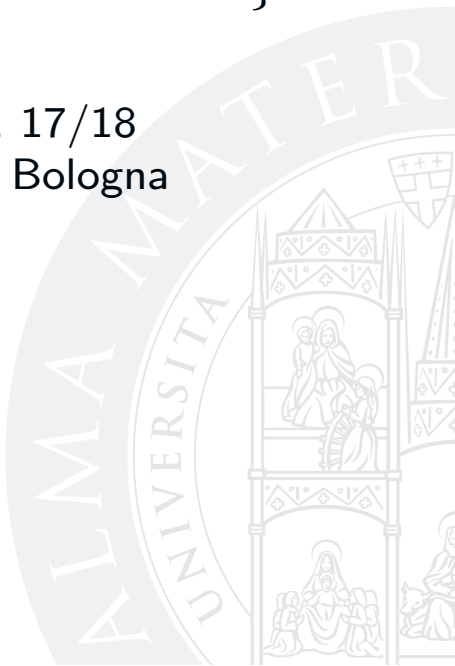
A taste of Microservices with Kotlin

Matteo Gabellini {matteo.gabellini2@studio.unibo.it}
Elia Di Pasquale {elia.dipasquale@studio.unibo.it}

Final Course Presentation @ Distributed Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

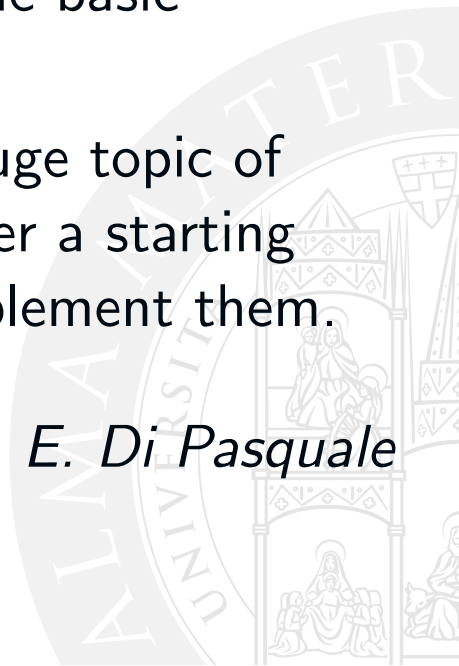
October 11, 2018



Introduction

- We start this presentation with an introduction to microservices in order to understand them in a more clear way.
- Afterwards we will introduce the Kotlin language and its key features.
- In the end we will show some little examples that use Kotlin and other different technologies in order to implement some basic applications modelled with microservice architecture.
- The scope of this presentation is not to face all the huge topic of microservices, but to show the key features and to offer a starting point to explore the enabling technologies used to implement them.

— M. Gabellini, E. Di Pasquale



Outline

Lecture Overview

Introduction to Microservices

- Since 2014 more and more enterprises say that they use microservices and microservice architecture to develop their systems.

Introduction to Microservices

There isn't a formal definition of microservice architecture, but an acceptable definition can be:

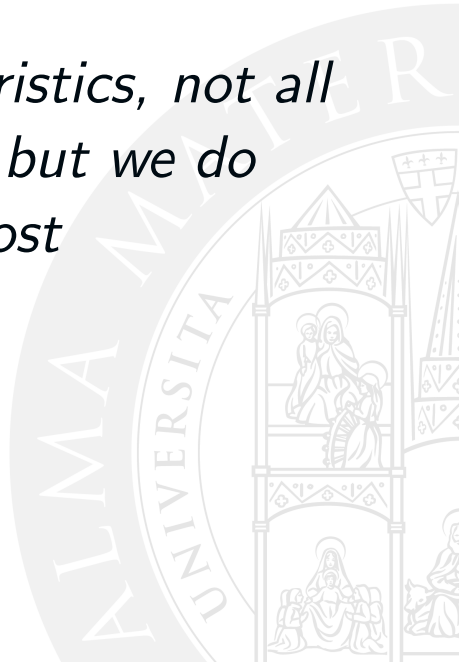
What are Microservices?

"In short, the microservice architectural style is an approach to developing a single application as **a suite of small services**, *each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."* – James Lewis and Martin Fowler [a]

[a] https://martinfowler.com/microservices/

Charatteristics of Microservice Architectures

- When you look at a set of projects that assert to use microservices, is possible to extract a common set of characteristics from the architecture that can be defined as the principal characteristics of an Microservice Architecture.
- *As with any definition that outlines common characteristics, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics.* [20]



Charatteristics of Microservice Architectures

Common charatteristics

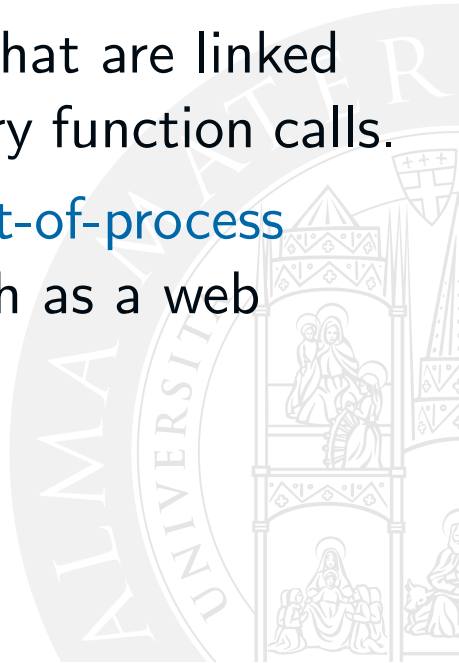
- Componentization via Services
- Organized around Business Capabilities
- Products not projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

Componentization via Services I

Component

In software development a component is a unit of software that is independently replaceable and upgradeable.

- In monolithic style we define libraries as components that are linked into a program and interact each other using in-memory function calls.
- In the microservices's style as classic SOA we have out-of-process components that communicate with a mechanism such as a web service request or remote procedure call.



Componentization via Services II

Key aspects

- Service are independently deployable: in ideally context a change to a service only require that service to be redeployed.
- Services have a more explicit component interface rather than libraries.
- Using explicit remote call mechanisms it make easy to define Published Interface (that is different from public interface [a]).

[a] https://martinfowler.com/bliki/PublishedInterface.html

but there are downsides...

Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained, which is often more awkward to use. [20]

Organized around Business Capabilites I

Conway's Law

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." – Melvyn Conway, 1967

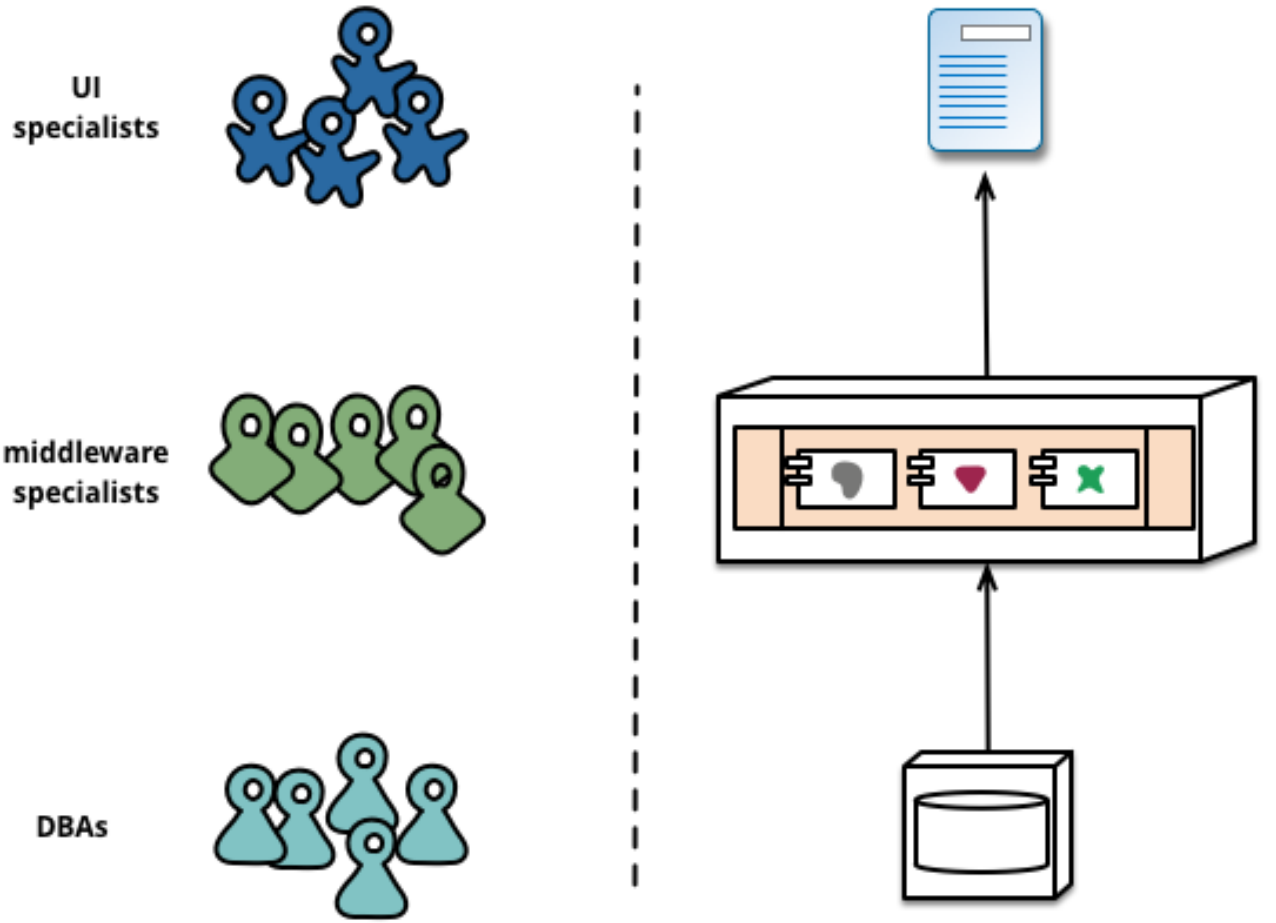
Organized around Business Capabilities II

In the classic monolithic style when a large application is split into parts, often management focuses on the technology layer leading to a division of the teams into:

- UI teams
- Middleware logic teams
- Database teams

This division of the teams brings to an architecture of the application where the components are split with the same criteria.

Organized around Business Capabilities III



[1]

[1] from: https://www.martinfowler.com/articles/microservices.html

Organized around Business Capabilities IV

Microservice organization approach

The microservice approach to division splits services around business capability. $\longrightarrow$ **Business-oriented decomposition**

Such services take a broad-stack implementation of software for that business area:

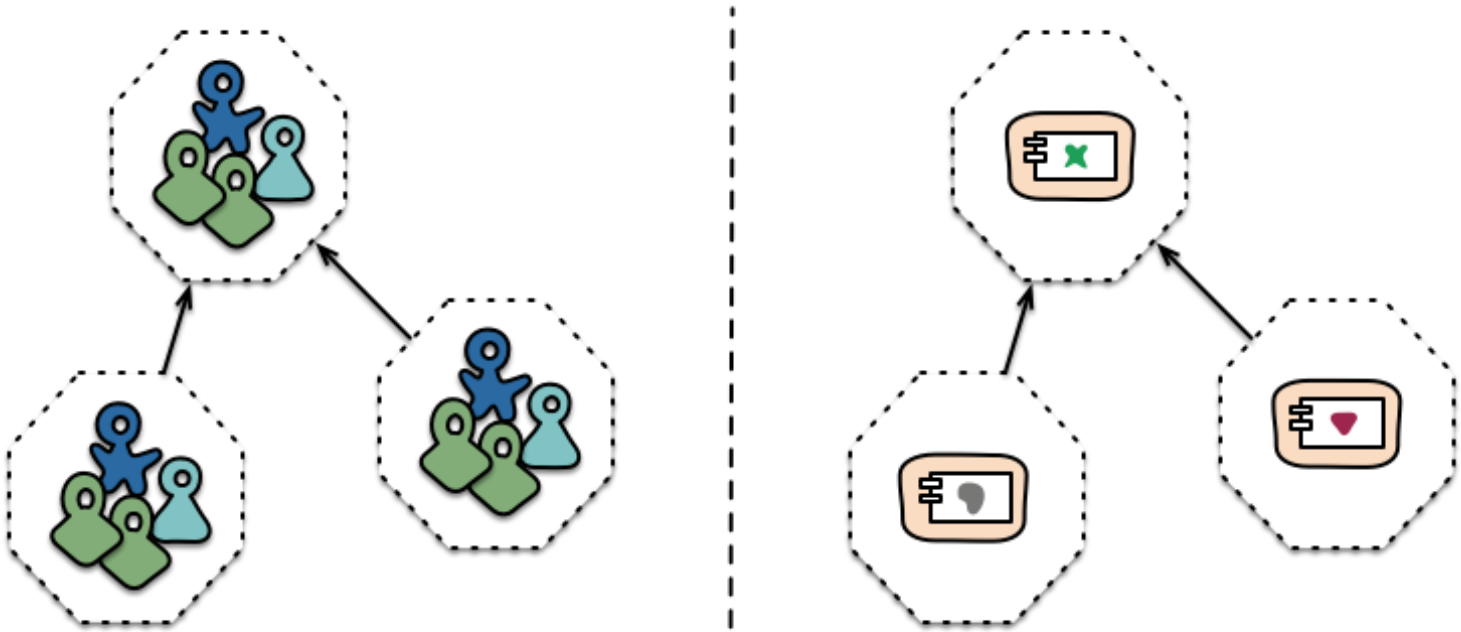
- user-interface;
- persistent storage;
- ... and any external collaborations.

