FACULTY OF ENGINEERING
AND ARCHITECTURE

SYSTEM DESIGN

Pieter Simoens – Academic Year 2018-2019

FACULTY OF ENGINEERING
AND ARCHITECTURE

PART 0:
COURSE INTRODUCTION

Pieter Simoens – Academic Year 2018-2019

Hello World!?

FROM BUILDING PROGRAMS TO DESIGNING SYSTEMS

Program

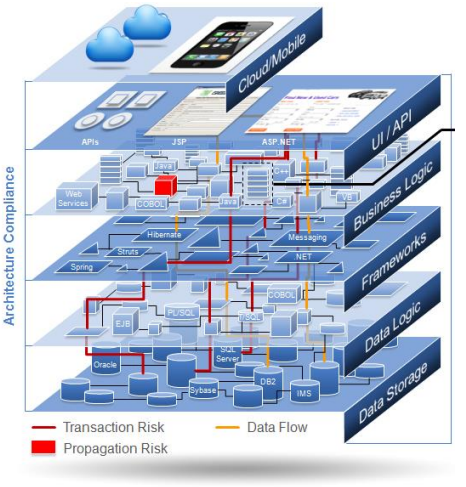
A set of instructions telling a computer what to do

System

- Set of distributed components (programs, databases)
- Designed to permit the user to perform a group of *functions, tasks, or activities*
- User expects performance, even under load and failure

System Design requires you to think about:

- Infrastructure to deploy your components
- Interaction between these components

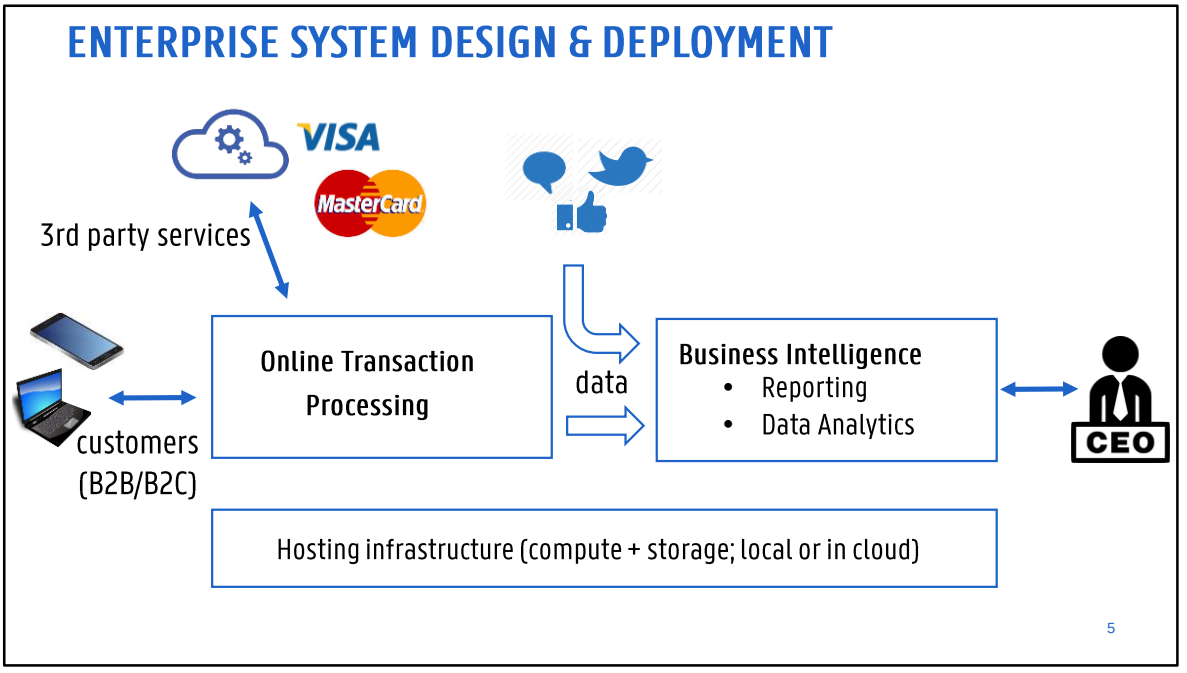


The world is becoming increasingly interconnected, and interconnected systems deliver the majority of our crucial applications and services. We are no longer building *programs,* which are basically sets of instructions transforming input to output. Instead, we are building *systems*, consisting of a multitude of components, who in and of themselves also can be systems.

System components are independent interacting processes. By intelligent design, these components appear as a single application to the end-user allowing him or her to perform specific functions, tasks or activities. The spatial distribution of the components may vary widely: they may be on datacenters in separate continents, in the same building or even on the same server in different virtual machines or allocated to different CPU cores.

At the same time users' expectations have become harder and harder to meet as everyday human life is increasingly dependent on the availability of systems to function smoothly. The system has to be performant: it does not matter if something provides the correct response if the response is not available when it is needed. This responsiveness must be maintained under failure and under load.

