecycle of domain objects. Domain objects may thus only be accessed via the corresponding repository. Often these approaches are already integrated in the DDD's architectural styles.

(SOLID) SOLID is an acronym that stands for five fundamental principles of software development [Ma02]: (1) Single Responsibility Principle, (2) Open-Closed Principle, (3) Liskov Substitution Principle, (4) Interface Segregation Principle, (5) Dependency Inversion Principle.

API	Application Programming Interface
DDD	Domain-Driven Design
SOLID	Single Responsibility Principle; Open-Closed Principle; Liskov Substitution Principle; Interface Segregation Principle; Dependency Inversion Principle
SoC	Separation of Concerns

[Ev03] Eric Evans: Domain-Driven Design – Tackling Complexity in the Heart of Software. Addision-Wesley Professional, 2003.
[Ma02] Robert C. Martin: Agile Software Development – Principles, Patterns, and Practices. Prentice Hall, 2002.

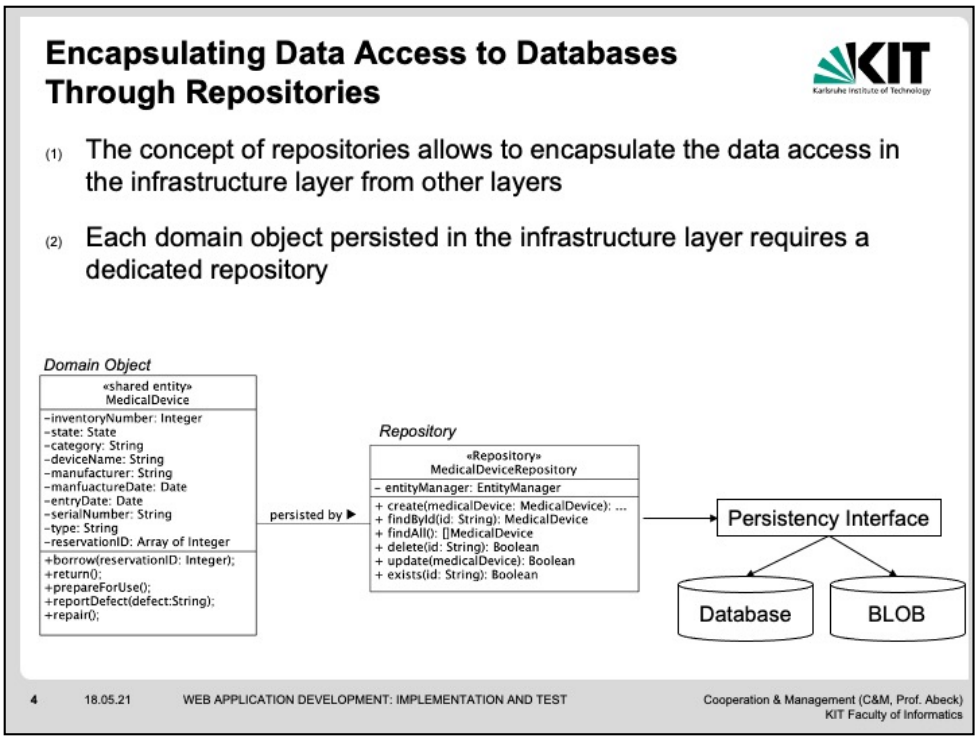


(1) A DTO is an envelope to transport data in a specific representation between the microservice and the microservice requester. By separating the data schemas, both can be modified mainly independently of each other. Furthermore, complying with API specification contracts is much easier.
(1.1) By merging domain objects into a DTO, the number of microservice requester requests required can be reduced.
(1.2) Omitting attributes can be used to implement certain microservice requester requirements. A DTO consequently includes fewer attributes than the original domain object that is hidden in the DTO. As an example, the DTO MedicalDevice does not have the attribute reservationID from the domain object MedicalDevice.

(2) An API will constantly evolve. Versioning can be introduced so that consumers of the API do not have to adapt their implementation with each new release of the API. A DTO is fundamental to such versioning by providing a DTO for each version. Thus, a new DTO can be defined for a new version, while for older versions the DTOs remain untouched.

(Microservice Requester, GET /api/v1/...) When the API is called by the microservice requester, the DTO MedicalDevice version 1 is initialized by the microservice API. The version can be taken from the path of the request. This DTO is a generalization of the domain object MedicalDevice.

DTO	Data Transfer Object
REST	Representational State Transfer

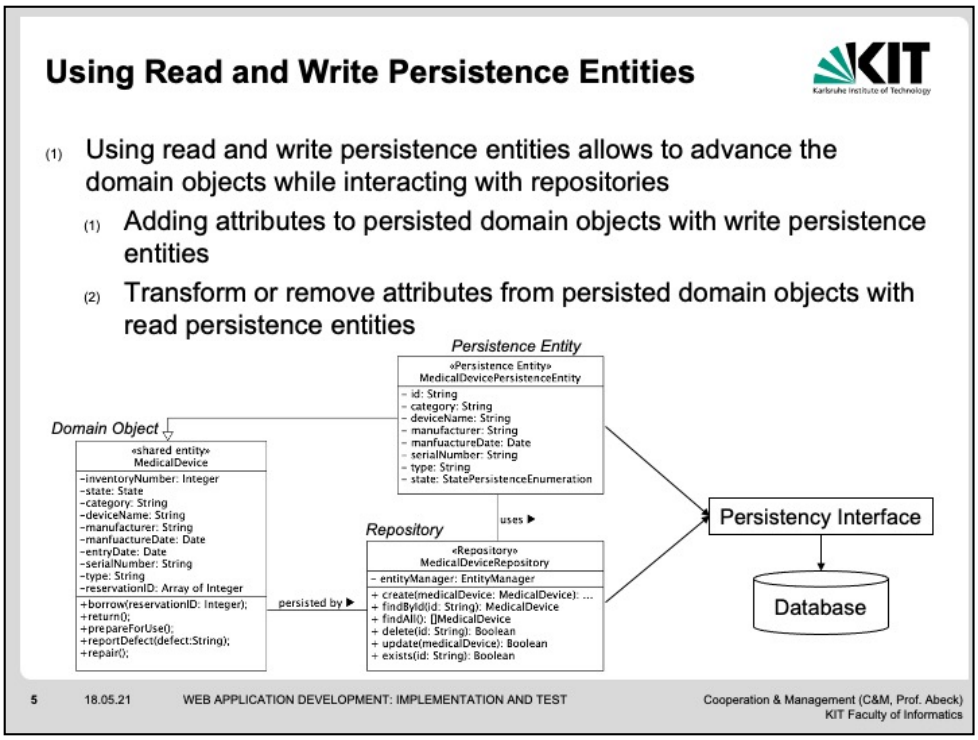


(1) Domain objects must be stored in the database. In addition, this data must also be accessed again. In order to be able to separate these two tasks from the rest of the logic, the concept of repositories exists. The repository is also suitable for implementing constraints placed on the domain objects by the domain. A repository is implemented as an independent class. In the source code, the domain objects must only be initialized, persisted or read from the database via the repository.

(2) Each domain object of the domain logic must necessarily be equipped with a repository. This applies to both entities and value objects. As long as these are persisted as an entity in the database, data access must be regulated.

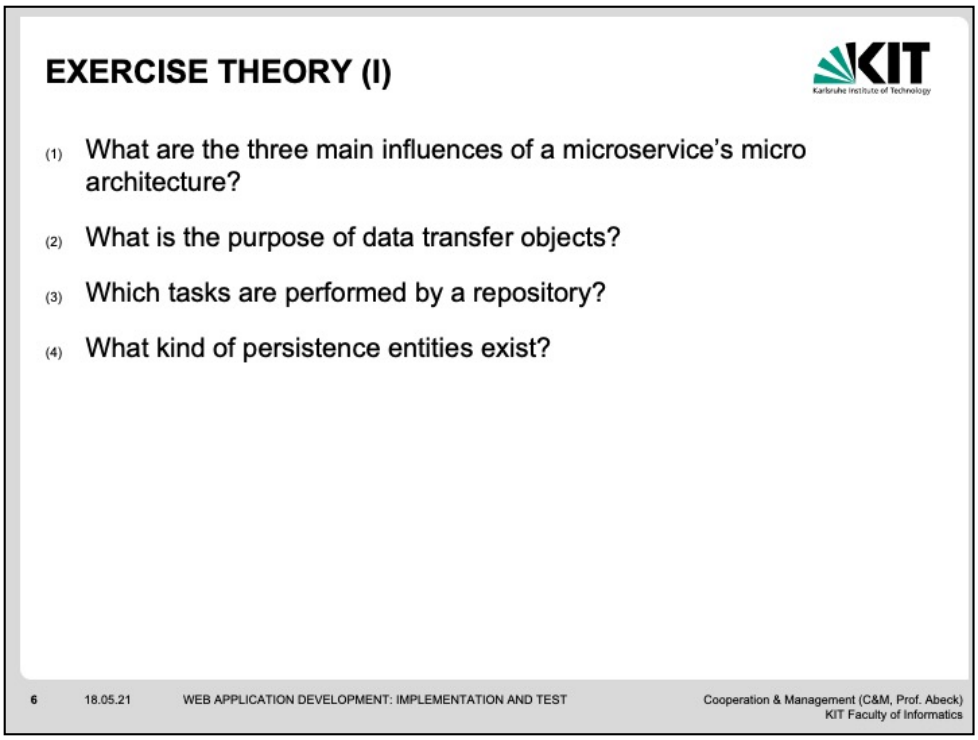
(MedicalDeviceRepository) The repository MedicalDeviceRepository writes instances of the domain object MedicalDevice to the database and reads them from the database. Writing a domain object medical device to the database is performed with the function create. This functions requires the domain object MedicalDevice as parameter. Reading domain objects from the database, the repository returns a functional domain object. The function findAll() returns an array of medical devices.

BLOB Binary Large Object



(1) In many situations, it may be necessary to persist information about domain objects in the database that is not contained in the domain object itself. This is usually meta information about the raw record, such as creation date or modification date, to enable auditing. Using persistence entities, this meta information can be added to the domain object at the time of persistence. In addition, persistence entities provide a data schema that is loosely coupled to the domain logic. It is distinguished between write and read persistence entities. Persistence entities are used only in the context of a repository.
(1.1) At the time of persistence, the write persistence entities come into play. These are typically used to add attributes to the domain object.
(1.2) The read persistence entities are a useful concept when the data in the database needs to be transformed. If a particular form of data is required, a read persistence entity can be created. A domain object can be accessed via several read persistence entities.

(Persistence Entity) The persistence entity MedicalDevicePersistenceEntity is a write persistence entity that is used by the repository MedicalDeviceRepository. The persistence entity has further technically relevant attributes such as the "id", which corresponds to a technical identifier. The repository must only access this persistence entity when writing the domain objects to the database. Before the repository can use the persistence entity, a mapping between the domain object and the persistence entity itself is needed. There are several ways to implement this. In C&M, an explicit mapper was implemented as a separate class.

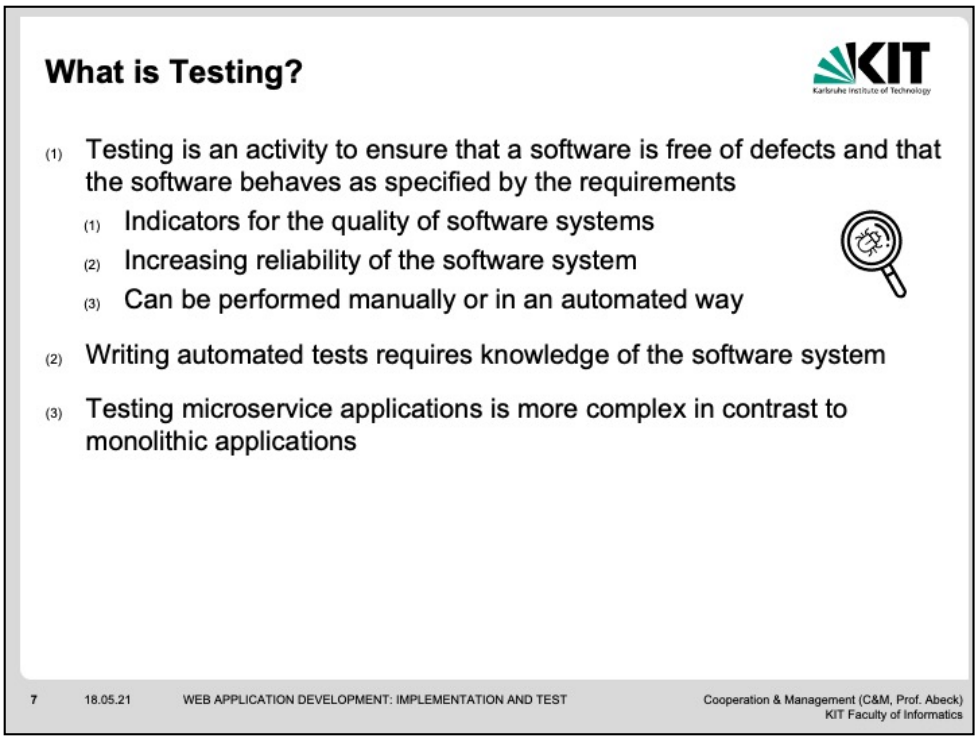


(1) ++Architectural Styles, Development Principles, and Approaches Affect the Micro Architecture++
The three main influences are architectural styles, development approaches and development principles.

(2) ++Utilization of Data Transfer Objects++
A DTO encapsulates the data schema required for the API from the data schema within the residual logic. Further, DTOs are the fundamental for versioning a RESTful API. Thus, in the C&M application CAM the DTO "MedicalDevice" is used.

(3) ++Encapsulating Data Access to Databases Through Repositories++
A repository is responsible for the life cycle of a specific type of a domain object. It initializes, persists, and read domain objects from the database. For example, the "MedicalDeviceRepository" repository stores the "MedicalDevice" domain object in the database and, if required, initializes this domain object again from the database during a read access.

(4) ++Using Read and Write Persistence Entities++
There are two kinds of persistence entities: (1) read persistence entities and (2) write persistence entities. The "MedicalDeviceRepository" repository must rely on the write persistence entity "MedicalDevicePersistenceEntity" when writing the "MedicalDevice" domain object to the database.

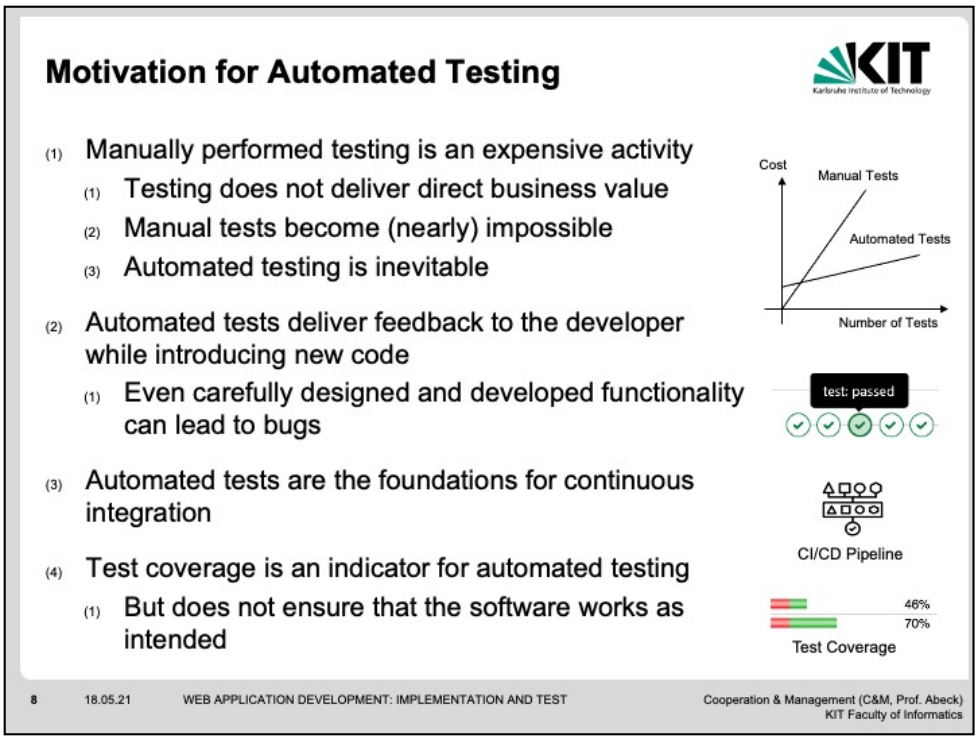


(1) It is important that the faults of a software system are detected with tests in order to prevent crashes and software failures. Next to possible faults, the correct software behavior is required in order to fulfill the customers requirements. Therefore, testing does not only cover bugs, but the specified requirements are tested as well.
(1.1) Testing should also address the quality indicators of software systems such as stability, performance, maintainability and code quality.
(1.2) As a result, the reliability of a software system is increased which leads to less crashes and a more stable performance of the software system.
(1.3) The testing activity involves running the software and testing it with manual or automated tests.

(2) Tests should not be experiments that assumes what could go wrong with the application [RJ20]. It is important that the developers are aware of the problems which can occur in the software system.

(3) Testing microservices is different from testing monolithic applications [Cl14]. A microservice-based application requires a different test strategy due the independently deployable microservices.

[Cl14] Toby Clemson: Testing Strategies in a Microservice Architecture, martinfowler.com, [Online], November 2014, vol. 30, 2014.
[RJ20] Casey Rosenthal, Nora Jones: Chaos Engineering. O'Reilly Media, Incorporated, 2020.



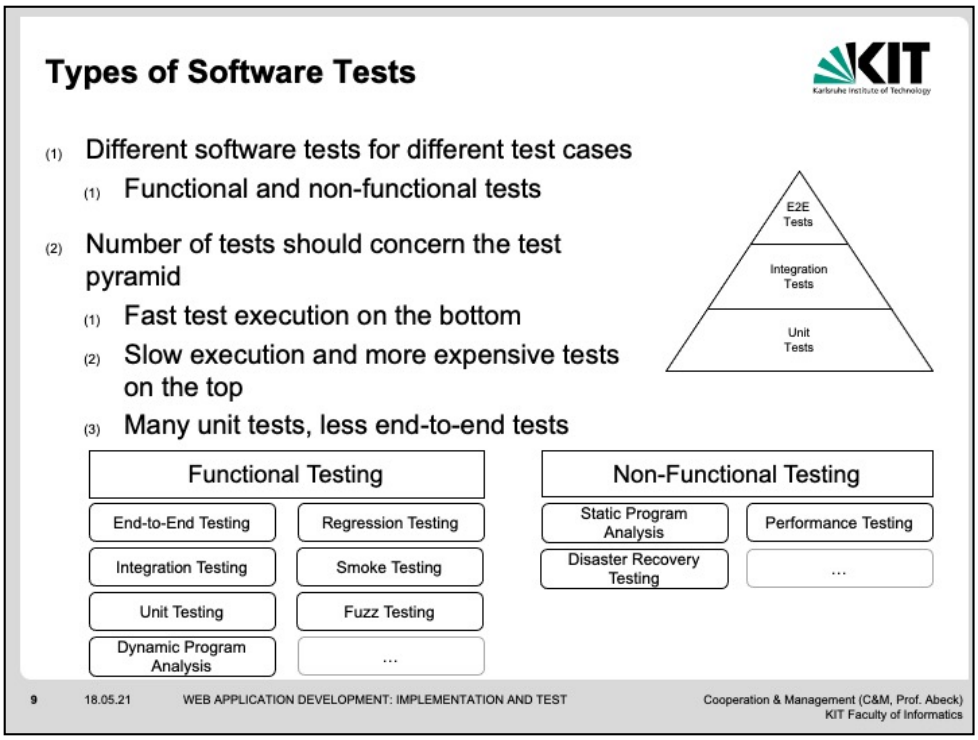
(1) A software system can be tested manually to detect faults. However, executing manual tests have a high effort since the tests need to be executed after each change in the software.
(1.1) Since testing is not paid for by the customer, it is often neglected In addition, involving developers/stakeholders in the testing process is costly.
(1.2) Further development of the system increases the complexity of the tests and therefore, manual tests are very difficult to execute. As a result, the cost of manual testing is increased.
(1.3) Automatic tests have a higher initial implementation effort than manual executed tests. However, the tests pay out in the long run since they can be executed after each change to ensure that the previous implemented functionality is still working as intended.

(2) For a developer, it is important to get feedback from the tests. Automated tests can be executed after the implementation of a new feature. If a test fails, the developer knows that the latest changes led to unseen consequences in the implementation. Executing these tests ensures that the previous functionality still works as intended, and no bugs were introduced by the implementation of the new features [AB+20].
(2.1) Even when the old code basis is nearly untouched, there is no guarantee that no error has been introduced to the changes made. Therefore, automated tests can be used to ensure that no bugs are introduced with the changes.

(3) A CI/CD pipeline can be used to automatically execute the tests. This ensures that the software does not only run on the local machine. Furthermore, software which fails the test in the pipeline is not handled further and an error message is sent to the developer.

(4) Test coverage displays how many lines of code are tested. However, it is a bad indicator since it does not cover how many test cases are not covered with tests.

[AB+20] Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, Adam Stubblefield: Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems. O'Reilly Media, 2020.



(1) For testing software, a huge variety of software test types exist. The test types itself can be categorized in functional and non-functional testing types.

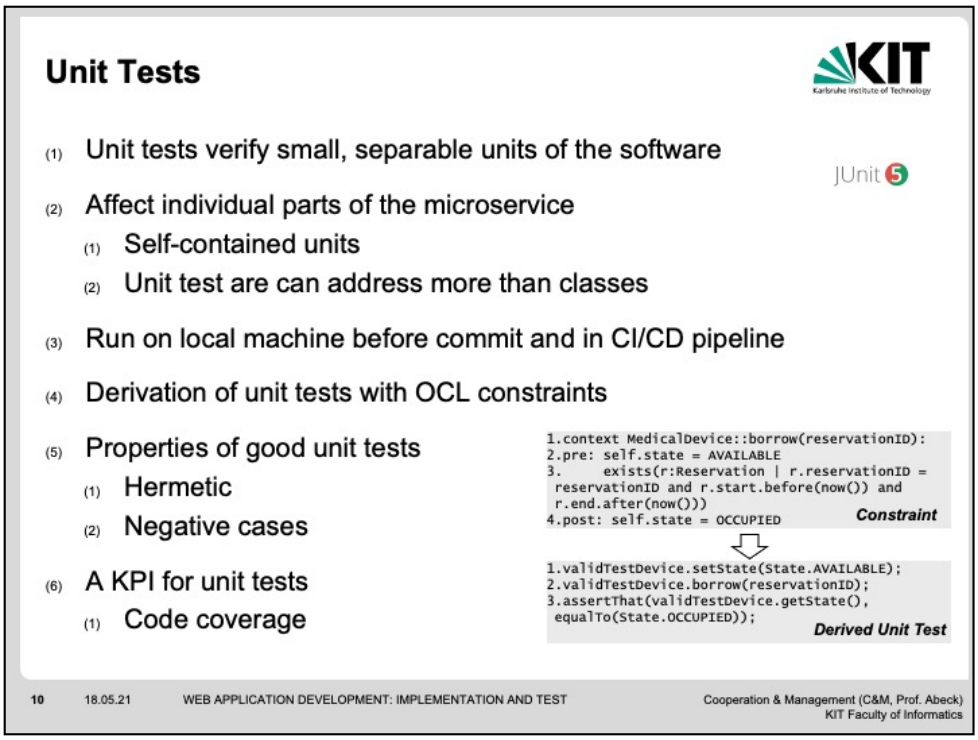
(2) Depending on the type of the tests, the test is more expensive than the other.
(2.1)(2.2) Unit tests are executed much faster than slow end-to-end tests which require to navigate through the user interface. In addition, unit tests are faster to implement. The test pyramid implicitly proposes that developers should write more unit tests than integration tests and so on [Co10].
(2.3) A microservice application itself requires more effort in integration and end-to-end-testing, since the interaction of microservices is important.

(End-to-End Testing) The goal is to test the software from a user's perspective. End-to-end testing involves testing the complete set of user interactions with the system to ensure that your software behaves as intended. Typically, testing starts with a clean system and runs through all of a user's interactions. These tests are particularly important in a microservice architecture, as they also test the interaction of the various services.

(Integration Testing) Integration tests verify the provided interfaces and interactions between the components to detect possible defects [Cl14]. In contrast to unit tests, integration tests aim to test whole subsystem and test their interactions. In the context of the microservice architecture, the interfaces of the individual microservices are addressed.
(Unit Testing) Unit tests are used to test the smallest parts of a software system for their functionality. This includes individual units of the source code, such as functions, modules or objects. Unit tests are a form of white-box tests, which aim to check a section of code for its correctness. This is intended to that code changes does not affected the previous implementation.
(Regression Testing) Changes in the software should not lead to braking changes in existing functionality. Regression tests ensure that changes do not influence the old software behavior.
(Smoke Testing) The name smoke testing is derived from water installations. It stands for the check if smoke comes out of the pipes after a deployment has been made. As with water installations, smoke testing in software projects is done after a deployment.
(Fuzz Testing) Random and invalid data is used for testing the software. The goal is to analyze crashes and exceptions.
(Dynamic Program Analysis) Dynamic program analysis tries to cover each path of the software during the execution of the software.

(Static Program Analysis) Mostly code artifacts are analyzed by this method.
(Performance Testing) Performance testing can be seen as a general term which contains tests such as load testing and stress testing.
(Disaster Recovery Testing) Simulations of disasters (such as loss of a data center).

[Cl14] Toby Clemson: Testing Strategies in a Microservice Architecture, martinfowler.com, [Online], November 2014, vol. 30, 2014.
[Co10] Mike Cohn: The Forgotten Layer of the Test Automation Pyramid. https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.



(1) Unit tests verify the smallest separable unit of software. It is widely accepted that a unit test verifies the smallest separable unit of software. Object-oriented programming tends to treat a class as a unit, while function or procedural programming might consider a function as a single unit [Fo14]. In the example of Java, JUnit is a testing framework which supports the development of unit tests.

(2) Unit tests can be used to test several parts of a microservice.
(2.1) Self-contained units of a microservice can be tested with unit tests. For example, the implementation of an entity and its operations, unit test can be written.
(2.2) Testing specific kind of HTTP requests to ensure if the microservice can handle them right is another aspect of unit testing. Furthermore, multiple classes can be involved in unit-testing [Cl14].

(3) Before the software is committed to the version control system, the unit tests should be executed locally. The test displays if some functionality is not working properly and needs to be fixed before committing.

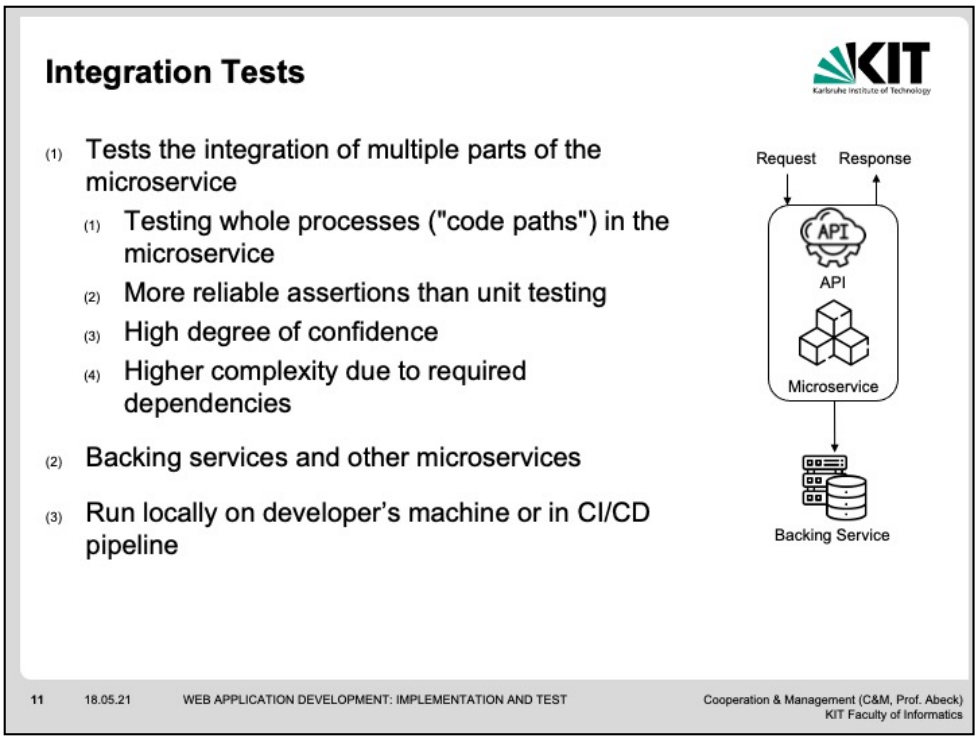
(4) It is important to understand what should be tested with unit tests. For the functionality (or logic) constraints are useful for the derivation of unit tests.

(5) Unit tests should consider several properties in order to provide good test cases.
(5.1) The test should be protected from influences such as databases or libraries.
(5.2) Unit tests should not only contain one success scenario. In addition, special cases for failures or faults of the system needs to be considered.

(6) Key Performance Indicators (KPIs) are for measuring the efficiency of unit tests. However, such metrics need to be handled with care since code coverage does not tell if the system works as intended.
(6.1) Popular metrics for unit tests are path coverage and statement coverage [Or19].