Consequently...

Team organization

The teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.

Organized around Business Capabilities V



[2]

[2] from: https://www.martinfowler.com/articles/microservices.html

Organized around Business Capabilities VI

- With this necessarily more explicit separation required by service components *makes it easier to keep the team boundaries clear.*
- So Microservices involve an organization change. [22]

Products not projects

Project Model

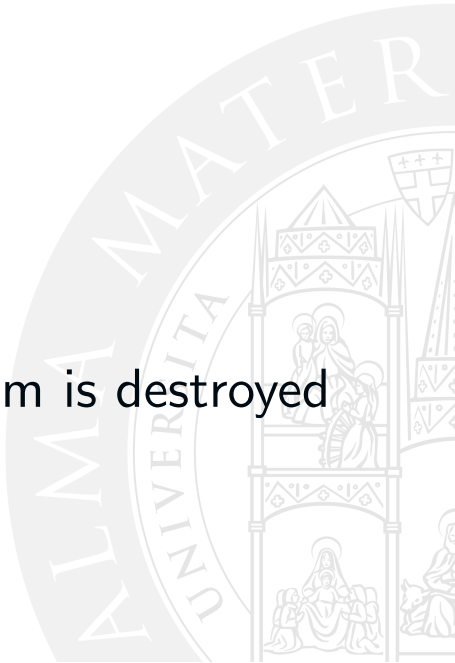
In the typical application development is used the project model where the aim is to deliver a piece of software which is then considered to be completed.
Developers *looking at the software as a set of functionality to be completed*.

Product Model

Microservice model instead prefers the notion that a team should own a product over its full lifetime.
There is an on-going relationship where the question is how can software assist its users to enhance the business capability

Products not projects: The "Amazon" Example and the DevOps culture

- A concrete example of this approach is the *Amazon*'s notion
  - **"you build, you run it"** [19]
- This approach brings developers to:
  - take full responsability for the software production.
  - increase the contact with the user
  - take some of the support burden
- DevOps culture [27]
  - the wall between development team and operation team is destroyed [25]



Smart endpoints and dumb pipes

Classic SOA strategy

Often in the classic SOA significant smarts are put in the communication mechanism itself (e.g Enterprise service bus - ESB where it includes sophisticated facilities for message routing, choreography, transformation, and applying business rules).

Microservice approach

The microservice approach instead is to put all the elaboration in the services *(smart endpoints)* and to use simple RESTish protocols instead complex protocols *(dumb pipes)*.

Smart endpoints and dumb pipes

- The two protocols used most commonly are:

First Approach: HTTP request-response with resource API's

Microservices teams use the principles and protocols that the world wide web is built on:

- HTTP
- REST API

Second Approach: Lightweight messaging

- The second approach in common use is messaging over a lightweight message bus.
- The infrastructure chosen is typically dumb as in acts as a message router only. Simple implementations such as RabbitMQ or ZeroMQ. They don't do much more than provide a reliable asynchronous fabric.

Decentralized Governance

- Often in the monolithic style is used a *single technology platform*.
- Microservice style instead prefers use **the right tool for the job** (*polyglot technologies*)

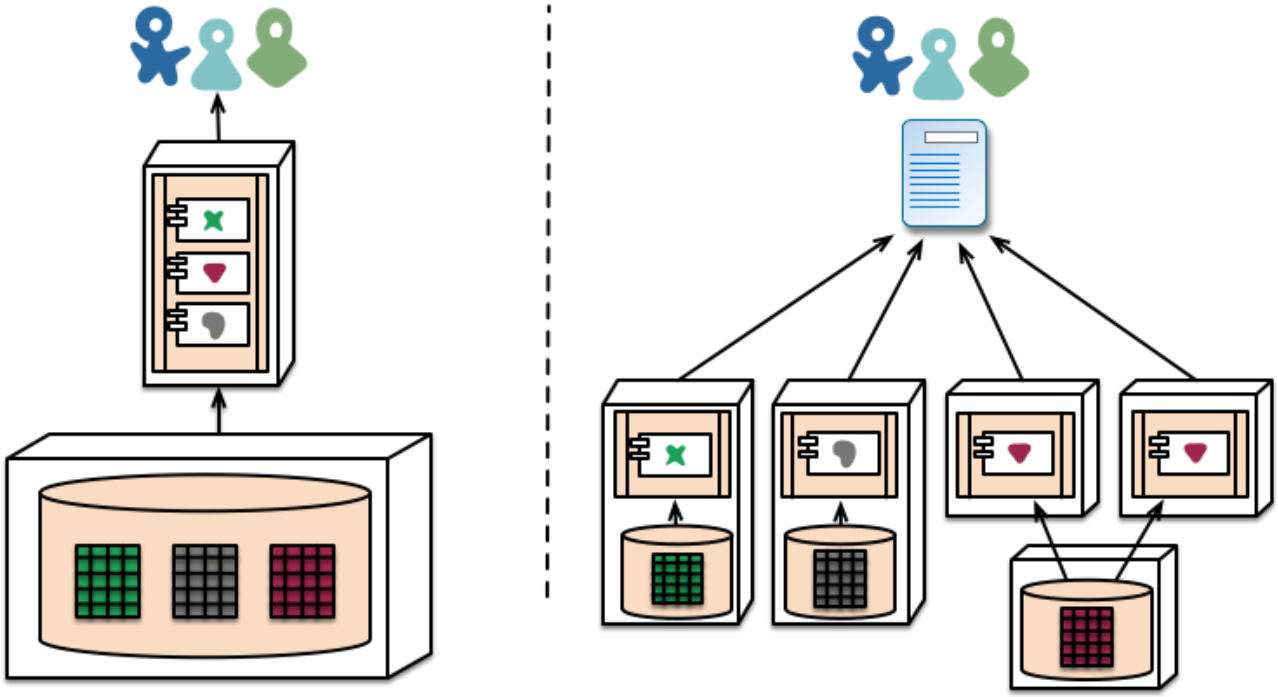
- Splitting the monolith's components out into services give us a choice when building each of them.
- So we have also the opportunity to analyze and choose a specific technology to implement each service.

Is an Opportunity not an Obligation

"Of course, just because you can do something, *doesn't mean you should*, **but partitioning your system in this way means you have the option**." [20]

Decentralized Data Management

Decentralization of data management presents in a number of different ways.



[3]

[3] from: https://www.martinfowler.com/articles/microservices.html

Decentralized Data Management: Polyglot modeling

Problem at the Conceptual Level

- The conceptual model of the world will differ between systems. (e.g the sales view of a customer is different from the support view)
- This issue is present between applications and sometimes also within applications.

How to chose the correct conceptual model?

- A way to treat this issue is to split the application entities using the notion of **Bounded Context of the Domain-Driven Design (DDD)**.
- Using this approach we are able to divide complex domain into multiple bounded contexts *(polyglot modeling)* and maps out the relationships between them.

Decentralized Data Management: Polyglot persistence

Microservice also decentralizes the data storage decisions.

Monolithic prefers:

- single logical database
- or in enterprise context, a single database across a range of applications.

Microservice prefers:

Each service manages its own database (polyglot persistence).
Two variant:

- different instances of the same database technology
- entirely different database systems

Decentralized Data Management: Managing updates I

There are implications for managing updates with the decentralizing of responsibility for data across microservices.

Typical approach

When updating multiple resources the common approach is to use transactions to guarantee consistency.

- Unfortunately transactions, as well know, impose significant temporal coupling, which is problematic across multiple services.

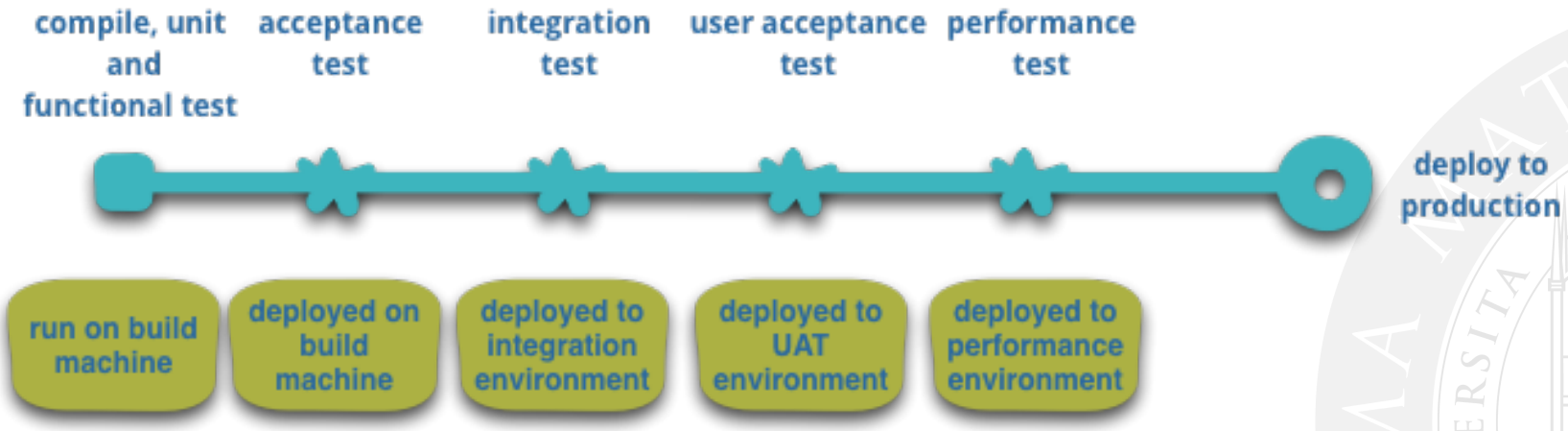
Decentralized Data Management: Managing updates II

Microservices approach

- Microservice architecture emphasizes transactionless coordination between services, with explicit recognition that
  - **consistency may only be eventual consistency**.
- Problems are dealt with by compensating operations, fixing mistakes.

Infrastructure Automation

- Teams that develop microservices use extensively Continuous Integration and Continuous Delivery tools.
- They run a lot of automated tests in order to have as much confidence as possible that their software working.



[4] from: https://www.martinfowler.com/articles/microservices.html

Infrastructure Automation

- This tools result particularly important during production.
- Today a lot of big companies, among which Netflix [5] and Facebook [6], develop open source tools (used first of all internally in this companies) that help developers to create, build, deploy and manage microservice systems.

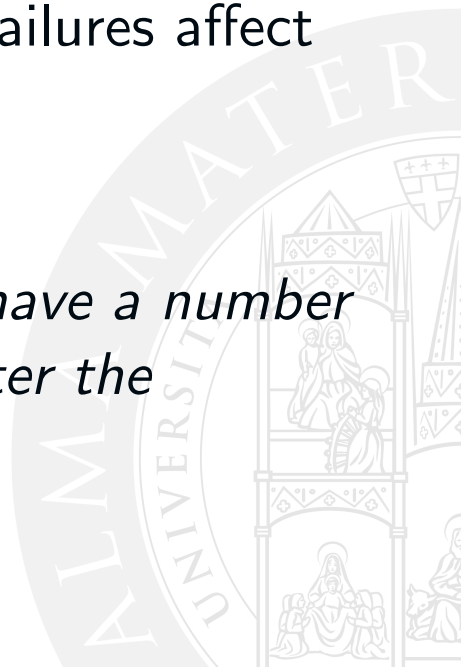
[5] http://netflix.github.io/
[6] https://opensource.fb.com

Design for failure

- The application that uses microservice architecture needs to be designed so that can tolerate the failure of a service.
- A service can fail at any time, so it's important to be able to detect failures quickly and, if possible, automatic restore the service.
- Microservice teams reflect constantly on how service failures affect the user experience.

Remember...

- Synchronous calls considered harmful: *Any time you have a number of synchronous calls between services you will encounter the multiplicative effect of downtime*.