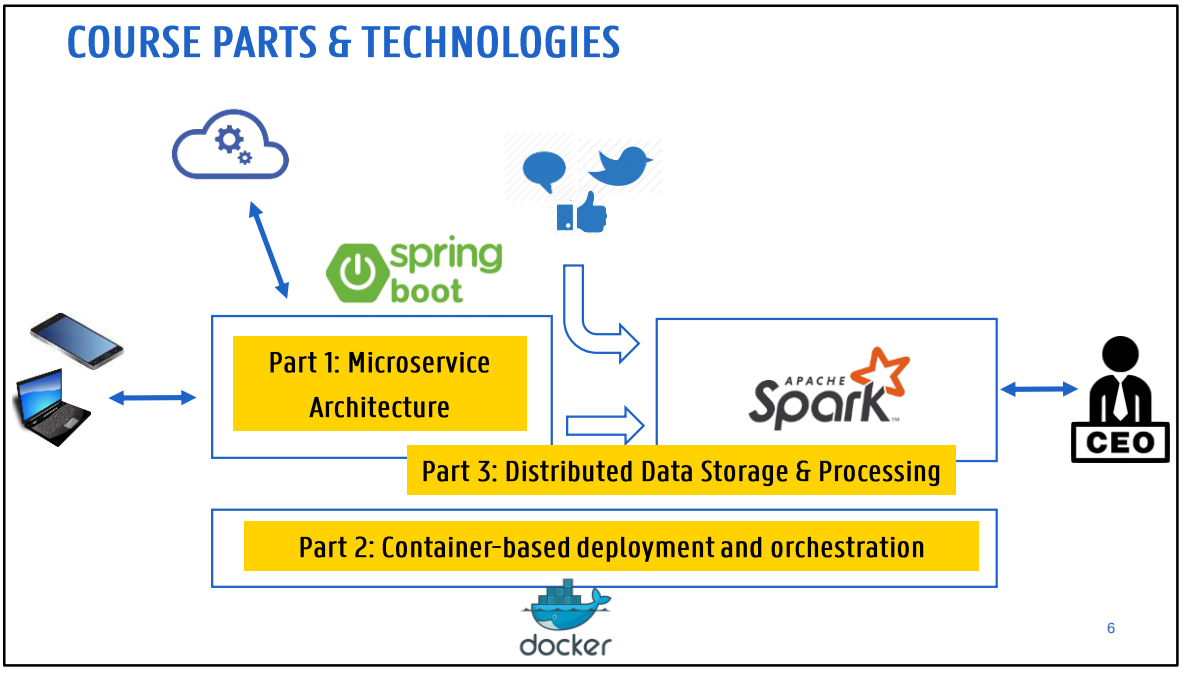
When designing a system, you thus need to think about the infrastructure on which you will deploy your components (e.g. the cloud), and how these components will interact (e.g. protocol, messaging…). Unsolid system design will harm the performance of your system, regardless of how well you coded your individual components.



This slide shows a typical architecture of a modern enterprise architecture. It consists of two major functional blocks, that are hosted on some infrastructure.

The first block is the Online Transaction Processing block. This block is responsible to handle requests coming from customers, which can be end-users or other companies. Each request is basically a business transaction: a purchase, a check-in of a tourist in a hotel, a factory machine communicating its status, etc.

The second block is the Business Intelligence. In this block, we analyze the data that is produced from the business transactions, possibly in combination with data coming from external sources. The output are reports and insights for the operational and financial management of the enterprise. For instance, you might generate historical trends on the revenue of solar panel installations, or produce a report that correlates social media comments about your company with the number of online purchases.


COURSE PARTS & TECHNOLOGIES
spring boot
Part 1: Microservice Architecture
APACHE Spark
CEO
Part 3: Distributed Data Storage & Processing
Part 2: Container-based deployment and orchestration
docker
6

COURSE CONTENT

- Microservices
  - Microservice architectural style
  - Strategies for system decomposition
  - Interaction styles and technologies
  - rich domain models with Domain-Driven-Design
- Container deployment and orchestration
  - elasticity and cloud provisioning
  - Linux Containers and Docker
  - Docker Swarm and Kubernetes
- Distributed Data Storage & Processing
  - Big Data
  - data lakes and distributed file systems
  - NoSQL data models
  - distributed data storage: replication, sharding, partitioning
  - Apache Spark