KPL	Key Performance Indicators
OCL	Object Constraint Language

[Fo14] Martin Fowler: Unit Test, martinfowler.com, [Online], 2014, https://martinfowler.com/bliki/UnitTest.html.
[Or19] Gerard O'Regan: Concise Guide to Software Testing, Springer International Publishing, Nature, 2019.

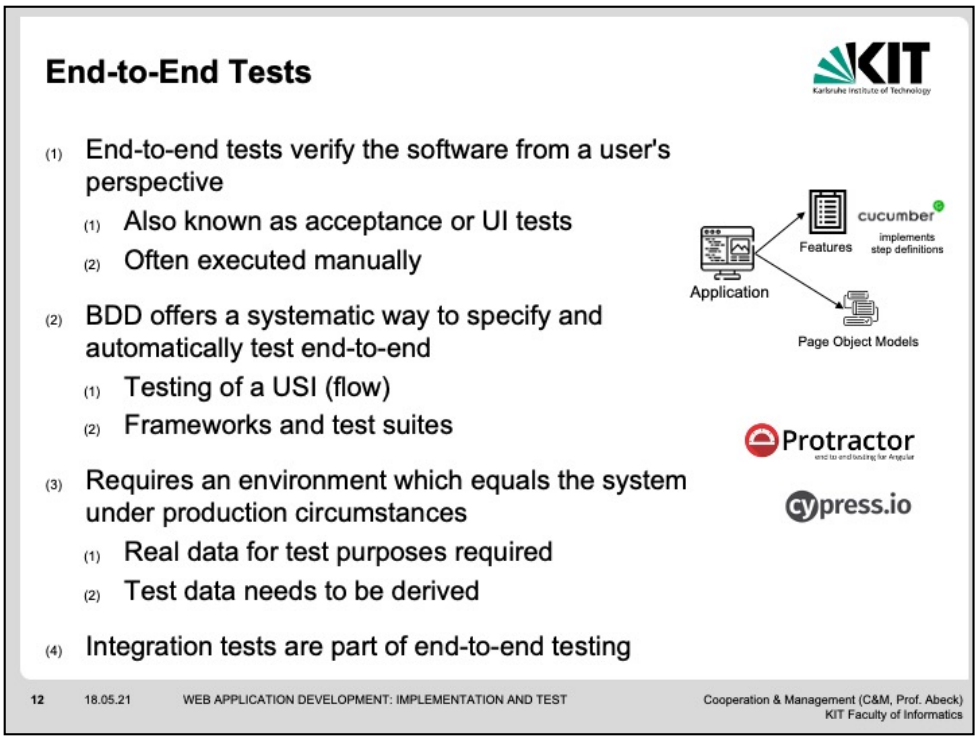


(1) Integration tests verify the provided interfaces and interactions between components to detect defects [Cl14].
(1.1) In the case of microservices, integration testing involves sending requests to the API of the microservice under test and verifying that correct responses are returned. This can even include external dependencies which are consumed by the microservice under test.
(1.2) Integration tests, in contrast to unit tests, collect software units together and test them as a subsystem in order to detect any incorrect interaction assumptions.
(1.3) As a result, the integration tests can verify that the microservice implementation fulfills its task on a higher level.

(2) One of the test cases are the implementations of backing services and other microservices. The integration tests verify if the code paths deliver the correct answer.

(3) During the test stage of the CI/CD pipeline, the unit and integration tests are executed. Therefore, the required services (e.g., backing services) need to be available. If the tests are run on a local machine, all required services need to be available as well.

[Cl14] Toby Clemson: Testing Strategies in a Microservice Architecture, martinfowler.com, [Online], November 2014, vol. 30, 2014.



(1) End-to-end tests ensure that the software works as intended from a user's perspective which means that frontend inputs lead to expected outcomes and the system behaves as intended.
(1.1) Therefore, the tests are also called acceptance tests since the system is tested following the specified requirements.
(1.2) Since end-to-end tests are often difficult (and costly) to implement, end-to-end tests are often executed by hand.

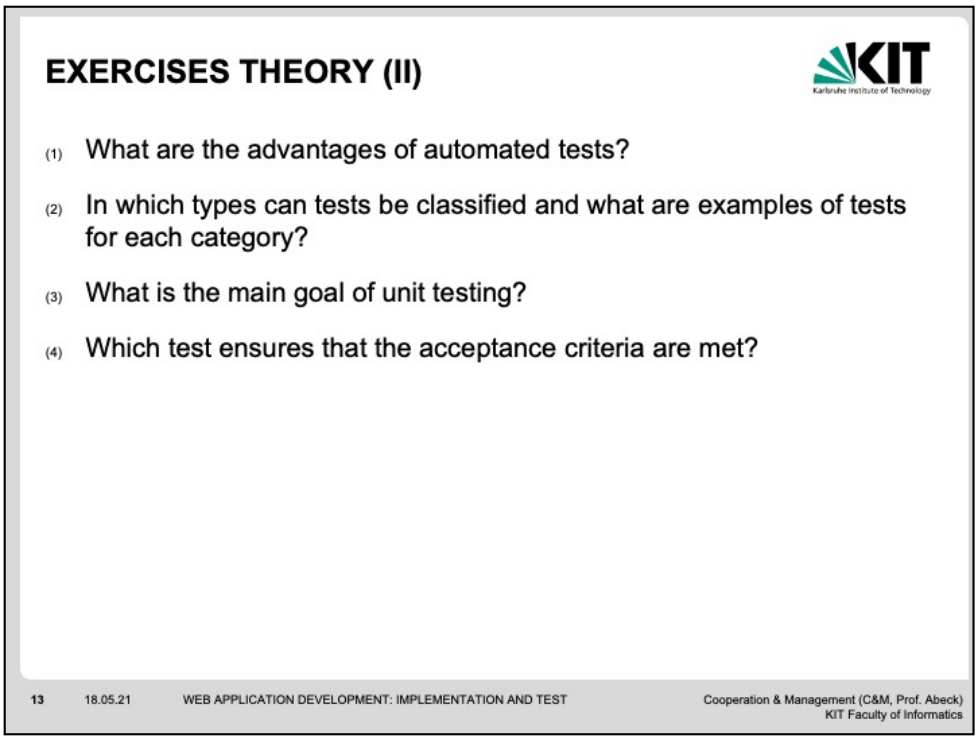
(2) One way to develop and implement end-to-end tests in an automatic way is by using the concepts of Behavior-Driven Development (BDD). In this way, it is ensured that the goals are achieved and the whole system works together without errors.
(2.1) User/system interactions can by systematically translated to scenarios and thus end-to-end tests. With BDD, the specified features and scenarios are transferred to end-to-end tests via step definitions.
(2.2) To simulate the user interactions with the software and especially the UI, page object models are used [Web-Pag]. All the necessary code to simulate the interactions with the browser UI and the page element assertions are implemented in those classes. Tools or testing suites such as Protractor (Angular) or Cypress (JavaScript) assist ease the implementation.

(3) For a microservice system this means that all microservices need to be available for testing. Furthermore, the databases need to be set up.
(3.1) Especially for edge cases, real test data needs to be provided. This test data needs to match data which is used in the production.
(3.2) Therefore, it is important to derive the required test data for each of the test cases. Furthermore, the database needs to be set in a specific state for further tests.

(4) Since end-to-end-tests require all parts of the system to work together, the integration is tested as well. However, end-to-end tests do not replace integration tests.

BDD	Behavior-Driven Development
UI	User Interface
USI	User/System Interactions

[Web-Pag] Webdriver.io: Page Object Pattern, https://webdriver.io/docs/pageobjects.html.

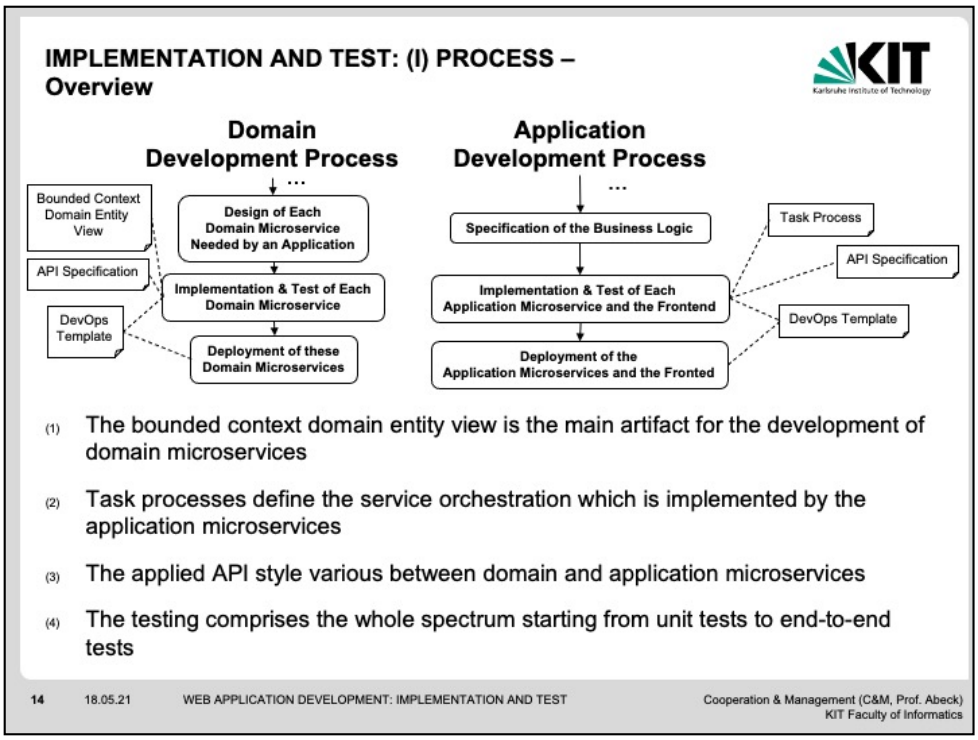


(1) ++Motivation for Automated Testing++
- Can be integrated into CI/CD
- Executable after every change
- Ensure that no faults are introduced into the system

(2) ++Types of Software Tests++
- Functional and non-functional tests
- Functional: end-to-end, integration, unit tests, …
- Non-functional: performance, code analysis, …

(3) ++ Unit Tests++
- Testing small, separatable parts of the software
- Ensure that changes do not lead to new faults

(4) ++End-to-End Tests++
- End-to-end test
- BDD can be used to write acceptance tests on the end-to-end side



The implementation can start after the design of the microservices was completed. In a following deployment phase the implemented microservices and the frontend are deployed on a container-based infrastructure.

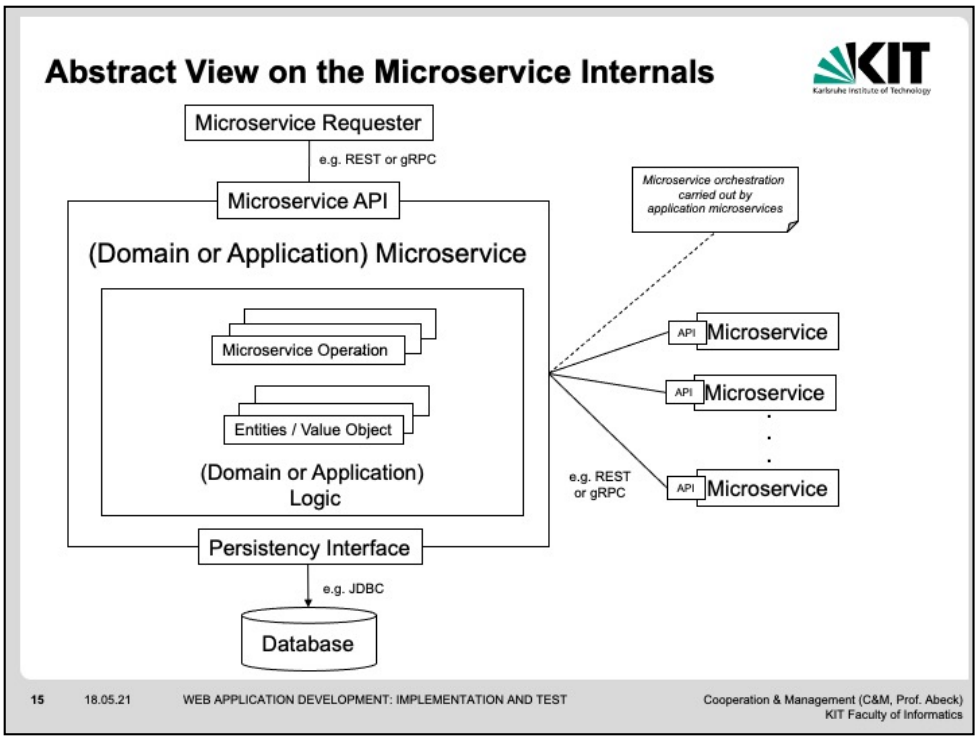
(1) The internals of the domain microservices are implemented as defined in the domain entity view. Each domain object in this design artifact is implemented as standalone class.

(2) While domain microservices are resource-centered services providing CRUD (Create, Read, Update, Delete) operations, application microservices include more complex logic which is provided by calling operations from underlying domain microservices and possibly operations from other application microservices.

(3) The APIs (Application Programming Interface) of domain microservices are specified in OpenAPI which allows to describe REST (REpresentational State Transfer) APIs in a standardized way. For the specification of application microservices the use of gRPC is recommended.

(4) The end-to-end tests concern the application microservices. These tests are defined by the Gherkin features and the included scenarios.

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
gRPC	gRemote Procedure Call
REST	REpresentational State Transfer



The figure illustrates an abstracted view on the internal structure of microservice. This structure holds both for domain microservices and the application microservices.

(Microservice Requester) The software unit which sends a request to the microservice and receives the response from the microservice. This software unit can be the frontend or another microservice. In the hexagonal architecture, this unit is called a client.

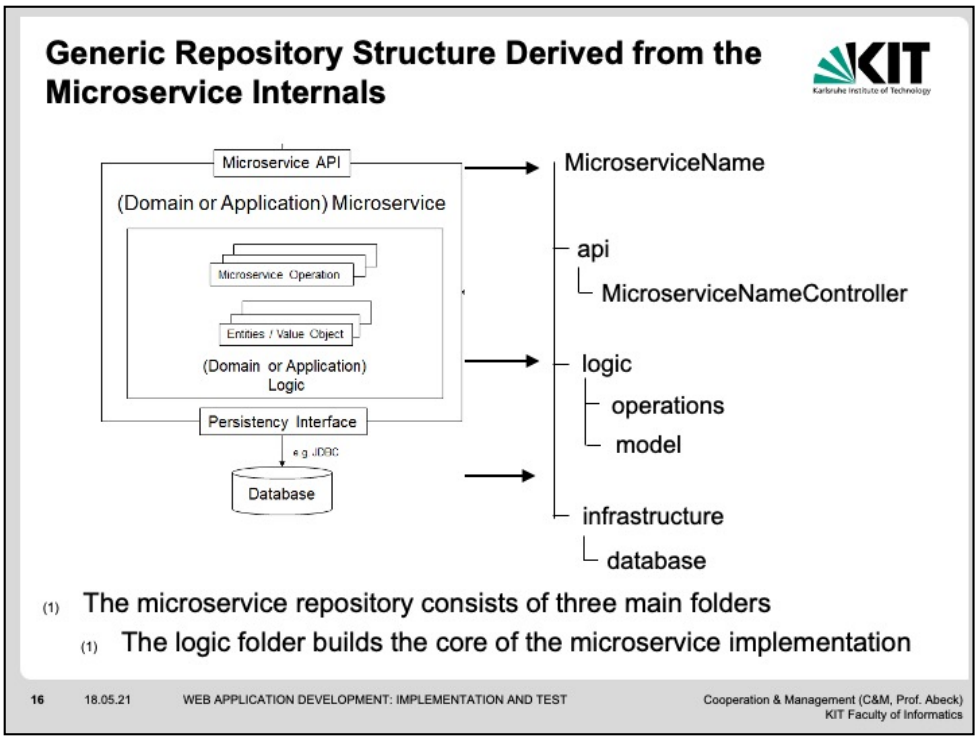
(Microservice API) Each microservice must implement and provide an API which is defined by the API specification of this microservice. The specfication is usually based on REST or gRPC depending on the type of microservice.

(Domain or Application Logic) This core part of the implements the domain logic or the application logic of the microservice.
(Operation) The logic is structured according to the operations defined by the API. In the case of a domain microservice the operations are CRUD operations on a domain entity. In the case of an application microservice an operation implements a user interaction as defined by the corresponding USI flow and task process.

(Microservice orchestration ...) In the case of an application microservice this microservice becomes a microservice requestor since it sends requests to one or more microservices according to the specification of the task process.

(Entities and Value Object, Persistency Interface, Database) This software unit provides the the data elements, i.e. the entities and value objects that are accessed by the operations. This unit provides a decoupling of the interface to the database by which the data elements are persisted.

AMS	Application Microservice
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DMS	Domain Microservice
JDBC	Java Database Connectivity

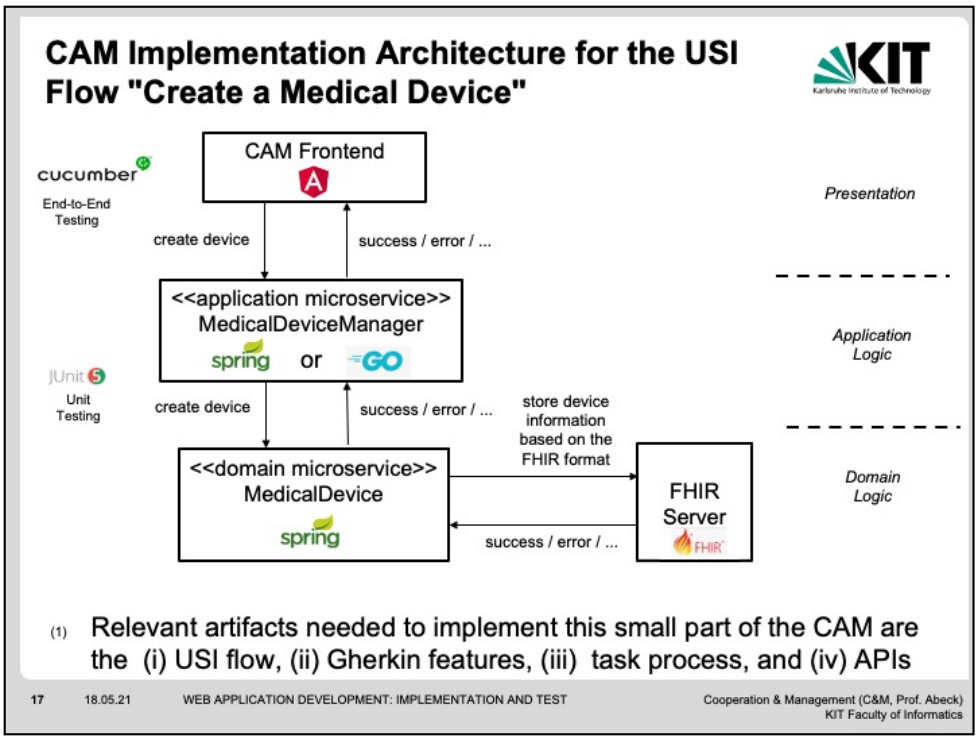


The generic structure used to organize the code artifacts in GitLab, reflects the microservice implementation architecture. Generic in this specific context means that the structure is independent from any implementation technology. This is no longer the case for the further structuring of the folders of the generic structure which is different for each implementation technology (e.g., Spring Boot or Node.js).

(1) The structure consists of three main parts or folders:
(api) This folder contains the controller by which the API is exposed. If REST is used the API is specified by the OpenAPI standard.

(logic) (1.1) This folder contains the implementation of the microservice's functionality. The code artifacts stored in this folder can be seen as the core of the microservice implementation.
(operations) Each operation specified by the API is implemented in the programming language chosen for the implementation. In the case of an application microservice, an operation implements the microservice orchestration as it is specified by the corresponding task process.
(model) This folder contains the code of the entities and value objects and the object-relational mapping to the database.

(infrastructure) This folder provides the technical basis of the microservice implementation which especially includes the database which is responsible for the persistency of the data.



This page focusses the part of the CAM implementation architecture by which the USI flow "Create a Medical Device" is realized.

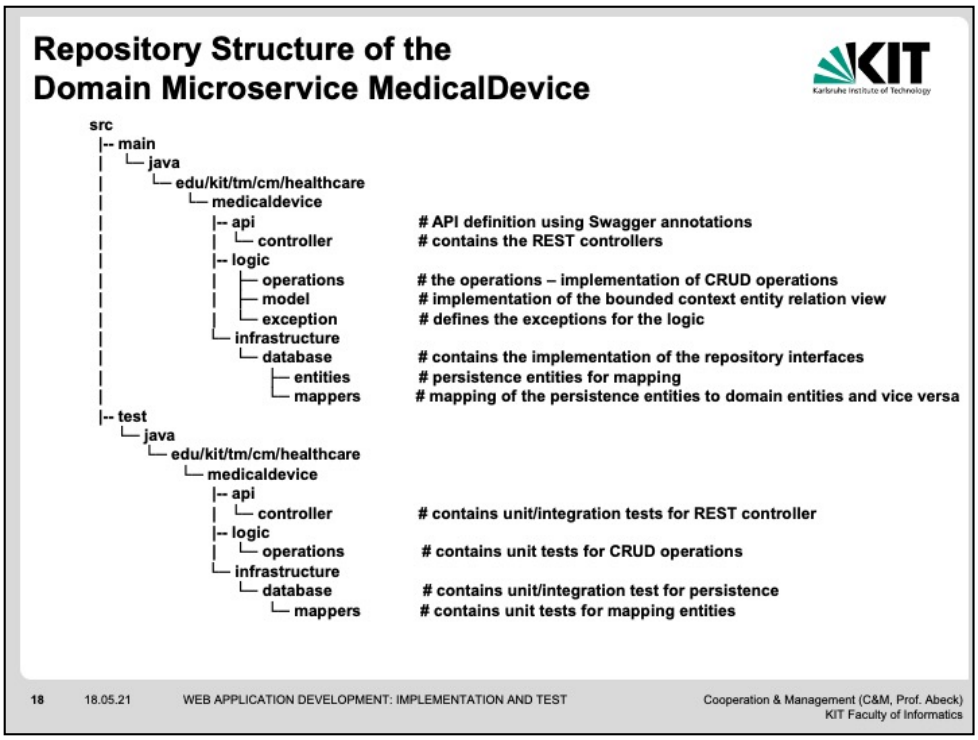
(CAM Frontend) This part of the frontend implements the UI elements specified by the USI flow based on the Angular frontend framework.

(MedicalDeviceManager) This application microservice implements the API which is specified based on the gRPC standard. According to the task process by which the dynamic aspects of this application microservice is specified, the only task is to validate the input information received from the frontend.

(MedicalDevice, FHIR Server) This domain microservice implements the API which is specified based on the REST standard. The main functionality which must be provided by this domain microservice is to transform the device information so that the FHIR server can be used.

(End-to-End Testing, Unit Testing) Testing is extremely relevant in order to make sure that the software fulfills the specified requirements.

(1) If necessary, the artifacts must be adapted in order to make sure that all analysis, design, and implementation artifacts fit together.

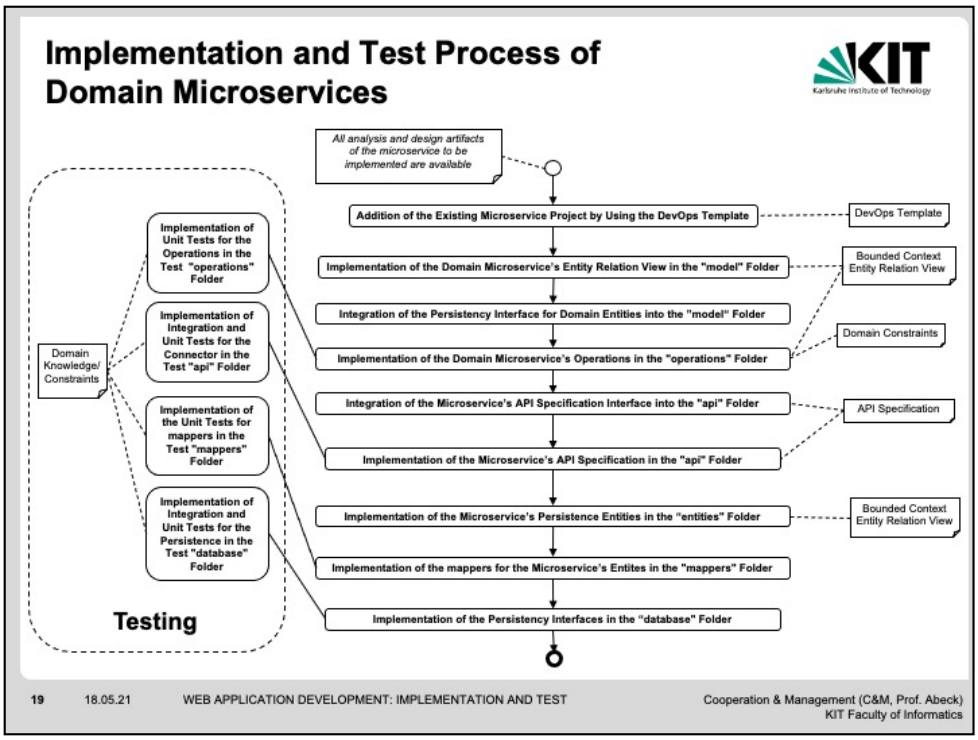


The repository structure is derived from the microservice internals. It is applied to all microservice implementations carried out by C&M. One of these microservice is the domain microservice MedicalDevice [CM-G-HMD] for which the repository structure is explained on this page.
The repository contains the implementation of the MedicalDevice in the "main" folder as well as the implementation of the unit and integration tests in the "test" folder.
Further, the repository structure depends on the used technology. In this case, the domain microservice MedicalDevice was implemented with the Java Framework Spring Boot [Spr-Bui] and consists of three main folders: (i) api, (ii) logic and (iii) infrastructure.

The implementation code of the domain microservice is built of:
(controller) The controller exposes a RESTful API for the specified operations derived from the OpenAPI specification.
(operations) The operations of a domain microservice is determined by the CRUD operations.
(model) Entities, value objects and enumerations from the bounded context entity relation view build the model of the domain microservice (e.g., MedicalDevice, DescriptionData, etc.).
(database) MedicalDevice has a database connection where medical devices are stored.
(entities) Persistence entities for mapping and data access.
(mappers) mapping of the persistence entites to domain entites and vice versa

Unit and integration tests concentrate on the formal correctness of the domain microservice which requires domain knowledge to derive domain constraints. The test code of the domain microservice is built of:
(controller) Contains unit/integration tests for REST controller.
(operations) Contains unit tests for the operation classes.
(database) Contains unit/integration tests for persistence.
(mappers) Contains unit tests for mapping entites.

[CM-G-HMD] Cooperation & Management: Healthcare MedicalDevice Domain Microservice. https://git.scc.kit.edu/cm-tm/cm-team/2-6.healthcare/1.domain/medicaldevice
[Spr-Bui] Spring: Building an Application with Spring Boot. https://spring.io/guides/gs/spring-boot/



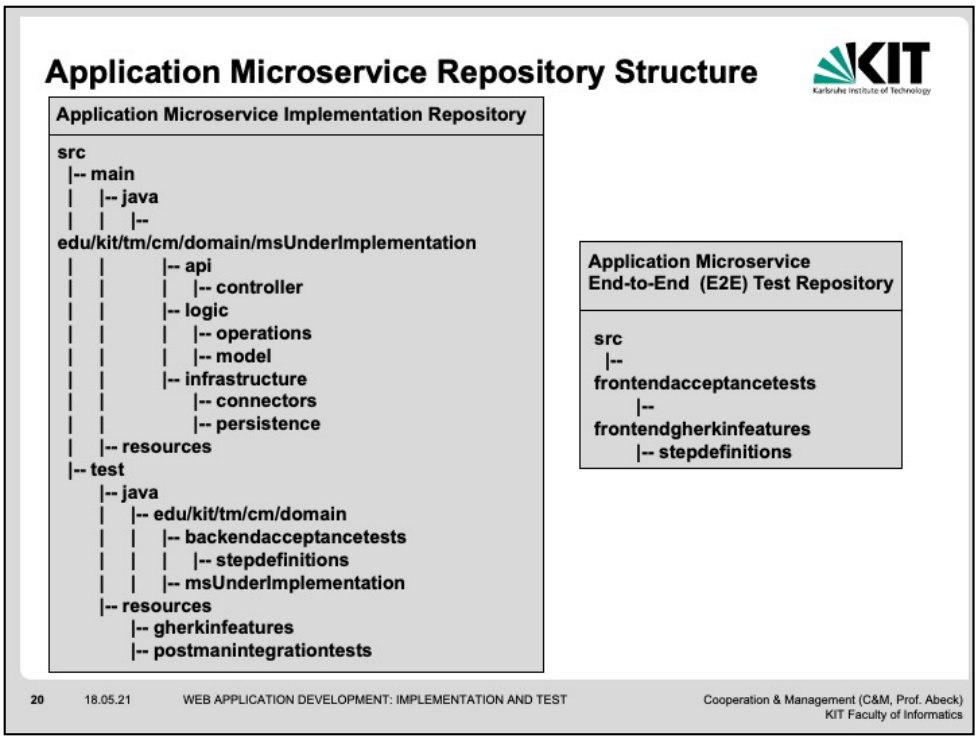
The activity diagram illustrates the steps of the implementation process of a domain microservice. The implementation process starts as soon as all analysis and design artifacts of the domain microservice are available.