Design for failure: Real-time Monitoring

The need of monitoring

In order to detect failures is extremely important to do real-time monitoring of application and check:

- architectural elements (e.g. how many requests per second is the database getting)
- business metrics (e.g., how many orders per minute are received)

Doing Semantic Monitoring can provide an early alert system of something going wrong. Is important doing monitoring at different level of abstraction from entire application until single service.

Design for failure: Testing on Production

With Microservice architecture became important also run tests directly on the production system.[23]

- This could reveal if services are really stateless [22] and independent each other.

Is also necessary to make automated destructive testing, because [22]:

- failure will occurs certainly and will occurs when engineers are away from office (during the night in the weekend)
- For this reason is necessary to simulate failure during working hours when engineers are at work. [7]

.

[7] See the nice example of the Chaos Monkey tool used by Netflix `https://www.youtube.com/watch?v=57UK46qfBLY`

Evolutionary Design

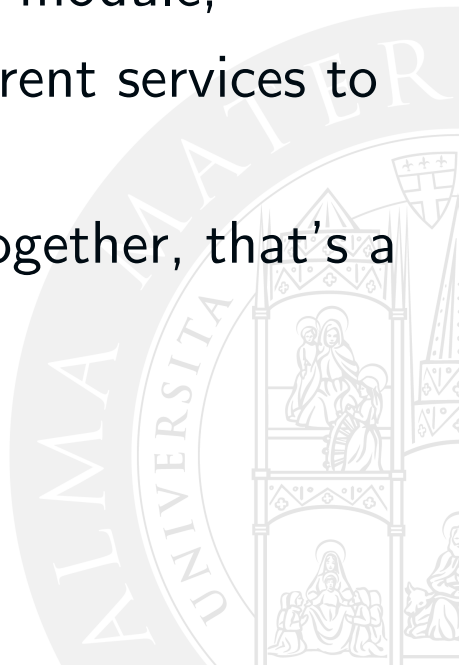
The key property of a component is the notion of indipendent replacement and upgradeability.

- Service decomposition can be seen as a tool to enable application developers to control changes in their application without slowing down change.[26]

Evolutionary Design: Replaceability

Emphasis on Replaceability that is a special case of the general principle of modular design, is used to drive modularity through the pattern of change and bring to:

- keep things that change at the same time in the same module;
- parts of a system that change rarely should be in different services to those that are currently undergoing lots of churn;
- if you find yourself repeatedly changing two services together, that's a sign that they should be merged



Evolutionary Design: Impact on deployement

Using services give the opportunity for more granular release planning.

With Monolith

any changes require a full build and deployment of the entire application

With Microservices

only need to redeploy the service(s) you modified.

- Pros: this can simplify and speed up the release process.
- Downsides: you have to worry about changes to one service breaking its consumers
  - traditional integration approach is to try to deal with this problem using versioning
  - microservice world prefer to only use versioning as a last resort
  - Solution: avoid a lot of versioning by designing services to be as tolerant as possible to changes in their suppliers (Remember Design for Failure)

Microservices vs SOA

SOA have a lot of different definitions [13] [24]. Depending on which definition of SOA is chosen, the relationship with the microservices also varies.

- For some people Microservices are totally different things from SOA.
- For others, Microservices are one form of SOA.
- For others again, the contrast between SOA and Microservices is as movements rather than technology, so the question "How Microservice architercure differed from SOA" is the wrong question. The right question is "what Microservices movement can learn from the SOA movement". [21]

Often with the term SOA we consider also those systems that use Enterprise Service Bus (ESB); this systems for example are not considered Microservices, because the latter use more lightweight mechanisms (key characteristics: *smart endpoints and dumb pipes*)

When use Microservices?

Microservices provide benefits...

- Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams.

- Independent Deployment: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.

- Technology Diversity: With microservices you can mix multiple languages, development frameworks and data-storage technologies.

...but come with costs

- Distribution: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.

- Eventual Consistency: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency.

- Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

8

[8] from: https://www.martinfowler.com/articles/microservice-trade-offs.html

When use Microservices?

- Microservices aren't the silver bullets. Their correct use need a discrete level of experience in the distributed system programming and CD/CI skills.[14]
- The microservices approach is all about handling a complex system, but in order to do so the approach introduces its own set of complexities called **MicroservicePremium**

MicroservicePremium [15]

The Microservices architectural style needs to manage automated deployment, monitoring, dealing with failure, eventual consistency and other factors that a distributed system introduces.

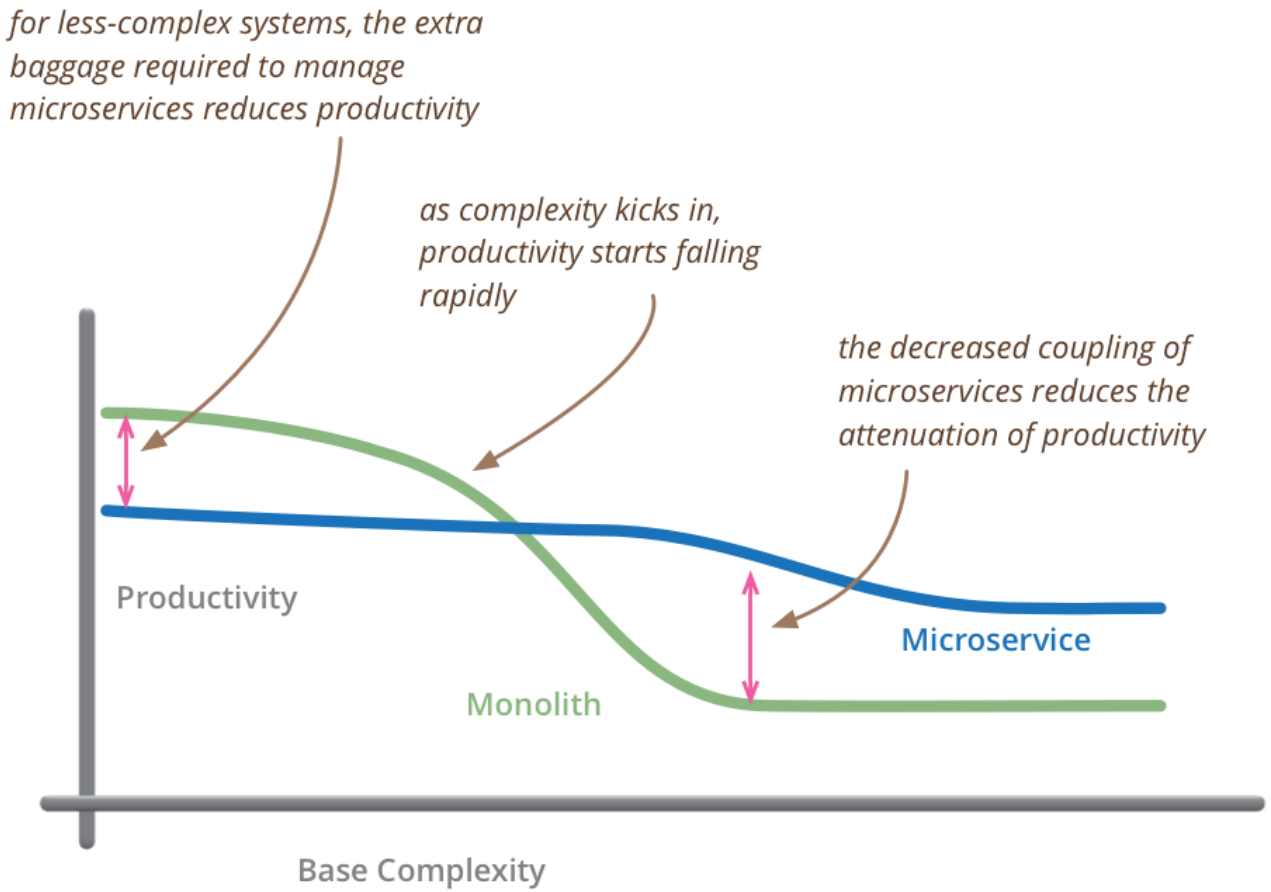
- This, if the system is not complex enough, could be add an overhead that decrease productivity instead of improves it.

When use Microservices?

So the primary guideline would be:

"don't even consider microservices unless you have a system that's too complex to manage as a monolith" [15]

When use Microservices?



but remember the skill of the team will outweigh any monolith/microservice choice [9]

[9] from: `https://martinfowler.com/bliki/MicroservicePremium.html`

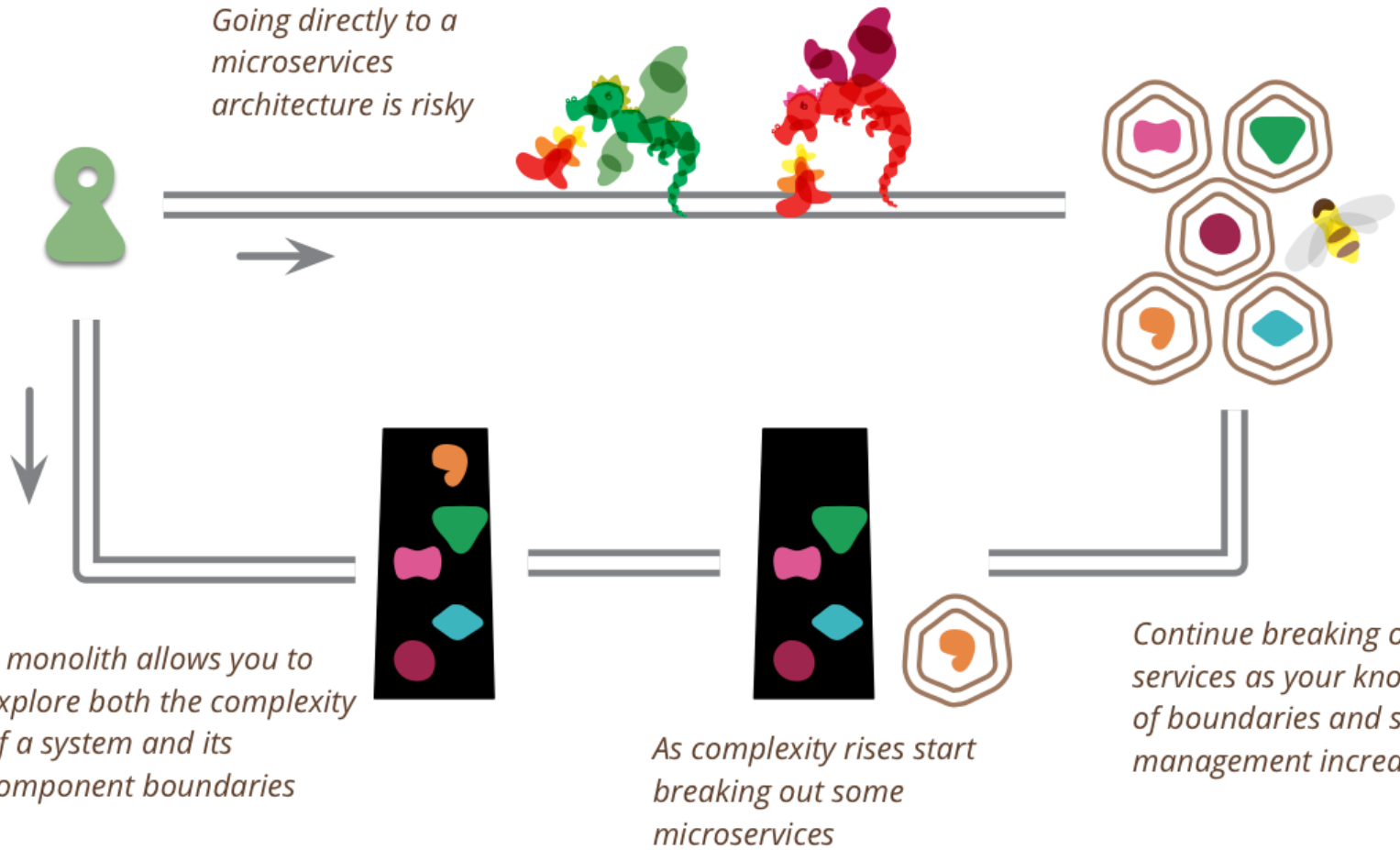
When use Microservices?: MonlithFisrt

Considering the **MicroservicePremium**, Martin Fowler recommended to adopt the strategy:

MonolithFisrt[18]

- *"you shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile."*...
- Start to *"build a new application as a monolith initially"*
- *"design a monolith carefully, paying attention to modularity within the software, both at the API boundaries and how the data is stored. Do this well, and it's a relatively simple matter to make the shift to microservices"* when complexity rises.
- After that start to extract services

When use Microservices?: MonlithFisrt

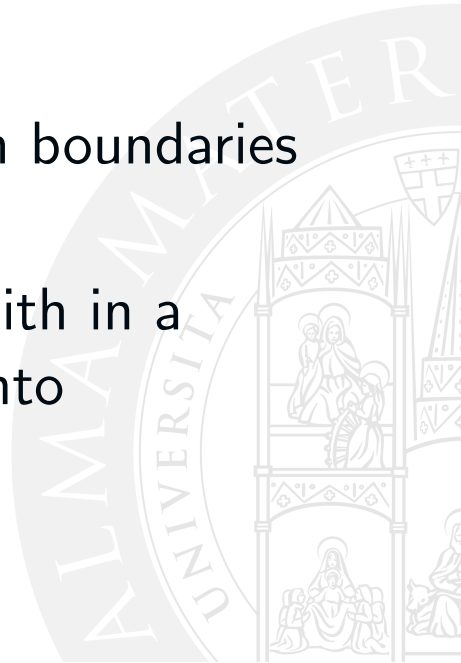


[10] from: https://www.martinfowler.com/bliki/MonolithFirst.html

When use Microservices?: Microservice first

Of course there are contrasting positions to **MonolithFirst** strategy. They assert that **starting with microservices**:

- allows you to get used to the rhythm of developing in a microservice environment
- having teams separated by service boundaries makes it much easier to scale up the development effort when you need to
- have a better chance of coming up with stable-enough boundaries early
- MonolithFisrt need a lot of discipline to build a monolith in a sufficiently modular way that it can be broken down into microservices easily.