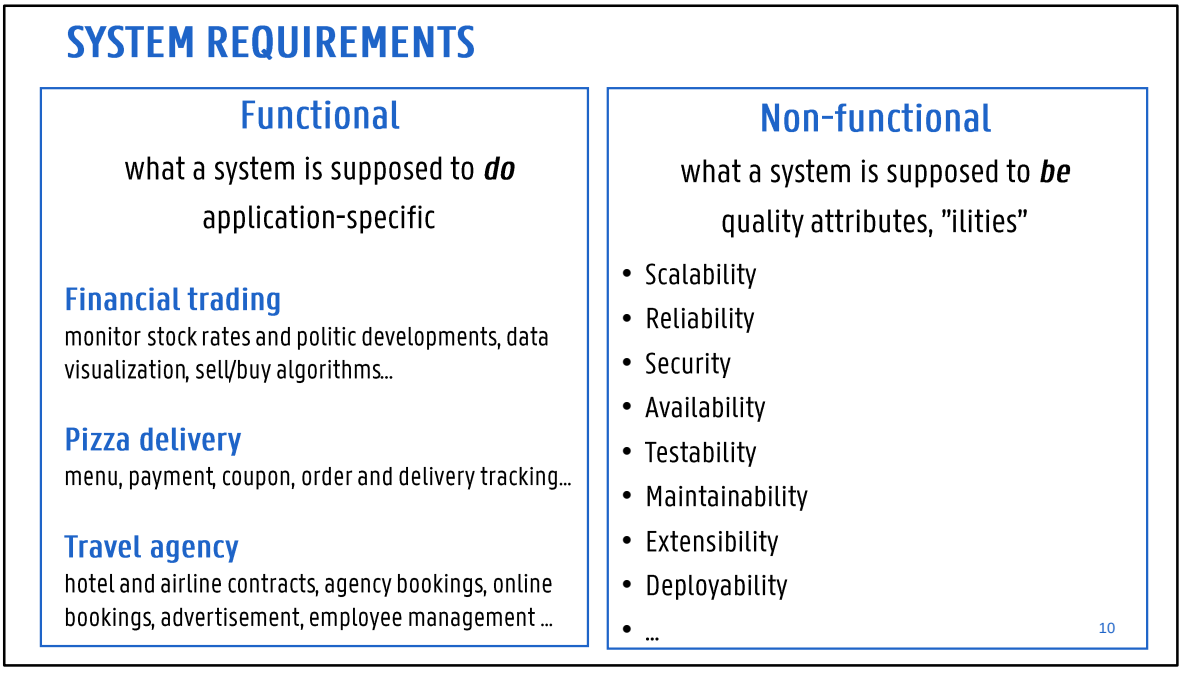
ORGANIZATION

- 6 weeks: theory classes of 2 x 1 hour [odd weeks]
- 6 weeks: lab sessions of 2 x 1.5 hour [even weeks]
  - Spring Boot + architectural design
  - guest lecture on Apache Spark and Apache Spark MLLib (by Infofarm)
- Check OASIS calendar: locations may vary per week

- Course Material via Minerva
  - annotated slides
  - lab assignments

- Exam
  - two parts: theory exam and hands-on exercise on PC
  - both parts count for 50% in the end-score
  - < 8/20 for one part → Aggregate score capped to 8/20

SOFTWARE ARCHITECTURE

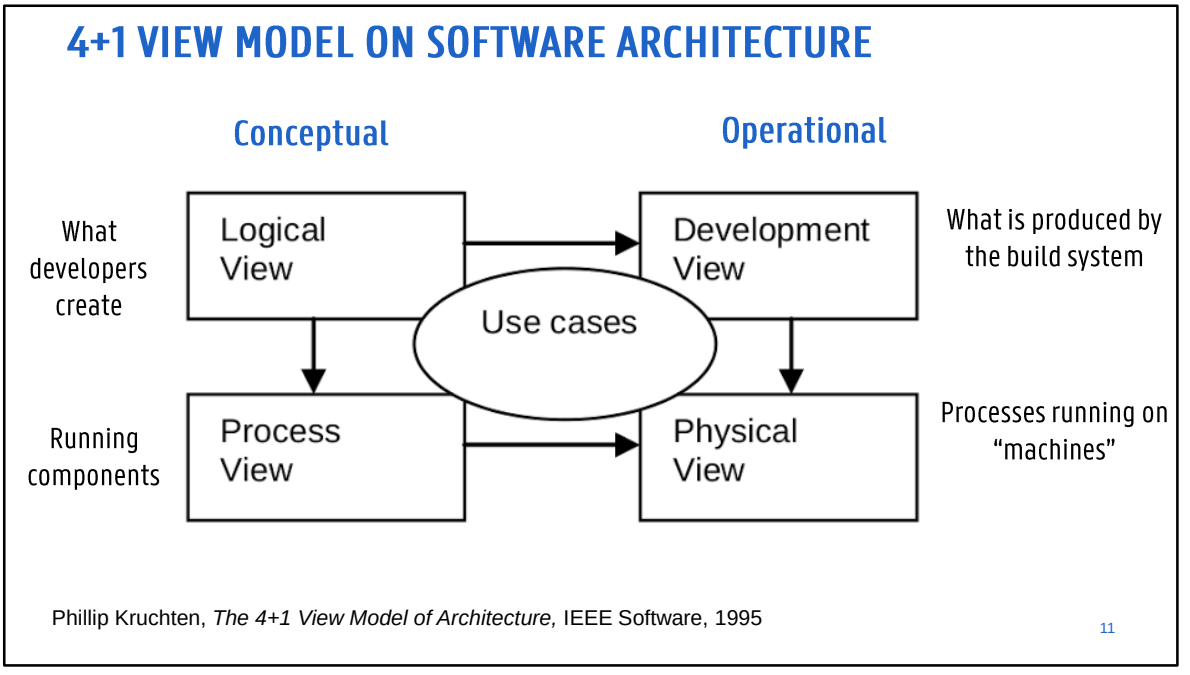


Any application system must meet functional and non-functional requirements.