(DevOps Template) DevOps templates for different technologies (e.g. Java, Go, Node.js) are made available in the GitLab DOCUMENTATION part [CM-G-Doc].
(Bounded Context Entity Relation View) The bounded context domain entity view must be mapped to the code level first so that operations and controllers can be implemented. For each entity and value object a separate class with their respective attributes and methods is created in the "model" folder.
(Persistency Interface) The persistency interface defines the methods for accessing and manipulating entities in the ”model" Folder.
(Domain Microservice’s Operations) The operations of a domain microservice are implemented in the "operations" folder.
(API Interface) An API interface is created in the "api" Folder which contains the API endpoints for the CRUD operations.
(Domain Constraints) Defined as separate artifact
(Implementation of API Interface) HTTP requests are handled by a controller and expose a RESTful API for the specified operations in OpenAPI. The controller is identified by the @RestController annotation and implements the API interface in the "api" Folder.
(Persistence Entities) An entity represents a table stored in a database and is created in the "entites" folder.
(Implementation of Mappers) They map persistence entities to domain entities or vice versa in the "mappers" folder.
(Implementation of the Persistency Interface) The persistency interface defined in the ”model" folder will now be implemented in the "database" folder.

(Testing) Unit tests and integration are implemented for the connectors, mappers, and persistence functionality of the domain microservice. The operations are only tested by unit tests which check if the domain microservice implementation implements the domain knowledge.

CRUD Create, Read, Update, Delete

[CM-G-Doc] Cooperation & Management: Documentation. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation

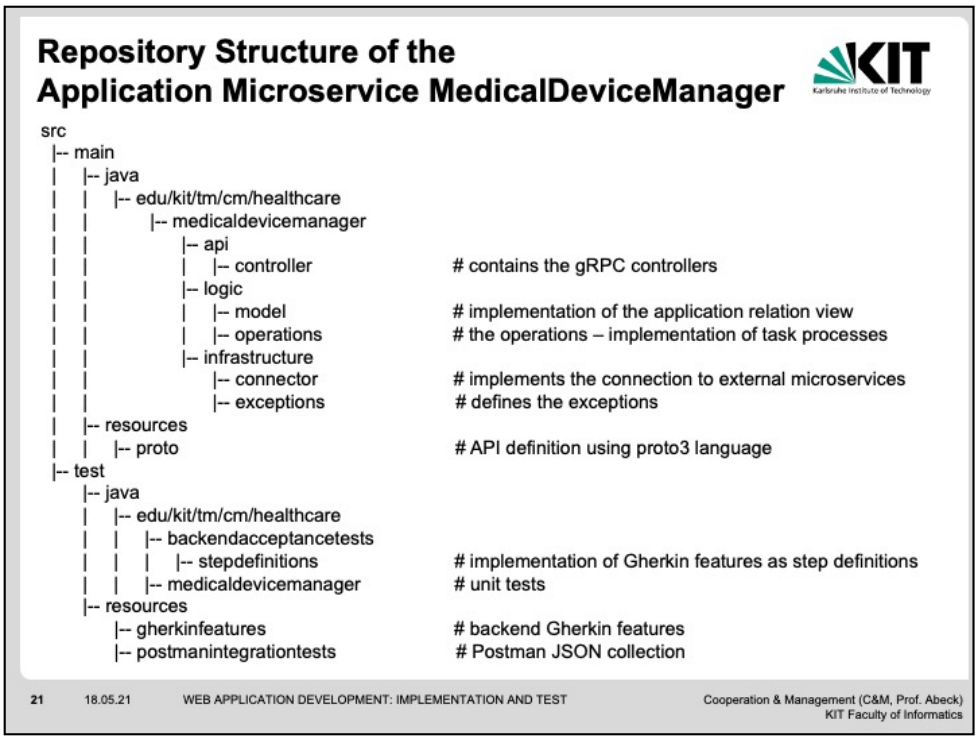


The application microservice's repository structure is an extension of the generic microservice structure.

(Application Microservice Implementation Repository) This repository contains the application code of the application microservice.
(controller) Implements the API specification by mapping API operations to pieces of code that use application microservice operations to provide the specified functionality.
(operations) An application microservice operation implements the functionality required for the computation of the resources exposed by the API specification operations. It may contain complex algorithms. Task flows provide information on which resources are fetched from other microservices, thus informing the developer which functionality should be implemented and which functionality is a part of another microservice.
(model) Entities, value objects, etc. from the application relation view
(connectors) The package contains connectors to other microservices. HTTP requests to a microservice and response decoding functionality (e.g., mapping response to Java objects) are implemented in a connector.
(persistence) An optional folder containing the database maintenance functionality. Some application microservices might not require data persistence. Other need to maintain a database. For example, MedicalDeviceManager should persist relationship between a device (one domain entity) and the staff (another domain entity).

(backendacceptancetests) The package contains the implementation of the Backend Gherkin features (the step definitions) and a runner class which automates execution of the step definitions, and the backend Gherkin features as acceptance tests during mvntest
(stepdefinitions) Contains the step definitions for the backend Gherkin features.
(msUnderImplementation) The structure of the folder mirrors the msUnderImplementation folder in src. Instead of application code, however, it contains unit and white-box integration tests. The white-box integration tests resemble unit tests in that methods are fed with test data and output is matched to expected output. The difference is that there is no mocking involved. They do not test functions in isolation. Consequently, for a successful execution of the tests a bigger part of the system is required
(resources) Contains test resources, amongst which are the backend Gherkin features and a Postman JSON collection for black-box API testing.
(gherkinfeatures) Contains backend Gherkin features. Those are features, which manipulate application code directly. They are used for the backend acceptance tests and can also be utilized for derivation of unit tests.
(postmanintegrationtests) Contains the JSON collection exported from Postman, which is run with Newman as black-box API tests.

(Application Microservice End-to-End (E2E) Test Repository) A separate end-to-end test repository is created and maintained for each application. It contains the frontend acceptance tests, which comprise of frontend Gherkin features and their step definitions.

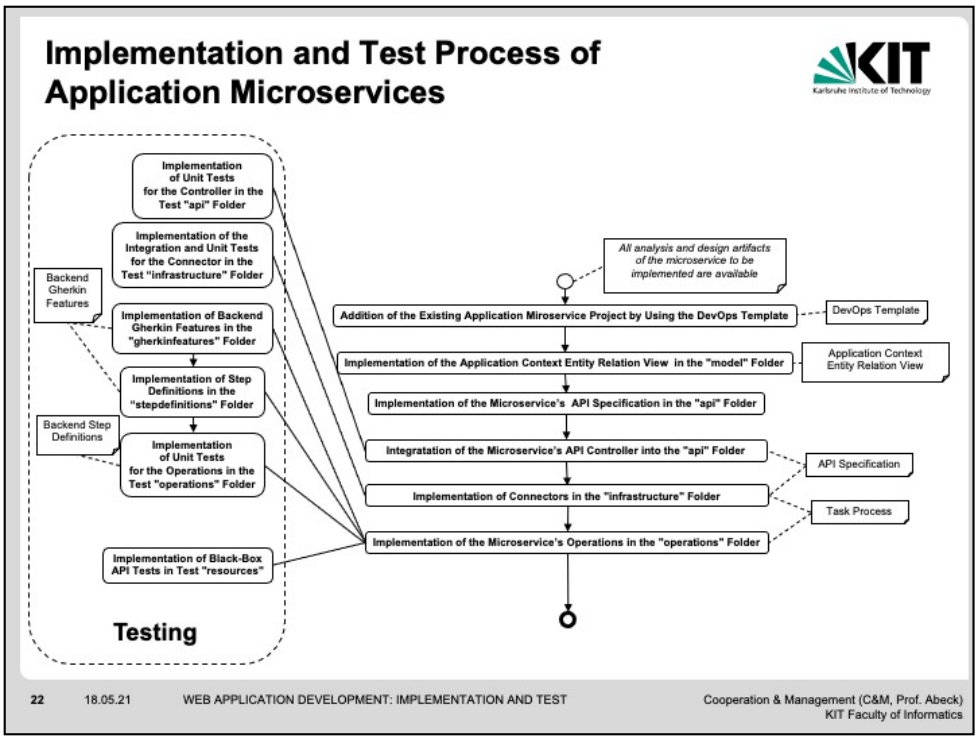


MedicalDeviceManager (MDM) is an application microservice developed in Java. It is part of the CAM application and it implements the capability "Management of Medical Devices"

The application code is built of
(controller) In the case of MDM the API controller is a gRPC service which implements the API definition, which is specified in a .proto file.
(model) Entities, value objects and enumerations from the application relation view build the model of the microservice (e.g. MedicalDevice, Staff, etc.).
(operations) Functionality for management of medical devices (e.g. assigning a device to a staff member)
(connector) MDM has a connector to the MedicalDevice microservice where medical device domain functionality is implemented. The devices themselves are fetched as resources from MedicalDevice.
(exceptions)
(proto) The gRPC API definition written with proto3.

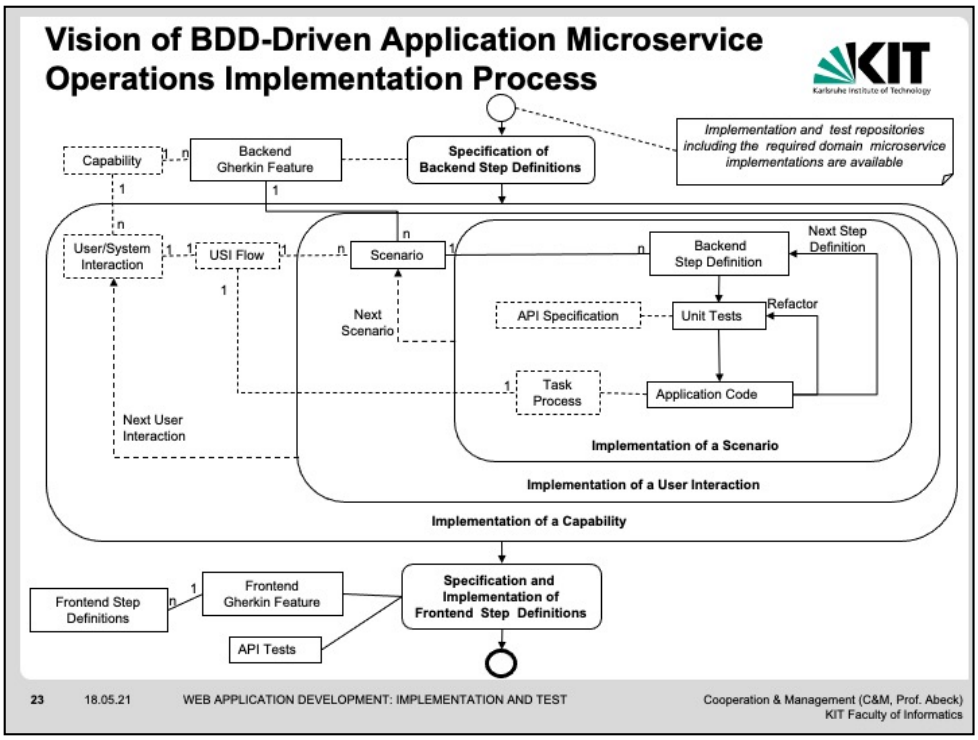
The test code is built of:
(backendacceptancetests) Exercise operation classes for scenarios defined in the backend Gherkin features.
(medicaldevicemanager) Contains unit tests for the operation classes, the connectors and the controller. Integration tests for the controller and the connector should be written as well.
(gherkinfeatures) Contains the backend Gherkin features
(postmanintegrationtests) Contains the JSON collection exported from Postman

[CM-G-HMD] Cooperation & Management: Healthcare MedicalDeviceManager Application Microservice. https://git.scc.kit.edu/cm-tm/cm-team/healthcare/clinicsassetmanagement/medicaldevicemanager



The implementation and test process for application microservice is infrastructure-driven meaning the infrastructural functionality, such as communication with other microservices or API controllers, is implemented first. Then, the application-related logic is implemented inside the operations folder utilizing BDD practices. At the end, end-to-end tests are developed at the application test repository.

(Addition of the Existing…) An application microservice implementation begins with the creation of a GitLab repository where the DevOps template for the technology chosen for the implementation is cloned. The pipeline of the template should be extended to trigger the pipelines of all microservices that request resources from the MS under implementation.
(... Application Relation View ...) Entities, value objects, etc. from the application relation view are implemented first as they build the model of real-world concepts.
(... API Specification ...) Creation of the API specification (e.g OAS).
(... API Controller ...) Implementation of the API specification as a controller. For example, in gRPC a gRPC service is created with the use of @GrpcService annotation. Unit tests are written for the controller. Mocks are utilized.
(... Connectors …) In the task processes the resources required from other microservices or external services (e.g., FROST server) are modelled. They are used for the creation of connectors which make HTTP requests to external resources and map responses to the format of the technology used for the implementation (e.g., map responses to Java objects). Unit tests and white-box integration tests are implemented. They resemble each other in that test data input is fed to the functions under test and the output is matched against expected output. The difference is that unit tests test the functions in isolation, while the integration tests require a bigger part of the system.
(... Backend Gherkin Features ...) These features manipulate application code directly. They act as application microservice acceptance tests. The features are derived from the Gherkin features from the requirements analysis phase.
(... Step Definitions …) Step definitions are the implementation of the Gherkin features. Both artifacts serve as input for Cucumber, which tests the microservice.
(... Microservice's Operation ...) Unit tests are derived from the step definitions. Then application code for successful execution of the unit tests is written.
(... Black-Box API Tests) Black-box API tests are developed with Postman. Requests and expected responses are defined. Then, they are exported as a JSON collection. The collection can be executed as black-box integration tests with Newman.



The implementation of the application microservices is based on the approach of Behavior-Driven Development (BDD). Therefore, BDD's Gherkin features are a central artifact in the implementation and test process described in the following step-by-step description.

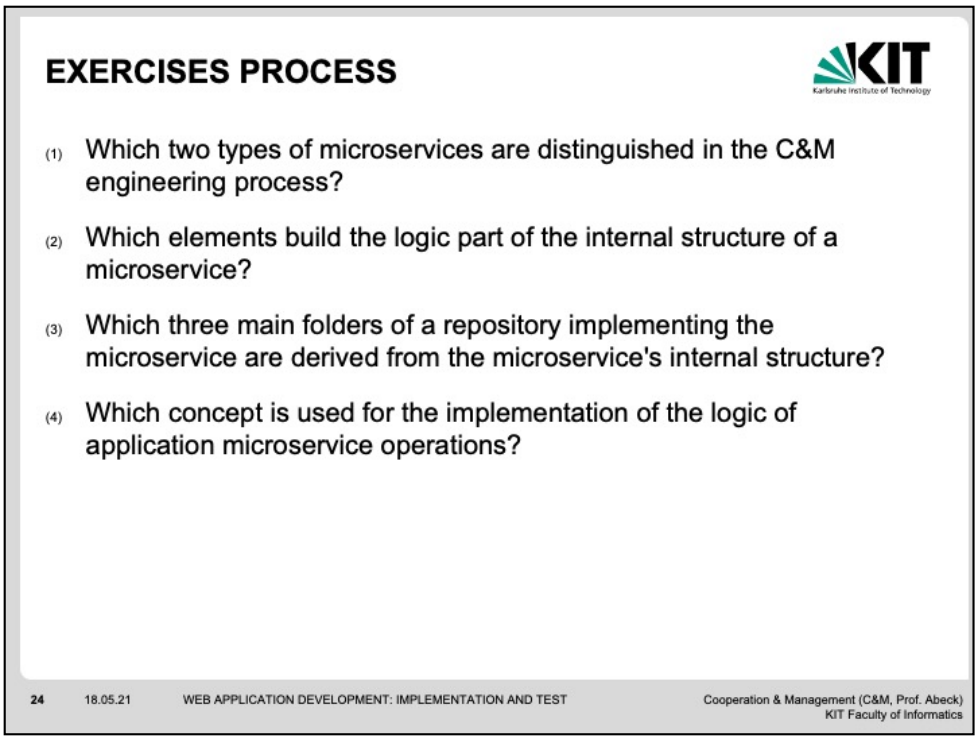
(Starting Point, Implementation and test repositories ...) The process assumes the repository structure as it was introduced in the previous pages. Further, it is presupposed that the implementation of the domain microservices which are accessed in a task process are available.

(Specification of Backend Step Definitions) Starting from a Gherkin feature and an including scenario, for each of the (Given When Then) steps the step definitions are are specified by coding the function calls on the backend side.

(Implementation of a Scenario) The application logic needed to carry out a backend step definition is implemented by writing the required unit tests in a first step and the application code that is necessary to pass the unit tests in a second step.

(Implementation of a User/System Interaction / Capability) After all scenarios belonging to one user interaction are implemented, the next user interaction of the capability starting with the first scenario is implemented. After all user interactions of one capability are implemented, the next capability is implemented scenario-by-scenario.

(Specification and Implementation of Frontend Step Definitions) For each Gherkin feature and including scenario, two types of Gherkin steps describing (i) the backend behavior and (ii) the frontend behavior exist. As soon as the implementation of a scenario fulfilling the backend behavior is available the implementation of the frontend part of this scenario can be carried out.

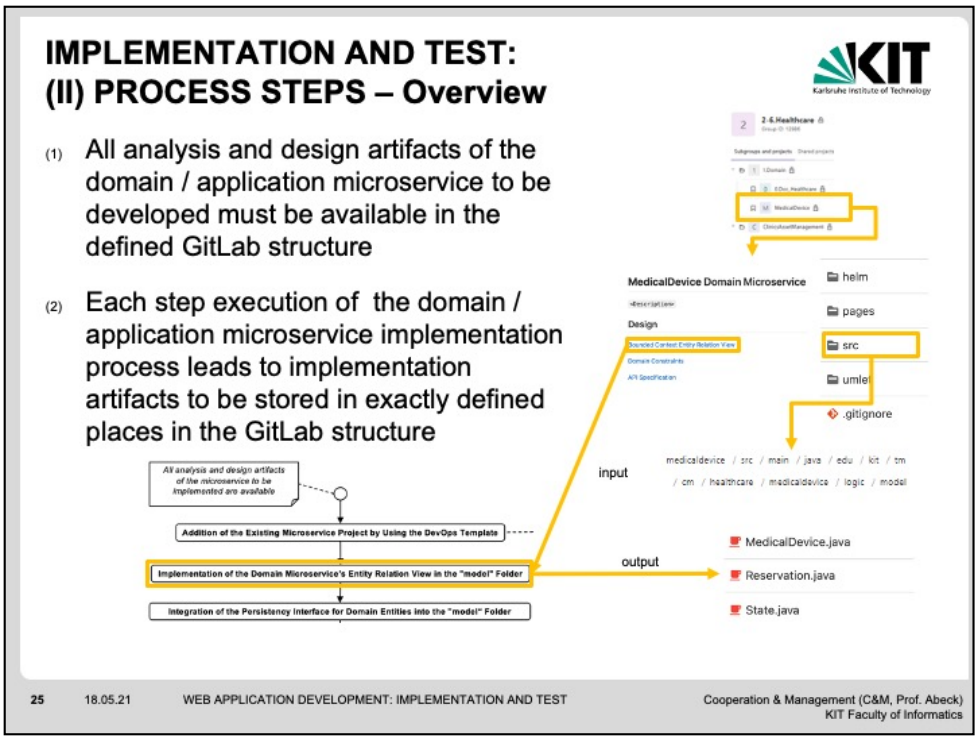


(1) ++Implementation Phase in the C&M Engineering Process++
(i) domain microservice implemented as part of the domain development process
(ii) application microservice implemented as part of the application development process

(2) ++Abstract View on the Microservice Internals++
(i) operations
(ii) entities and value objects

(3) ++Generic Repository Structure Derived from the Microservice Internals++
- The three folders are: (i) api, (ii) logic, (iii) infrastructure

(4) ++BDD-Driven Application Microservice Operations Implementation Process++
- Behavior-Driven Development
- Backend Gherkin features



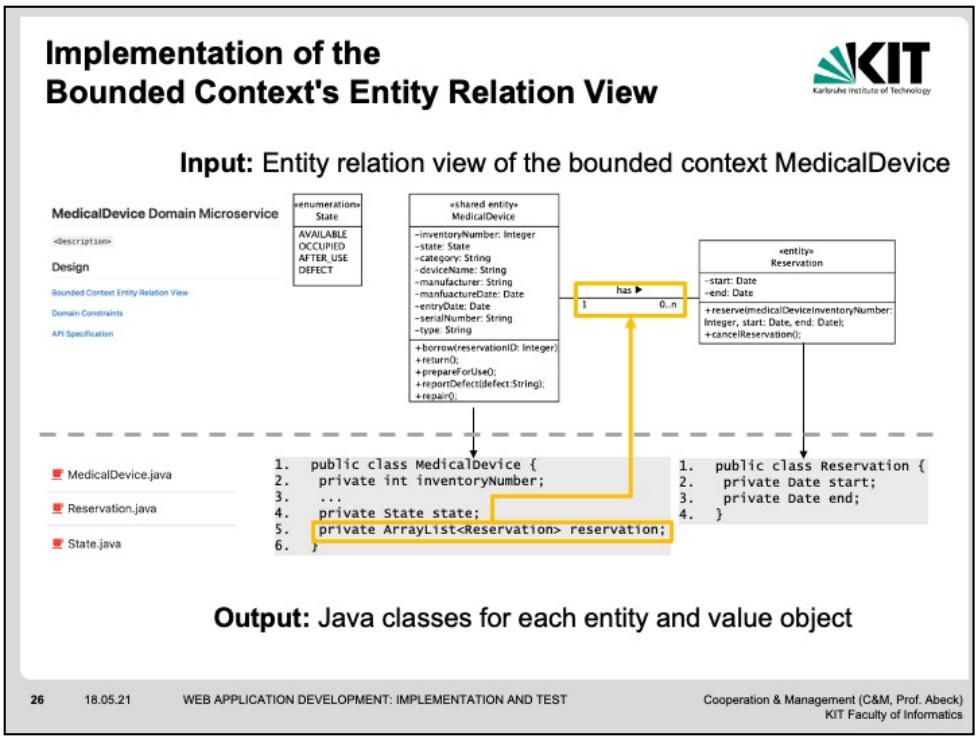
In the second part of the IMPLEMENTATION AND STEP chapter, some of the most relevant process steps are described in detail. The guideline [CM-G-CAM] provides a more complete description of such implementation and test details.

(1) The defined GitLab structure is necessary to be able to establish a precise step description. The structure was introduced in a previous chapter named TOOL ENVIRONMENT.

(2.6 Healthcare) This is the subgroup which contains healthcare domain and all applications of this domain, such as the ClinicsAssetManagement application which serves as a continuous example in this course unit.

(2) Each step requires as input certain analysis and/or design artifacts which must be available at a defined place in the GitLab structure. The output of each step is one or more implementation artifacts which must be placed at predefined places in the GitLab structure.

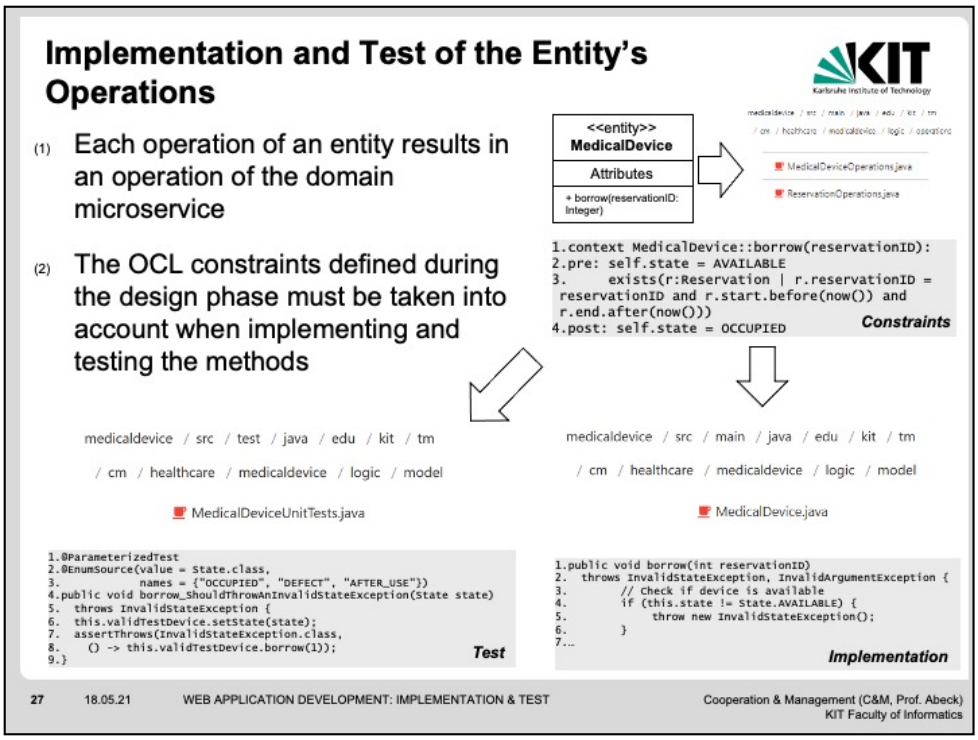
(Implementation of the Domain Microservice's Entity Relation View in the "model" Folder) The input/output concept is shown with the example of this step which was introduced in ++Implementation and Test Process of Domain Microservices++.
(Design Bounded Context Entity Relation View, input) The Bounded Context Entity Relation View, which in this example stems from the MedicalDevice domain microservice, serves as input.
(medicaldevice/src/.../logic/model, output) The output are Java classes which are stored in the stated folder of the GitLab structure.



This implementation step "Implementation of the Domain Microservice's Entity Relation View in the "model" Folder" takes the entity relation view of a bounded context as input and provides a class for each entity and value object as output. The bounded context MedicalDevice leading to a domain microservice serves as an example. The technology used for the implementation is Java and the framework Spring Boot.

(Input: entity relation view of the bounded context MedicalDevice) The entity relation view is a DDD-based UML diagram modeled with UMLet. The diagram can be found in the repository at HealthCare > Domain > <<subdomain>> ClinicsManagement > <<bounded context>> MedicalDevice

(Output:Java classes for each entity and value object) The implementation of the classes follow the modeled entity relation view. Each modeled attribute is implemented accordingly as an attribute of the Java class in the code. Relations are implemented according to their meaning. In the CAM case, a MedicalDevice has Reservation. Therefore, the implementation of a MedicalDevice implements Reservation as an attribute.



In this implementation step, the operations of the entities being part of the entity relationship view of the bounded context are implemented and tested.

(1) The implementation of the methods is stored in the GitLab as a "src" subfolder ".../microservice_name/logic/operation". (+ borrow..., MedicalDeviceOperations.java) In the following this implementation step is shown with the example of the operation borrow() of the entity MedicalDevice.

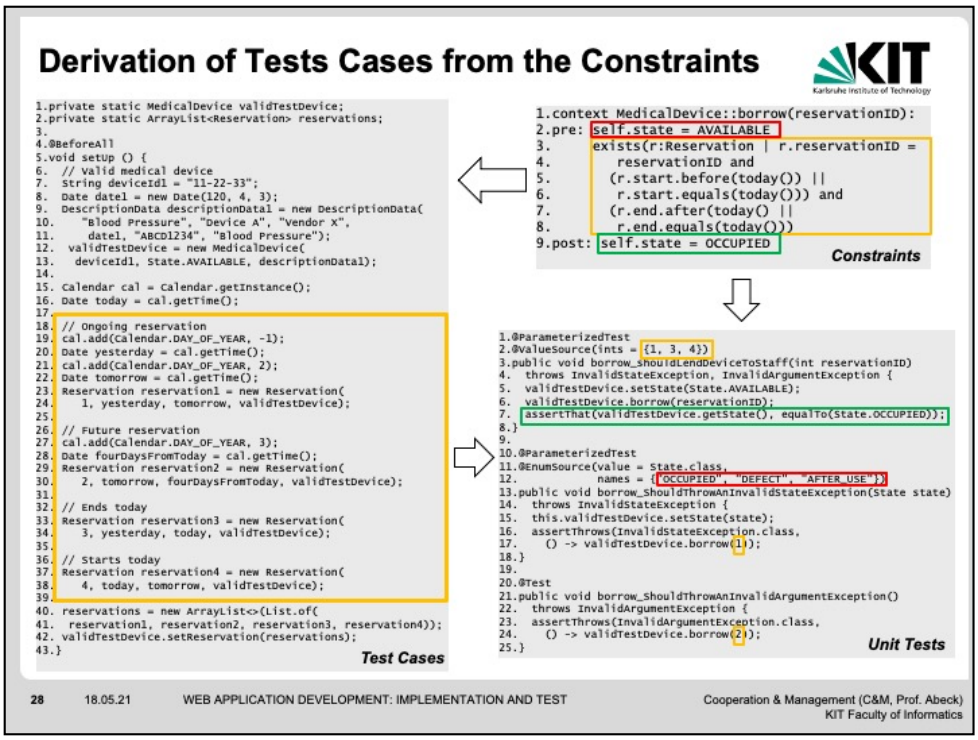
(2) The OCL constraints are defined based on the bounded context's entity relation view as explained in the chapter DESIGN. During implementation, preconditions and attribute invariants are enforced through "if" expressions in the source code. For the tests, the postconditions are important as well. Each method has multiple unit tests. One unit test feeds the method under test with valid input derived from the preconditions and attribute invariants and asserts the outcome matches the expected outcome, which is derived from the postconditions. Then, the ability of the method under test to detect input that violates the domain constraints is put to the test. Test data, which violates the domain constraints is fed to the function under test.