When use Microservices?

So

"you shouldn't start with microservices unless you have reasonable experience of building a microservices system in the team." [18]

Lecture Overview

Introduction to Kotlin

What is Kotlin

Kotlin is a powerful statically-typed programming language that runs on Java Virtual Machine (JVM) and also targets Android, JavaScript and Native. Built by JetBrains, the development company behind IntelliJ, PhPStorm and a number of IDE solutions, the language is completely open source. [4]

Introduction to Kotlin

The growth

It was introduced back in 2011. However, the interest for Kotlin peaked in 2017, following the announcement about its official support as first-class Android development language at Google I/O.
As a result, the language, for the first time entered into the top 50 languages in June 2017, according to TIOBE Index, over 30 positions up since May. [a]

[a] https://www.tiobe.com/tiobe-index

Paradigms

Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two (first-class support for features such as higher-order functions, function types and lambdas).

Kotlin platforms

Operating platforms

Kotlin can be used for any kind of development, be it server-side, client-side web and Android. With Kotlin/Native (currently in the works) support for other platforms such as embedded systems, macOS and iOS is coming.
Examples of use: mobile and server-side applications, client-side with JavaScript or JavaFX, data science, etc.

Kotlin for server side

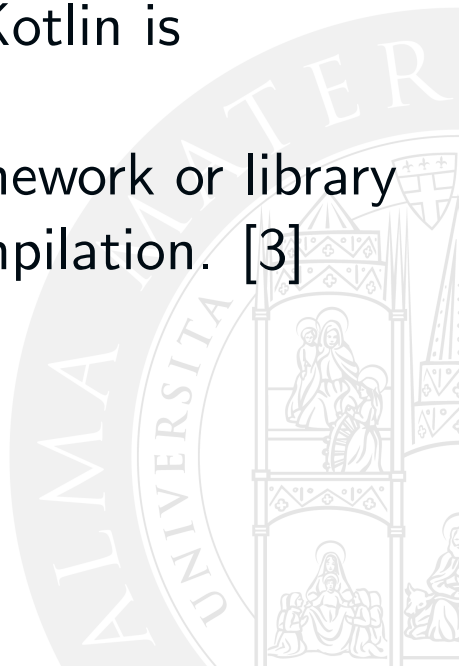
Kotlin is compatible with the JVM and as such you can use any existing frameworks such as Spring Boot, vert.x or JSF. In addition there are specific frameworks written in Kotlin such as Ktor.
Kotlin applications can therefore be deployed into any host that supports Java Web applications, including Amazon Web Services, Google Cloud Platform and more.
Furthermore Kotlin's support for coroutines helps build server-side applications that scale to massive numbers of clients with modest hardware requirements. [10]

Kotlin for Android

Kotlin is supported as a first-class language on Android, bringing all of the advantages of a modern language to the Android platform without introducing any new restrictions.
In fact, Kotlin is fully compatible with JDK 6, ensuring that Kotlin applications can run on older Android devices with no issues.
The Kotlin tooling is fully supported in Android Studio and compatible with the Android build system. [9]

Kotlin for JavaScript

Kotlin provides the ability to target JavaScript; it does so by transpiling Kotlin to JavaScript.
When you choose the JavaScript target, any Kotlin code that is part of the project as well as the standard library that ships with Kotlin is transpiled to JavaScript.
However, this excludes the JDK and any JVM or Java framework or library used. Any file that is not Kotlin will be ignored during compilation. [3]



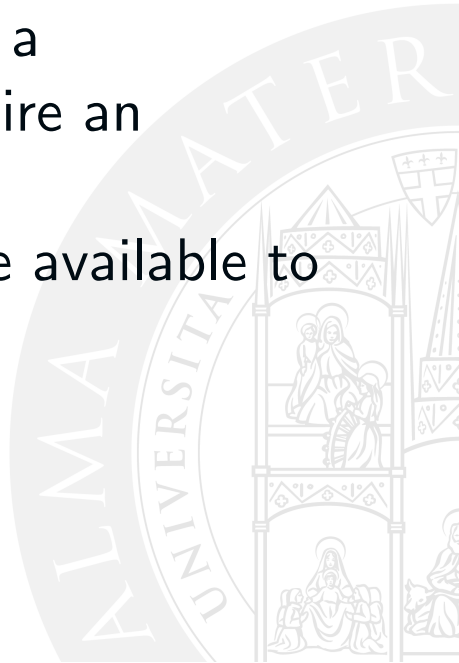
Kotlin for JavaScript

Kotlin compiler goals

- Provide output that is optimal in size
- Provide output that is readable JavaScript
- Provide interoperability with existing module systems
- Provide the same functionality in the standard library whether targeting JavaScript or the JVM (to the largest possible degree)

Kotlin/Native

It's a technology for compiling Kotlin to native binaries that run without any VM.
Kotlin/Native is primarily designed to allow compilation for platforms where virtual machines are not desirable or possible (such as iOS or embedded targets), or where a developer needs to produce a reasonably-sized self-contained program that does not require an additional runtime.
However it is currently in development; preview releases are available to try. [5]



Kotlin/Native

Platforms currently supported

- Windows (x86_64 only at the moment)
- Linux (x86_64, arm32, MIPS, MIPS little endian)
- MacOS (x86_64)
- iOS (arm32 and arm64)
- Android (arm32 and arm64)
- WebAssembly (wasm32 only)

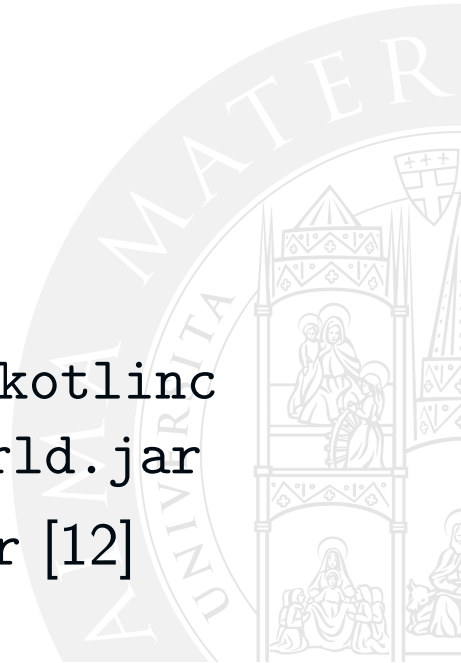
...with the IDE (IntelliJ)

- Install a recent version of IntelliJ IDEA; Kotlin is bundled with IntelliJ IDEA starting from version 15
- Create a New Project selecting Java Module, and select the SDK; Kotlin works with JDK 1.6+. Also, select the Kotlin (Java) checkbox and create the project
- Let's create a new Kotlin file under the source folder (it can be named anything)
- Once we have the file created, we need to type the main routine, which is the entry point to a Kotlin application, after which we can create and run the first "Hello world" in Kotlin [2]

```
fun main() {
  println("Hello World!")
}
```

...with the Command Line Compiler

- Download the compiler
- Installation
  - manual: unzip the standalone compiler into a directory and optionally add the bin directory to the system path
  - with a tool
    - SDKMAN! (UNIX based systems)
    - Homebrew (OS X)
    - MacPorts
    - Snap package (Ubuntu)
- Create the first "Hello world" in Kotlin
- Compile the application using the Kotlin compiler: `$ kotlinc hello_world.kt -include-runtime -d hello_world.jar`
- Run the application: `$ java -jar hello_world.jar` [12]



...with build tools

Supported tools

- Ant
- Maven
- Gradle
- Griffon (external support) [11]

General philosophy

General philosophy

Designed to be an industrial-strength object-oriented language, and a "better language" than Java but still be fully interoperable with Java code, allowing companies to make a gradual migration from Java to Kotlin.

Extension

Extensions allow resolving a static method/property onto an object that already exists, meaning we call the object as if it has the method or property.
They are a good replacement for static utility methods.
Under the hood, this method is just a static method with the first parameter of the method used as the receiver; when declaring it in Kotlin, we can access the class as if we're in the body of the method.

```
fun String.greet():String {
    return this.plus(" we welcome you!")
}
print("John".greet())
```

High order function

Kotlin functions are first-class, which means that they can be stored in variables and data structures, passed as arguments to and returned from other higher-order functions. You can operate with functions in any way that is possible for other non-function values.
A higher-order function is a function that takes functions as parameters, or returns a function.

```
fun performOperation(operation:(Int,Int) -> Int):Int {
    return operation(10,20)
}
performOperation{num1, num2 -> num1 * num2} // Output: 200
performOperation{num1, num2 -> num1 - num2} // Output: -10
```

Null safety

Kotlin's type system is aimed at eliminating the danger of null references from code.
One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception (NullPointerException in Java).
In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references).

```
var cantBeNull: String = "this is a string"
cantBeNull = null // Compilation error
```

To allow nulls, we can declare a variable as nullable string, written `String?`:

```
var canBeNull: String? = "this is another string"
b = null // No error
print(b)
```

Data class

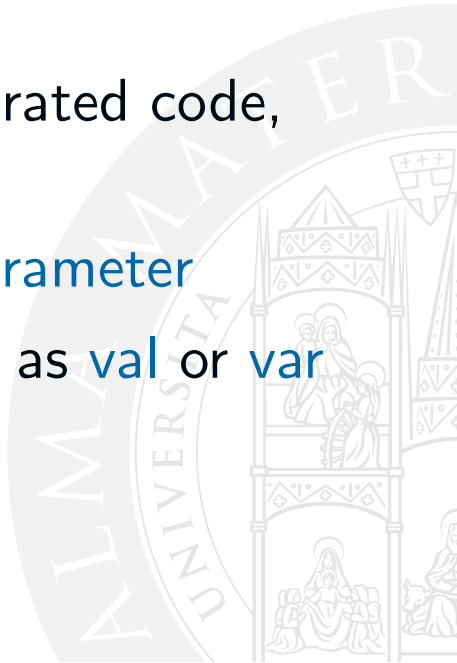
In Kotlin, is possible to mark a class as data class:

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives the following members from all properties declared in the primary constructor: equals(), hashCode(), toString(), componentN() functions corresponding to the properties in their order of declaration, copy() function.

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

- The primary constructor needs to have at least one parameter
- All primary constructor parameters need to be marked as val or var
- Data classes cannot be abstract, open, sealed or inner



Ranges

In Kotlin we can use ranges to express the concept of series of number. Range expressions are formed with rangeTo functions that have the operator form `..` which is complemented by `in` and `!in`.
Range is defined for any comparable type, but for integral primitive types it has an optimized implementation. Integral type ranges (IntRange, LongRange, CharRange) have an extra feature: they can be iterated over. With function like `downTo`, `step` and `until` it is possible to implement particular behaviors that go beyond simple numerical progression.
Examples:

```
if (i in 1..10) {          // Equivalent of 1 <= i && i <= 10
    println(i)
}
for (i in 4 downTo 1 step 2) print(i)

for (i in 1 until 10) {    // i in [1, 10), 10 is excluded
    println(i)
}
```

Pros and cons

Pros

- **Increases team's productivity**: language is compact, clear and efficient, it has a concise and intuitive syntax. It takes less time to write and deploy new code
- **Compatible with existing code**: Kotlin has 100% Java interoperability, including all related tools and frameworks, which provides a rich ecosystem
- **Maintainability**: built and supported by JetBrains, Kotlin has a stellar support for a number of IDEs, including Android Studio
- **Reliable**: Kotlin is a mature language. Being introduced back in 2011, it has gone through multiple Alfa and Beta stages before its final 1.0 release

Pros and cons

Cons

- **Slow learning curve**: being very close to Java, Kotlin still differs in many aspects. Thus, a certain learning curve is involved for a developer who wants to switch languages
- **Slow compilation speed**: in some cases Kotlin it's significantly slower than Java in compilation speed
- **Little community**: while Kotlin is rapidly it still has a small developer community. This leads to limited resources for learning the language and makes finding the answers to any questions more difficult
- **Lack of qualified workers**: as Kotlin is still relatively new to most of the developers, it might be hard to find experienced professionals in this domain

Lecture Overview

Disclaimer

- How discussed in the introduction, the use of Microservices is for the implementation of complex systems and requires a discrete knowledge and experience in distributed system programming and CD/CI.
- So the following exercise doesn't pretend to show all aspects of the development of a microservice system (which would require one or more dedicated university courses), but will show the basic elements of interaction between one or more services and modelling a little application using services.
- The exercises will offer also an example of usage of frameworks and technologies that can be used to create Microservice Architectures.
- For further information about development of microservice systems see the references at the end of the presentation. They also contain different links to other sites and books that deal the argument more in depth.