Functional requirements are the functions that need to be provided by the application to its users (customers, employees, etc.). These reflect the problem domain in which you are working. If you develop enterprise software for financial trading, the functional requirements will likely cover monitoring of stock rates, visualization of trade volumes, sell/buy algorithms based on machine learning, etc… The enterprise software system of a travel agency will have totally different functional requirements: managing the contracts with hotels and airlines, a booking procedure for physical travel agencies, an online shop, etc. The system may also contain a module for employee (HR) management, e.g. to calculate bonuses for highly performing employees.

Non-functional requirements (NFR) specify quality attributes that can be used to judge the operation of a system. This category of requirements aren't features of the system, but are a required characteristic. You cannot write specific lines of code to implement them, rather they are emergent properties that arise from the entire solution. Systems with entirely different functional requirements may have very similar non-functional requirements.
Example NFRs are:

- availability, e.g. 99.9999% uptime
- deployability, e.g. a newly coded feature of the application must get into production no later than 10 minutes after it was released by the development team



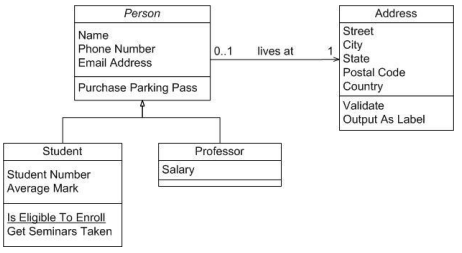
Software architecture design has very little to do with functional requirements. You can implement a set of use cases – an application's functional requirements – with any architecture. Architecture matters, however, because it affects the quality attributes: with a wrongly designed architecture, your system will not be maintainable, testable, extensible, etc.

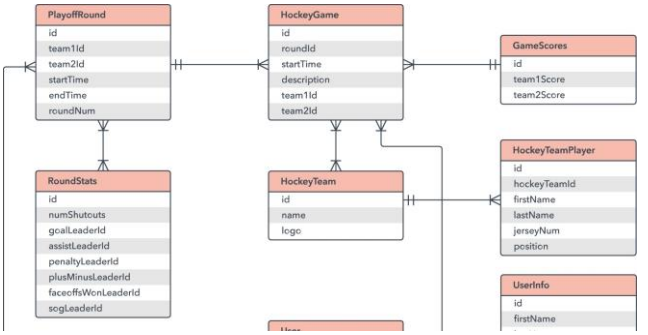
But what is exactly meant with the term software architecture? There are numerous definitions. In essence, **the software architecture defines the structure of the components (software elements) and their relationships**. It is the decomposition of the architectures into parts and the relationship between those parts that determine the application's quality attributies (-"ilities").

It is not possible to capture all details of a system architecture from a single perspective. The "4+1" view model defines four different views of a software architecture. Each view describes a particular aspect of the architecture and consists of a particular set of elements and relationships between them. The four views are: logical view, implementation view, process view and deployment view.

In addition to these 4 views, there are the scenarios (the +1 in the 4+1 model) that animate views. Each scenario describes how the various architectural elements within a particular view collaborate in order to handle a request. A scenario in the logical view, for example, shows how classes collaborate. Similarly, a scenario in the process view shows how the processes collaborate. Scenarios do not bring additional information, but they are a good checkpoint for the system architecture.

LOGICAL VIEW





UML class diagram

Entity-Relationship diagram

- **Elements**: classes and packages
- **Relations**: inheritance, association, depends-on, etc.
- Allows to identify common mechanisms and design elements
- Advanced use of design patterns if you use object-oriented languages

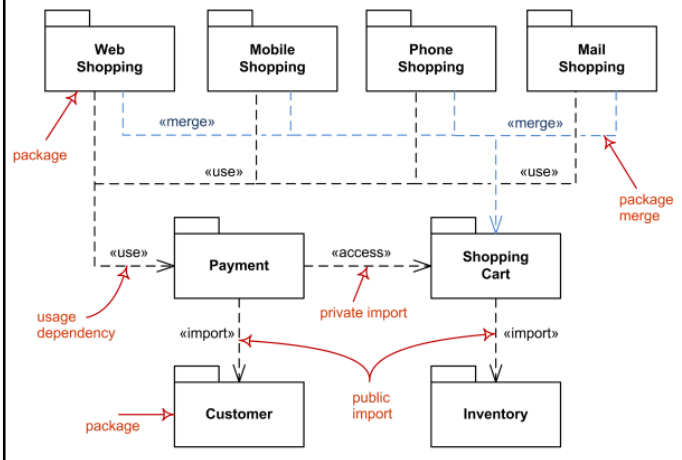
12

The logical view describes the software elements that are created by developers. When using an object-oriented language, these elements are classes and packages. The relations between them are the relationships between classes and packages including inheritance, associations and depends-on.