(Constraints) (1. context MedicalDevice ...) In the constraint [CM-G-Med] two preconditions are defined which must be met when the method borrow() is executed:
(i) The medical device must be available (line 2.).
(ii) The reservationID must be valid which means that the current time in which the method is called (now()) lies between the start time and the end time of the reservation identified by reservationID (line 3.).
(4. post ...) When the method is executed, the medical device changes its state to occupied.

(Implementation) (4. if (this.state = ...) In the example, before the borrow() method executes, the state of the medical device is checked.
(5. throw ...) If the device is not available, an InvalidStateException is thrown.

(Test) (2. @EnumSource ...) (3. names = ...) In the example, the method under test receives each state that should lead to an exception as input.
(7. assertThrows ...) The unit test asserts whether the correct exception is thrown. A minimal unit test suite tests each constraint with one faulty case.

[CM-G-Med] Cooperation & Management: MedicalDevice Domain Constraints, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/2-6.healthcare/1.domain/medicaldevice/-/blob/master/pages/domain_constraints.md



This page clarifies the utilization of the domain constraints during testing of a domain microservice. The domain constraints (Constraints) are used for the derivation of test data (Test Cases). The data covers different test cases, which are used by the unit tests (Unit Tests), resulting in tests that assure the system complies to the constraints. In the following, a closer look is taken at the test example from the previous page.

(Constraints) This listing shows the constraints for the method borrow() of the entity Medical Device. The constraints describe the rules of the domain. In this case, the medical device must be available (marked with red) and the reservation id given by the user must match the id of the reservation that is currently active (marked with yellow). If the domain constraints are violated, the execution of the method must be stopped and the user must be notified. Often the domain constraints affect the outcome of a method. In the example, if the preconditions are fulfilled, the state of the medical device should change to occupied (marked with green). The manifestations of the domain constraints in the test cases and unit tests are marked with the respective color.

(Unit Tests) The domain constraints can be violated either by the system or by the user. Hence, two types of unit tests can be differentiated:
(i) (3. public void borrow_...) This unit test assures correct functionality of the method under test. It covers test cases where the system might violate the domain constraints due to improper implementation of the constraint validation or computation errors if the method modifies data. Edge test cases are used as input. In the example, the edge cases are represented by reservations 3 and 4.
(ii) (10. @ParameterizedTest ...) (20. @Test ...) These two unit tests assure the user is not allowed to carry out actions which violate the domain constraints. They cover test cases where the user violates the constraints by giving incorrect input to the system. For example, the unit test at line 10 covers the cases where the user tries to borrow a medical device, which is not available. The unit test on line 20 covers the cases where an incorrect reservation id is given (e.g. an id of a future reservation).

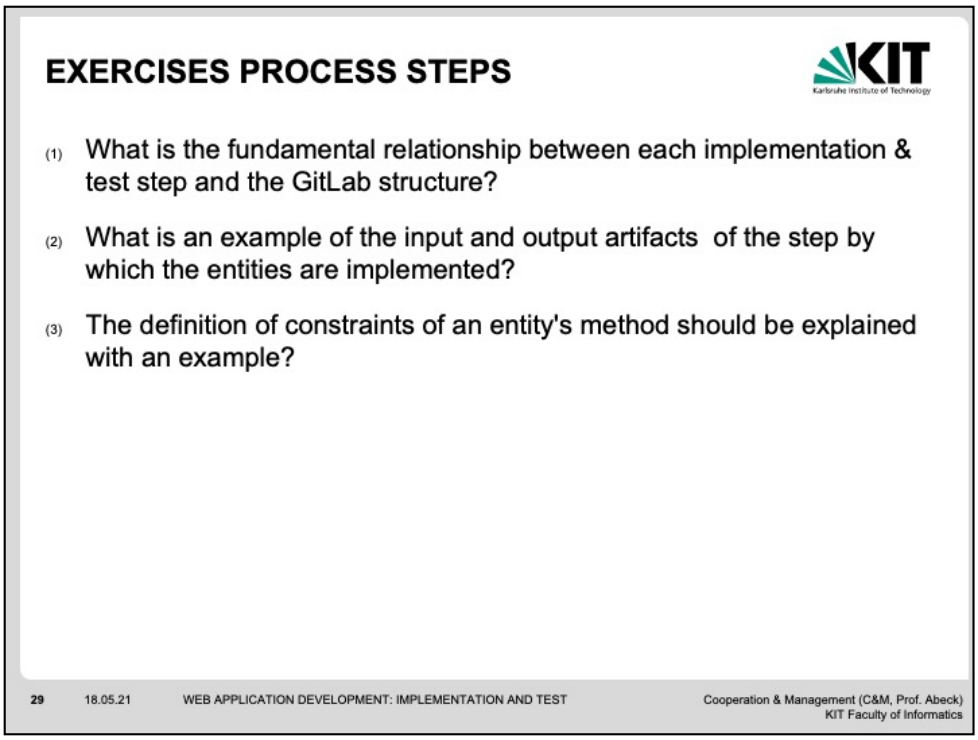
(Test Cases) Test cases are derived from the constraints, whereby the tester considers how they can be violated, either by the user or the system under test. An example for a user violation is the future reservation at line 26. Test cases for a system violation usually are edge cases (e.g. reservations 3 and 4). A minimal test suite should have one faulty test case per constraint. Advanced test suites have multiple test cases per constraint, which cover edge cases.

(5. void setup(){}) There are different ways to initialize test data for the test cases. In the example, the data is set up in the method setup(). However, in more complicated cases a custom ArgumentsProvider can be implemented where the test data can be loaded from external resources (e.g. test database).

(Unit Tests) (12. names = ...) Test cases for the state of the medical device are marked with red. Each test case represents a state that does not allow the execution of the method under test. They are given as input to the test which asserts that the InvalidStateException is thrown properly (line 16).

(Test Cases) (18. // Ongoing reservation) This test case assures the system behaves as expected when correct data is received. It is used as input for the unit test that assures the system does not violate the constraints (Line 1 in Unit Tests)
(32. // Ends today) (26. Starts today) The reservations that end today and start today cover edge cases. They are used as input for the unit test at line 1 in Unit Tests as well.
(26. // Future reservation) The future reservation should lead to an InvalidArgumentsException and is used as input for the unit test which checks whether the exception is thrown properly (Unit Tests, line 20).

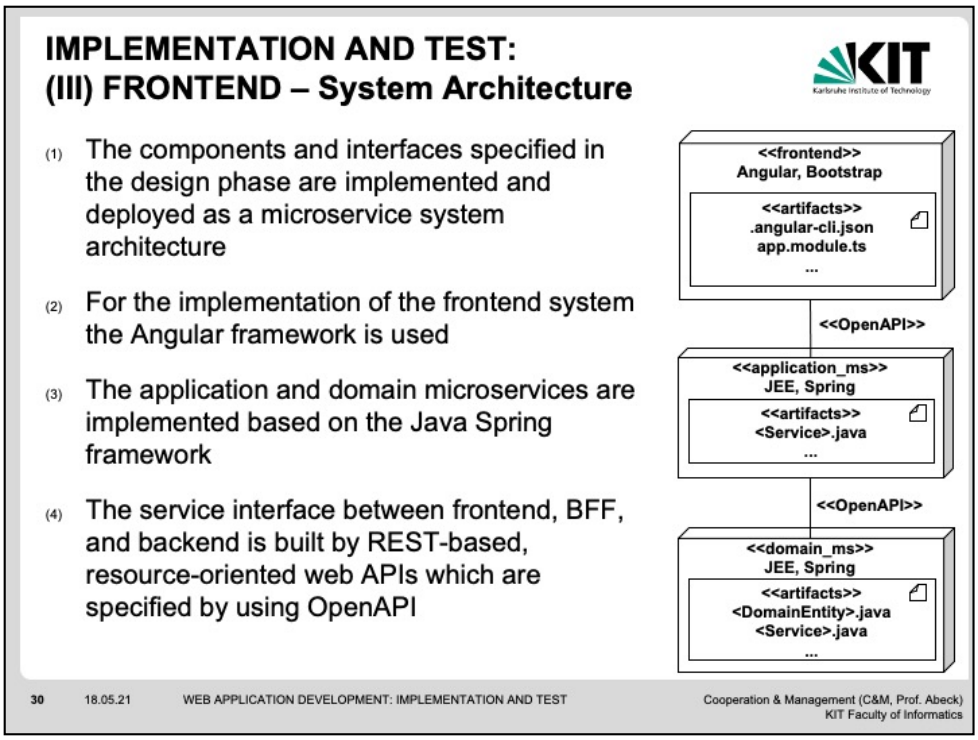
(Unit Tests) (7. assertThat ...) Assertions for the method under test are derived from the post conditions (marked with green). In the example, it is asserted that the medical device state was changed from available to occupied.



(1) ++Overview++
- The Gitlab structure provides all input artifacts of a step.
- All output artifacts are stored in a defined way and place in the GitLab structure.

(2) ++Implementation of the Bounded Context's Entity Relation View++
- Input artifact: Entity Relation View
- Output: Java classes

(3) ++Implementation and Test of the Entity's Methods++
- Defined by preconditions and postconditions
- Example for the method borrow(reservationID):
-- precondition: self.state = AVAILABLE
-- postcondition: self.state = OCCUPIED



The microservice system architecture consists of a frontend, backend-for-frontend (BFF) and a backend. Angular is used as implementation technology for the frontend and Spring as implementation technology for the BFF and the backend microservices.

(1) A microservice architecture divides the software system into three distributed subsystem (or system parts) frontend, BFF, and backend. These systems communicate via resource-oriented web APIs.

(Diagram) The Unified Modeling Language (UML) provides the deployment diagram to model this physical view on the software application [MH06] as shown on the right hand side of the slide. The three-dimensional boxes represent the UML modeling element of a so-called node (which describes a computer system). On a node contain so-called artifacts are located which describe software programs running on that node.

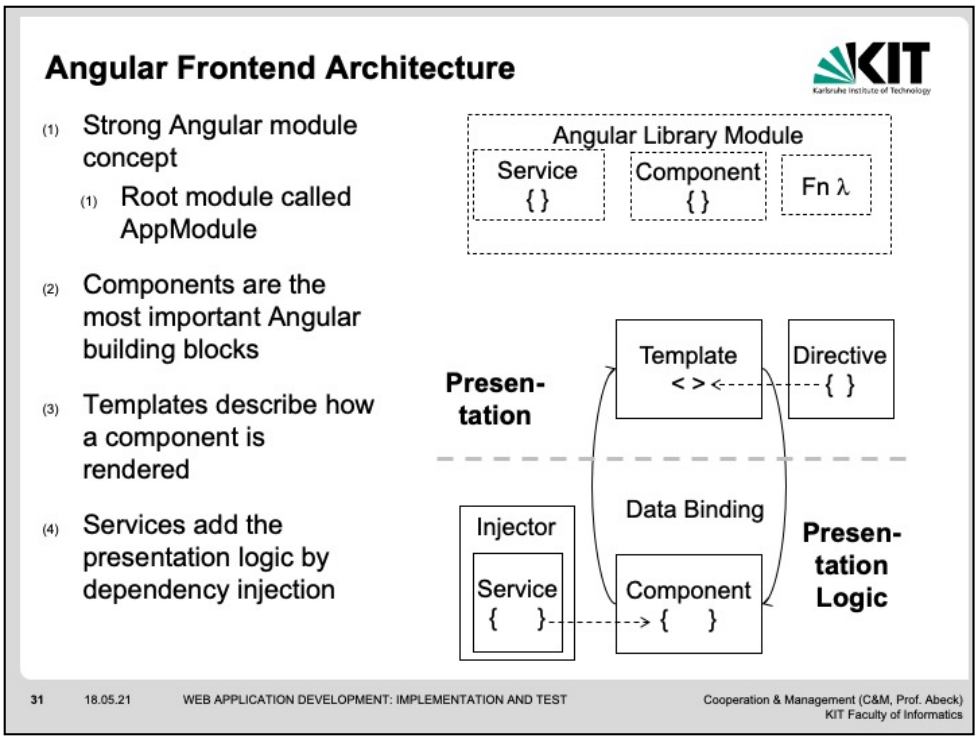
(2) Angular is a popular open-source framework to implement the frontend based on HTML (HyperText Markup Language) and Typescript which is an extension of JavaScript.
(.angular-cli.json) A JSON file (JavaScript Object Notation) which contains configuration information of the whole project setup.
(app.module.ts) A Transcript file which defines the modules that are bootstrapped by the framework at its start.

(3) Spring is one of the most popular open-source development framework for Java-based application servers. Many available open JavaEE technologies, such as Java Persistence API (JPA) are used and extended by Spring.

(4) The three system parts of the microservice architecture (usually) run on distributed client and server systems which communicate via the Web. The way how specification of the web APIs are specified is described in a previous chapter.

HTML	HyperText Markup Language
JPA	Java Persistence API
JSON	JavaScript Object Notation
UML	Unified Modeling Language

[MH06] Russ Miles, Kim Hamilton: Learning UM 2.0, O'Reilly, 2006.



This page gives an overview of the modular architecture the Angular framework is based on.

(1) An Angular module is a class decorated with NgModule. Decorators (such as @NgModule) are functions that modify JavaScript classes.
(1.1) Each Angular App has at least one such Angular module class, the root module (conventionally called AppModule).
The Angular module system is complementary to the JavaScript module system in which each file is a module and all objects defined in the file belong to that JavaScript module. The two types of module systems must be well separated which is not that easy since both use the same vocabulary of "imports" and "exports".
(Library Module) A library module is a collection of JavaScript modules. Angular ships with several such library modules, called Angular libraries each beginning with the @angular prefix. Examples are: @angular/core, @angular/platform-browser.

(Metadata) Describe how a class needs to be processed. They are inserted into classes with decorators.

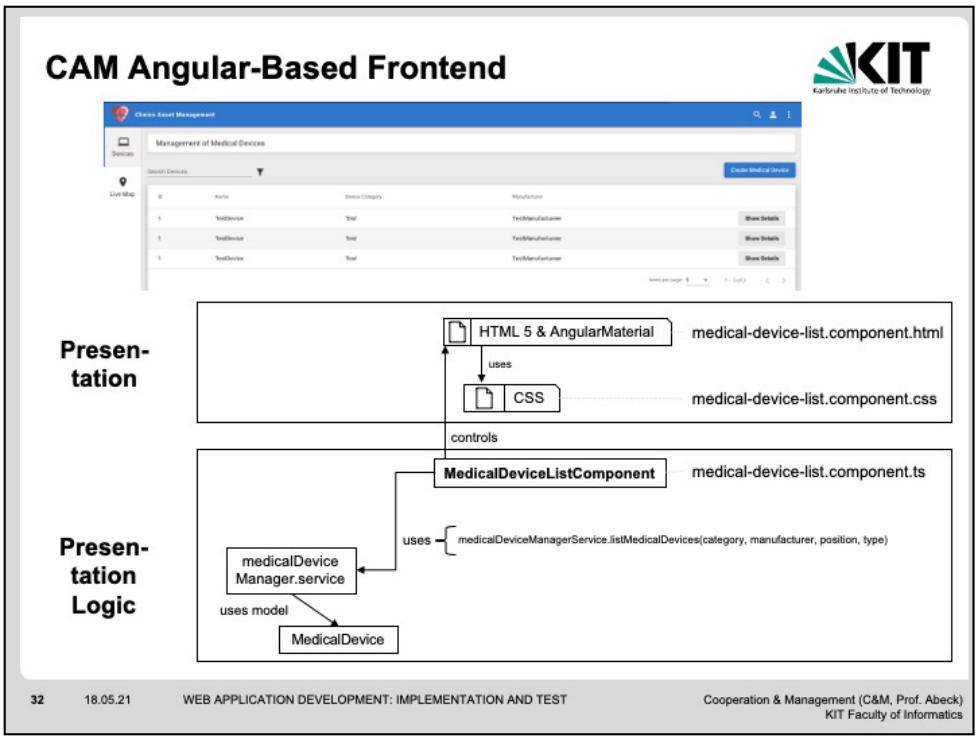
(2) A component controls parts of the screen called view and each defined by a template. The component supports the view by providing presentation logic which is implemented as a class.

(3) A template is a form of HTML that tells Angular how to render the component. Directives give instructions on how templates should be rendered.
(Property Binding, Event Binding) The mechanism for coordinating parts of a template and parts of a component is supported by the data binding concept of Angular. Data binding is conducted by adding markup bindings (specific forms) to the HTML template to tell Angular how to connect the component with the Document Object Model (DOM).

(4) A service can be nearly everything, such as a logging service or data service. Or it can be a tax calculator or an application configuration.
There is nothing specifically Angular about services. Angular has no definition of a service. There is no service base class, and no place to register a service.
(Injector) Angular uses dependency injection to provide new components with the services they need. When Angular creates a component, it first asks an injector for the services that the component requires.

DOM Document Object Model

[Ang-Arc] Angular: Architecture Overview. https://angular.io/docs/ts/latest/guide/architecture.html



The main structure of the Angular architecture is demonstrated with the example of the part of the CAM frontend which presents the data and functions related to a medical device list.

(Clinics Asset Management, Management of Medical Devices) This is the scrrenshot of the CAM UI which covers the aspects related to the list of medical devices.

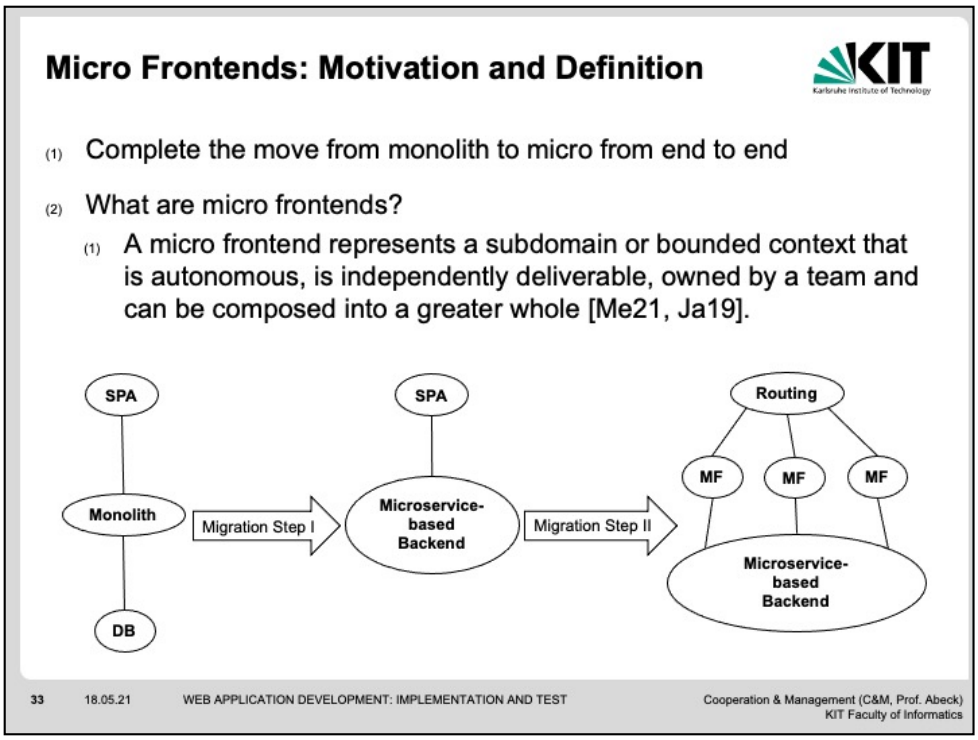
(HTML 5 & AngularMaterial) The HTML defines which elements are shown in the browser. It includes Angular syntax like "*Ngfor" which is used for addressing each list element. The todolist.component.html uses the Bootstrap element "card".

(CSS) Elements can use custom style sheets (CSS). These are defines in the "*.css" classes.

(MedicalDeviceListComponent) The file "medical-device-list.component.ts" handles the actions from the HTML elements. For example, if a button on the web page is pressed, it is forwarded to this file. If an action requires a request, this is forwarded to the service.

(medicalDeviceManager.service) This is the interface to the application microservice MedicalDeviceManager which forwards every request. The medicalDeviceManager.service class is integrated by dependency injection.

CSS Cascaded Style Sheet



Currently used frontend frameworks, such as Angular or React, lead to a monolith on the frontend side. A new frontend concept is to apply the microservice approach not only on the backend side but also on the frontend side. This concept is called micro frontend.

(1) The micro frontend concepts complements the overall move from monolithic to micro architectures in the frontend. As of today, many software projects design scalable microservice-based backend systems, while the corresponding frontend application is a Single-Page-Application (SPA), which can be considered as a UI monolith.

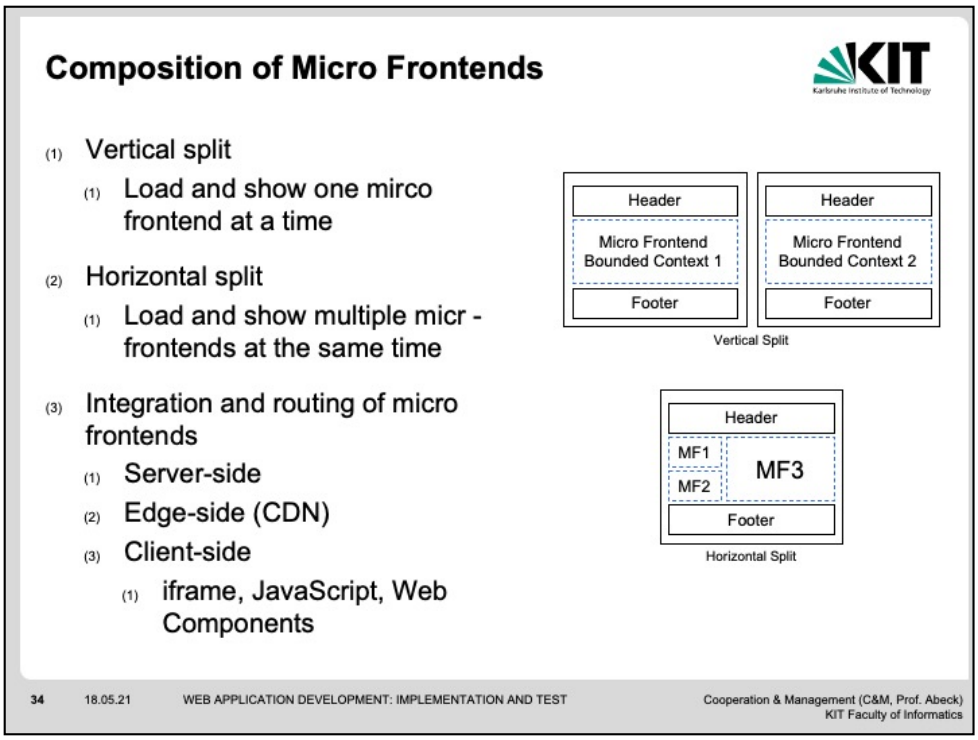
(2) The micro frontend concept aims to scale the delivery and operation of frontend applications, by providing small, less complex, independently deployable applications that are easier to maintain and allow a higher degree of innovation.

(Migration Step I) This step leads to a transformation of the backend monolith into application and domain microservices on the backend side.

(Migration Step II) By this step the frontenend monolith, i.e. the single-page application (SPA), is transformed into micro-frontends (SPA).

SPA Single-Page Application

[Me21] Luca Mezzalira: Building Micro-Frontends, 2021.
[Ja19] Cam Jackson: Micro Frontends, 2019. https://martinfowler.com/articles/micro-frontends.html



(1) In the vertical split scenario one team owns an entire view and is fully responsible for it. This approach enables a clearer separation of concerns, where one team is responsible for one view and the corresponding bounded context/subdomain.

(2) The horizontal split can be more challenging. The loading of multiple micro frontends within a single view could result in the duplication of dependencies, which could have a negative performance impact. To meet this challenge it can make sense to share dependencies across micro-frontends, but the way this is done needs to be considered carefully. Additionally, this approach most likely requires some sort of alignment between the different micro frontend teams.

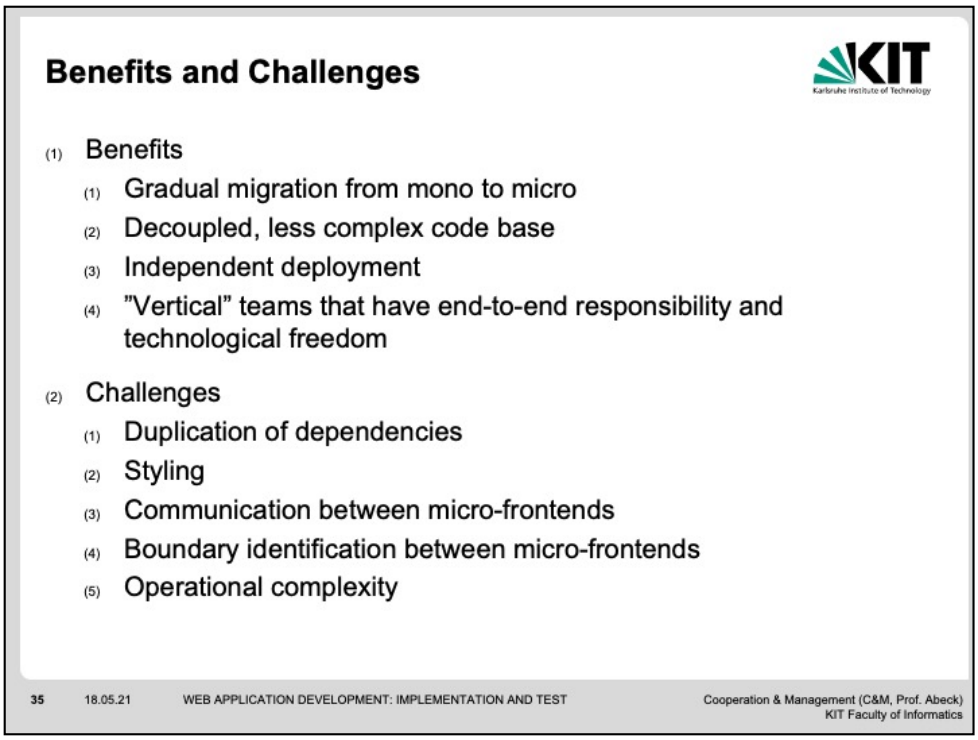
(3) For composing different micro frontends a container application, which handles cross-cutting concerns like the rendering of common page elements (header, footer), authentication and navigation is commonly used [Ja19].

Requested pages can be server-side composed by template composition. For example, depending on the requested URL server-side includes is used to plug-in different HTML templates/fragments into an index.html file that contains the common page elements.

Edge-side composition is possible with the tag-based edge-side includes (ESI). Next to ESI, content delivery networks (CDN) can be used to cache a previously server-side composed page, which can then be served to other clients requesting that page. This works well as long as there is no or limited personalization of displayed content.

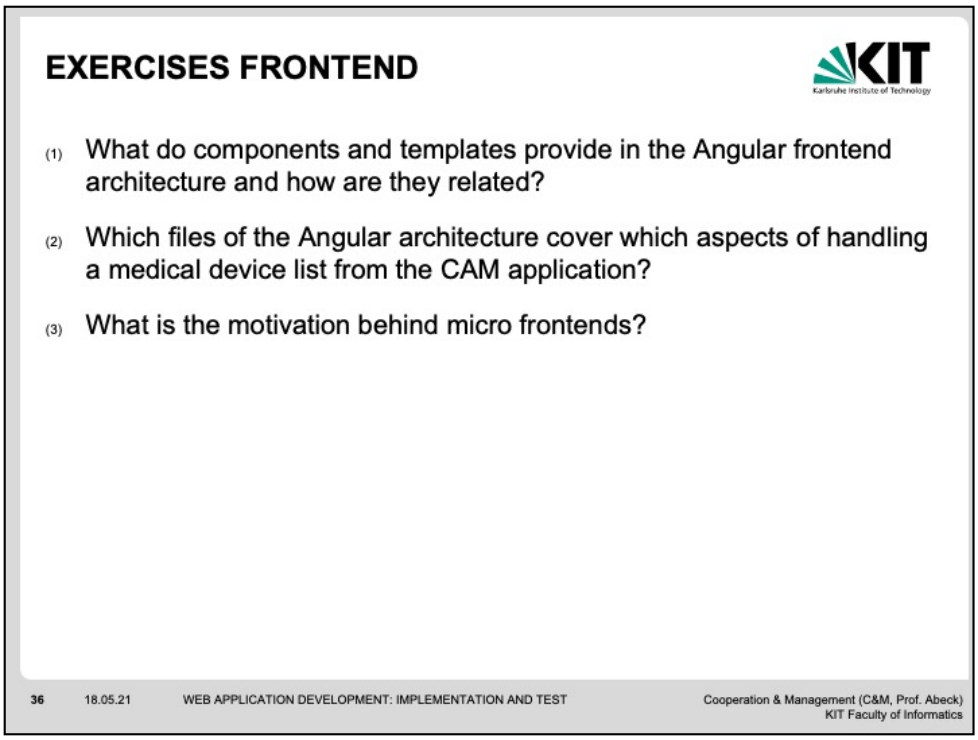
For the client-side integration of micro frontends several technological approaches are possible including iframes, web components or appending of nodes to the document object model (DOM) using JavaScript.