Level 1: Hands on communication

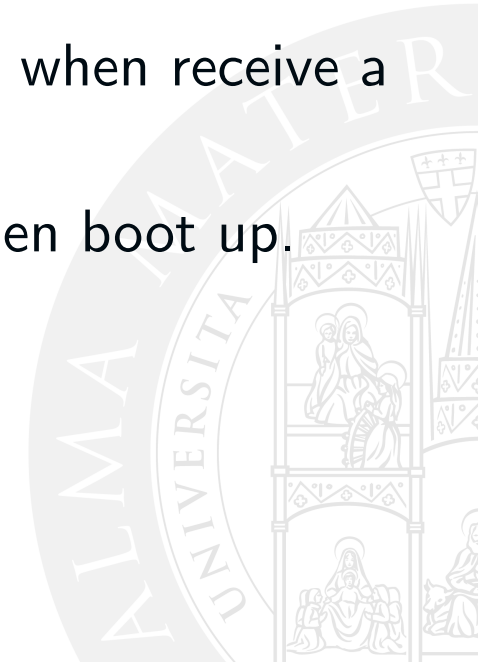
The first exercise shows the usage of AMQP and REST protocol in order to implement a basic communication between two services.

Ping Pong

The goal of the following exercises is to create two services:

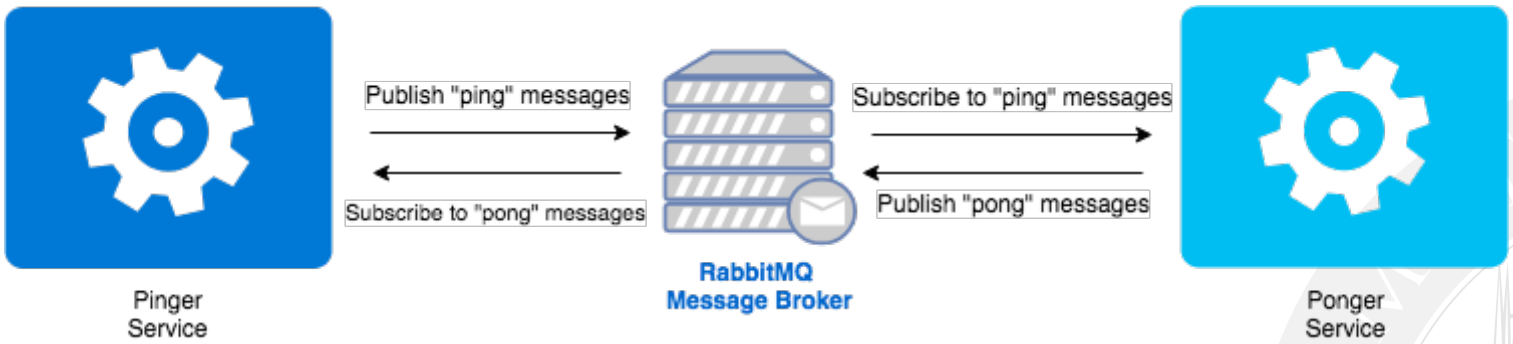
- "Pinger" $\rightarrow$ sends a "ping" message to the "Ponger" when receive a "pong" message
- "Ponger" $\rightarrow$ sends a "pong" message to the "Pinger" when receive a "ping" message

"Pinger" start the exchange sending a "ping" message when boot up.



PingPong: RabbitMQ

- For the first exercise we try to implement the Ping Pong exercise using a Publish-Subscribe architecture with RabbitMQ.



PingPong: RabbitMQ

- RabbitMQ is an open source message broker, it supports multiple protocol among wich AMQP and MQTT.
- This is the list of features directly from the official site [11]

Asynchronous Messaging

Supports multiple messaging protocols, message queuing, delivery acknowledgement, flexible routing to queues, multiple exchange type.

Developer Experience

Deploy with BOSH, Chef, Docker and Puppet. Develop cross-language messaging with favorite programming languages such as: Java, .NET, PHP, Python, JavaScript, Ruby, Go, and many others.

Distributed Deployment

Deploy as clusters for high availability and throughput; federate across multiple availability zones and regions.

Enterprise & Cloud Ready

Pluggable authentication, authorization, supports TLS and LDAP. Lightweight and easy to deploy in public and private clouds.

Tools & Plugins

Diverse array of tools and plugins supporting continuous integration, operational metrics, and integration to other enterprise systems. Flexible plug-in approach for extending RabbitMQ functionality.

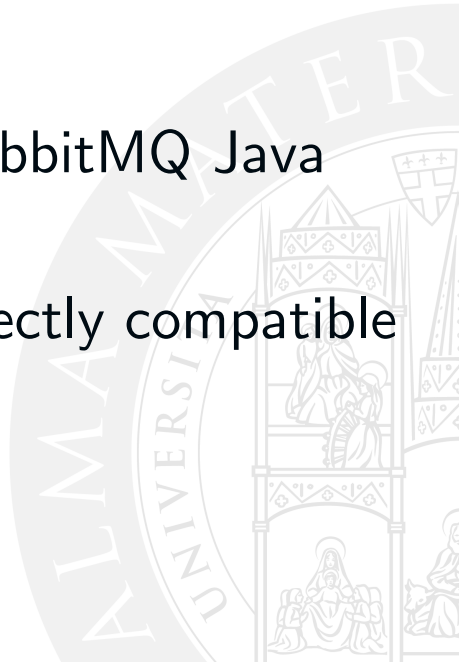
Management & Monitoring

HTTP-API, command line tool, and UI for managing and monitoring RabbitMQ.

[11] https://www.rabbitmq.com/

PingPong: RabbitMQ

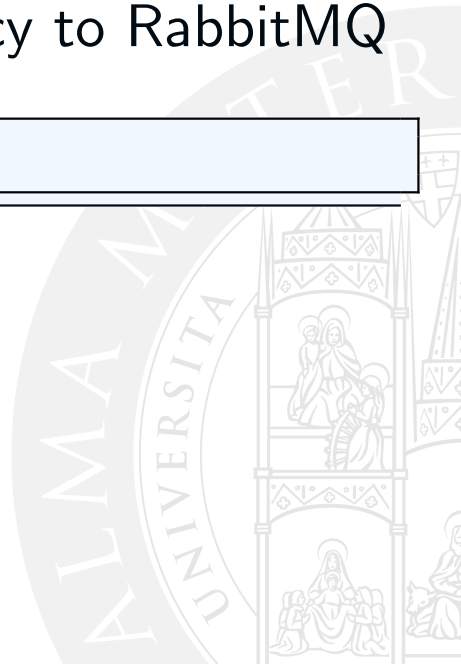
- Before start to coding, It's necessary to install the Message Broker server on your machine
- To install and run it, follow the tutorials on the official website according to your operating system:
  - `https://www.rabbitmq.com/download.html`
- To date it is not available yet an official porting of RabbitMQ Java libraries in Kotlin...
- ...but don't worry! As mentioned before Kotlin is perfectly compatible with Java, so we will use directly Java libraries.



PingPong: RabbitMQ - Project setup

- Now open Itelli-J and create a new Project type: Gradle - Kotlin(JVM)
- Open the `build.gradle` script and add the dependecy to RabbitMQ

```
compile group: 'com.rabbitmq', name: 'amqp-client', version: '5.1.2'
```



PingPong: RabbitMQ - Communication model

Let's take a look at the elementary block of the RabbitMQ Communication model

- A producer is a user application that sends messages.
- Queue is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a queue. A queue is only bound by the host's memory & disk limits, and it's essentially a large message buffer. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue
- A consumer is a user application that receives messages.

12

[12] from `https://www.rabbitmq.com/tutorials/tutorial-one-java.html`

PingPong: RabbitMQ - Communication model [13]

- The core idea in the messaging model in RabbitMQ (AMQP Protocol) is that the producer never sends any messages directly to a queue.
- *Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.*
- Instead, the producer can only send messages to an exchange.

[13] from `https://www.rabbitmq.com/tutorials/tutorial-three-java.html`

PingPong: RabbitMQ - Communication model [14]

- An exchange on one side receives messages from producers and on the other side it pushes them to queues.
- The exchange must knows exactly what to do with a message that it receives.
  - Should it be appended to a particular queue?
  - Should it be appended to many queues?
  - Or should it get discarded?
- The rules for that are defined by the exchange type.

[14] from https://www.rabbitmq.com/tutorials/tutorial-three-java.html

PingPong: RabbitMQ - Communication model

There are 4 different exchange types:

- direct: the routing algorithm behind a direct exchange is a message goes to the queues whose *binding key* exactly matches the *routing key* of the message. (bk = rk) [15] [16]
- headers: a headers exchange is designed for routing on multiple attributes that are more easily expressed as message headers than a routing key. Headers exchanges ignore the routing key attribute. Instead, the attributes used for routing are taken from the headers attribute. A message is considered matching if the value of the header equals the value specified upon binding. [17]

[15] https://www.rabbitmq.com/tutorials/tutorial-four-java.html
[16] https://www.rabbitmq.com/tutorials/amqp-concepts.html#exchange-direct
[17] https://www.rabbitmq.com/tutorials/amqp-concepts.html#exchange-headers

PingPong: RabbitMQ - Communication model

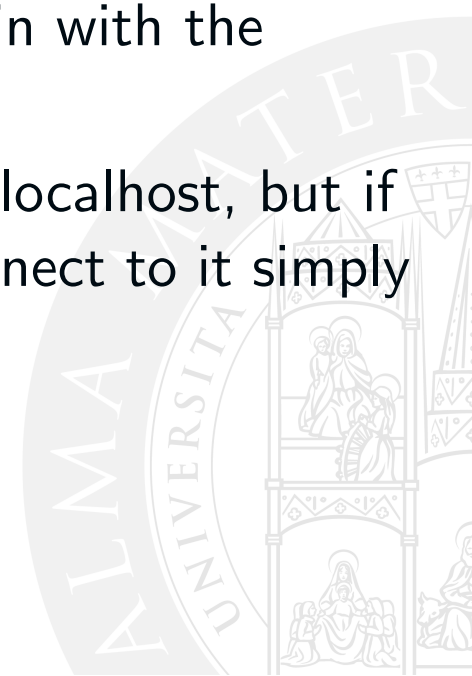
- fanout: in this modality exchange just broadcasts all the messages it receives to all the queues it knows. [18] [19]
- topic: messages sent to a topic exchange can't have an arbitrary *routing key* - they must be a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message. The logic behind the topic exchange is similar to a direct one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys:
  - * (star) can substitute for exactly one word.
  - # (hash) can substitute for zero or more words

  [20] [21]

[18] https://www.rabbitmq.com/tutorials/tutorial-three-java.html
[19] https://www.rabbitmq.com/tutorials/amqp-concepts.html#exchange-fanout
[20] https://www.rabbitmq.com/tutorials/tutorial-five-java.html
[21] https://www.rabbitmq.com/tutorials/amqp-concepts.html#exchange-topic

PingPong: RabbitMQ

- In order to implement the exercise we can use different strategies
- For our exercise we will use one exchange of direct type and filter messages by the routing key.
- In order to learn how to use RabbitMQ API we'll begin with the classic HelloWorld example
- In this example we deploy the message broker on the localhost, but if you want you can deploy it on other machine and connect to it simply specifying its name or its IP address.