The logical view is e.g. a UML class diagram showing a set of classes and their logical relationships: composition, inheritance, usage. Such a class diagram will also allow you to identify common mechanisms and design elements, allowing you to improve your object-oriented code using design pattens like Factory, Adapter, etc…

Note that the logical view may also use other forms than the UML class diagram. For example, in a data-driven application you can use an Entity-Relationship diagram. An example of such a diagram is shown at the right side of the slide.

IMPLEMENTATION VIEW (OR DEVELEOPMENT VIEW)



- **Elements**: modules (e.g. JAR) and components (WAR, executables)
- **Relations**: their dependencies
- Organisation of the logical entities in packaged software modules/executables
- Describes import and export relationships
- Packaging can be based on:
  - commonality
  - programming language
  - functionality (layered architecture: presentation, business logic, data)
  - management decisions

13

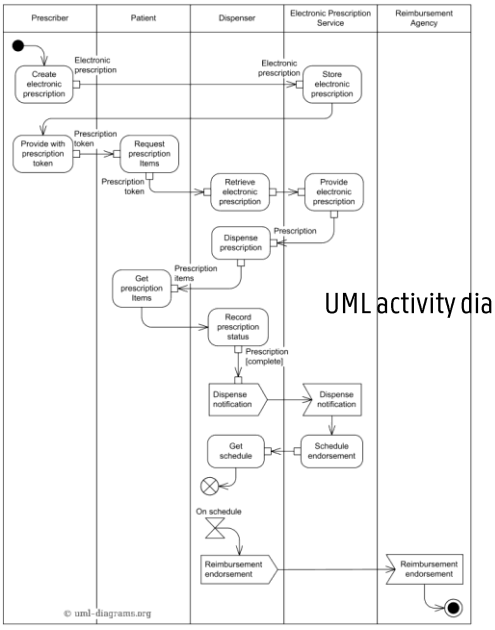
The implementation view describes the output of the build system. This view consists of **modules**, which represent packaged code, and **components**, which are executable or deployable units consisting of one or more modules. In Java, a module is a JAR file and a component is typically a WAR file or an executable JAR file. The relations between them include dependency relationships between modules and is-composed-of relationships between components and modules.

The implementation architectural view of the system is represented by module and subsystem diagrams, showing the “export” and “import” relationships. How the entities of the logical view are split in different modules can be driven by:

- commonality: e.g. many subsystems require logging, user authentication, etc…
- programming language (not all parts are necessarily in the same programming language)
- functionality: e.g. a 3-layered architecture: presentation, business logic, data
- management: e.g. the responsibilities assigned to a team of developers

PROCESS VIEW

- Elements: processes
- **Relations**: inter-process communication
- Flow of execution between processes
- Processes are tactical controls
  - start/stop/duplicate/...
- Important impact on non-functional requirements:
  - Blocking threads/processes
  - Failing/slow remote service
  - Synchronization of state



UML activity diagram

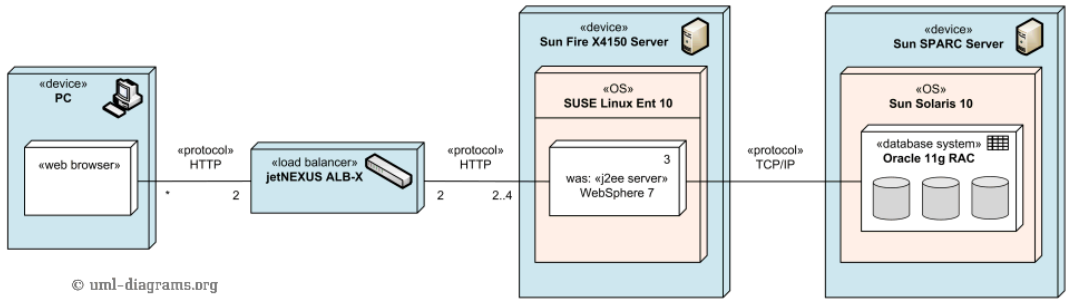
The process view describes the components at runtime. Each element is a process and the relations between processes represent inter-process communication (IPC). Process views can e.g. be UML activity diagrams, allowing you to model:

- decision points (e.g. choose between several payment options when ordering online)
- failure paths (e.g. what should be done if the credit card was refused)
- communication (synchronous or asynchronous method calls)
- synchronization (what should be executed concurrently and what sequentially)

Processes are defined as executable units that can be *tactically controlled*: started, recovered, reconfigured and shut down. In addition, processes can be replicated for increased distribution of the processing load, or for improved availability. It is thus clear that the process view looks at non-functional requirements such as performance and availability. For instance:

- wrongly designed synchronization may result in deadlocks with processes or threads waiting indefinitely for each other
- you should anticipate for larger network delays when contacting an external service (e.g. an external payment provider)
- you should think about the consistency and synchronization of object and session state if you have multiple replicas of the same process.