[Me21] Luca Mezzalira: Building Micro-Frontends, 2021
[Ja19] Cam Jackson: Micro Frontends, 2019, https://martinfowler.com/articles/micro-frontends.html



(1) Micro frontends come with several benefits, which are similar to the benefits associated with the microservice architecture for backend systems. The biggest benefit is of organizational nature, since development teams have full ownership of their part of the system with clear lines of responsibility and the ability to freely innovate.

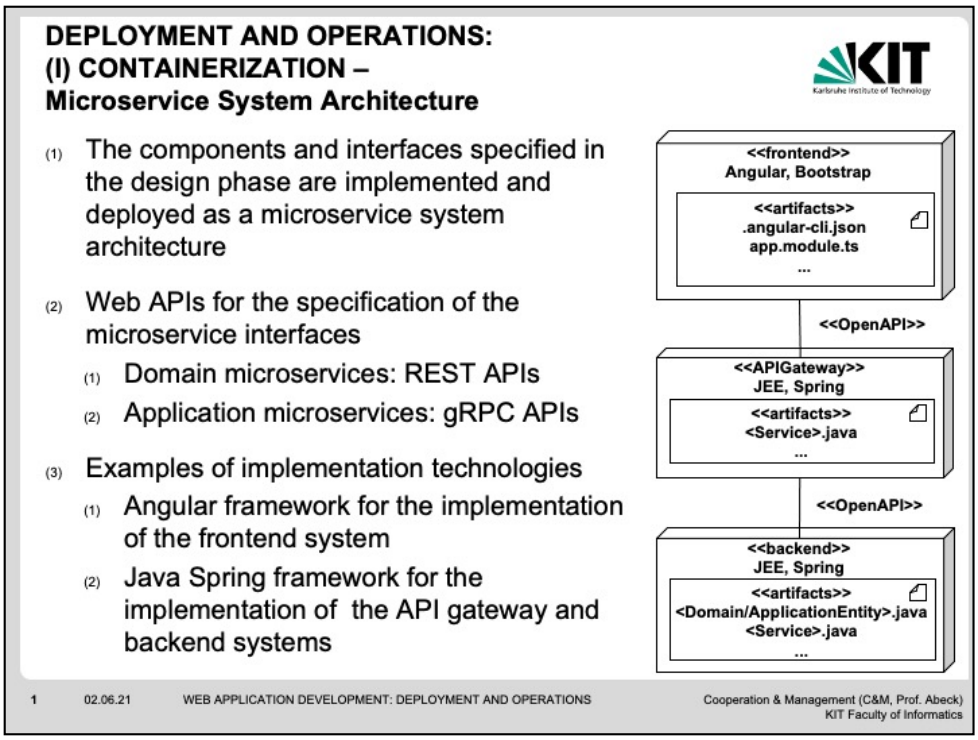
(2) Micro frontends have several advantages, but they are not the right fit for any system. It always needs to be decided case by case if micro frontends are the right choice for the frontend application, since the approach also has its downsides.
(2.2) Avoid conflicting style declarations, Shared UI component library is an option.
(2.3) Avoid communication as it can result in direct coupling between micro frontends. When communication is necessary the URL, custom events or the passing callbacks and data downwards can be used.
(2.5) Common DDD challenge
(2.6) Common microservice challenge



(1) ++Angular Frontend Architecture++
- Component: controls parts of the screen called view which are defined by a template. The component supports the view by providing application/presentation logic which is implemented as a class.
- Template: is a form of HTML that tells Angular how to render the component.
- Relation: The mechanism for coordinating parts of a template and parts of a component is supported by the data binding concept of Angular. Data binding is conducted by adding markup bindings (specific forms) to the HTML template to tell Angular how to connect the component with the DOM (https://angular.io/guide/architecture#data-binding).

(2) ++CAM Angular-Based Frontend++
- medical-device-list.component.htm: defines which elements are shown in the browser
- medical-device-list.component.css: defines the layout of the HTML elements
- medical-device-list.component.ts: defines the presentation logic

(3) ++Micro Frontends: Motivation and Definition++
- The split of UI monoliths (esp. Single Page Applications, SPA) to a micro architecture in the frontend



The microservice system architecture consists of a frontend, an optional API gateway and backend systems for each domain microservice an application microservice.

(1) A microservice architecture divides the software system into three types of distributed subsystem (or system parts). These systems communicate via resource-oriented web APIs.

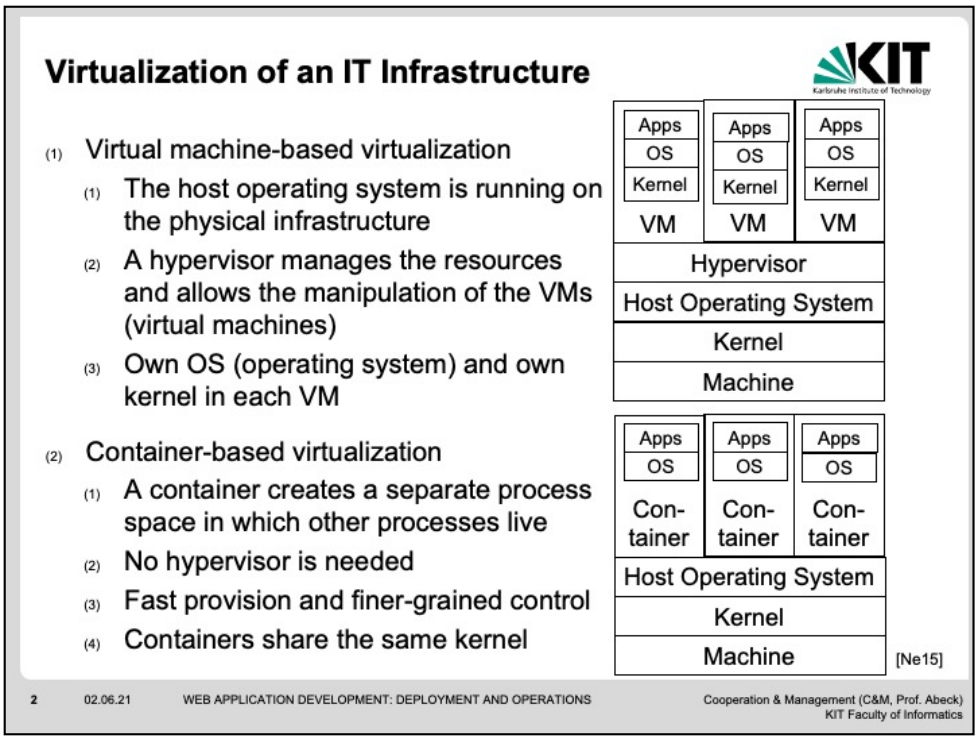
(2) The three system parts of the microservice architecture (usually) run on distributed client and server systems which communicate via the Web.
(2.1) (2.2) The way how web APIs are specified is described in a previous chapter.

(3.1) Angular is a popular open-source framework to implement the frontend based on HTML (HyperText Markup Language) and Typescript which is an extension of JavaScript.
(.angular-cli.json) A JSON file (JavaScript Object Notation) which contains configuration information of the whole project setup.
(app.module.ts) A Transcript file which defines the modules that are bootstrapped by the framework at its start.
(3.2) Spring is one of the most popular open-source development framework for Java-based application servers. Many available open JavaEE technologies, such as Java Persistence API (JPA) are used and extended by Spring.

(Diagram) The Unified Modeling Language (UML) provides the deployment diagram to model this physical view on the software application [MH06] as shown on the right hand side of the slide. The three-dimensional boxes represent the UML modeling element of a so-called node (which describes a computer system). On a node contain so-called artifacts are located which describe software programs running on that node.

HTML	HyperText Markup Language
JPA	Java Persistence API
JSON	JavaScript Object Notation
UML	Unified Modeling Language

[MH06] Russ Miles, Kim Hamilton: Learning UM 2.0, O'Reilly, 2006.



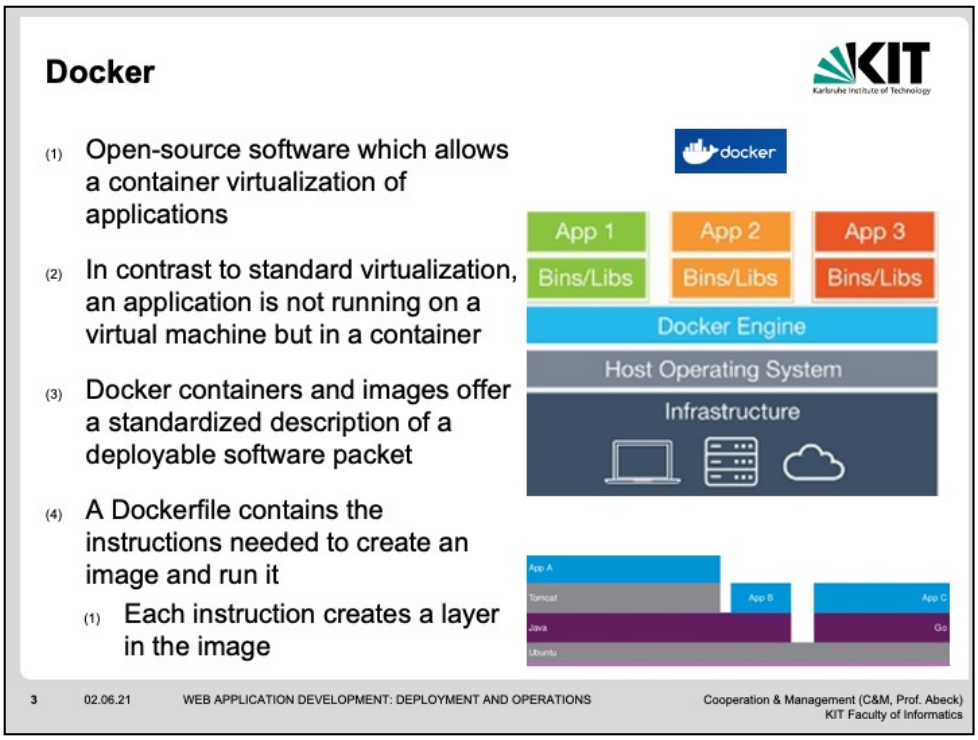
Virtualization allows to divide a physical system into separate virtual parts (machines, containers) each of which can run different things. This concept is especially interesting for microservice architectures where a microservice is one of these things.

(1) The virtualization based on virtual machines (VM) is also called "standard" virtualization.
(1.1) The use of VM-based virtualization technology among Unix, Linux and Unix-like operating systems started around 2005.
(1.2) If the hypervisor directly runs on the hardware, it is called type 1 virtualization. In the case it runs on top of another operation system, it is a type 2 virtualization. Examples of type 2 virtualization technologies are VMWare, AWS, and Xen. Main resources managed by the hypervisor are CPU and memory.
(1.3) Each VM can be considered almost hermetically sealed machines, kept isolated from the underlying physical host and the other virtual machines by the hypervisor.

(2) The concept of a container-based virtualization stems from Linux, more concrete from LXC (LinuX Container) which is available in any modern Linux kernel.
(2.1) A container is a subtree of the overall system process tree.
(2.2) The allocation of resource to a container is done by the kernel.
(2.3) The missing hypervisor and the lightweight design lead to a much faster provisioning of a container (a few seconds) compared with a virtual machine (many minutes). Resources can be assigned to containers in a more targeted way.
(2.4) The kernel is where the process tree lives and must therefore be shared.

AWS	Amazon Web Service
LXC	LinuX Container
VM	Virtual Machine

[Ne15] Sam Newman: Building Microservices – Designing Fine-grained Systems. O'Reilly, 2015.



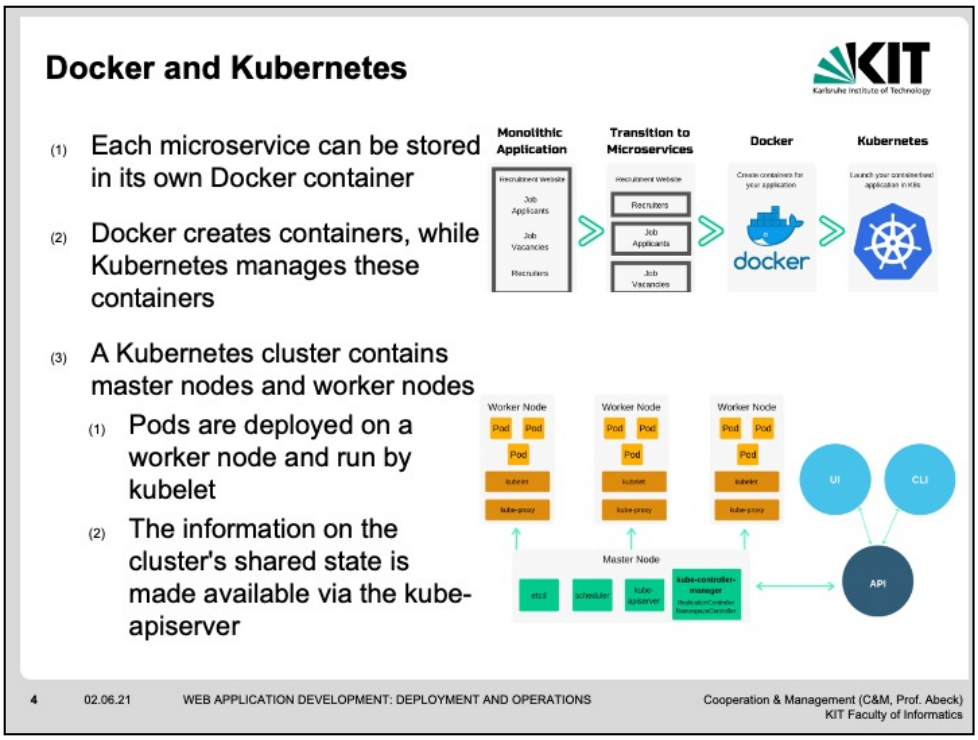
Docker is a software product developed in Go by Docker Inc. which published the first version in 2013.

(1) Docker extends the existing container technology from Linux in a way that it provides a complete and easy-to-use solution for the creation and the operation of containers.

(2) Container virtualization offer applications the isolated usage of resources (CPU, RAM, network) without use of heavyweight virtual machines. Each container runs as an isolated process on the host operating system. This offers important advantages compared to virtual machine virtualizations, such as lightweight resource consumption (esp. CPU, storage), shorter start time, easier distribution.

(3) A container is a runnable instance of an image while an image is a read-only template with instructions for creating a Docker container. The standardized description leads to a portability of containers. In addition, deployment becomes easier and more stable.

(4) The instructions are formulated in a simple syntax. An example of a command that can be used in a Dockerfile is "docker run".
"$ docker run -i -t ubuntu /bin/bash" (i) creates a new container containing an Ubuntu image, (ii) allocates a read-write filesystem as its final layer, (iii) creates a network interface including the assignment of an IP address to the container, (iv) starts the container and executes the /bin/bash command.
(4.1) One important property of layers is that they are shared between all installed images.
In the example, the (operating system) layer "Ubuntu" is shared by all three applications A, B and C, while the (programming language) layer Java is shared by the applications A and B.



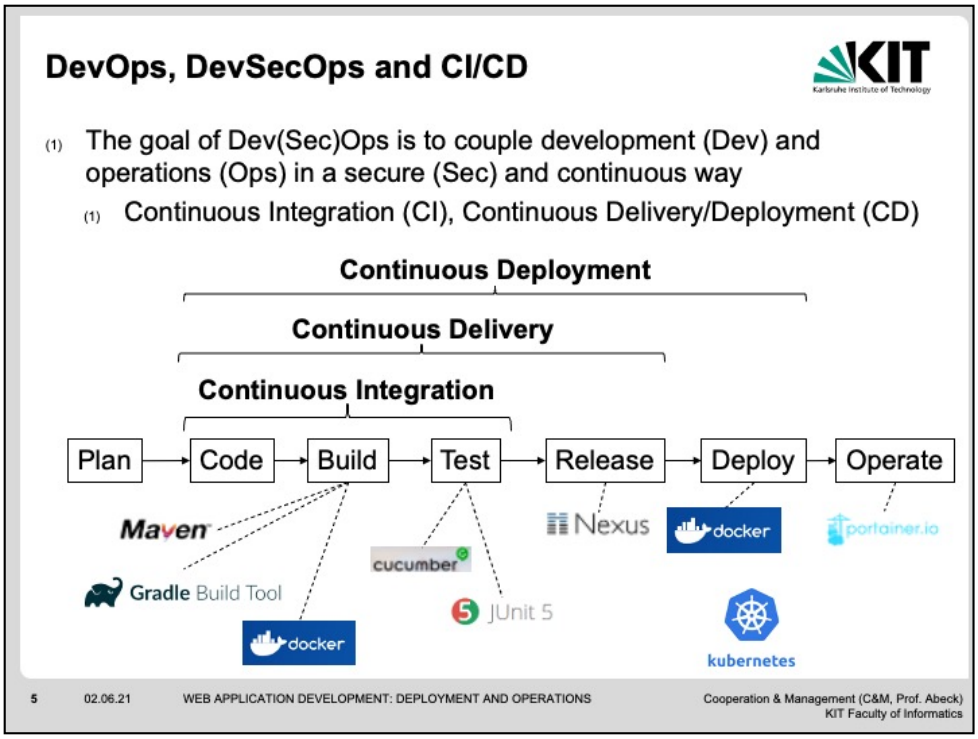
It is important to understand the fundamental concepts Kubernetes is based on [Byt-Kub].

(1) A container is a standardized structure which can store and run any application. Each microservice can be stored in its own container, along with everything it needs to run (settings, libraries, dependencies etc.). This creates an isolated operating environment so the container can run on any machine.

(2) Docker is not an alternative to Kubernetes but the two technologies are used in combination.

(2) Nodes are the hardware components. A node is likely to be a virtual machine hosted by a cloud provider or a physical machine in a data centre. But, it can simpler to think of nodes as the CPU/RAM resources to be used by your Kubernetes cluster, rather than just as unique machines. This is because pods aren't constrained to any given machine at any given time, they will move across all available resources to achieve the desired state of the application. There are two types of node, worker and master, which form the master-slave architecture.
(2.1) (pod, kubelet, kube-proxy) A pod holds one or more container(s). Pods are the simplest unit that exists within Kubernetes. Any containers in the pod share resources and a network and can communicate via the kube-proxy with each other – even if they are on separate nodes. The kubelet is an agent which receives the instruction from the master node.
(2.2) The kube-apiserver provides a bridge between the Kubernetes API and other Kubernetes object (e.g. pods, controllers, services). So all interaction between components goes through the kube-apiserver.

[Byt-Kub] Bytemark: Kubernetes Terminology: Glossary. *https://docs.bytemark.co.uk/article/kubernetes-terminology-glossary/#introduction*

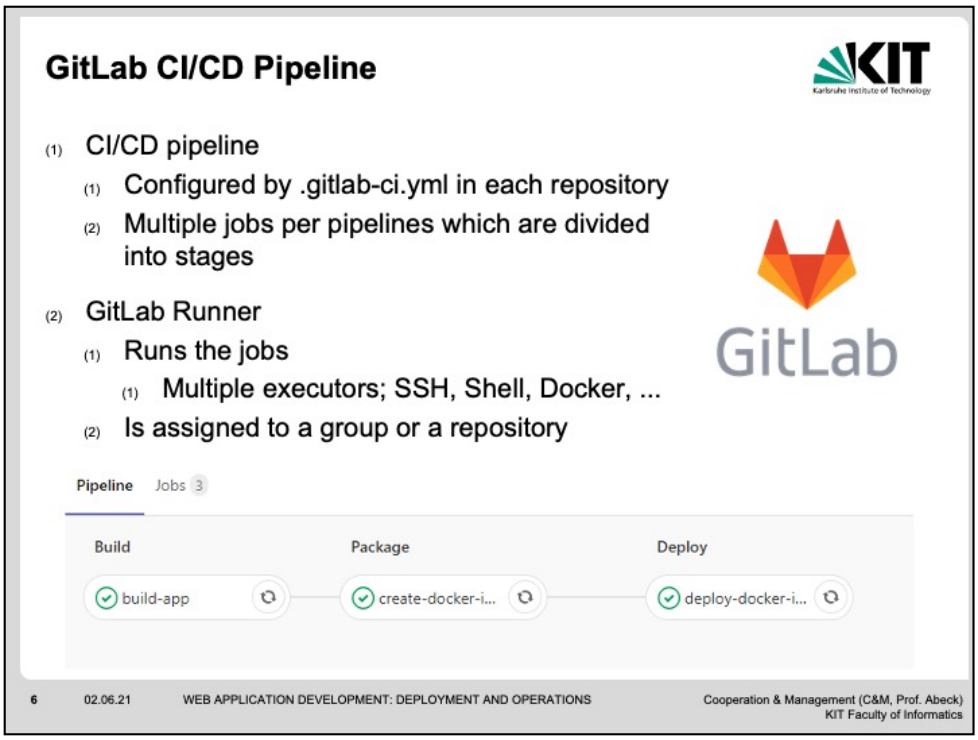


After the software has been implemented it must be built and deployed which requires another specific set of tools.

(1) DevOps contains the two terms development and operations. Since security is an important issue, DevOps is extended to DevSecOps. It summarizes all practices that bring together software development and software operations in a way that the development cycles are shortened and the deployment frequency is increased. Microservice architectures support the DevOps concept since microservices have the property that they can be tested and deployed independently from the whole software system. This is not the case for monolithic systems.
(1.1) Continuous integration (CI) is a DevOps practice by which the code from the involved developers are built and tested in short time intervals (e.g. several times a day) in order to discover and solve integration problems as soon as possible.
Continuous delivery (CD) extends the CI practice by releasing the software after it was built and tested (i.e. continuously integrated).
Continuous deployment (also abbreviated by CD) which is often confused with continuous delivery, means that every change is automatically deployed to production. Since the software must be released before it can be deployed, continuous deployment is an extension of the DevOps practice of continuous delivery.

(Maven) A Java tool by which a project's build, reporting and documentation can be managed.
(Gradle) A Java-based build management tool (comparable to Apache Maven) which uses a domain-specific language
(Cucumber) A library by which user acceptance tests based on Gherkin features can be carried out.
(Junit5) A framework to test Java programs based on unit tests.
(Nexus) A repository for build artifacts produced by Docker, Maven, npm, and others.
(Docker) A software system which supports container virtualization in order to deploy applications in isolated containers.
(Portainer) An open-source management user interface to manage a Docker host or swarm cluster.
(Kubernetes) An open source solution for the container management which is describes in further detail on succeeding pages.

DevOps	Development and Operations
DevSecOps	Development and Security and Operations



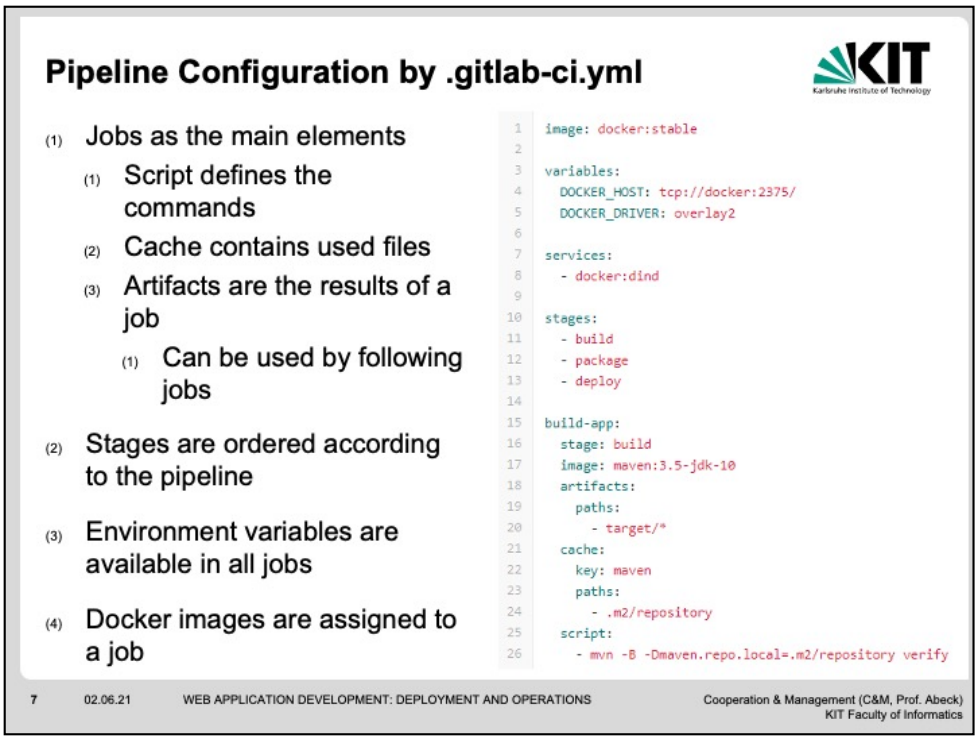
(1) GitLab provides a CI/CD pipeline for each repository.
(1.1) The configuration of the CI/CD pipeline is described in a YAML file named ".gitlab-ci.yml". This file is stored in the root directory of the repository and it contains the definitions of the jobs and the order of their execution. A complete configuration reference can be found in the GitLab documentation [GL-CIC].
(1.2) Each pipeline can contain multiple jobs which can be divided into several stages. Jobs of the same stage can be executed in parallel. Jobs of the next stage can only be executed when all jobs of the previous stage have been finished successfully.

(2) The code of the GitLab Runner is open source, written in Go and designed to run on GNU/Linux, macOS and Windows operating systems.
(2.1) The GitLab Runner is needed to execute the pipeline and its jobs. It can be run on a server or a desktop computer.
(2.1.1) Multiple ways to execute the jobs in so-called executors are supported:
(SSH) The SSH executor can be used to connect to a remote machine and execute the commands of the pipeline job on this machine over SSH.
(Shell) The Shell executor is used to execute the commands of pipeline jobs locally. Depending on the operating system that is used, different shells are used.
(Docker) GitLab Runner can execute jobs on Docker images which are specified in the ".gitlab-ci.yml" by using the Docker executor. To run Docker, commands inside the Docker container needs to be in privileged mode. Executors for VirtualBox, Kubernetes and more are also available.
(2.2) A GitLab Runner can be assigned to a group or a single repository. If a GitLab Runner is assigned to a group, all child groups and repositories of that group can use it.

(Pipeline) The image shows a CI/CD pipeline with three stages: Build, Package, Deploy. Each stage contains one job and the execution of the jobs was finished successfully, as shown by the green checkmarks.

YAML YAML Ain't Markup Language

[GL-CIC] GitLab: CI/CD Pipeline Configuration Reference. *https://docs.gitlab.com/ee/ci/yaml/*



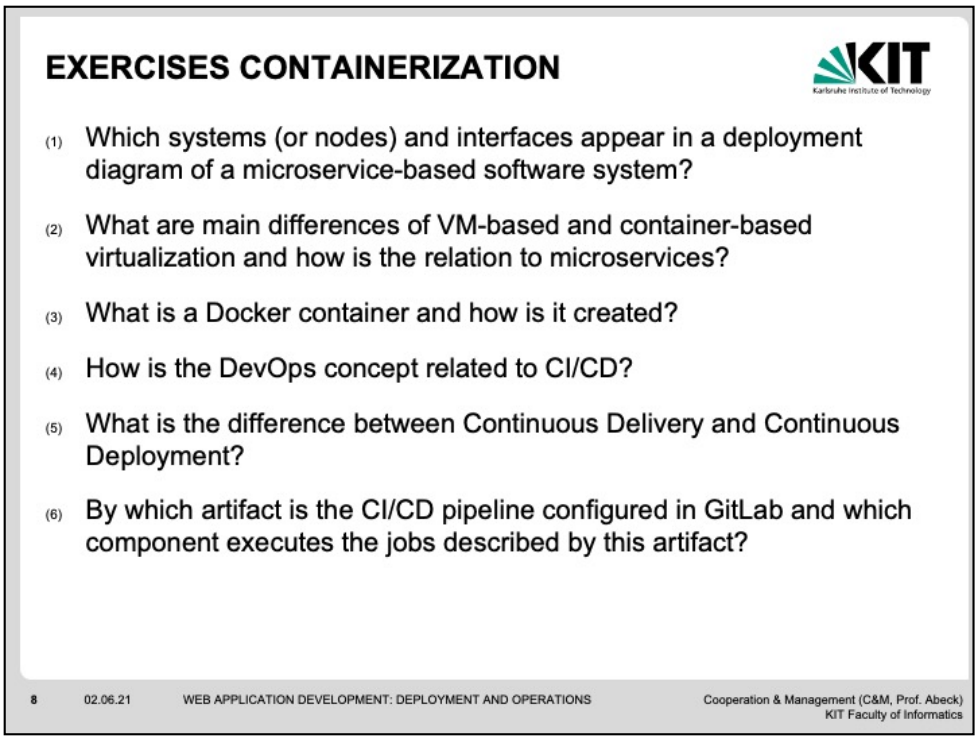
The screen dump on the right shows an excerpt of the .gitlab-ci.yml that is used in the microservice repository of the ElectricCarCharger project.

(1) The .gitlab-ci.yml file contains the configuration of the GitLab CI/CD pipeline for a GitLab repository. The main elements of the pipeline configuration are the jobs. A job has no predefined name or prefix, it can have any name possible. The only limitations are already existing keywords like image, services, stages, and a couple more. The full list can be found in the GitLab documentation [GL-CIC].
(1.1) (25. script) Every job requires the script keyword which defines the commands that are executed in this job. These commands are executed by the GitLab Runner that runs the script in a shell .
(1.2) (21. cache) The files that are used by a job can be cached. This way the files do not need to be downloaded or created for every run. As an example, the build-app job (15.) caches the Maven repository (24.) so it does not have to be created in every run.
(1.3) (18. artifacts) With the artifacts keyword the artifacts that were created by running the job can be copied and attached to the job on GitLab after the job finished successfully. The artifacts of a job can also be used by jobs which are part of a following stage.

(2) Jobs can be assigned to stages. Jobs of the same stage can be executed in parallel. Jobs of the next stage can only be executed if all jobs of the previous stage have been finished successfully.
(10. stages) The stages are defined outside the jobs by using the stages keyword, followed by the stages that should be used. The stages are ordered by their order in the pipeline config.

(3) (3. variables) With the variables keyword it is possible to define environment variables that are available in all jobs of the pipeline.

(4) (1. image) The GitLab CI/CD pipeline allows the use of Docker images. The default Docker image for the pipeline can be set with the image keyword on the same layer in the pipeline config as the stages and jobs. Inside each job the image keyword can be used to assign another Docker image to a job.



(1) ++Microservice System Architecture++
- Nodes: Fontend, API Gateway, Backend system
- Interfaces: Web APIs, e.g. specified by using OpenAPI

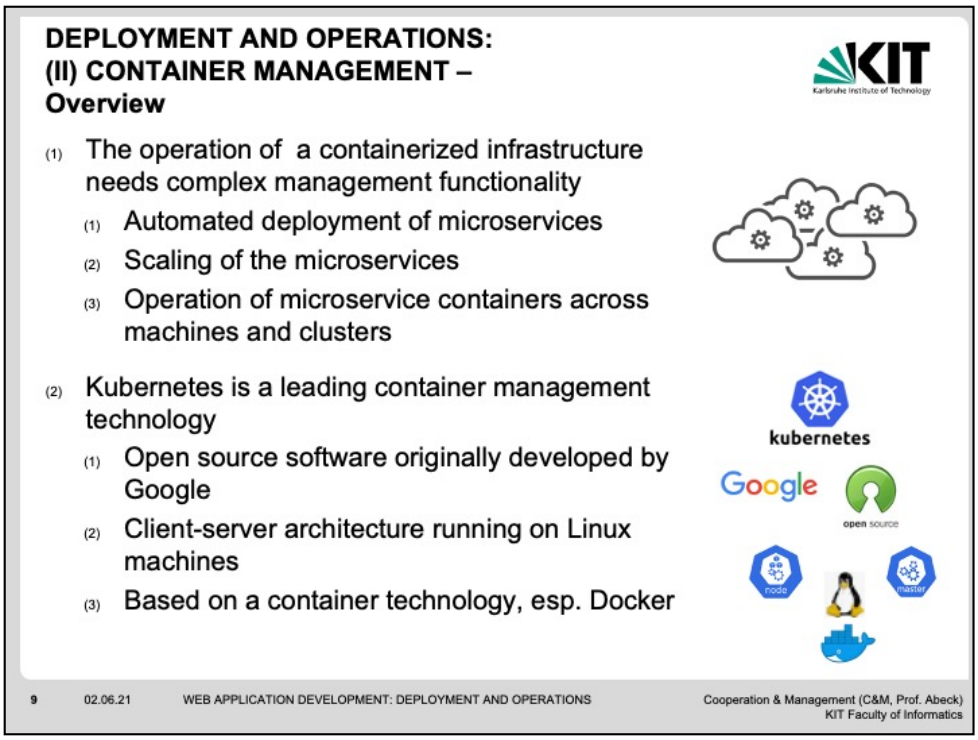
(2) ++Virtualization of an IT Infrastructure++
- Differences
(i) Granularity: Container are much more fine-grained and lightweight compared to VMs
(ii) Performance: A container can be started and stopped much quicker than a VM
(iii) Architecture: Container share a common kernel and need no hypervisor
(iv) Security: VMs run in a more close environment and their protection is easier
- Microservices
-- Microservices usually run on virtual nodes
-- A container is the favorite virtualization unit for a microservice (especially due to the horizontal scaling aspect)

(3) ++Docker++
- A Docker container runs as an isolated process on the host operating system and provides a virtualized environment to run applications.
- A Docker container is a running instance of an image.
- Created by instructions in a Docker file

(4) ++DevOps, DevSecOps and CI/CD ++
- DevOps uses the CI/CD approach to couple development (Dev) and operation (Ops) in a continuous way

(5) ++DevOps, DevSecOps and CI/CD ++
- Continuous Deployment extends Continuous Delivery by automatically deploy a delivered release onto a (usually virtualized) infrastructure

(6) ++GitLab CI/CD Pipeline++
- The artifact is a YAML file named gitlab-ci.yml
- The component is called GitLab Runner which supports multiple ways to execute the jobs (e.g by SSH, Shell, Docker) (1) ++Container Management++
- Automated deployment of a microservice
- Scaling of the microservices
- Operation of microservice containers across machines and clusters



Source: WEB APPLICATION DEVELOPMENT, Chapter DEPLOYMENT

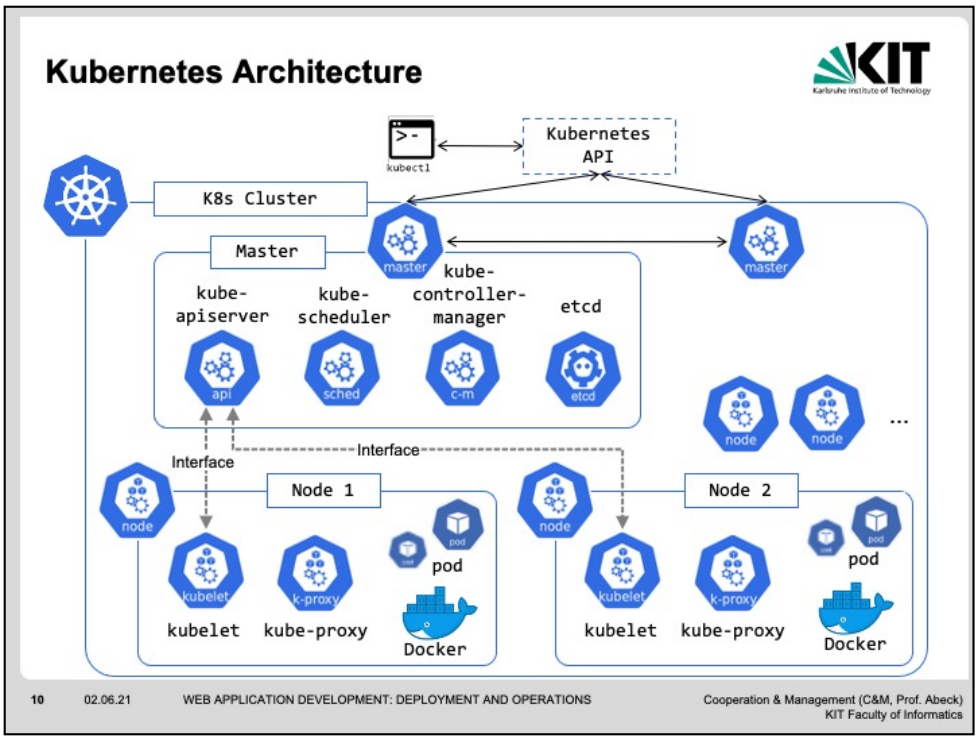
In order to run microservices efficiently, a containerized infrastructure is well suited.

(1) A container technology, such as Docker, provides the technology to establish a containerized infrastructure. But it does not provide the functionality that is needed to manage such an infrastructure.
(1.1) The automated deployment is a central aspect of the DevOps (Development and Operations) concept which makes sure that a change of the software system is continuously integrated into and deployed to the infrastructure. In the case of a microservice architecture deployed on a container-virtualized infrastructure this concerns the container on which the microservice is running.
(1.2) Scaling is concerned with the distribution of resources (CPU, storage) to the microservices. Resources can be added to or withdrawn from a microservice by vertical or horizontal scaling.
(1.3) A complex containerized infrastructure usually consists of more than one cluster and microservice-based applications can be distributed over more than one cluster. Therefore, a powerful container management must provide functionality for a multi-cluster management.

(2) Alternative solutions for the management of container-virtualized infrastructures include Docker Swarm or Kubernetes (short name K8s) [Lin-Kub] which today is one of the most important and powerful technologies in this field.
(2.1) Kubernetes is hosted by the Cloud Native Computing Foundation (CNCF) which serves as the vendor-neutral home for many of the fastest-growing open source projects.
(2.2) In the case of K8s, the client is a so-called node and the server is a so-called master as further illustrated on a succeeding page. Each node runs on a separate Linux machine.
(2.3) K8s supports the Container Runtime Interface (CRI) which is not only provided by the container technology Docker, but also by other technologies such as XEN and Rocket.

CNCF	Cloud Native Computing Foundation
CRI	Container Runtime Interface
K8s	Kubernetes

[Lin-Clo] The Linux Foundation: Cloud Native Computing Foundation. https://kubernetes.io/docs/concepts/
[Lin-Kub] The Linux Foundation: Kubernetes Concepts. https://kubernetes.io/docs/concepts/



Kubernetes (short name K8s, Greek for governor) was first announced by Google in 2014 and its design is heavily influenced by Google's Borg system. While Borg is entirely written in C++, the rewritten Kubernetes is implemented in Go.
The main goal of Kubernetes is to automate the deployment, scaling, and management of applications that run in a container-virtualized infrastructure. One of the most relevant supported container tools is Docker. Since many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) it is a key technology in the field of cloud computing.
Google describes Kubernetes as a platform for the orchestration of containers. Orchestration in this context means that all resources are made available (in a highly elastic and efficient way) in order to provide the services.

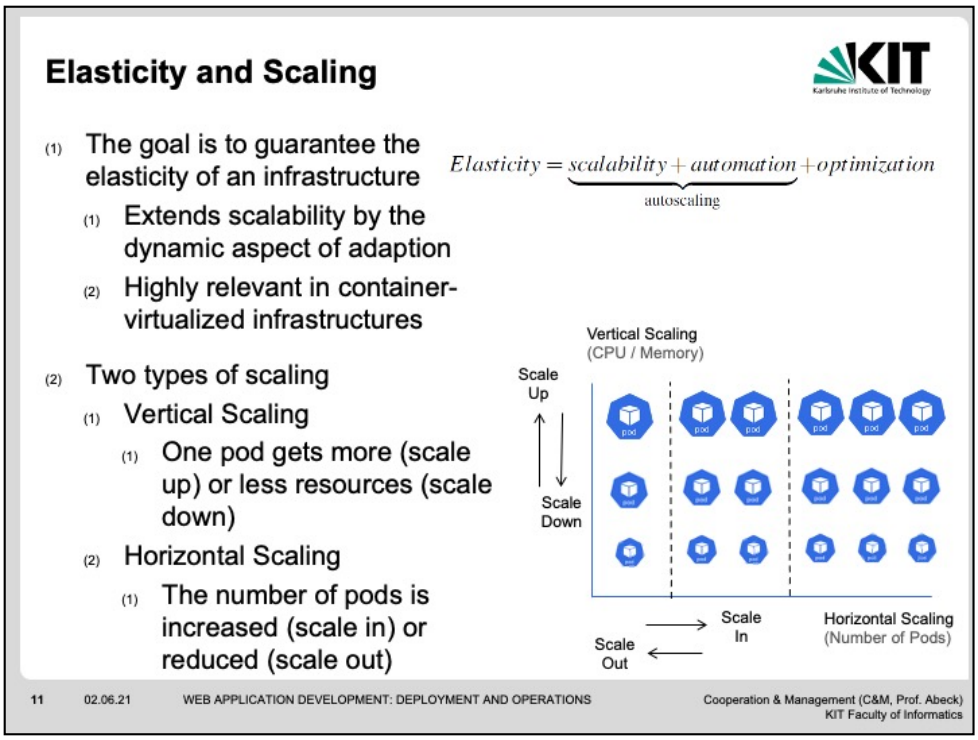
(kubectl) (Kubernetes API) The official Kubernetes command line interface tool (kubernetes control) that serves for the communication with the Kubernetes API which passes the information onto the master node. Via the API Kubernetes objects (e.g. pods and nodes) can be created, inspected, changed, replicated and destroyed [Byt-Kub].

(master) The central management unit of a Kubernetes cluster consisting of the four components (i) kube-apiserver, (ii) kube-scheduler, (iii) kube-controller-manager, and (iv) etcd.
(kube-apiserver) Provides the whole Kubernetes API.
(kube-scheduler) Schedules the pods by placing them in a queue from where they are assigned to a node. Opposed to schedulers in operating systems, it is not responsible for executing the pods.
(kube-controller-manager) Manages all of the controllers (e.g., ReplicationController, NamespaceController).
(etcd) Stores all metadata and configuration data of Kubernetes in a key-value store (i.e. configuration data "/etc" distributed).

(node, Docker) A Kubernetes node, also called minion, is a single server for containers. The container technology used by Kubernetes, such as Docker, must be installed on each node.
(kubelet) Fulfills the task of executing a pod. Since kubelets run on a node, an overload of the master is prevented.
(kube-proxy) Distributes the requests to the container based on a load balancing algorithm.
(pod) This is a running process on the Kubernetes cluster. Each pod contains one or more containers. The one-container-per-pod concept is used in most situations. Containers are able to intercommunicate between pods which have their own internal IP address.

etcd	/etc distributed
IaaS	Infrastructure as a Service
K8s	Kubernetes
PaaS	Platform as a Service

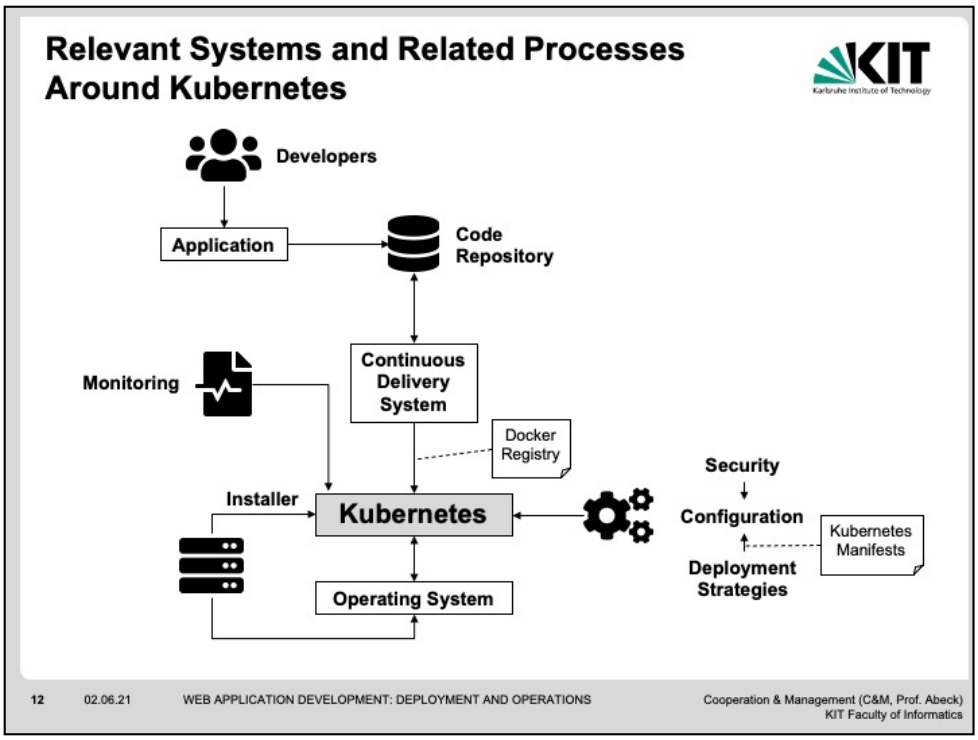
[Byt-Kub] Bytemark: Kubernetes Terminology: Glossary. https://docs.bytemark.co.uk/article/kubernetes-terminology-glossary/#introduction



(1) Elasticity can be expressed by the "formula" scalability + automation + optimization [AP+18] which means that elasticity is an automation of the concept of scalability (a static property of a system) aiming the optimization of the needed resources.
(1.1) Scalability is the time-independent property of a system to be technically able to cope with increasing load by using additional resources. Elasticity adds a dynamic aspect of adaption which leads to the concept of autoscaling.
(1.2) The automated scaling, also called autoscaling, is one of the key features of Kubernetes.

(2) The difference of the two types of scaling is illustrated by the figure.
(2.1) Vertical scaling generally means that a computing node (computer, virtual machine, container, pod) receives more or less resources.
(2.1.1) In the context of Kubernetes, horizontal scaling means changing the resource requests and limits of pods for CPU or memory.
(2.2) Horizontal scaling in general, means that the number of computing node (computer, virtual machine, container, pod).
(2.2.1) In the context of Kubernetes the number of pods changes. This means that more or less pods are running.

[AP+18] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, Philippe Merle: Elasticity in Cloud Computing: State of the Art and Research Challenges, IEEE TRANSACTIONS ON SERVICES COMPUTING, VOL. 11, NO. 2, March/April 2018.



The figure illustrates relevant aspects which must be taken into account when introducing and running a Kubernetes-based infrastructure.

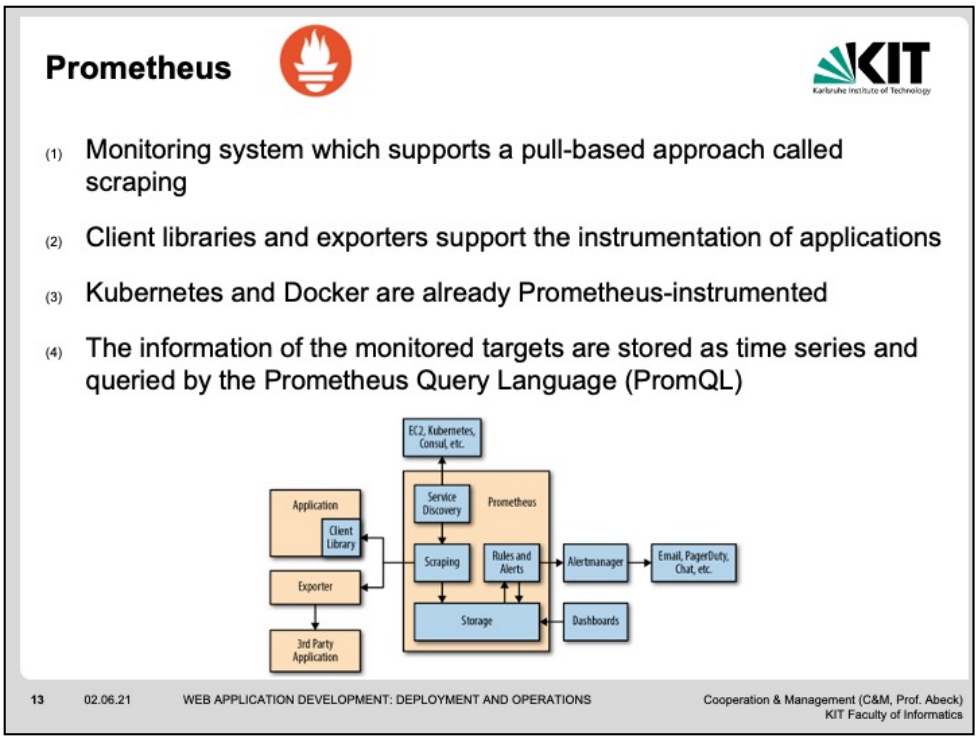
(Developers, Application, Code Repository, Continuous Delivery System, Docker Registry) This part of the figure concerns the influence of Kubernetes on the software development process. The only requirement of Kubernetes is that the application must be containerized. The outcome of the CI/CD pipeline is the containerized application in a Docker registry. This means, the pipeline is not responsible to distribute the application in the Kubernetes cluster.
In the C&M environment, GitLab is used as continuous delivery system. In order to integrate a Kubernetes cluster into GitLab, the address of the cluster (uniform resource locator of the API and the port) and and authorization information is needed.

(Deployment Strategies, Kubernetes Manifest, Configuration) In order to distribute the application in the Kubernetes cluster, the developer or operator must manually apply the Kubernetes manifest files.

(Operating System) Well established operating systems, such as Ubuntu, are not appropriate for the use with Kubernetes. The reason is, that these operating systems contain redundant elements (e.g. package manager) which make their use in the context of Kubernetes insecure. Instead, specific operating systems tailored to the needs of Kubernetes do exist (e.g. RancherOS, k3OS, Talos).

(Installer) The installation of Kubernetes requires the installation and integration of several components, such as the master, the key-value database etcd, and the start of a node using kubelet. Many different installation tools (e.g. Minikube, MicroK8s, k3s, Kubeadm, Kubespray) do exist which all have their specific advantages and disadvantages.

(Monitoring) Kubernetes allows to use any monitoring tool that fulfills the requirements of the involved roles (operator, developer, end user). In general, it must be distinguished between monitoring of the Kubernetes cluster and monitoring of the pods.
A relevant monitoring tool is called Prometheus that offers the possibility to record metrics in Kubernetes. By user-defined queries and the usage of an HTTP interface, pods and controllers are able to access the measurement data.



Prometheus is an open source, metrics-based monitoring system which is – as Kubernetes and Helm – supported by the Cloud Native Computing Foundation (CNCF). It is primarily written in Go and licensed under the Apache 2.0 license.

(1) Opposed to most monitoring systems which get the data from the applications, Prometheus asks the status of the monitored applications via defined interfaces. This concept is also called scraping.

(2) Prometheus client libraries in all the popular languages and runtimes, including Go, Java/JVM, C#/.Net, Python, Ruby, Node.js, Haskell, Erlang, and Rust. For third-party software that exposes metrics in a non-Prometheus format, there are hundreds of integrations available. These are called exporters, and include HAProxy, MySQL, PostgreSQL, Redis, JMX, SNMP, Consul, and Kafka.

(3) Prometheus is able to monitor metrics on Kubernetes in order to take scaling decisions.

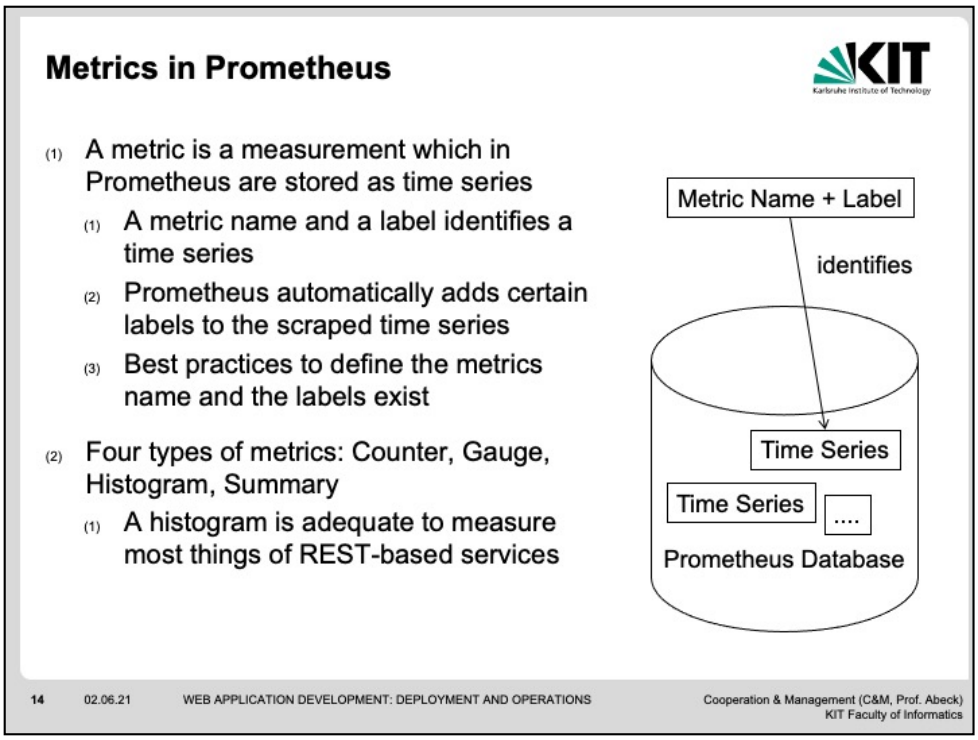
(4) The targets to be monitored by Prometheus are defined by endpoints for which the time interval of monitoring can be individually set.
The data model identifies each time series not just with a name, but also with an unordered set of key-value pairs called labels.
The PromQL query language allows aggregation across any of these labels, so the stored data can be analyzed not just per process but also per datacenter and per service or by any other labels that was individually defined.

(Dashboards) The results gained from PromQL queries can be graphed in dashboard systems such as Grafana.

(Alertsmanager) Alerts can be defined using the exact same PromQL query language that is used for graphing. Labels make maintaining alerts easier, as a single alert covering all possible label values can be created. In some other monitoring systems one would have to individually create an alert per machine/application.

(Service Discovery) Relatedly, service discovery can automatically determine what applications and machines should be scraped from sources such as Kubernetes, Consul, Amazon Elastic Compute Cloud (EC2), Azure, Google Compute Engine (GCE), and OpenStack.

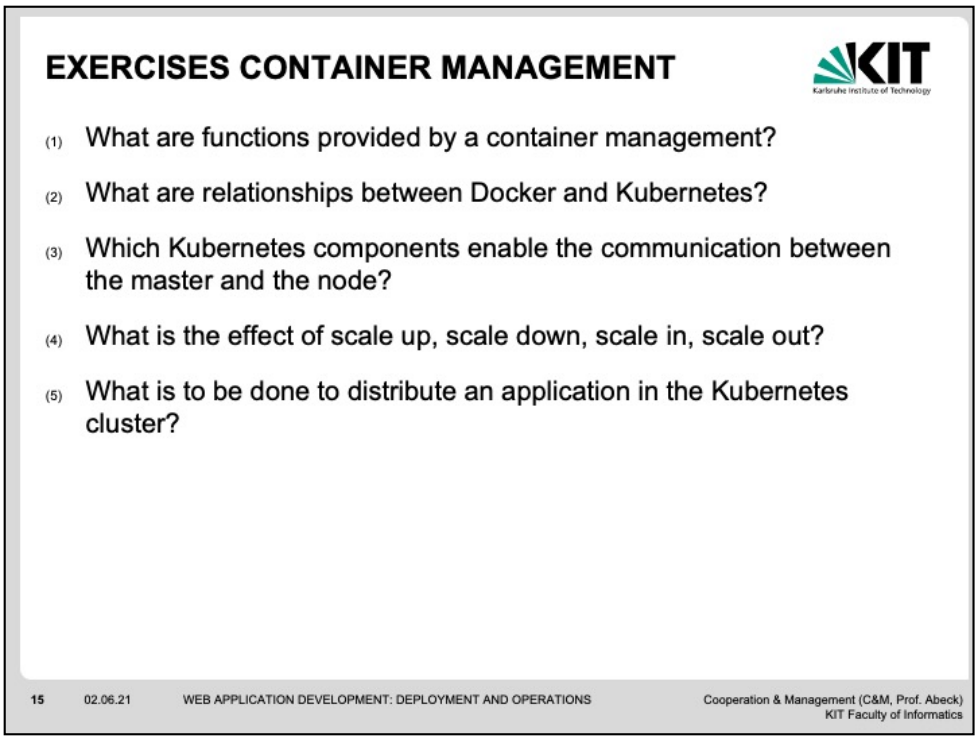
[ORe-Pro] O'Reilly: Prometheus: Up & Running by Brian Brazil. *https://www.oreilly.com/library/view/prometheus-up/9781492034131/ch01.html*
[Pro-Doc] Prometheus: Docs. *https://prometheus.io/docs/*



(1) A time series corresponds to a measurement with a timestamp [Win-Int].
(1.1) A label is a key-value pair which represents a dimension of a metric. The following format is used to address metrics:
<metric name>{<label name>=<label value>, ...}
(1.2) Example: scrape_duration_seconds{job="<job-name>", instance="<instance-id>"}: duration of the scrape
(1.3 These include to use consistent domain-based prefixes (e.g. http_... for all HTTP related metrics) and consistent units (e.g. seconds instead of milliseconds and bytes instead of megabytes).

(2) Prometheus caters for different types of measurements by having four different types of metrics:
(i) Counter: A cumulative metric that only ever increases (e.g. requests served, tasks completed, errors occurred).
(ii) Gauge: A metric that can arbitrarily go up or down (e.g. temperature, memory usage).
(iii) Histogram: Binned measurement of a continuous variable (e.g. latency, request duration, age).
(iv) Summary: Similar to a histogram, except the bins are converted into an aggregate (e.g. 99% percentile) immediately.
(2.1) This is because you not only get a series of bins, but also a count and a total sum of the data too. So you can use a histogram to calculate request rates, error rates and durations with only a single metric.

[Pro-Met] Prometheus: Metric and Label Naming. *https://prometheus.io/docs/practices/naming/*
[Win-Int] Winder: Introduction to Monitoring Microservices with Prometheus. *https://winderresearch.com/introduction-to-monitoring-microservices-with-prometheus/*



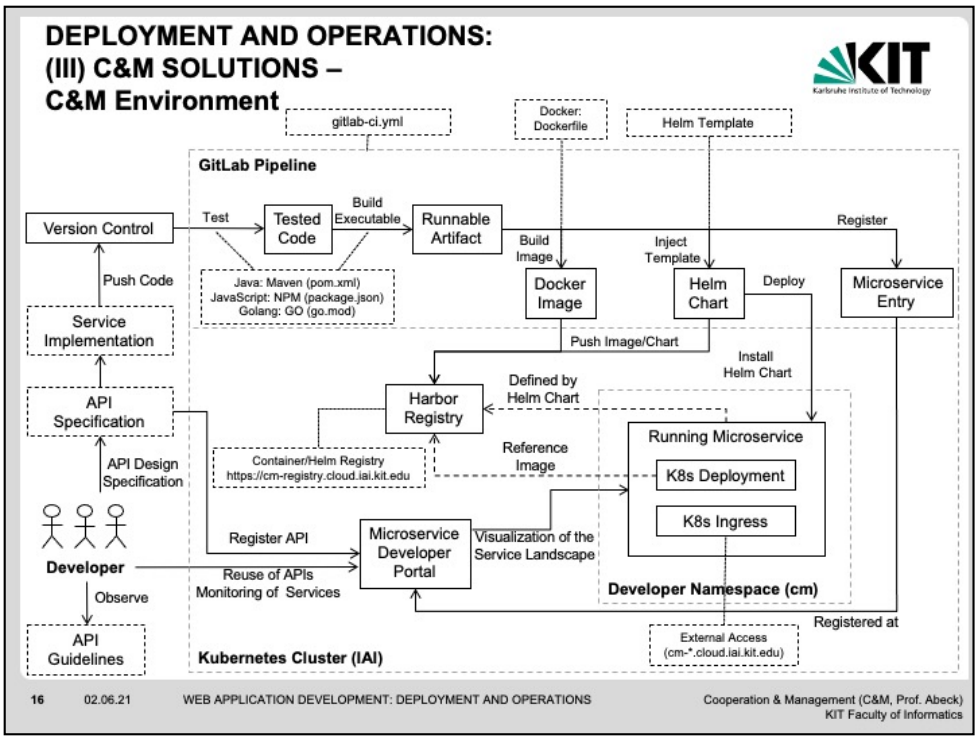
(1) ++Container Management++
- Automated deployment of a microservice
- Scaling of the microservices
- Operation of microservice containers across machines and clusters

(2) ++Container Management++ ++Kubernetes Architecture++
- Kubernetes can be based on Docker as container technology.
- One or more Docker containers are contained in a Kubernetes pod.

(3) ++Kubernetes Architecture++
- kube-apiserver: Master component which provides the whole Kubernetes API.
- kubelet: Node component which fulfills the task of executing a pod.

(4) ++Elasticity and Scaling++
scale up, scale down: Resources are added (up) to or withdrawn (down) from a node -> vertical scaling
scale in, scale out: An additional node is created (in) or an existing node is deleted (out) -> horizontal scaling

(5) ++Relevant Systems and Related Processes Around Kubernetes++
- Kubernetes manifests must be applied by the operator or the developer
- From the viewpoint of the CI/CD pipeline no further demands related to Kubernetes must be fulfilled.



This page illustrates the microservice deployment and operations environment as it is established at C&M.

(MicroserviceDeveloperPortal) The MicroserviceDeveloperPortal (MDP) is used during the development to guarantee a consistent overview of the existing services. The MDP visualizes the service landscape by offering API descriptions as well as the state (i.e., monitoring data) and metadata (e.g., dependencies) about the microservices.

(Version Control) The source code of the services is organized by the version control system Git and stored at GitLab.

(GitLab Pipeline) The CI/CD pipeline is a customized pipeline executed from the GitLab Runner whenever code is pushed to the repository. Hereby a step initiates the next one, when its run was successful. Each step runs in its own Docker container as defined by the "gitlab.ci.yml" which is stored in the version control system of each GitLab project.

(Kubernetes Cluster) For the building and deployment of all the application we use a Kubernetes Cluster, which is provided by the IAI.

(Tested Code) It is validated whether all information, like dependencies, are available. After a successful test of the source code, it is compiled. The project information and dependencies which are needed are given by the specific project description of the framework. For Java this is realized by the "pom.xml".

(Runnable Artifact) Furthermore the used framework compiles code to an executable. In the case of Java this a lightweight "jar" file. This "jar" file is then stored as a pipeline artifact to be used in the next step.

(Docker Image, Harbor Registry) Next, the runnable artifact is copied into a bare image, the result is the so called Docker image. This Docker image is pushed to the Harbor registry.

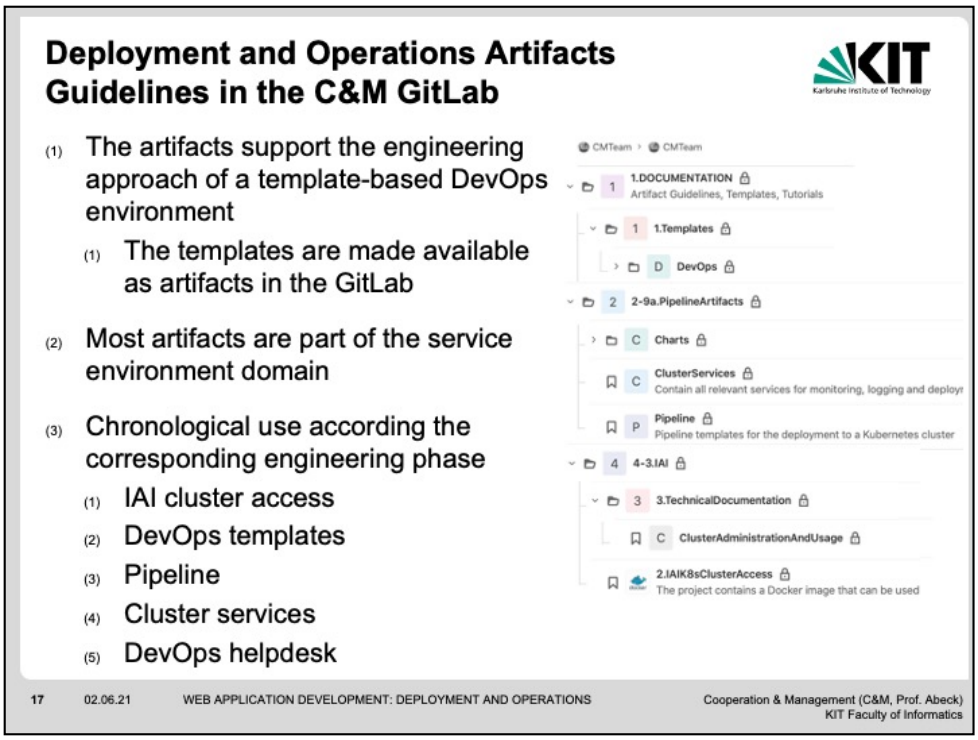
(Harbor Registry) Stores and manage all the Docker images and charts. All resources which are used within the Kubernetes cluster are pulled from the Harbor registry.

(Helm Chart) Each project provides a Helm chart which consist of a Helm template, metadata as well as a configuration file ("values.yaml"). The template represents all the Kubernetes manifest resources, which will be installed in the cluster. Before the Helm chart get packaged to the Harbor registry some project-specific values are injected into the template. Afterwards, the Helm chart is installed into the cluster.

(Microservice Entry) Contains information about the deployed microservice (e.g., name, deployment name, version). The entry is registered at the MDP.

(Running Microservice) The microservices are deployed by the GitLab Runner. For the deployment the Helm chart is used. Each service consists of a Kubernetes deployment which executes the build Docker image. To access the service within the intranet, a Kubernetes ingress can be activated which allows access to the application under defined DNS entries (which in the C&M context is "cm-*.cloud.iai.kit.edu").

CI/CD	Continuous Integration / Continuous Deployment (or Delivery)
IAI	Institut für Automation und angewandte Informatik
MDP	MicroserviceDeveloperPortal
OIDC	OpenID Connect



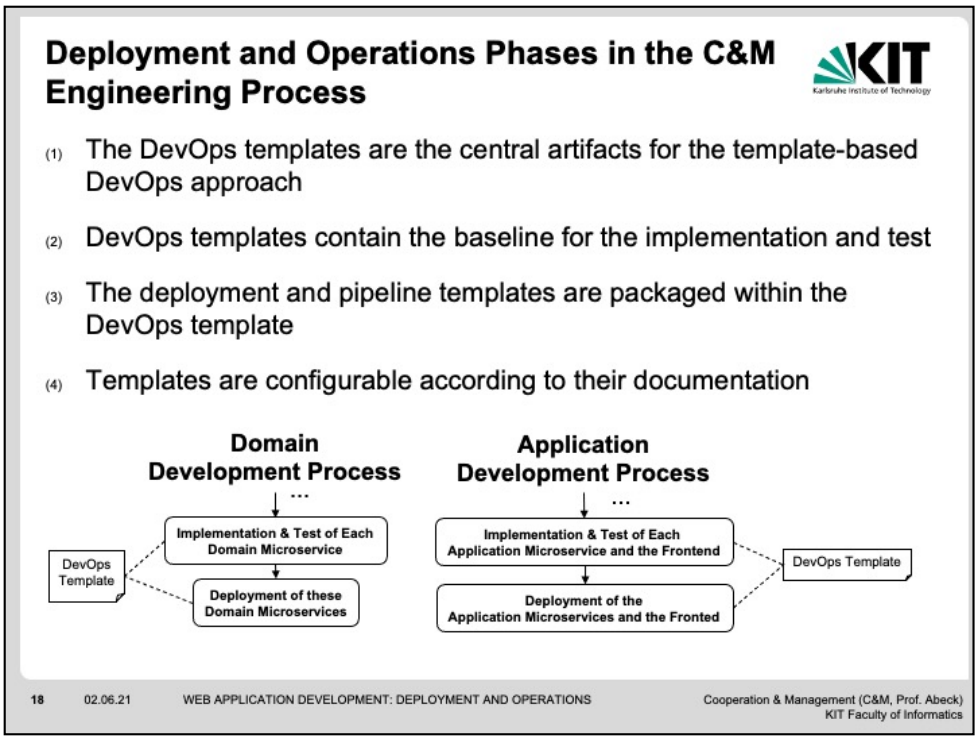
The deployment and operations phase is the fourth and last phase in the C&M microservice engineering process. For all artifacts used in this phase, guidelines are made available in the C&M GitLab [CM-G-Gui].

(1) The template-based engineering approach aims to reduce the maintenance effort for the operational team as well as lowers the entry barriers for developers to provide services within a container orchestration.
(1.1) This is a central aspect of engineering, which means that the developers and DevOps engineers use these artifacts according to a clear guideline to efficiently run through the deployment and operations process.

(2) The ServiceEnvironment Domain describes development artifacts, the CI/CD pipeline, running microservices and the monitoring of the system and the deployed microservices. At C&M the ServiceEnvironment Domain is realized through GitLab and Kubernetes.

(3) An overview of all artifacts and the process of the engineering process is described in the guidelines for the construction of artifacts [CM-G-Gui]
(3.1) The project 2.IAIK8sCluster Access contains a Docker image that can be used as a jump box to access the Kubernetes cluster at Institute for Automation and Applied Informatics (IAI). In addition, the procedure for requesting cluster access and establishing a connection to the cluster using the provided jump box image is described. Further description of IAI cluster specific features can be found in the Cluster Administration and Usage technical documentation.
(3.2) DevOps templates provide a baseline for all services to support for the GitLab-based CI/CD pipeline to rollout out an application to a Kubernetes cluster.
(3.3) The pipeline is part of the service environment and consists of pipeline templates which serve for the deployment to a Kubernetes cluster. Pipeline templates are shared between the projects which reduces the maintenance effort for all project pipelines.
(3.4) The cluster services include all relevant cluster-wide services, e.g., monitoring, logging, deployment services and messages brokers.
(3.5) Description how to get support for development on operational or deployment related issues, which can not be solved within the project team.

[CM-G-Gui] Cooperation & Management: Guidelines for the Construction of Artifacts, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation/3.artifactguidelines/-/blob/master/README.md



The deployment and operation phase is the last phase of the C&M microservice engineering process and is a follow-up of the implementation and test phase. Since the CI/CD pipeline can also support developers during the implementation and test phase, it is recommended to start the deployment and operation phase after the first initial implementation. The implementation can then be extended iteratively. Additionally this will ensure that other development teams can already use the latest state of the services and can test against them.

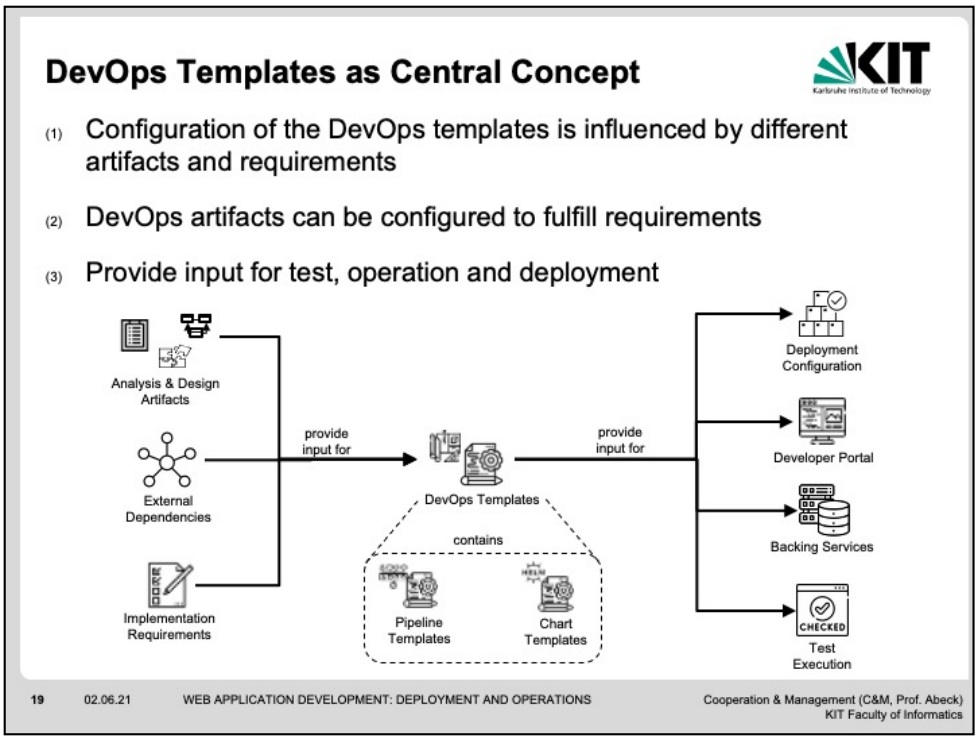
(1) The DevOps template incorporates the DevOps practices developed by C&M which aims to minimize the barriers for development to integrate continuous integration and delivery to their services.

(2) The DevOps templates follow a zero-configuration approach. This means that all DevOps templates are directly functional after the integration, if the recommendation for the implementation has been followed. However, deviations from the recommendation are also supported and can be made according to the documentation of the DevOps artifacts.

(3) Further components that represent a central role for the implementation of the DevOps approach are integrated in the DevOps template. These include the integration of the pipeline templates in the supplied pipeline configuration, as well as the base configuration for the chart template.

(4) Both the pipeline templates and the included chart template allow easy customization via variables. The variables for this can be found in the GitLab documentation of the pipeline templates [CM-G-PIP] and the chart template [CM-G-TEM].

[CM-G-Pip] Cooperation & Management: Pipeline Templates Cluster, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/pipeline
[CM-G-Tem] Cooperation & Management: Base Chart, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/charts/template-chart



The DevOps templates represent the central intersection where operational requirements are captured using specific DevOps artifacts. The DevOps artifacts are then picked up by the pipeline to perform the corresponding operations to implement the operational requirements.

(1) The DevOps template is influenced by the various artifacts from the analysis and design phase. An example are the dependencies to other domain microservices which an application service needs to communicate with. Other factors that influence the DevOps template can be external dependencies such as message brokers, databases or monitoring integrations. Additional implementation requirements such as the fulfillment of certain Service Level Objectives (SLO) also have an influence on the configuration of the DevOps templates.

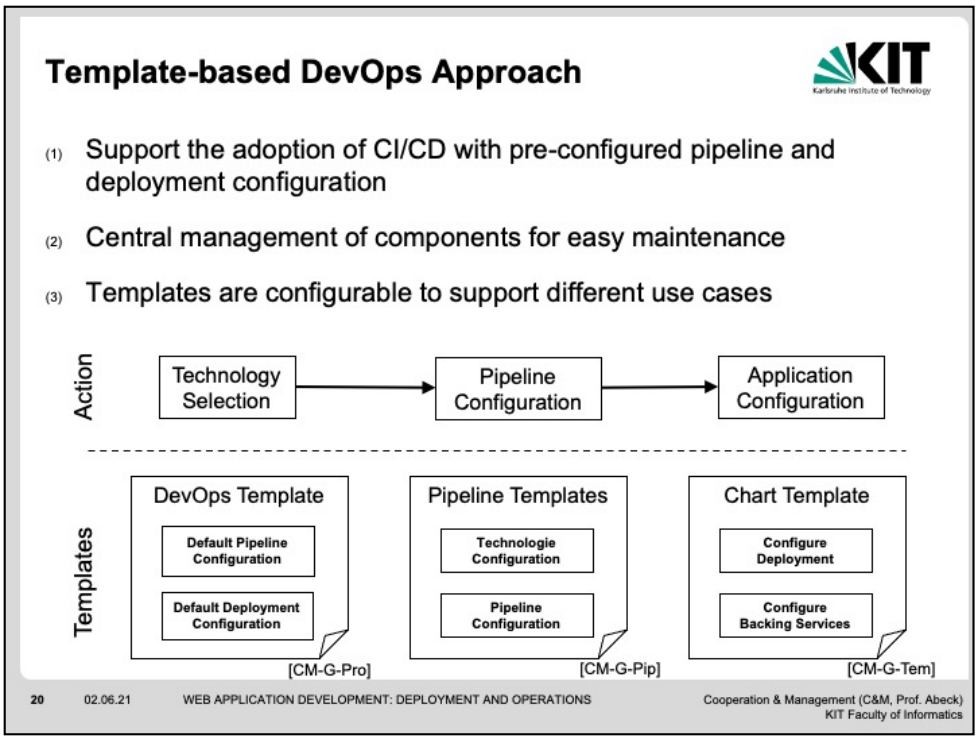
(2) In order to implement these requirements, the DevOps template and the two components of the DevOps template provide a number of DevOps related artifacts, which allow easy customization for developer. Each DevOps artifact within the project can be adjusted and extended without affecting other projects.

(3) The DevOps template can be used to configure a variety of operations and external dependencies. This includes the integration of backing services that directly support the application and are an essential part of the application. A classic example of a backing service is a database that manages the data of the microservice.

SLO Service Level Objective

[CM-G-Pip] Cooperation & Management: Pipeline Templates Cluster, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/pipeline

[CM-G-Tem] Cooperation & Management: Base Chart, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/charts/template-chart



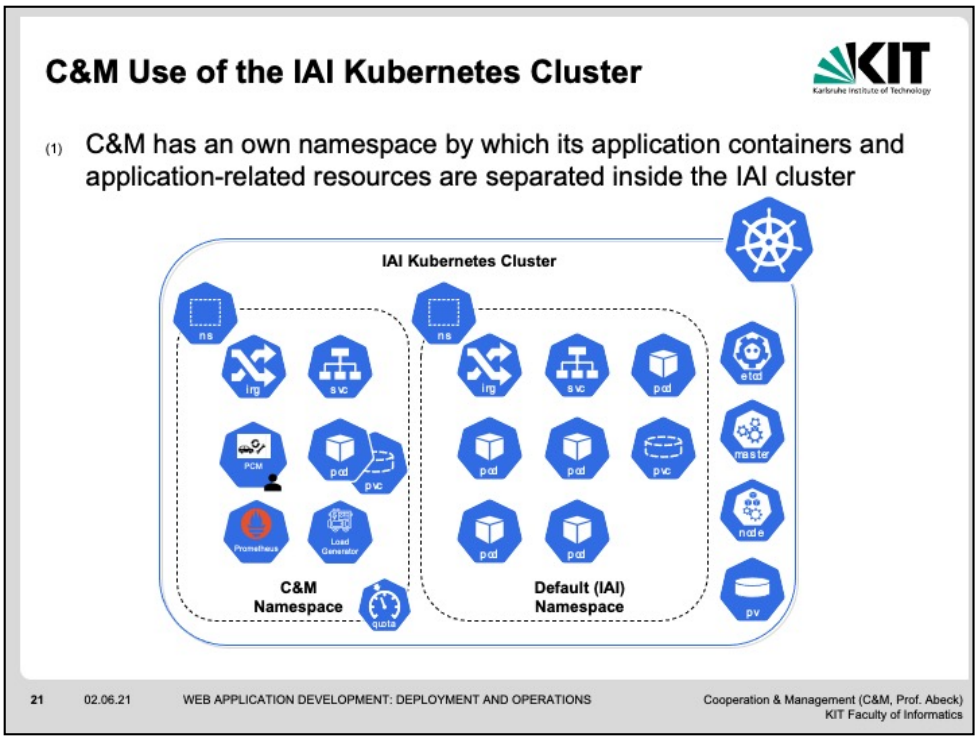
A good integration of CI/CD makes it possible to quickly detect errors and correct them. However, the integration of these automatisms is time-consuming and is often forgotten in the process of development, since it does not offer any functional added value for the created product. One goal that is increasingly pursued with DevOps is the improvement of daily work. One element of achieving this is the templates-based DevOps approach.

(DevOps Template) The template-based DevOps [CM-G-Dev] approach supports developers in adopting DevOps principles. By integrating the DevOps templates into the projects, a basic CI/CD pipeline is provided as well as a basic configuration for the deployment of a microservice.

(2) The majority of the pipeline configuration consists of external pipeline fragments which are maintained within the pipeline template project. Each pipeline fragment contains the necessary logic and steps to accomplish a task to implement the CI/CD pipeline. Adjustments to these can thus be made centrally and do not require adjustments in the individual projects. A similar concept is used for the chart template, which represents the configuration of the delivery artifacts to the cluster.

(3) To support all use cases with this approach, it is required to allow the configuration of these external components and adapt to the needs of the microservice. This configuration for the pipeline can be done via variables in the pipeline configuration file. Each of the variables are documented in the pipeline templates project. The configuration of the deployment artifacts is stored in a separate values file in the project. The documentation therefore can be found in the project [CM-G-Tem].

[CM-G-Dev] Cooperation & Management: DevOps Templates, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation/1.templates/2.devops
[CM-G-Pip] Cooperation & Management: Pipeline Templates Cluster, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/pipeline
[CM-G-Tem] Cooperation & Management: Base Chart, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/charts/template-chart



The Institute for Automation and Applied Informatics (IAI) runs a Kubernetes (K8s) cluster which consists of four nodes having 32 cores and 128 GR RAM. A well-defined part of the cluster is used in the IAI-C&M cooperation project. A central research question deals with the (continuous) deployment and operation of microservice applications, such as the PCM, in the K8s cluster [Br20] [CM-W-WEB].

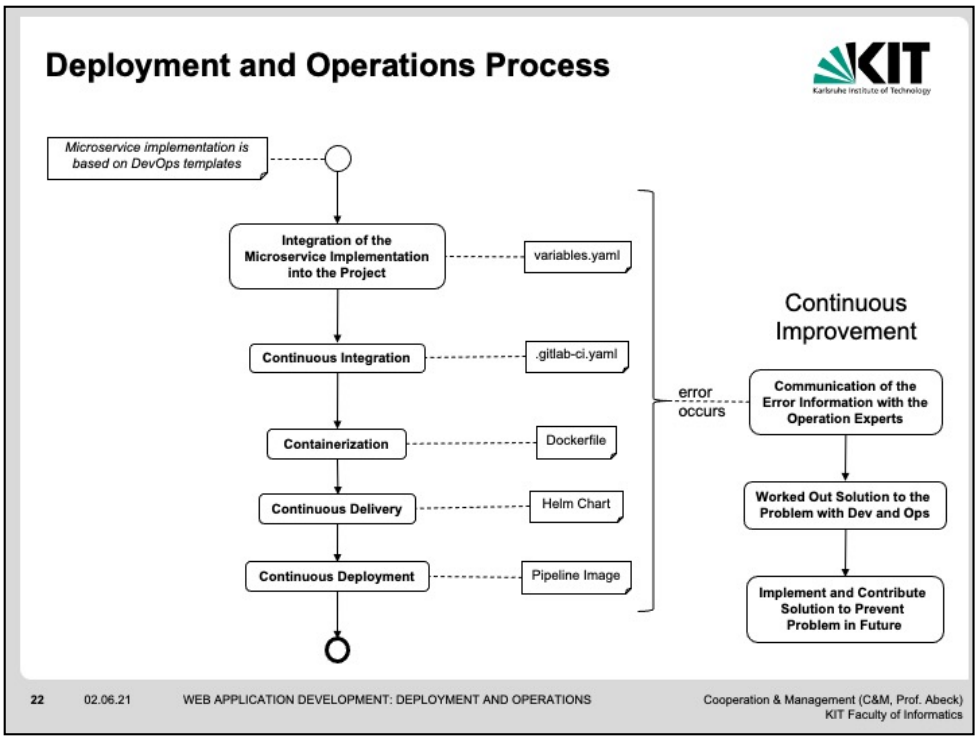
(1) In Kubernetes, a namespace is used to support multiple virtual clusters within one physical cluster [Kub-Sha].

(Kubernetes Cluster, master, node, etcd, pv) In the cluster, the master takes the role of the server and the nodes are the clients. Master and node are as well as etcd ("/etc distributed" storing all configuration data) and pv (persistent volume) are infrastructure components which do not belong to any namespace.

(C&M Namespace, PCM, Load Generator, ing, svc) This namespace includes all containers (pods) containing the applications and other application-related resources, like ingress (ing) and services (svc). This especially includes the PCM application and the Load Generator. Only these resources can be modified by C&M.
(quota) A reasonable resource quota is defined for the C&M namespace. The quota limits the total amount of resources that this namespace and its pods, services, controllers, etc. can consume.

IAI	Institute for Automation and Applied Informatics
ing	ingress
pv	persistent volume
svc	service

[Br20] Kevin Braun: Development and Evaluation of a Machine Learning Approach for Scaling Container-virtualized Infrastructures, Master Thesis, Karlsruhe Institute of Technology (KIT), C&M (Prof. Abeck), 2020. *https://team.kit.edu/sites/cm-tm/Archiv/2020/MT_Braun*
[CM-W-WEB] Cooperation & Management: WEB APPLICATION DEVELOPMENT. WASA Course Unit. *https://team.kit.edu/sites/cm-tm/Mitglieder/2-2.WASA_Lecture*
[Kub-Sha] Kubernetes: Share a Cluster with Namespaces. *https://kuber-netes.io/docs/tasks/administer-cluster/namespaces/*



The deployment of a microservice is carried out in several steps which are illustrated on this page. In order to continuously improve the process, it is required to communicate problems during deployment and operation in order to work out a common solution to the problem.

(Microservice implementation ...) The deployment process can start when the first implementation of the microservice has been made. With the integration of the DevOps template during the implementation a commitment to a technology has been made. By default, the DevOps template comes with a pre-configured pipeline and a demo application which can be used as the first implementation.

(Integration of the Microservice Implementation …) The DevOps template provide a predefined structure for the microservice implementation. The structure follow the best practices from C&M for the development of microservices based application. If the predefined structure is not suitable for the service, the configuration for the build process can be adapted via the variables.yaml file.

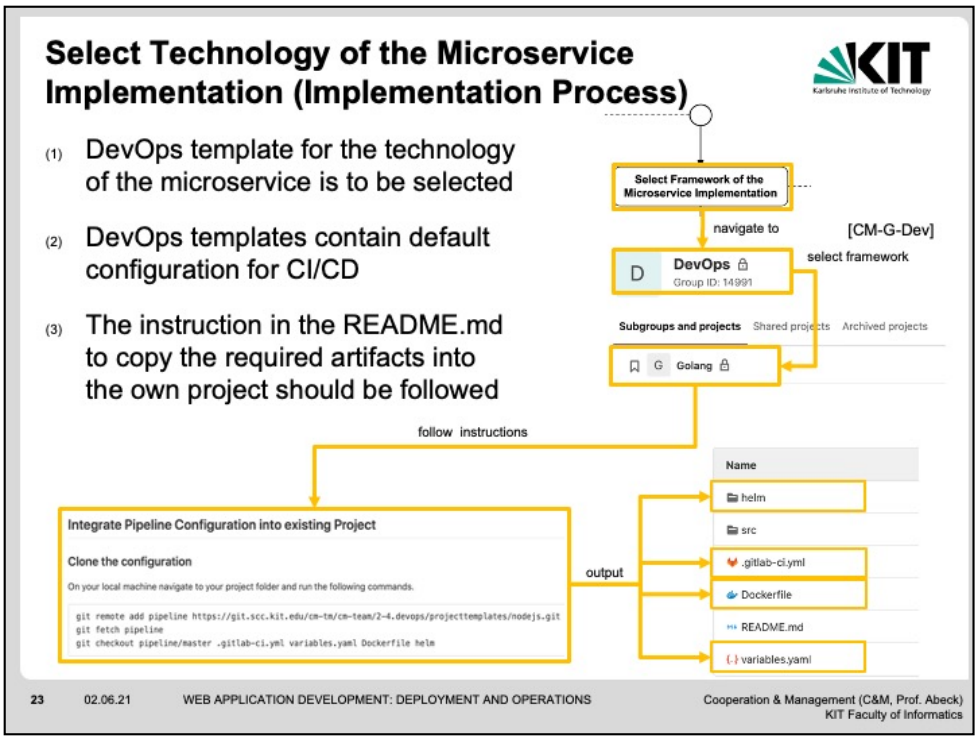
(Continuous Integration) The pipeline configuration which was copied from the DevOps template, performs automated tests with every change. Individual pipeline fragments in the .gitlab-ci.yaml are configurable and can be customized as needed.

(Containerization) The application logic is containerized for delivery. A Dockerfile describes how the application is containerized and which dependencies it has.

(Continuous Delivery) In addition to containerizing the application itself, a framework for delivering the application into a container orchestration must be provided. Helm allows the packaging of network, storage, and backing services with the application and managing the deployment to a Kubernetes cluster.

(Continuous Deployment) Every build artifact of the pipeline is automatically delivered to a Kubernetes cluster and therefore visible to all. To support this, multiple dependencies and configurations are required, which have been bundled within a single Docker image. Developers with appropriate permissions can connect to the cluster using the image.

(Continuous Improvement) DevOps tries to avoid barriers but does not prevent specialization and separation of individual teams. To avoid repetitive work or solutions that cannot be integrated into other projects, solutions for should be developed across teams. This ensures that all perspectives have been taken into account. The solution should be made available to everyone so that other teams can benefit from it.



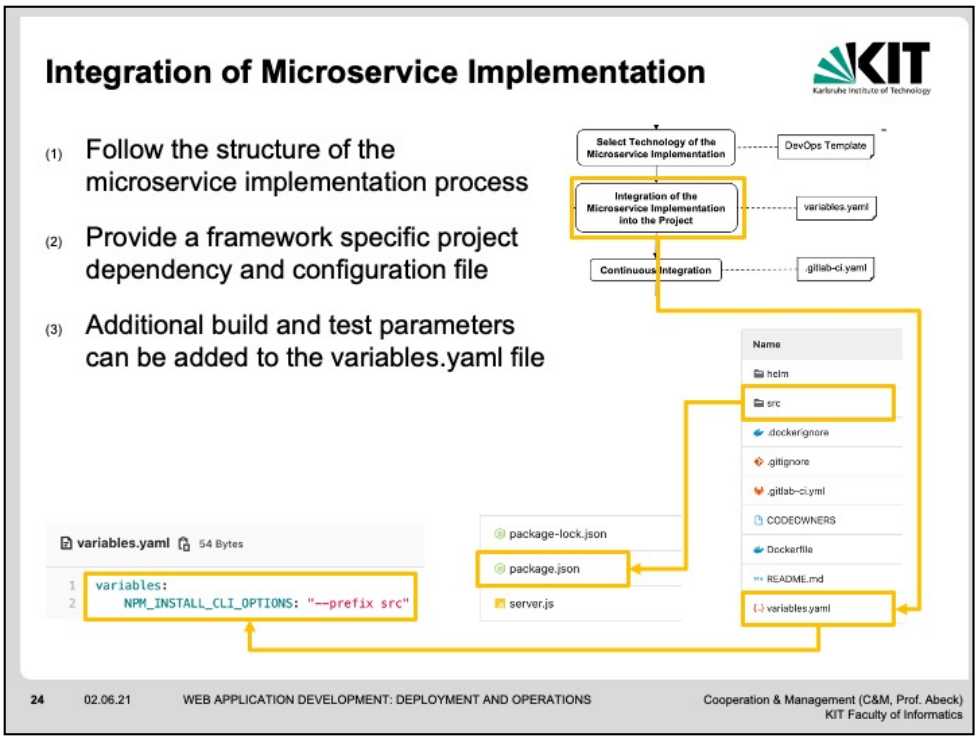
To minimize the entry barrier C&M offers a variety of prefabricated DevOps templates which are available in GitLab [CM-G-Dev].

(1) All supported technologies are listed in the DevOps template group [CM-G-Dev]. Missing technologies which are required for the project should be communicated with the DevOps experts or contributed upstream.

(2) Each of the DevOps templates contains several predefined files which are configured to support the technology. This includes always a Dockerfile, the .gitlab-ci.yml as well as a variables.yaml and a folder for the Helm chart. Different projects can have additional files.

(3) The artifacts can be cloned into the own microservice project via the provided Git command. The instructions therefore are documented in the README.md file. The cloned files have no connection to the used DevOps template and can therefore be modified in the project without any conflict.

[CM-G-Dev] Cooperation & Management: DevOps Templates, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/1-1.cmdocumentation/1.templates/2.devops

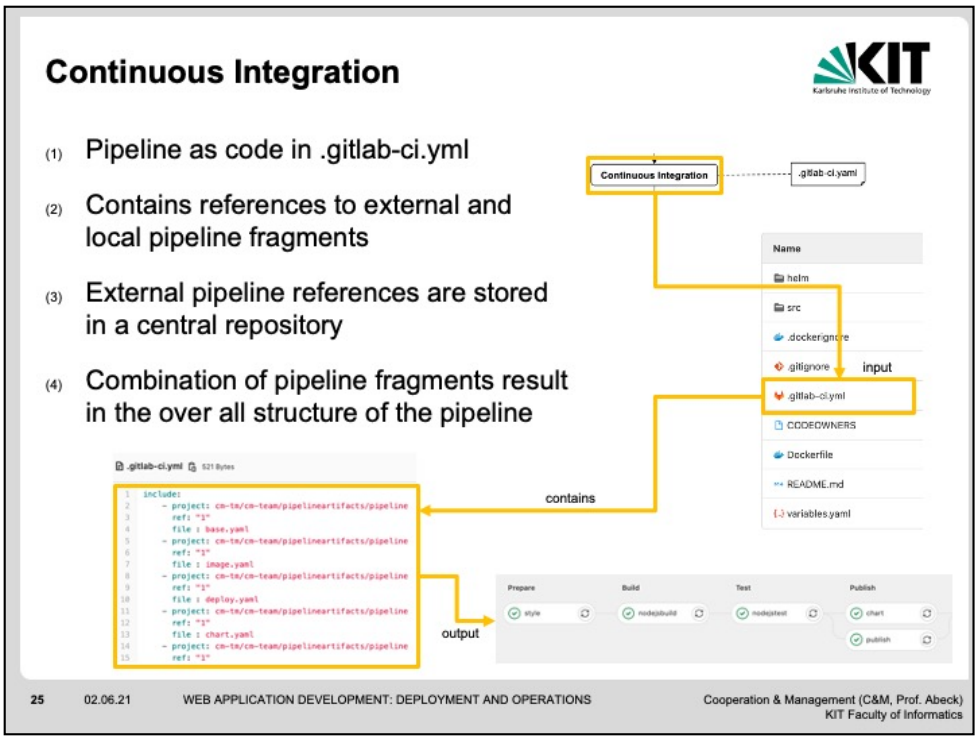


Since the development of microservice is a continuous process and consist of many iterations the deployment and pipeline need to be adjusted to support the requirements of the microservice.

(1) The artifacts of the DevOps templates are configured to use the provided structure of the microservice implementation process therefore it is recommended to follow these process.

(2) Most of the technologies provide a project specific configuration file which is required to support the build and test process in the pipeline. A dummy project specific configuration file in the default location is provided in each DevOps template.

(3) Additional build and test parameters can be configured in a separate variables.yaml file. The variables are then injected in the build command within the pipeline fragment of the used technology. Adjustment to the location of the project specific configuration needs to be configured here as well.

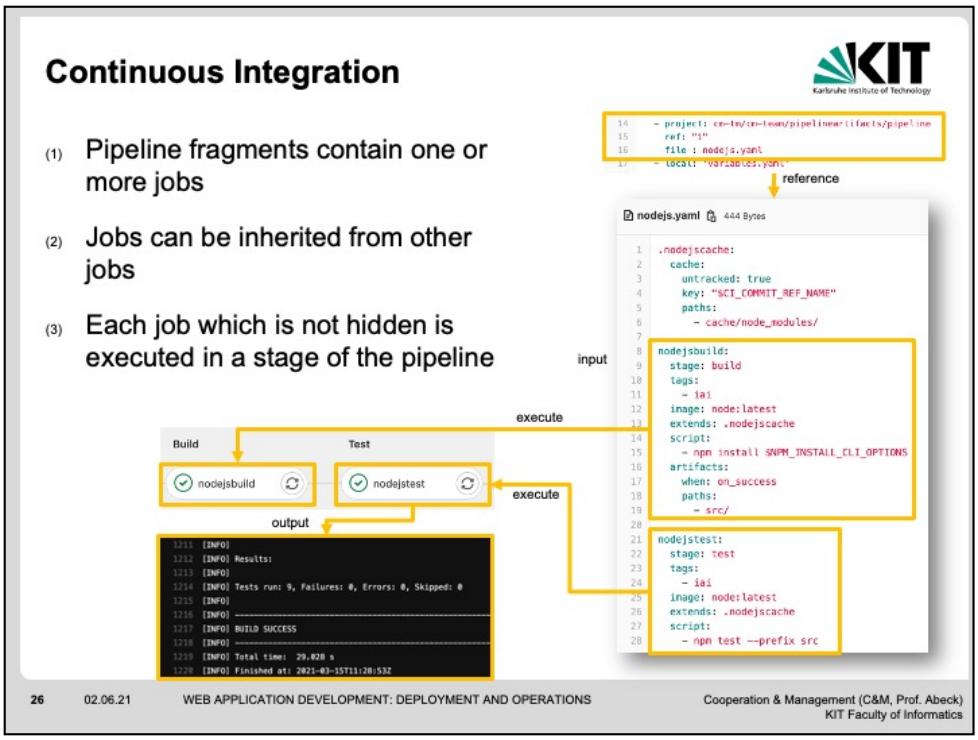


(1) The pipeline in GitLab is defined as a YAML file and follows the configuration as code approach. The structure and content of the file define the resulting Pipeline. The pipeline itself consist of stages which are ordered as a graph.

(2) The provided .gitlab-ci.yml file from the DevOps templates contains only external and local references to pipeline fragments. External pipeline fragments are defined by the path to the origin project of the fragment, a reference to a branch or tag of the origin and the name of the file in the origin containing the pipeline logic.

(3) The external pipeline references are stored in the central Pipeline template repository and maintained by the DevOps experts. Modified fragments can be tested by changing the reference to the created branch name in the Pipeline Templates project. Contributions are always welcome and can be added as merge request to the master branch.

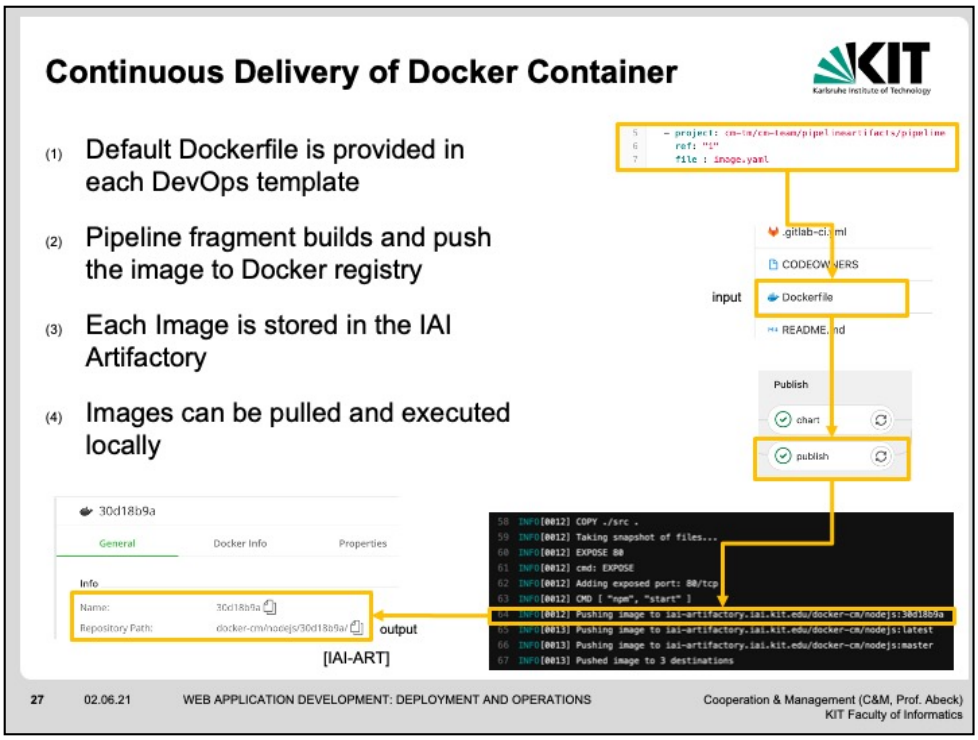
(4) The overall pipeline and the structure is a result of all pipeline fragments which are merged to a single file and then executed. Therefore, name conflicts of stages, jobs and variables should be avoided.



(1) In this example, the fragment for node.js consist of two regular jobs for building and testing the source code. The order of the execution is defined by the stage and the optional dependencies to other jobs. The stages run sequentially according to the definition in the external pipeline fragment of the base.yaml file.

(2) Jobs can inherit from each other. In the example above the two regular jobs extend the hidden job .nodejscache. To define a job as hidden the name of the job should start with a dot. Hidden and non hidden jobs can be inherited with the extends parameter. GitLab support the inheritance of regular and hidden jobs over multiple depth similar to the concept of inheritance in Java and abstract classes.

(3) Hidden jobs are not executed in the pipeline and serve only to share configuration for multiple jobs. Each regular job result in a separate executed step in the pipeline and provide a specific output in the console. Changes within a job are not stored and therefore does not affect following jobs. To share artifacts between different jobs an external storage or the build in artifact storage in GitLab should be used. Due to limitation of the storage size of the artifact storage it is recommended to not store packages or binaries.



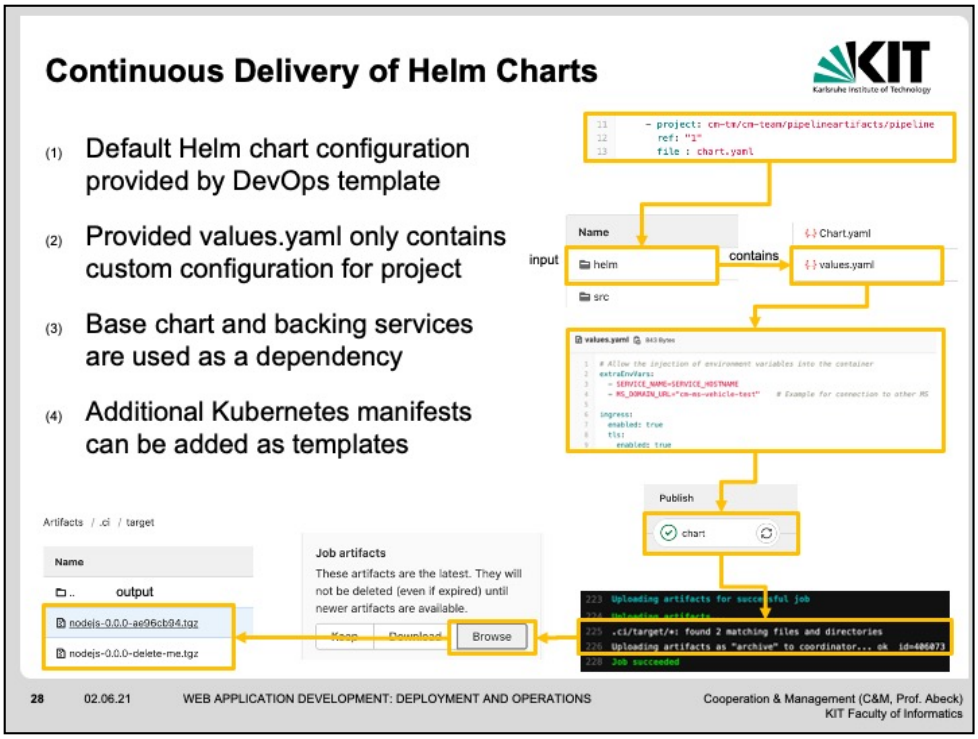
(1) A default Dockerfile for the used technology is provided with each DevOps Template. The Dockerfile often contain a multi-stage build to separate the build and runtime dependencies from each other. This result in smaller image size, faster startup time and a minimization of the attack vectors.

(2) The build image is afterwards tagged and pushed to a container registry. According to the trigger of the pipeline an image will be tagged differently. By default, each image is tagged with the short commit sha of the git commit and the branch name. Additionally, images build from released version in git will be tagged with the same tag as the git commit.

(3) Published images can be viewed and pulled from the IAI Artifactory. Each student with cluster access can therefore go to the Artifactory [IAI-ART] and login with the KIT credentials.

(4) To pull build images from the Artifactory a login to the Docker registry is required. Afterwards the images can be downloaded, inspected and executed locally.

[IAI-ART] Institut für Automation und angewandte Informatik: Artifactory Repository. https://iai-artifactory.iai.kit.edu/artifactory/webapp



For the delivery of the application to the Kuberentes Cluster we use Helm Charts. Helm itself is the name of the software, charts are the packages created by the software, which are later reinstalled on the cluster using Helm. In addition to the creation and installation of charts, Helm allows version management of charts and therefore allow rollback in the cluster to previews versions. Each build chart version is stored as a pipeline artifact within the publish step of the chart and can be inspected from there. Alternative the chart can also be downloaded from Harbor or Chartmuseum.

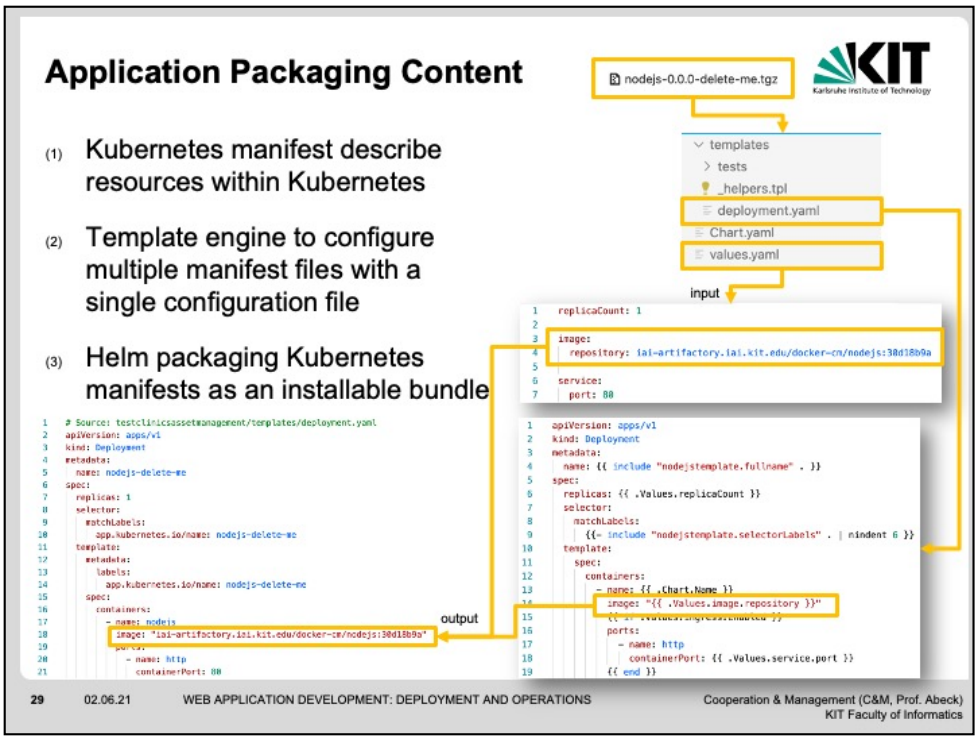
(1) The Helm charts provided by the DevOps template are pre-configured and allow the application to be delivered directly to the Kubernetes cluster. The initial configuration for this is minimal, but can be easily customized via values.yaml. All customizable parameters within values.yaml in DevOps template is documented within the file.

(2) The values.yaml included in the DevOps template contains only the most important section of the configuration. The complete configuration can be viewed in the Base Chart project [CM-G-Tem]. All configurations within the base chart can also be overwritten within the own values.yaml file.

(3) The base chart and other backing services such as databases are configured in the chart as external dependencies. The dependencies are defined within the requirements.yaml. The configuration of dependencies can also be overwritten. To do this, simply add the name of the external chart as the root path to the configuration parameters.

(4) In addition to the inclusion of external charts, it is also possible to add your own Kubernetes manifest files. To do this, these must simply be placed in the templates orders within the helm orders. These are then packaged and later installed in the Kubernetes cluster.

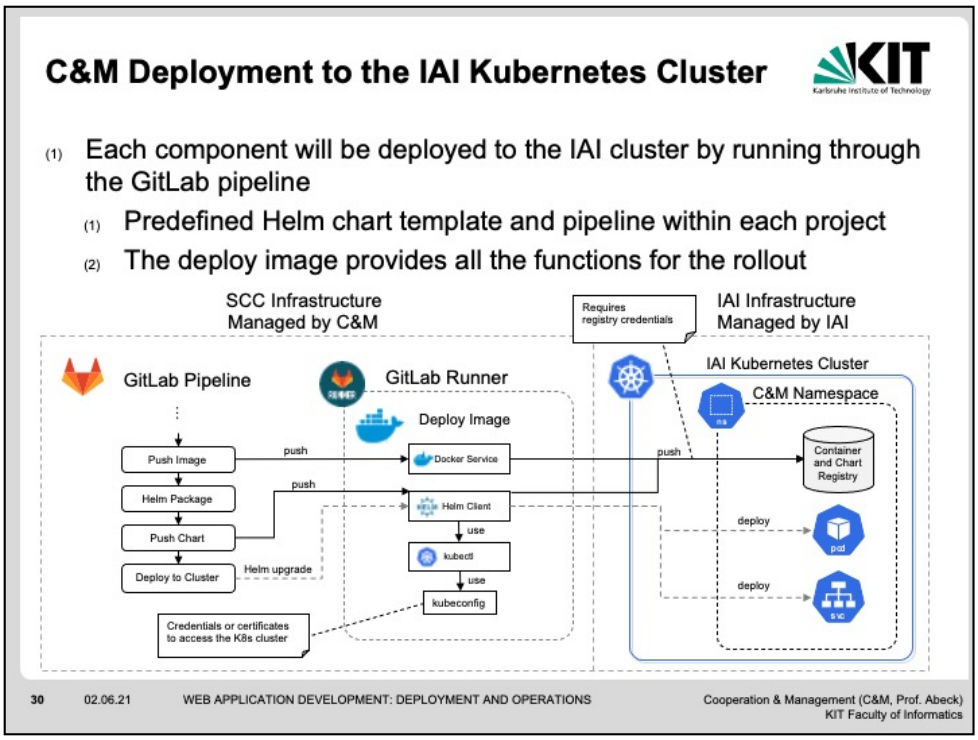
[CM-G-Tem] Cooperation & Management: Base Chart, C&M GitLab. https://git.scc.kit.edu/cm-tm/cm-team/pipelineartifacts/charts/template-chart



(1) Helm packages all files on the level of the Chart.yaml file to a .tgz file. For this the metadata from the Chart.yaml file are taken as basis for the naming as well as the versioning of the chart. The version is additionally overwritten in C&M to ensure a unique assignment to the pipeline. Accordingly, each chart generated by the pipeline is published with the prefix "0.0.0-" as well as the commit sha and the branch name.

(2) Kubernetes manifests can be installed or removed from a cluster. Each manifest is also versioned within Kubernetes itself. Typically, Kubernetes manifests are used as declarative descriptions that define the desired state of the cluster. Individual manifests can also contain multiple manifests which can be viewed as separate resources in the cluster. An example of this would be a deployment that also creates one or more ReplicaSets resources.

(3) The primary content of the packaged chart consists of the templates for the Kubernetes manifests files, a chart.yaml file with metadata of the chart, and the values.yaml. Helm uses the go template language and additional libraries to create the deployable Kubernetes manifest files from the templates and values.yaml files. The templates are not replaced with the parameters from the values.yaml until the time of delivery.



(1) All microservices should be deployed to the IAI Kubernetes cluster and communicate with each other within this environment. The orchestration of the created microservices and other additional services like database and message broker will run within the cluster.
(1.1) Project templates are provided for all programming languages used in C&M development projects. This includes a basic GitLab pipeline with the necessary steps for the deployment to the IAI Kubernetes cluster.
(1.2) The functionality for the deployment to the IAI Kubernetes cluster being part of the deploy image consists of: (i) the Docker service, (ii) the Helm client, (iii) kubectl and (iv) kubeconfig.

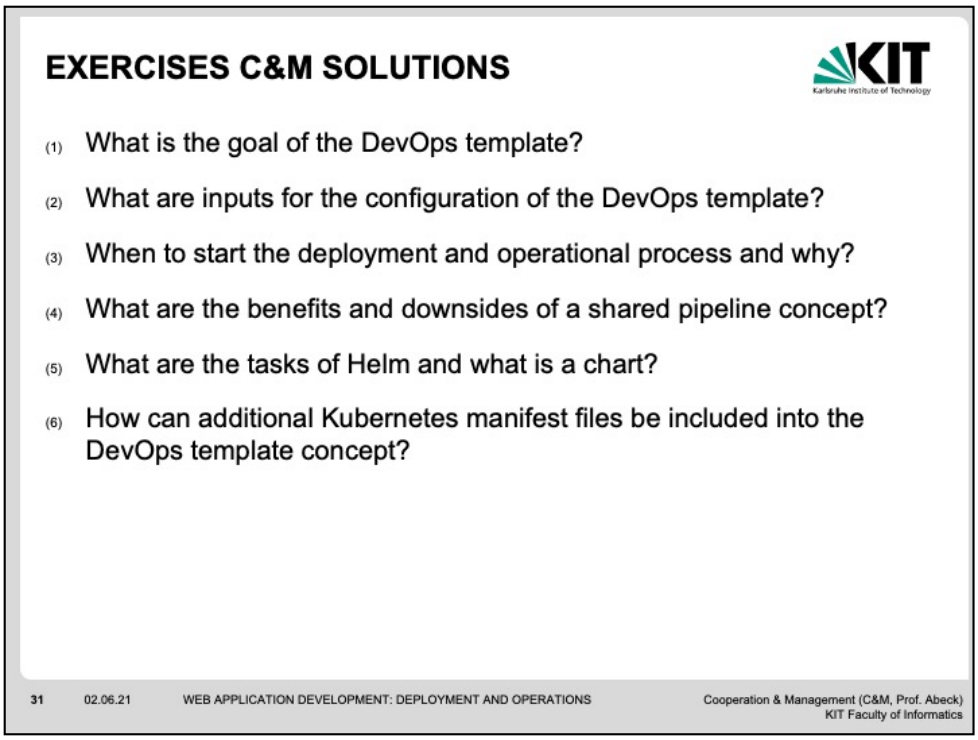
(GitLab Pipeline) A Gitlab pipeline defines the order, parameters and steps of a pipeline run. Each step defines own processes and can use different images or execution units.

(GitLab Runner) Executes all the steps of the build pipeline. For the execution of each of the steps, a Docker image can be provided for the GitLab Runner which will then be used. For the deployment to the IAI cluster, a custom image will be used which provides the necessary tools. The credentials for private image registries can be set within the GitLab Runner image.

(Helm Client) (kubectl) Lists, checks and deploys the Helm charts in a Kubernetes cluster. It uses the default connection of kubectl to the Kubernetes cluster.

(kubectl) (kubeconfig) For the connection to the cluster, kubectl requires a valid certificate for the cluster. This certificate is stored in a kubeconfig file. The certificate is provided from the SCC OpenID Connect (OICD) provider which can be accessed with a kubectl plugin from the IAI.

OICD OpenID Connect



(1) ++Deployment & Operation Phases in the C&M Engineering Process++
The DevOps template is intended to reduce the barriers to entry the DevOps practices. Developers can thus fall back on existing pipelines and concepts for delivery. In addition, the use of DevOps templates simplifies the integration of tests and the integration of external dependencies.

(2)) ++DevOps Templates as Central Concept++
Inputs can be the analysis and design artifacts, as well as dependencies to other services or the inclusion of implementation requirements.

(3) ++Deployment and Operations Process++
It is recommended to implement the deployment and operation phase after the first initial implementation. Further implementations and test extensions can then be carried out iteratively. In addition to supporting development, this will ensure that other development teams are already using the services and can test the connection to them.

(4) ++Continuous Integration++
The central management of the pipeline allows to roll out fixes or features for all projects. This means that not all pipelines have to be maintained separately, which greatly reduces the maintenance effort. Due to the centralized concept, changes can lead to problems in other projects, which remain undiscovered. Accordingly, a good versioning and testing concept is necessary to avoid this.

(5) ++Continuous Delivery of Helm Charts++
Helm itself is the name of the software, charts are the packages created by the software, which are later reinstalled on the cluster using Helm. In addition to the creation and installation of charts, Helm allows version management of charts and therefore allow rollback in the cluster to previews versions

(6) ++Continuous Delivery of Helm Charts++
The Kubernetes manifest must simply be placed in the templates folder within the helm folder. These are then packaged and later installed in the Kubernetes cluster.