PingPong: RabbitMQ - HelloWorldPublisher

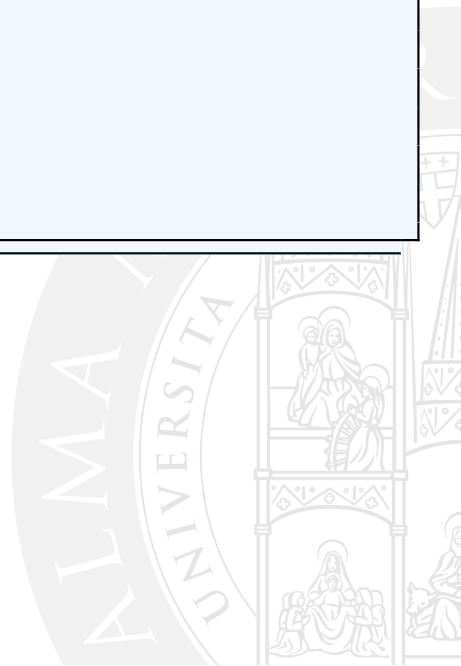
```
class HelloWorldPublisher {
    fun publish() {
        //Establish connection to the Broker
        var factory: ConnectionFactory = ConnectionFactory()
        var connection: Connection
        var channel: Channel

        factory.host = "localhost"
        connection = factory.newConnection()
        channel = connection.createChannel()

        val exchangeName = "HelloExchange"
        //create the exchange
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT)
```

PingPong: RabbitMQ - HelloWorldPublisher

```
        val routingKey = "helloMessage"

        val message: String = "Hello World RabbitMQ"

        //Publish message to the exchange with the defined routing key
        channel.basicPublish(exchangeName, routingKey, null, message.toByteArray(charset(
    "UTF-8")))
        println(" [x] Sent '$message'")

        channel.close();
        connection.close();
    }
}

fun main(argv: Array<String>) {
    val pub = HelloWorldPublisher()
    pub.publish()
}
```

PingPong: RabbitMQ - HelloWorldSubscriber

```
class HelloWorldSubscriber {

    fun subscribe() {
        //Establish connection to the Broker
        var factory: ConnectionFactory = ConnectionFactory()
        var connection: Connection
        var channel: Channel

        factory.host = "localhost"
        connection = factory.newConnection()
        channel = connection.createChannel()
```



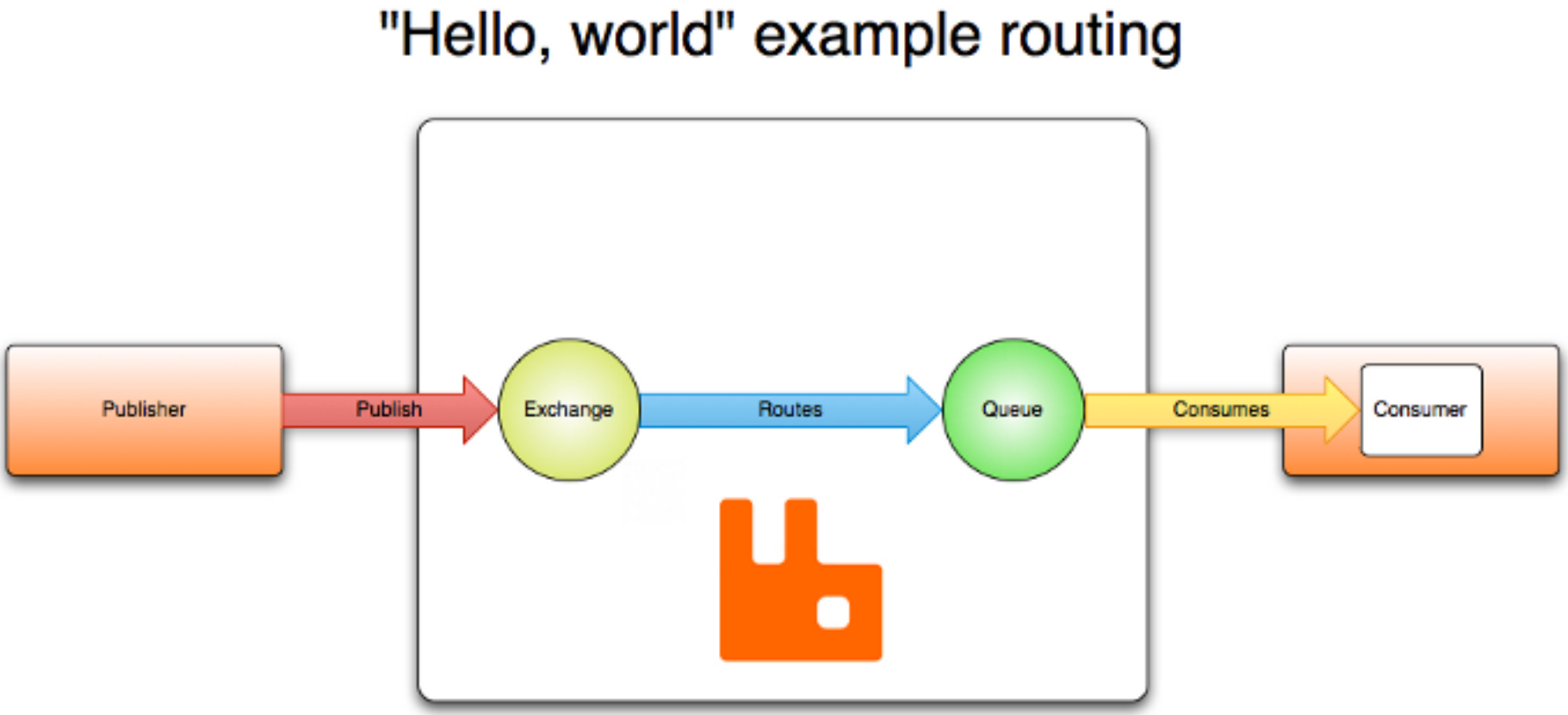
PingPong: RabbitMQ - HelloWorldSubscriber

```
    // create a
    // non-durable (the queue will not survive a broker restart)
    // exclusive (used by only one connection and the queue will be deleted when that
 connection closes)
    // autodelete (queue that has had at least one consumer is deleted when last
consumer unsubscribes)
    // queue with a generated name:
    val queueName: String = channel.queueDeclare().queue

    val exchangeName = "HelloExchange"
    //declare the exchange
    channel.exchangeDeclare(exchangeName, BuiltinExchangeType.DIRECT)

    //bind the queue to the exchange
    //after this the queue will receive from the exchange all message with the
specified routing key
    val routingKey = "helloMessage"
    channel.queueBind(queueName, exchangeName, routingKey)
```

PingPong: RabbitMQ - HelloWorldSubscriber

```
        val helloWorldMessagesConsumer: Consumer = object : DefaultConsumer(channel) {
                @Throws(IOException::class)
                override fun handleDelivery(consumerTag: String,
                                            envelope: Envelope,
                                            properties: AMQP.BasicProperties,
                                            body: ByteArray) {
                    val messageContent = String(body, Charsets.UTF_8)
                    println("Received message with content: " + messageContent)
                }
            }


        //bind a consumer to the queue
        channel.basicConsume(queueName, true, helloWorldMessagesConsumer)

    }

}

fun main(argv: Array<String>) {
    val sub = HelloWorldSubscriber()
    sub.subscribe()
}
```

PingPong: RabbitMQ - HelloWorld



22

[22] image from https://www.rabbitmq.com/tutorials/amqp-concepts.html

PingPong: RabbitMQ - HelloWorld

In order to execute the above example with different process

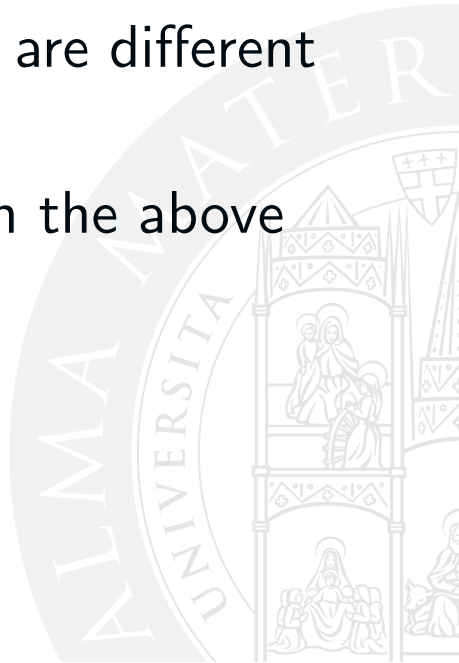
- make sure you have started the RabbitMQ Server
- directly from *IntelliJ* start the main function of the subscriber..
- and after start the main of the publisher

PingPong: RabbitMQ - Suggestions

- As you may have guessed the *Pinger* and the *Ponger* are both publishers and subscribers
- If you want you can create a singleton where is defined all the code used for the connection to the Broker.
- Now it's your turn!

PingPong: RabbitMQ - Solution

- Exercise solution avaiable at:
  `https://github.com/mattigabo/PingPongRabbitMQKotlin`
- Before see the solution try yourself to implement some solution.
- Remember that there isn't a single solution, but there are different correct solutions.
- So don't worry if your implementation is different from the above linked solution.



PingPong: Spring

- As a second version of the same exercise, we now try to implement the Ping Pong exercise using the REST architecture style with the framework Spring.

PingPong: Spring

- The Spring framework is an open source application framework and inversion of control container for the Java platform.
- Spring framework includes several modules that provide a vast range of services.

PingPong: Spring

- This framework is associated with many other projects, which have composite names like Spring Boot, Spring Cloud, Spring Cloud Data Flow, and so on. These projects have been designed to provide additional functionality to the framework. [7]

PingPong: Spring - Features

Spring Boot

- Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production ready applications.

PingPong: Spring - Features

Spring Cloud

- Built directly on Spring Boot's approach to enterprise Java, Spring Cloud simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.

PingPong: Spring - Features

Spring Cloud Data Flow

- Connect the Enterprise to the Internet of Anything-mobile devices, sensors, wearables, automobiles, and more. Spring Cloud Data Flow provides a unified service for creating composable data microservices that address streaming and ETL-based data processing patterns.

PingPong: Spring - RESTful principles

- REST stands for Representational State Transfer. It is an architecture style for designing loosely coupled applications over HTTP, that is often used in the development of web services.
- REST does not enforce any rule regarding how it should be implemented at lower level, it just put high level design guidelines and leave you to think of your own implementation.

PingPong: Spring - RESTful principles

REST defines 6 architectural constraints which make any web service a true RESTful API [6]

Principles

- Uniform interface
- Client–server
- Stateless
- Cacheable
- Layered system
- Code on demand (optional)

PingPong: Spring - RESTful principles

Uniform interface

- Planning of RESTful APIs you must decide APIs interface for resources inside the system which are exposed to API consumers and follow religiously. A resource in the system should have only one logical URI and that should provide a way to fetch related or additional data.
- Any single resource should not be too large and contain each and everything in its representation.
- The resource representations across system should follow certain guidelines such as naming conventions, link formats or data format (xml or/and json).
- All resources should be accessible through a common approach such as HTTP GET and similarly modified using a consistent approach. [a]

[a] https://restfulapi.net/rest-architectural-constraints/#uniform-interface

PingPong: Spring - RESTful principles

Client–server

- Client application and server application must be able to evolve separately without any dependency on each other (a client should know only resource URIs).
- Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered. [a]

[a] https://restfulapi.net/rest-architectural-constraints/#client-server

PingPong: Spring - RESTful principles

Stateless

- Make all client-server interaction stateless. Server will not store anything about latest HTTP request client made. It will treat each and every request as new (no session, no history, etc).
- If client application needs to be a stateful application for the end user, where user logs in once and do other authorized operations thereafter, then each request from the client should contain all the information necessary to service the request, including authentication and authorization details.
- No client context shall be stored on the server between requests. The client is responsible for managing the state of the application. [a]

[a] https://restfulapi.net/rest-architectural-constraints/#stateless

PingPong: Spring - RESTful principles

Cacheable

- In REST, caching shall be applied to resources when applicable and then these resources must declare themselves cacheable. Caching can be implemented on the server or client side.
- Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance. [a]

[a] https://restfulapi.net/rest-architectural-constraints/#cacheable

PingPong: Spring - RESTful principles

Layered system

- REST allows to use a layered system architecture where you deploy the APIs on server A, and store data on server B and authenticate requests in Server C, for example.
- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. [a]

[a] https://restfulapi.net/rest-architectural-constraints/#layered-system

PingPong: Spring - RESTful principles

Code on demand (optional)

- When you need to, you are free to return executable code to support a part of your application (e.g. clients may call your API to get a UI widget rendering code). [a]

[a] https://restfulapi.net/rest-architectural-constraints/#code-on-demand

PingPong: Spring - Operations

Use HTTP methods to map CRUD (create, retrieve, update, delete) operations to HTTP requests. [8]

HTTP methods

- **GET**: used to retrieve information. GET requests must be safe and idempotent, meaning regardless of how many times it repeats with the same parameters, the results are the same.
  E.g. Retrieve an address with an ID of 1: `GET /addresses/1`
- **POST**: request that the resource at the URI do something with the provided entity; often POST is used to create a new entity.
  E.g. Create a new address: `POST /addresses`

PingPong: Spring - Operations

HTTP methods

- **PUT**: store an entity at a URI. PUT can create a new entity or update an existing one. A PUT request is idempotent (idempotency is the main difference between the expectations of PUT versus a POST request).
  E.g. Modify the address with an ID of 1: `PUT /addresses/1`
  Note: PUT replaces an existing entity.
- **PATCH**: update only the specified fields of an entity at a URI. A PATCH request is neither safe nor idempotent (that's because a PATCH operation cannot ensure the entire resource has been updated). E.g. `PATCH /addresses/1`
- **DELETE**: request that a resource be removed; however, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.
  E.g. Delete an address with an ID of 1: `DELETE /addresses/1`

PingPong: Spring - Project setup [1]

- First we need to create a Spring Boot application, which can be done in a number of ways.
- We will use Gradle build system since this is the most popular in Kotlin ecosystem.

Using the "*Initializr*" website

- Go to `https://start.spring.io` and choose Kotlin language.
- Then choose Gradle build system and fill in the remaining fields indicating the names to be used and the dependencies to be included.

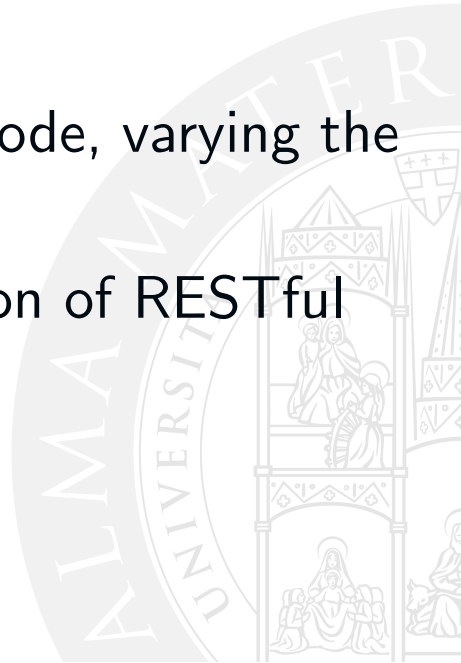
PingPong: Spring - Project setup [1]

Using IntelliJ IDEA

- Spring Initializr is also integrated in IntelliJ IDEA and allows you to create and import a new project without having to leave the IDE for the command-line or the web UI.
- To access the wizard, go to `File | New | Project`, and select `Spring Initializr`.
- Follow the steps of the wizard filling the fields indicating the names to be used and the dependencies to be included.

PingPong: Spring

- As for the RabbitMQ version it is possible to use different modes for the resolution of the exercise.
- Once the application has been started, the browser or API testing software such as Postman can be used to verify the correctness of the operations.
- For simplicity we will use localhost as a deployment mode, varying the port number to handle communications.
- The following is an example of a simple implementation of RESTful APIs with Spring.

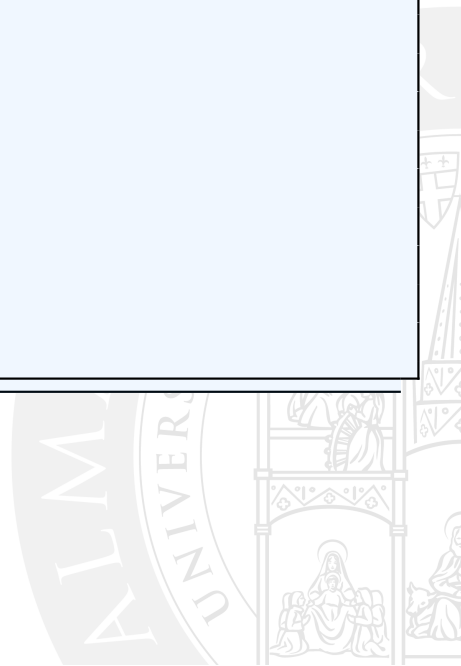


PingPong: Spring - HelloWorldController

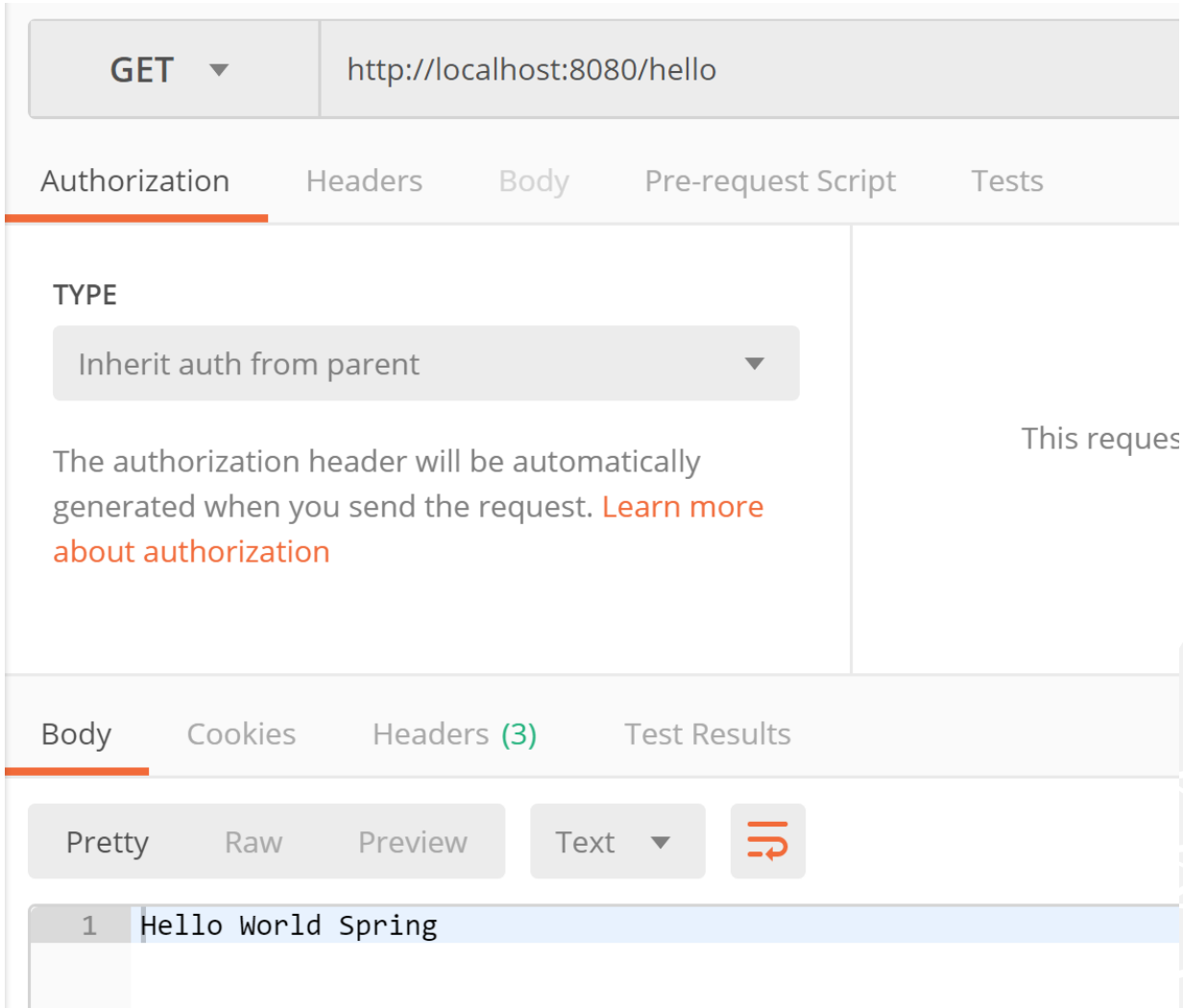
```
/**
 * The controller of the application.
 */
@RestController
class HelloWorldController {

    @RequestMapping("/hello")
    fun ping() = "Hello World Spring"
}
```

PingPong: Spring - HelloWorldApplication

```
/**
 * The main entry point to the application.
 */
@EnableAutoConfiguration
@Configuration
internal class HelloWorldApplication {
    @Bean
    fun controller() = HelloWorldController()
}

/**
 * Run the application
 * @param args The command line arguments
 */
fun main(args: Array<String>) {
    SpringApplication.run(HelloWorldApplication::class.java, *args)
}
```



PingPong: Spring - Verification



PingPong: Spring - Bootstrap

- You can specify a different port number than the default port number (8080) in the `application.properties` resource file.
- It could be useful to create separate modules for better management of Spring's infrastructure
- Now try it yourself!
- Exercise solution available at:
  `https://github.com/eliadp/PingPongSpringKotlin`

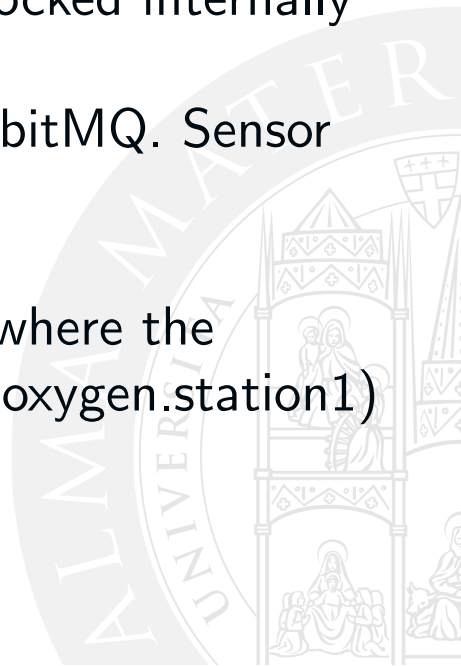
Mix two communication technology

- After we learned how to implement a basic communication between two services, now we'll try to create an application with more services and put some more logic in each of them.
- For the next exercise we will also use both of the previous technology in order to show that is not necessary to use the same technology between all services.
- This exercise can be done in group (max 5 people)

Sensor Manager

The goal of this exercise is to create a little application that manage different sensor stations. We will have two type of service:

- The first type is the *SensorStation*. This service will represents service that manages a set of sensors mounted in a room.
  - The service reads data from sensors (sensor can be mocked internally at the service with a random generator)
  - It also sends data to the *SensorManager* through RabbitMQ. Sensor value will be sent every 5 seconds.
  - Every station have 2 sensors: (Oxygen, Humidity)
  - For this example we will use the topic exchange type where the routing-key is composed by sensor.station-name (e.g. oxygen.station1)



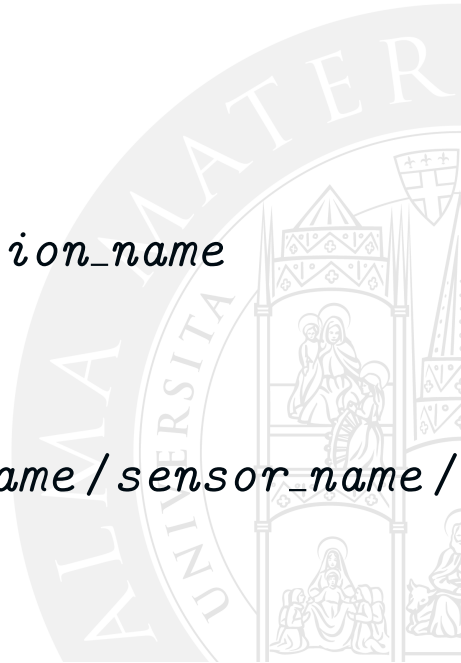
Sensor Manager

- The second type of service will be the *SensorManager*. This service have two purpose
  - To collect data from the registered *SensorStation*s through *RabbitMQ*
  - The service keeps in memory only 10 last values of each sensors (you can choose what data structure use e.g list, queue, set ecc..)
  - To offers a *REST* interface that will be used by a generic User application.

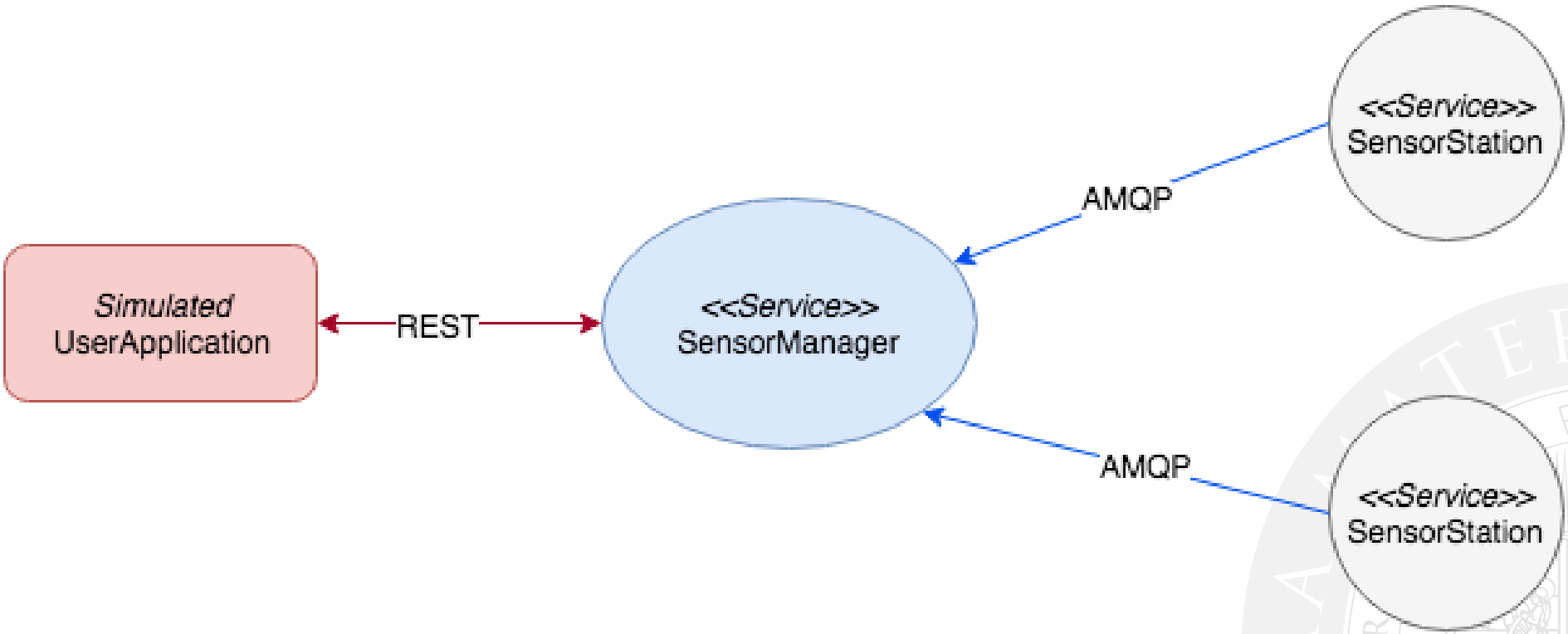
Sensor Manager

The REST interface will offers the following APIs:

- Add a new station, when this API is used the station manager add a listener to the value sent from the new station. The name of the station will be sent in the body of the message
  - **POST**: `http://{hostname}:{port}/addstation`
- Get all station info (name + current sensors values)
  - **GET**: `http://{hostname}:{port}/stations/`
- Get info of a specific station
  - **GET**: `http://{hostname}:{port}/stations/`*`station_name`*
- Get the sensor values of a station
  - **GET**: `http://{hostname}:{port}/stations/`*`station_name`*`/`*`sensor_name`*`/`



Sensor Manager



Sensor Manager

- The following code shows a User app simulation of interaction with the REST interface of the *SensorManager*, with the purpose to test the whole flow of the application.

```
/**
 * This is a basic function that simulate a external user application that interact with
     the SensorManager REST API
 */
fun main(args: Array<String>) {
    //Before start this process boot all the other services

    val SENSOR_MANAGER_ROUTE = "http://localhost:8080/"

    println("Adding the station: Station1...");

    var request = Fuel.post(SENSOR_MANAGER_ROUTE + "addstation").body("{ \"name\" : \"
     Station1\" }")
    request.headers["Content-Type"] = "application/json";
    request.response();

    Thread.sleep(6000)
```

Sensor Manager

```
println("Requesting all station data...")

request = Fuel.get(SENSOR_MANAGER_ROUTE + "stations")
request.headers["Content-Type"] = "application/json";
var response = request.response();

println("RESPONSE...")
println(response.second);

println("Requesting oxygen sensor data of Station1...")

request = Fuel.get(SENSOR_MANAGER_ROUTE + "stations/Station1/" + SensorTypes.OXYGEN)
request.headers["Content-Type"] = "application/json";
response = request.response();

println("RESPONSE...")
println(response.second);

println("Adding the station: Station2...");

request = Fuel.post(SENSOR_MANAGER_ROUTE + "addstation").body("{ \"name\" : \"
 Station2\" }")
request.headers["Content-Type"] = "application/json";
request.response();

Thread.sleep(6000)
```

Sensor Manager

```
    println("Requesting humidity sensor data of Station1...")

    request = Fuel.get(SENSOR_MANAGER_ROUTE + "stations/Station1/" + SensorTypes.HUMIDITY
     )
    request.headers["Content-Type"] = "application/json";
    response = request.response();

    println("RESPONSE...")
    println(response.second);



    println("Requesting oxygen sensor data of Station2...")

    request = Fuel.get(SENSOR_MANAGER_ROUTE + "stations/Station2/" + SensorTypes.OXYGEN)
    request.headers["Content-Type"] = "application/json";
    response = request.response();

    println("RESPONSE...")
    println(response.second);
}
```

Sensor Manager

Solution avaiable at:
`https://github.com/mattigabo/SensorManagerExample`

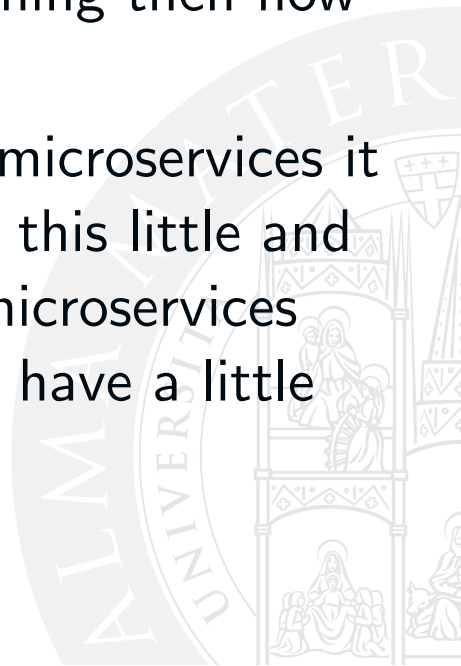
Sensor Manager

- In this example has been shown the possibility to interact with services through different protocols. The User App interaction has been mocked but it can be evolved in another real service or web app that interact directly through REST API.
- In order to implement the code that interacts with the RabbitMQ server of the *SensorManager* is possible to choose different strategies. In the linked solution above, the client of the RabbitMQ server was manually developed using the Java library, but considering that to develop the REST interface was used the Spring framework is possible to use directly the *Spring* library that implements an AMQP client [23].

[23] http://spring.io/projects/spring-amqp

A taste of Microrvice with Kotlin - Conclusions

- In this lesson we started with an introduction to microservices in order to understand in a more clear way: *What are they? When to use them? What's the route to take in order to implement them?*.
- Afterwards we introduced Kotlin, starting to explain the basics of the language itself to introduce its operating mode, explaining then how to use it materially in a project.
- In conclusion, like discussed above, in order to create microservices it is necessary to know a lot about CD and CI. But with this little and easy example it has been possible to understand the microservices characteristic of *smart endpoints and dumb pipes* and have a little taste of microservices implementation with kotlin.



References I

[1] Building web applications with Spring Boot and Kotlin.
https://spring.io/guides/tutorials/spring-boot-kotlin/.
[Online].

[2] Getting Started with IntelliJ IDEA.
https://kotlinlang.org/docs/tutorials/getting-started.html.
[Online].

[3] Kotlin JavaScript Overview.
https://kotlinlang.org/docs/reference/js-overview.html.
[Online].

[4] Kotlin (programming language).
https://en.wikipedia.org/wiki/Kotlin_(programming_language).
[Online].

[5] Kotlin/Native for Native.
https://kotlinlang.org/docs/reference/native-overview.html.
[Online].

[6] REST Architectural Constraints.
https://restfulapi.net/rest-architectural-constraints/.
[Online].

References II

[7] Spring: the source for modern java.
https://spring.io/.
[Online].

[8] Understanding REST.
https://spring.io/understanding/REST.
[Online].

[9] Using Kotlin for Android Development.
https://kotlinlang.org/docs/reference/android-overview.html.
[Online].

[10] Using Kotlin for Server-side Development.
https://kotlinlang.org/docs/reference/server-overview.html.
[Online].

[11] Working with Build Tools.
https://kotlinlang.org/docs/tutorials/build-tools.html.
[Online].

[12] Working with the Command Line Compiler.
https://kotlinlang.org/docs/tutorials/command-line.html.
[Online].

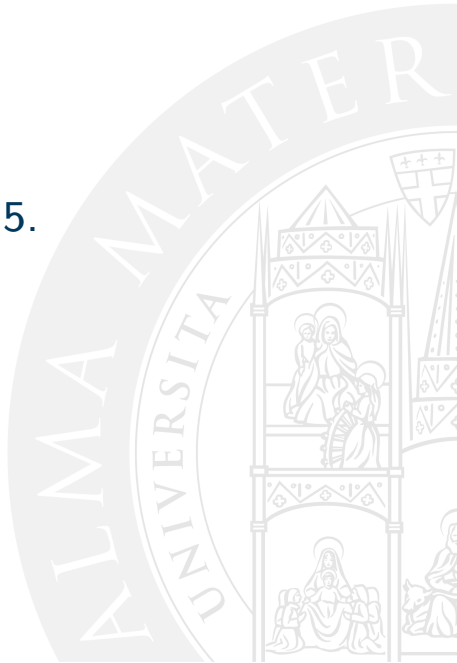
References III

[13] Martin Fowler.
ServiceOrientedAmbiguity.
https://www.martinfowler.com/bliki/ServiceOrientedAmbiguity.html, 2005.
[Online; accessed 15-Semptember-2018].

[14] Martin Fowler.
MicroservicePrerequisites.
https://martinfowler.com/bliki/MicroservicePrerequisites.html, 2014.
[Online; accessed 15-Semptember-2018].

[15] Martin Fowler.
MicroservicePremium.
https://martinfowler.com/bliki/MicroservicePremium.html, 2015.
[Online; accessed 15-Semptember-2018].

[16] Martin Fowler.
Microservices Resource Guide.
https://martinfowler.com/microservices/, 2015.
[Online; accessed 8-September-2018].



References IV

[17] Martin Fowler.
MonolithFirst.
https://martinfowler.com/bliki/MonolithFirst.html, 2015.
[Online; accessed 11-September-2018].

[18] Martin Fowler.
MonolithFirst.
https://martinfowler.com/bliki/MonolithFirst.html, 2015.
[Online; accessed 15-Semptember-2018].

[19] Jim Gray.
A Conversation with Werner Vogels: Learning from the Amazon technology platform.
https://queue.acm.org/detail.cfm?id=1142065, 2006.
[Online; accessed 16-Semptember-2018].

[20] James Lewis Martin Fowler.
Microservices.
https://www.martinfowler.com/articles/microservices.html, 2014.
[Online; accessed 9-September-2018].

References V

[21] Matt McLarty.
Learn from SOA: 5 lessons for the microservices era.
https://www.infoworld.com/article/3080611/, 2016.
[Online; accessed 15-Semptember-2018].

[22] R. Meshenberg.
GOTO 2016 - Microservices at Netflix scale: Principles, Tradeoffs Lessons Learned.
https://www.youtube.com/watch?v=57UK46qfBLY, 2016.
[Online; accessed 13-June-2018].

[23] Sam Newman.
Sam Newman: "Deploying and testing microservices".
https://www.youtube.com/watch?v=FotoHYyY8Bo, 2014.
[Online; accessed 26-April-2018].

[24] Andrea Omicini.
Standard Services for Distributed Systems: Web Services.
http://campus.unibo.it/259765/1/T5%20-%20Standard%20Services%20for%20Distributed%20Systems%20-%20Web%20Services.pdf, 2016.
Slide of the university course of Distributed Systems.

References VI

[25] Pivotal.
DevOps.
https://pivotal.io/devops, 2018.
[Online; accessed 16-Semptember-2018].

[26] Alessandro Ricci.
Engineering Distributed Systems - Service-Oriented Architectures, 2017.
Slide of the university course of Concurrent and Distributed programming: module-5.2 - course url http://www.ingegneriarchitettura.unibo.it/it/corsi/insegnamenti/insegnamento/2016/412598.

[27] Rouan Wilsenach.
DevOpsCulture.
https://martinfowler.com/bliki/DevOpsCulture.html, 2015.
[Online; accessed 16-Semptember-2018].

A taste of Microservices with Kotlin

Matteo Gabellini {matteo.gabellini2@studio.unibo.it}
Elia Di Pasquale {elia.dipasquale@studio.unibo.it}

Final Course Presentation @ Distributed Systems Course, A.Y. 17/18
Dipartimento di Informatica, Scienza e Ingegneria—Università di Bologna

October 11, 2018