DEPLOYMENT VIEW



- Elements: physical or virtual machines
- Relations: networking
- Deployment affects non-functional requirements: availability, reliability, scalability, performance…
- In many situations there are multiple deployment configurations
  - development vs. production
  - per region
  - per customer

15

The deployment view of the architecture shows how the various processes are mapped to physical or virtual machines. The relations between machines represent networking.
This view describes how many machines are used, what is deployed on what machine, hardware specifications, etc… The deployment architecture thus impacts non-functional requirements such as availability, reliability, scalability and performance. Some examples: the latency between two components will be much less when the components are deployed on the same server, some components will be heavily accessing the hard disk whereas others are more CPU-intensive, etc…

In many situations, multiple physical configurations will be used. For example, you can have a separate physical deployment for development and testing, and a more stable production environment. Another example is where there are deployments of the system on various sites (e.g. in different continents) or for different customers. The mapping of the software to the machines therefore needs to be highly flexible and have a minimal impact on the source code itself.

5TH VIEW: SCENARIOS/USE CASES

Use Case 7.5 The *Book Flight* Use Case with Extension Points

Book Flight

1. The use case begins when the agent specifies a travel itinerary for a client.
2. The system searches a set of appropriate flights and presents them to the agent.
   (frequent flier a)
3. The agent chooses Select Flight.
4. The system verifies that space is available on the flight and reserves a seat on the flight.
   (frequent flier b)
5. The agent finalizes the booking by supplying payment information.
   (frequent flier c)
6. The system books the seats and issues the ticket.

Alternatives

4a: Seat is not available in ticket category:

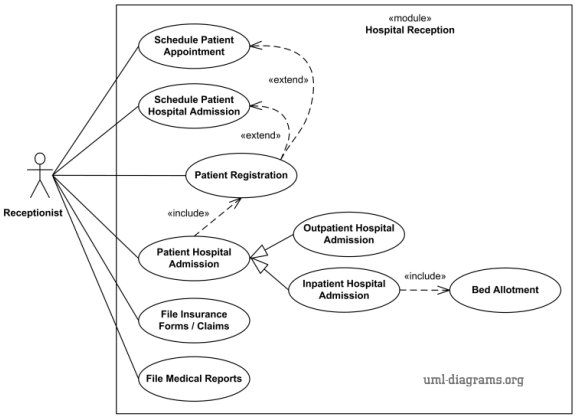
4a1. The system informs the agent that no seats are available in the client's chosen price category.

4a2. The agent specifies another price preference.

4b: Seat is not available (flight fully booked):

4b1. The system informs the agent that no seats are available at all.

4b2. The agent specifies another set of departure time preferences.
(frequent flier d)



Redundant, but very useful view:

- helps in discovering the architectural elements of each view
- validation (starting point of tests)

16

The description of an architecture is illustrated using a small set of use cases (or scenarios), which become a fifth view. A use case involves an actor and the flow that a particular actor takes in a given functionality or path. These often get grouped so you have a "set" of use cases to account for each scenario. A scenario involves a situation that may have single or multiple actors that take a given functionality or path to resolve the scenario.

Each scenario describes how the various architectural components within a particular view collaborate in order to handle a request. Executing a scenario in the logical view, for example, will show you how the classes collaborate. Similarly, executing a scenario in the process view will show you how the processes collaborate.

Because this view does not bring additional information, it is redundant with the other ones (hence the "+1"), yet it plays two critical roles:

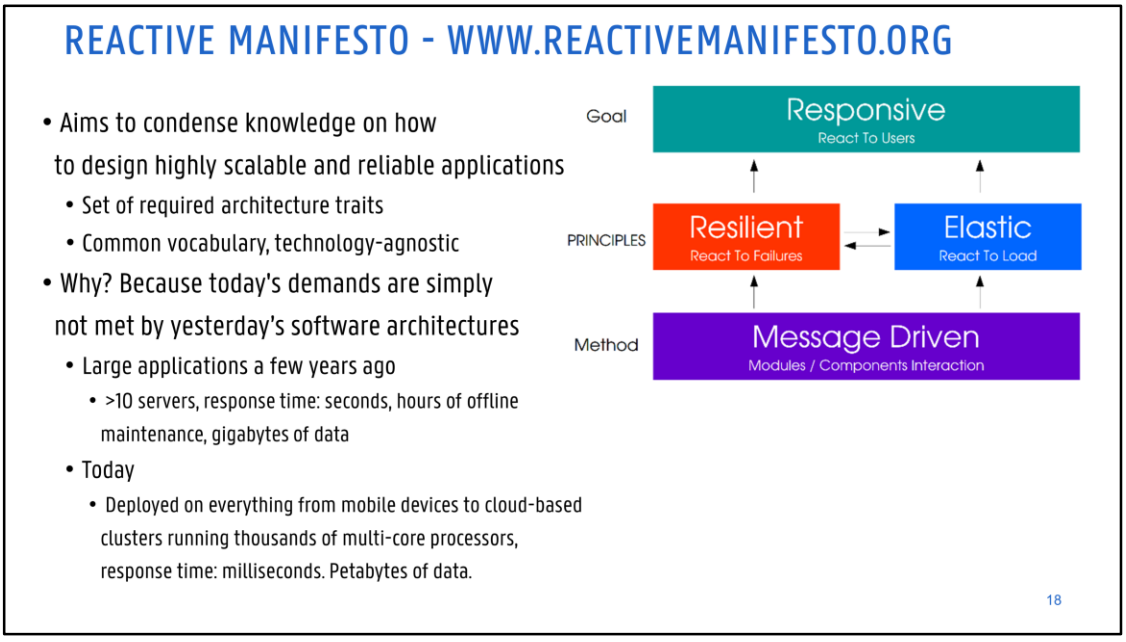
- it helps in discovering architectural elements during the design. Note that this applies to all 4 views: scenarios and use cases are not only for defining your logical view!
- it validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

REACTIVE MANIFESTO

17

After having defined what is a software architecture, let us now look at what is required to have a *good* software architecture. Because functional requirements are really specific to your application, in the next slides we will focus on non-functional aspects.

We will use the *reactive manifesto* as our guideline. The reactive manifesto was published by a number of professionals and should be seen as a blueprint for modern distributed system design.



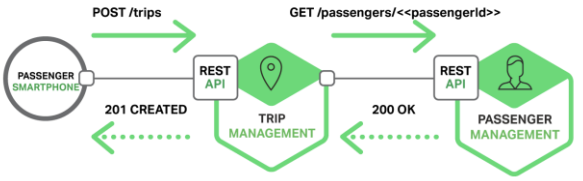
The reactive manifesto aims to condense the knowledge on how to design highly scalable and reliable applications into a set of required architecture traits, defining a common vocabulary between all participants in that process: developers, architects, project leaders, management, etc. Reason for this is that today's application demands are simply becoming too high to be met by software architectures of yesterday.

Another way to look at the Reactive Manifesto is to call it a dictionary of best practices, some of which have been known for a long time. Its benefit is to collect a consistent and cohesive set of principles and to define names for them and their key building blocks to avoid confusion and facilitate ongoing dialog and improvement.

A reactive system is an architectural style that allows multiple individual applications to coalesce as a single unit, reacting to its surroundings, while remaining aware of each other—this could manifest as being able to scale up/down, load balancing, and even taking some of these steps proactively. The manifesto lists 4 architectural traits: message-driven, resilience, elasticity and responsivity.

ARCHITECTURAL TRAIT: MESSAGE DRIVEN

- Asynchronous message-passing between components
  - Addressable recipients await the arrival of messages and react to them
  - Establishes a boundary that enables
    - Loose coupling (the message is the interface)
    - Decoupling in time (recipient can become alive later)
    - Decoupling in space: location transparency (you only require an address)
- Enables load management, elasticity and flow control by monitoring and shaping the message queues in the system
- Non-blocking: less system overhead, ability to delegate errors



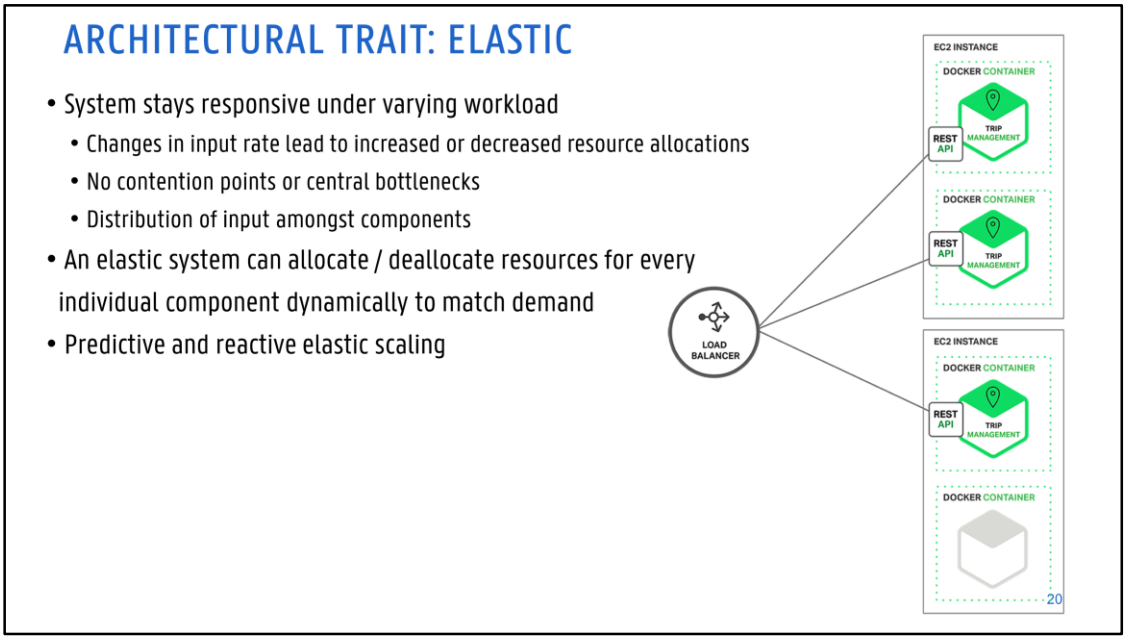
19

Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures:

- loose coupling: components only need to agree on the message format and work over well-defined protocols (e.g. HTTP)
- decoupling in time: sender and receiver can have independent life cycles (the recipient can become active later). Sender and receiver can work at their own tempo.
- decoupling in space (defined as location transparency) means that the sender and receiver do not have to run in the same process, but wherever it is most efficient – which might change during an application's lifetime. The sender only needs to know the recipient's locator, not where the recipient is exactly residing. Think of a phone number or an email address: knowing this is sufficient to reach your contact, regardless of the physical location of that person. This is a very useful property in modern systems, as the recipient process may have migrated to another server or even another datacentre.

Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system.

Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead because one component must not wait for another component. It also means that we will delegate failures as messages to other components.



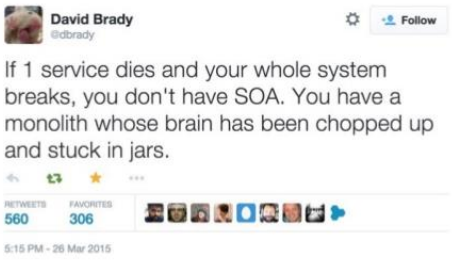
The system stays responsive under varying workload. Reactive systems can react to changes in the input rate by **increasing or decreasing the resources** (CPU cores, memory, VMs, etc…) allocated to service these inputs. In other words, elasticity means that you can scale your system up in response to increasing user load, but also scale down again to save cost if the load diminishes. An elastic system can allocate and deallocate resources **for every individual component**.

This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive systems support predictive and reactive scaling algorithms. Predictive scaling is based on historical knowledge, e.g. it can be expected that many students will access Oasis at the same time when exam results have been published. Reactive scaling is based on live performance measures and is needed to cope with unanticipated peaks.

ARCHITECTURAL TRAIT: RESILIENT

- Any service call can fail, but the system will *self-heal*
- Resilience is achieved by
  - containment and isolation of failures
  - replication for high availability
  - delegation of failures
- Detect failures quickly by monitoring
  - operational metrics (e.g. requests per second)
  - business metrics (e.g. orders per minute received)

  and automatically restore services when issues are detected
- Provide fallback services, e.g. Netflix graceful degradation
  - if recommendation service is down revert to most popular movies instead of personalized picks



Resilience is about reactivity *to failure*. It is an inherent functional property of the system: something that needs to be designed for, and not something that can be added in retroactively. Resilient systems keep processing even when there are (transient or persistent) stresses or component failures disrupting normal processing. Resilience is beyond graceful degradation, it is about being able to fully recover from failure: to *self-heal*.

Resilience is achieved by:

- containment and isolation of failures, so that parts of the system can fail and recover without compromising the system as a whole
- replication: ensuring high-availability, if one replica goes down another one can take over
- delegation: if a failure occurs in a component, the component should sent this failure as a message to a supervisor component, rather than throwing an exception back to the client. Clients should not have the responsibility of handling server failures.

Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Real-time monitoring of the application is important, checking both operational metrics (such as how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received).

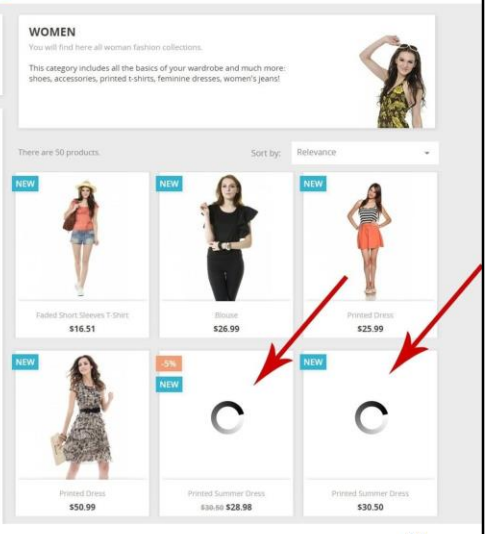
ARCHITECTURAL TRAIT: RESPONSIVE

The system responds in a timely manner

- lazy loading: first load important stuff and show it ASAP
- show progress
- individual slow performing service should not slow down others

Problems: detect quickly, deal with effectively

As far as users know, when the response time exceeds their expectation, the system is down



22

While 10-15 years ago it was normal for websites to go through maintenance or to have a slow response time, today everything should be online 24/7 and should respond with lightning speed. If a system is slow or down, users would prefer an alternative service. Today, slow means unusable or broken.

Responsiveness means that the system responds to users/clients in a timely manner if at all possible. User should see progress, e.g. using a mechanism like lazy loading. Responsiveness is the cornerstone to how users like your system (usability and utility), but it also means that problems must be detected quickly and dealt with effectively.

TL;DR, short for "**too long; didn't read**", is Internet slang to say that some text being replied to has been ignored due to its length. It is also used as a signifier for a summary of an online post or news article.

TL;DR

- Modern software architectures are systems of distributed components
- These systems must meet functional and non-functional requirements
- The involved complexity requires to evaluate the architecture from 4 different viewpoints
  - logical
  - process
  - development
  - physical
- Reactive architectures have the following traits
  - message driven
  - elastic
  - resilient
  - responsive, even under load and failure

BIBLIOGRAPHY

BIBLIOGRAPHY

- Phillip Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, 1995
- www.reactivemanifesto.org
- https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems