Politecnico di Torino

Web Applications II

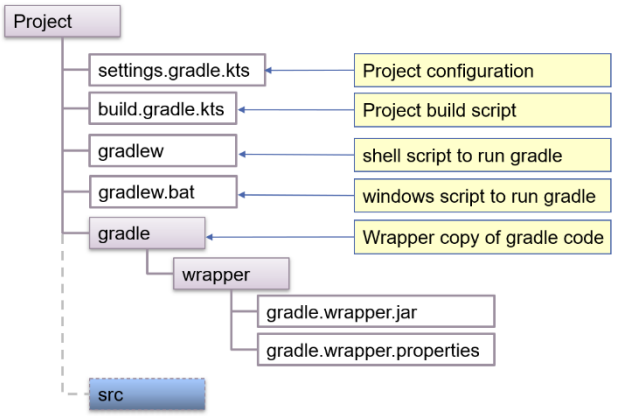
Slides recap from the course of Prof. Giovanni Malnati

A.A. 2021/22

Web Applications II

The Gradle build tool

Gradle is a build and automation tool implemented in Java and open source. It is used to build apps in different ecosystems (Java, Kotlin, C, Android). It is a **convention-based** tool that makes assumptions about the locations of source files, test files and resources. It supports **multi-project** builds, and it is customizable via **scripts**. It supports dependency management by automatically downloading needed modules and libraries from listed repositories. The entire build process is described in one or more script files.



The **project script file** contains a description of the build process in terms of **plugins** (define which tasks should be executed), **repositories** (web sites from which libraries can be downloaded) and **dependencies** (list of libraries with their version that are part of the project) together with **further** configuration information.

A project is the description of how a given software artifact is built. Sometimes, more than one artifact needs to be built: in this case a build can have more than one project. A project has one or more tasks and **plugins extend projects**. The project script file (*build.gradle.kts*) specifies all the tasks, while an optional settings file (*settings.gradle.kts*) provides build level information (i.e., pertaining all projects).

The build system is triggered by a set of commands, each targeted to a specific goal. The set of all available task is accessible using the "tasks" task name. Each task can depend on zero ore more other tasks: if Gradle is requested to execute a given task, all the tasks it depends on are executed first.

Plugins are software artifacts that extends project's capabilities that are Introduced by the plugins block. Each plugin can introduce custom tasks, set-up dependencies, trigger specific behavior.

By default, Gradle operates on conventions, so it expects source files to be located in folder "./src/main", test files in "./src/test". It is possible to deviate from these conventions by specifying proper entries in the project script file.

By applying the 'application' plugin, the 'run' task becomes available and it allows to launch the artefact that has been built. The application plugin extends the java plugin, and the application block is used to specify the name of the main class.

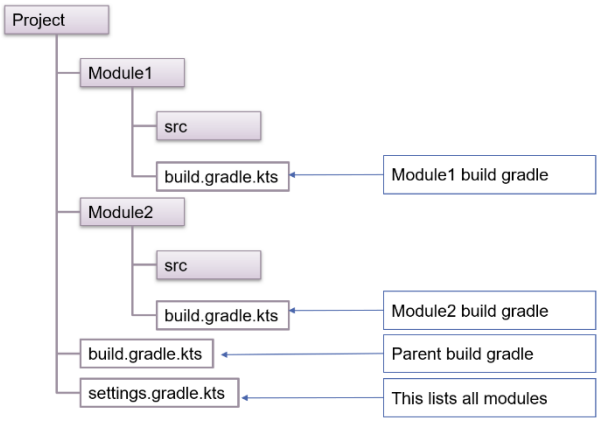
When the java plugin (or any other plugin that extends it) is applied, the java block can be used to configure the compilation and runtime environments.

The kotlin plugin must be applied, specifying the language version. The kotlin block can then be used to specify alternates folder for source and test files, if needed. To define the proper jvm target version, the compileKotlin and compileTestKotlin tasks can be overridden.

A project often requires libraries or other externally provided resources. These are collectively named **dependencies**. Being out of control of the project, care should be taken to identify the correct **version** of the artefact and **location** where it can be downloaded from. Gradle allows to specify this information in different blocks: the **dependencies** block lists the artefacts needed by the project, together with their version while the **repositories** block list the web sites where they should be looked up. Dependencies can be located: In other projects, in the local file system, in Maven repositories or in Ivy repositories. Dependencies are needed:

at **compile time** and all though the artefact's life; at **compile time only** (being provided form other sources at runtime); at **runtime only;** at **test compilation time;** at **test execution time**.

Given a set of locations where dependencies may be download from, two more pieces of information are needed: what exact piece of dependency is needed and in which phase of the build process it is needed. Dependencies are named out of three pieces of information: the **group name**, the **artifact name,** and the **artifact version.**



Most projects start with a single codebase and build script (build.gradle.kts) but, as they grow, they are often split into several interdependent modules to improve readability and maintainability.

A module is a sub-project that targets the creation of an individual software artifact (e.g., a jar file): modules have their own codebases, and they can have different dependencies.

Kotlin Annotations

Annotations are a special kind of class used to define custom metadata and bind them to other elements (declarations, expressions, functions) of the source code. Frameworks and processing tools use this feature to express configuration or to instrument code. Annotations are introduced by the @ character.

The Kotlin language defines a **syntax to declare an annotation** class, a **syntax to annotate** source code, **reflection APIs** to access annotations bound to a given class, function, or expression, a **standard way to store annotations** in bytecode and a **set of platform specific mechanisms** to let the compiler access and process code annotations.

Annotations can be used to:

- **Inform** the compiler about programmer intents and knowledge of possible misbehaviour (@Suppress, @UseExperimental, …)
- **Generate** extra code/data at compile/deploy time: using an annotation processor plugin, the compiler can be augmented with some extra-code that further processes the original content
- **Affect program execution**: using reflection, code can be inspected at run-time, detecting annotations, and behaving accordingly

Annotation classes implicitly derive from the Any class and the empty Annotation interface. Annotation classes can have neither subtypes nor supertypes, other than Any and **no code** can be placed inside an annotation (act as a container for their properties). Properties are **immutable** and **non-nullable**.

Annotation classes can be annotated with some special **meta-annotations**. These provide specify how the compiler should manage the declared annotation.

Annotations can be processed:

- At **compile time**, by using the kapt compiler plugin
- By **inspecting compiled classes**, using a tool capable to process bytecode
- At **runtime**, via reflection

In the precompile phase the Java compiler, that backs up the Kotlin compilation process, is capable of scanning annotations and can delegate their processing to a suitable plug-in. This can derive a new source file from the existing one, that will be further inspected and then compiled.

The $javax.annotation.processing.AbstractProcessor$ class can be extended to supply the custom annotation processing code. All annotation processors must extend the $AbstractProcessor$ class and $process(\dots)$ is the core method. It must return true to state that annotations have been properly processed. Returning false, allows other processors to deal with the current annotations.

The annotation processing happens in rounds. In each round, only some annotation may be considered: these are passed in the first argument. The second argument contains all the elements of the source file that contain one of the currently processed annotations.

To be available to the compiler, a Processor need to be included in a Jar file to be supplied at compile time. The Jar file must contain a $META-INF/services/javax.annotation.processing.Processor$ text file listing all the processor class names that should be considered by the compilation process. Moreover, the kapt plugin should be declared as part of the Gradle configuration.

If a given annotation has **runtime retention**, it can be processed via reflection. Given an object obj, it is possible to get a reference to its class via obj.javaClass. The Class object contains a lot of information about it, among the others the **set of annotations it was labelled with.** Annotation's properties can be accessed and used to guide computation.

Annotations provide a standard way to decorate code with meta-data and offer a powerful tool to support automatic data-structure and programming pattern generation. Annotations do not have any direct consequence on code semantics, but they allow to largely reduce the amount of boiler-plate code needed to implement repetitive patterns.

Architectures of Web Applications

Distributed systems

A distributed system is a system composed of two or more interconnected sub-systems that co-operate to achieve a common goal. The interconnection is supported by a network while communication is based on message passing.

The driving forces are:

- **Scalability** – can deal with larger problems or larger number of users
- **High Availability** – can leverage redundancy to support partial failure
- **Resource sharing** – make available (location-, capacity-, …) constrained resources to larger audiences

The **challenges of distributed systems** are:

- **Complexity** in design, implementation, testing, deployment, operations mainly due to *concurrency* (subsystems work in parallel), complex interactions, time dependence, …
- **Partial failures**: a single process is either working or broken. One or more subsystems can slow down, stop working or network may be partitioned, preventing communication.
- **Lack of a global clock:** coordination needs to be based on message exchange (which introduces a delay between the sender and the receiver)
- **Security:** it is possible to tamper with the system by injecting, removing, replacing, duplicating messages across the network. Extra complexity must be introduced to deal with this issue.

Reliable distributed systems

To overcome the challenges, and benefit from the opportunities provided by the distributed paradigm, a **proper architecture** must be chosen as well as be sure that each of underlying subsystem support a bunch of properties that make them **robust** and **reliable** on their own. Moreover, care must be taken not to fall in one of the well-known fallacies of distributed systems that have been distilled along the years by the developer community.

Software architecture

The software architecture of a given product/service defines:

- the **fundamental structures** of a software system
- the **software elements** they are made by
- the **relationships** among them
- the **properties elements** and **relationships** have

This high-level picture helps making the major design choices and evaluate their implication by choosing the proper compromise in terms of both functional and non-functional features the system must exhibit.

Reliable application properties

- **Idempotence:** an application will not duplicate the effect if it receives a duplicate message. Helps dealing with unreliable communications
- **Immutability:** no data is ever overwritten, nor it is deleted: information increases *monotonically*. It helps keeping a record of what happened in a system
- **Location independence:** the behaviour of an application does not depend on where it is deployed. It helps composing the overall system out of single parts
- **Versioning:** an application will tolerate changes over time of the data it manages and of the behaviour it exerts. It helps dealing with change.

Fallacies of distributed systems (by Peter Deutsch, 1994):

- The network is **reliable**
- **Latency** is zero
- **Bandwidth** is infinite
- The network is **secure**
- **Topology** does not change
- There is one **administrator**
- **Transport** cost is zero
- The network is **homogeneous**

Embracing failures

System must be designed knowing that something can always go wrong, so all problems must be detected and failing services should be restarted. Moreover, clients should not send requests to failed component instances and when problems are corrected, request must be resumed: this implies that client must be **resilient**. Dealing with these problems manually is not feasible, if the number of services increases. **Orchestrator** platforms like Kubernetes are used to this purpose.

Network unreliability

API calls are not like a function call, because a function call will either succeed (returning a result) or fail (throwing an exception or returning an error code). In a network communication, the request may not reach the recipient (thus never being processed), or the response **can fail** to reach back the invoker (preventing the result of the remote computation from being known to the caller). **Timeouts** need to be introduced to deal with this kind of failures, but they only allow to detect the problem.

Implementing **idempotent APIs** helps addressing the problem:

- **REST based architectures** tend to be error-prone since they allow POST messages which need not to be idempotent by design.
- **POST methods** are typically used to create new resources in the recipient, returning the identifier of those resources: if such ID is generated on the server-side, **idempotence is violated**.

Topology changes

People tend to assume that resources (e.g., a centralized DBMS) are stable and there is no need to fail over to a backup one. **Location independence**, on the other side, assumes that if the same message is sent to multiple instances of the same application, they must all produce the same effect.

DBMSs generating auto-increment IDs tend to break this rule: the generated ID is dependent on the location where it is produced. Again, client-generated IDs (UUIDs) and content-addressed storage (hash-based IDs) helps solving with this problem.

Network inhomogeneity

Since systems evolve across time, it is possible that **multiple versions of the application exist at different locations**, so software artifacts (code, data, APIs) need to be explicitly **versioned**. CVSs make it simple to manage code versions together with CI and CD pipelines. Versioning data is more complex since it can entail handling different DB representations. Versioning APIs is the most challenging one: while several options exist, no one is perfect.

Web applications

Web applications are a kind of distributed systems leveraging on **ubiquitous standard web technologies** (the HTTP(S) protocol, browsers) to reach the widest audience and limit deployment costs to the server-side only. This provides a large opportunity and a constraint at the same time: since it prescribes part of the design choices, forcing to adopt existing tools, losing some flexibility.

Web applications support users in performing business tasks according to some specific business logic. Their distributed nature automatically provides **concurrent access**, raising the complexity of the system. **Business logic** specifies how the information managed by the application is **transformed** and/or **computed** (e.g., business logic determines how a tax total is calculated from invoice line item) and specifies how it is **routed** to people or software systems (workflow).

Web application design

Web application design is a broad process spanning across many different domains and covering multiple dimensions. It is aimed at defining the **purpose** and **behaviour of the system** and making suitable choices about its **architecture** and **implementation**.

Design can be split into:

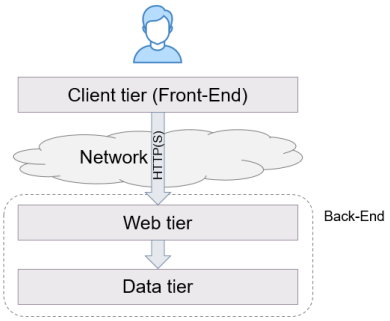
- **High level design** – where a suitable architecture is chosen, and major system components are identified
- **Module level design** – where each single component is defined, pointing out its functionalities, constraints, and implementation-level details

Software architectures

Software architectures describe the major components of a system, their relationships, and how they interact with each other. A **given architecture** provides an abstraction to manage the system complexity and establish communication and coordination among components. It helps define a solution to meet all the functional, technical, and operational requirements by trying to **balance diverging forces** like cost, time-to-market, performance, maintainability, overall quality, and security.

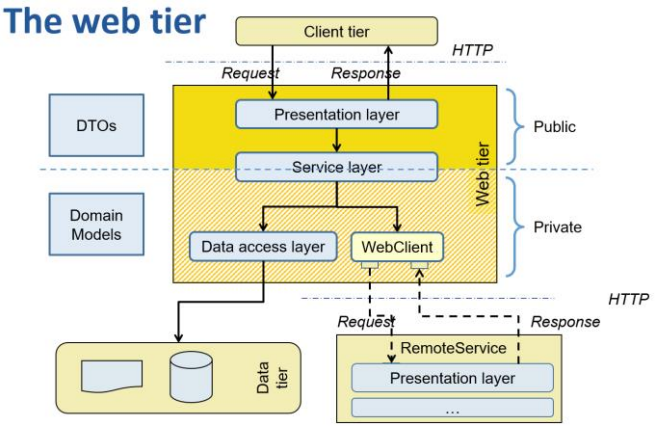
Web application architecture

The **client tier** is usually hosted in a web browser. It offers a generic, sandboxed, execution environment suitable to display the graphical user interface (GUI) and providing easy access to the remote system via HTTP. Modern browsers typically offer very high performances both in computing and in presenting media content (2D and 3D graphics, audio, video), as well as in network communications



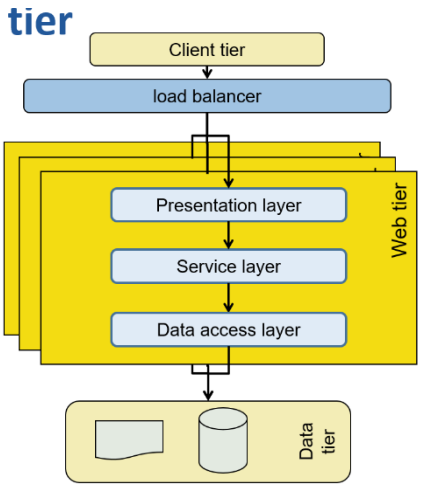
In some cases, the client tier may be implemented as a full-fledged application thus allowing to overcome the limitations placed by the browser sandbox.

The **server tier** is often built using a **layered approach**, as well. Each layer only interacts with the adjacent ones, limiting mutual dependencies and specifying contact points in term of software interfaces. Such an approach boosts code modularity and separation of concerns among modules. Layers can be part of a single, monolithic, process or can be implemented as separate (micro-)services.



Each layer is responsible of a very specific task and relies on the layer(s) immediately below, for further processing. The contact point between two adjacent layers is usually specified via a **software interface**. Since HTTP is stateless, the whole web tier can – in principle – be replicated in many different instances, to scale horizontally. A **load balancer** can be used to evenly spread requests among all available processes.

The **presentation layer** is in charge to process all **incoming requests** and produce suitable **responses**. It is compliant to the HTTP standard, so requests have verbs (GET/POST/PUT/DELETE), URLs, headers and (sometimes) a body.



Requests are coming **concurrently** from **multiple clients**: session IDs may be used to track and relate requests coming from the same client. Arrangements need to be taken if multiple instances of web tier are present.

Generated responses can consist of full pages or be just data structures presented on the client side by some kind of presentation logic (SPA – Single Page Application).

The **service layer** implements the application business logic. It defines the logical contract between the user and the application, but it has none of its side effects, which are delegated to the underlying layer.

Elementary operations defined by application logic maps 1:1 to the methods it provides. They coordinate the execution, in a transactional way, of the various data manipulations defined by the business logic. Since the operations need to be atomic (because of the transaction), the service layer is often **state-less**.

The service layer manipulates **Data Transfer Objects** (DTOs): they contain the **public representation** of the concepts coming from the application domain. DTOs will be mapped to the underlaying representation defined by the data tier (i.e., **Domain models**) by suitable functions (object mappers).

Care must be taken not to leak any information pertaining to the data tier out of the service layer: this makes the layer testable and mockable.

Domain models represent domain concepts in a way suitable for the chosen data tier. According to it, they can be named **entities** (when using RDBMS), **documents** (in some noSQL DBMS), or **nodes** and **edges** (in case of graph DBMS).

The implementation of this representation, being it private, may undergo relevant tuning actions, to improve the performance of the system while guaranteeing that the business rules are preserved.

The **data access layer** persists and retrieves entities or documents into/from the data tier:

- It is responsible to **issue basic commands** to the data layer (e.g., SQL statements or other kinds of DBMS languages)
- Its **implementation** is often **automatically derived** from some high-level description of the task to be carried out (basic CRUD operations, custom searches, …) by specific frameworks (ORM or ODM)
- It usually assumes that manipulated entities are labelled with a unique ID

The **data tier** is typically based on one or more DBMS in charge of persisting the application data and guaranteeing its consistency with respect to the model and business constraints. Different kinds of DBMS may be used: relational (SQL based), document-, graph-based. Sometimes, files and file systems may be part of the data tier: this may introduce some rigidity in managing the physical deployment of the tier.

Software components

Software components are high level elements which provide a well-defined functionality/behaviour:

- DBMSs
- Message brokers
- Cache servers
- Load balancers
- Backend application servers
- User interfaces

Each component may exist as a stand-alone process or be provided as a library to be embedded into another component. The number of processes to be deployed and monitored may grow fast and overwhelm operations.

Web application models

Some web applications models are:

- **One application server, one database** also known as the **monolith**. This is the simplest and less reliable model
- **Multiple application servers, one database:** more reliable and potentially scalable, this model requires a stateless design of the server application
- **Multiple application servers, multiple databases:** most reliable and complex model, since the multiplicity of databases introduces the limitations of the CAP theorem

Distributed state

State is "*a condition or stage in the physical being of something*" (Merriam-Webster). Virtually everything has a state, and, in computing, this is useful to make decision or take actions.

In a **centralized world**, state is represented as a set of values stored in some object. The state evolves because of events and values can be "atomically" updated to reflect the evolution of the system.

In a **distributed system**, each process can have its **own copy** of the state. Since event propagation requires time, these copies can become **incoherent** and lead to diverging decisions. Locking may be introduced to limit divergence, limiting scalability.

Distributed state issues:

- **Strong consistency**
  - How to make the state synced across the nodes such that all nodes return the same state even if queried separately?
- **Staleness/freshness**
  - Is the state returned older than the state at other nodes?
- **Distributed transactions**
  - How to roll back to previous state in case of part of the workflow failed?

Performance issues

- Most heuristics that apply in monolithic applications are not suitable in a distributed environment, since the time scale is largely different, when seen from the CPU vintage point
- Unfortunately, it is difficult for the designer easily grasp this time scale, since the involved delays are often too far from the everyday experience to be suitably appreciated
- In order to grasp the problem, it is useful to adopt a scaled-up version of the delays

Synchronous vs asynchronous programming

Computing systems are often designed based on a **synchronous pattern.** Sides effects are produced by function calls that block the invoker until the effect is ready, returning the outcome of the operation. Some operations may block the execution for an unknown amount of time (e.g., Anything concerning I/O and networking, delay functions, synchronizations). This may lead to a large **waste of time**.

Keeping a thread blocked while waiting for their outcome, makes the code **easier to be written** and maintained since it is possible to adopt an imperative coding approach, since the current computation remains "**stable**" (i.e., variables keep their values, the history of function invocation does not change, …) while waiting.

To **support many concurrent requests**, many **threads need to be pre-allocated** and whenever a new request arrives, a thread is selected to deal with it. If the thread blocks, other incoming requests will not be affected since they will be managed by a different thread. Once the request is done, the thread returns to the pool, and can be selected for managing other requests. Unfortunately, this approach requires a **lot of resources** since each thread need to be pre-allocated together with its execution stack.

Performance and scaling

An enterprise application must process data as fast as possible. The two most important metrics to evaluate its performances are **response time** and **throughput.** The response time is the *overall time needed to complete a transaction* while the throughput is the *rate of completing incoming load*, which can be measured as the number of transactions executed in each time interval. Since the system is intrinsically distributed and concurrent, measurements should properly take into consideration these features.

Response time

To evaluate **response time**, a statistical approach is chosen:

- On a single client, *Count* transactions are continuously invoked in a loop, and the overall time t is computed
- The average execution time is thus $\frac{t}{Count}$ while the throughput for the single client is $$X(1) = \frac{Count}{t}$$
- This value is the baseline for further calculations

Throughput

The concurrent nature of the system allows several transactions, coming from different clients, to be managed at the same time. The throughput of N concurrent connections can be modelled as follows:

$$X(N) = X(1) \times C(N)$$

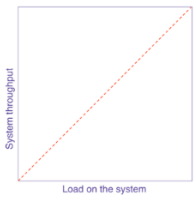
Where $X(1)$ is the **single-client throughput** and $C(N)$ is the **relative throughput gain** due to concurrent client support. If the system were perfectly scalable, increasing the number of clients would yield a proportional increase in the throughput. This does not happen, in practice, due to resource contention and the cost of maintaining coherence across all requests (which implies acquiring locks).

The Universal Scalability Law

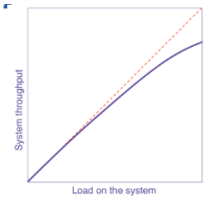
The Universal Scalability Law (Gunther – 1993, 2018) states that scalability (i.e., the system throughput as function of system load) can be approximated by the following equation, derived from queuing theory

$$X(N) = \frac{N\gamma}{1 + \alpha(N-1) + \beta N(N-1)}$$

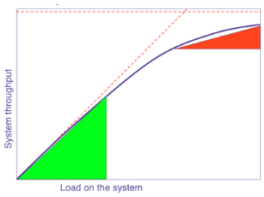
Parameter α represents internal resource **contention** due to waiting or queueing for shared resources. Parameter β represents the **coherency cost** due to the delay for data to become consistent due to point-to-point exchange of data between resources that are distributed. Parameter γ represents the **scalable** part of the system.



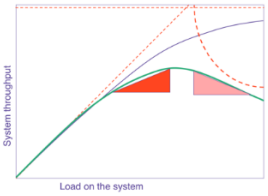
α = 0, β = 0



α > 0, β = 0



α >> 0, β = 0



α >> 0, β > 0

The need for system coherency, cause the β parameter to be non-null. This, in turn, sets a limit beyond which the throughput will not increase any more, and may start falling. The absolute maximum capacity gain is obtained when

$$N = \sqrt{\frac{1-\alpha}{\beta}}$$

To reduce contention, a lot of resources need to be poured in the system. This is the reason for which, in blocking system, hundreds of threads are allocated, and a corresponding amount of memory need to be provided.

The *usl4j* library can be used to estimate the USL parameters of a system by providing at least 6 throughput measurement at different concurrency level.

Spring Boot

In the object-oriented programming approach, an application consists in a set of objects that interact to provide/consume services. Each object encapsulates a part of the state, and it can contain references (dependences) to other objects. Application code typically mixes pure **business logic** (expressed via the class hierarchy and the methods each class offers) and **configuration** (object instantiation and bindings). This increases complexity and makes it difficult to maintain and evolve the code base.

Splitting an application into stand-alone components

Choosing and connecting dependent objects can be extracted from procedural code and transformed into declarative knowledge. The Spring framework provides a set of **abstractions** and **mechanisms** that allow to **instantiate**, **assemble**, and **manage** the life cycle of elementary components to create the final application. The programmer is responsible to define the class of each component (named **bean**) and declare (via **annotations**) its configuration. The framework will inspect the annotations and generate the glue code that is necessary to make the system work.

The Spring framework

The main element that allows this behaviour is named "**container**" or "**application context**". It stores a reference to all components the application is based upon, controls their life cycles, and is in charge of retrieving and connecting them one another. The container represents the **core of an architecture**, consisting of many more layers resting on top of it. One of these layers, **Spring Boot**, offers a mechanism to **simplify** and **speed up** the creation of applications based on a single-entry point.

Spring Boot

Spring Boot is a compact and easy-to-learn **framework for developing (web) applications**. It targets fast development based on conventions and **opinionated** assumptions. When required, conventions can be overridden with suitable configurations. It provides a set of pluggable, **ready-made** building blocks, easy to customize and assemble together with custom code. Code is selectively provided based on deduction of what might be necessary inspecting the class path and the project properties.

All available components (explicitly provided by the programmer or implicitly provided by the framework) are automatically wired together based on their type and on their annotations. This design pattern is known as **convention over configuration**. Spring Boot also provides support for building the entire project as a single, fat JAR file, including all the dependencies thus simplifying the deployment in container-based environments.

The Gradle Boot plugin

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle: it **merges all project jar files** in a single executable jar archive. It looks in the project for the $main()$ function signature and makes it the jar entry point. It automatically applies the *Dependency Management Plugin* and configures it to import the spring-boot-starter-parent BOM. This allows to omit version numbers when declaring dependencies that are managed in the BOM.

Starter packs

Spring Boot provides over 100 starter packs which offer a curated collection of libraries and configurations aimed at solving specific problems:

- **Web programming** (both in blocking and non-blocking versions), access to SQL and NoSQL DBMS, real-time messaging, security, I/O, caching, cloud-based support (configuration, discovery, routing, security)
- When a dependency on a specific starter pack is added to the project, Spring Boot will automatically configure and instantiate the components contained therein, based on **opinionated assumptions**.

Program structure

Independently of the application kind (stand-alone, web), a Boot program has a **single-entry point**. If necessary, a (reactive) embedded web server is automatically added to the project and started.

How can a single annotation allow to configure, assemble, and activate a complex system made of many layers (controllers, services, repositories, message handlers, …), properly wiring them together? In order to understand what happens behind the curtains, it is necessary to step back and examine the basic principles upon which the Spring framework is based.

The Spring Framework

The framework core is based on the following concepts:

- **Inversion of Control (IoC)**
- **Dependency Injection (DI)**
- **Aspect Oriented Programming (AOP)**

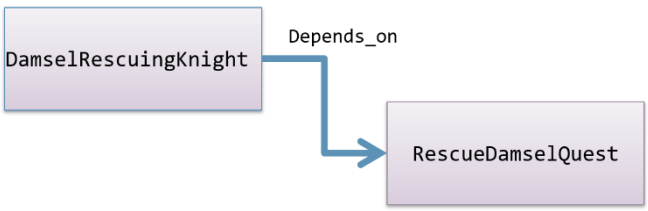
Differently from other frameworks, Spring heavily relies on **POJOs** (POKOs) that means **Plain Old Java (Kotlin) Object**. That is, most components are recognized and used because of what they **declare to do**, rather than for what they are (their inheritance hierarchy).

Data coupling

All object-oriented applications are based on a set of objects that interact according to some logic. In traditional application, each object is responsible to obtain all the references to the other objects it needs to interact with: these are called "**dependencies**". This leads to create tightly coupled applications (**data coupling**).

```
class DamselRescuingKnight : Knight {
    private val quest = RescueDamselQuest()

    override fun embarkOnQuest() {
        quest.embark()
    }
}
```



The first class depends on the second one: it cannot be compiled, tested, deployed if the other is **missing**. This makes the system fragile and difficult to maintain.

Letting two objects interact with each other is necessary in every program but class tight coupling makes code difficult to test, to reuse, and to understand. The two classes need to become **independent**. At source code level, removing the reference to the second class from the first one but keeping the run-time **wiring** between the two classes, since it expresses the purpose of the code.

Inversion of Control

We get rid of direct dependencies in source code, by introducing **interfaces** that abstract actual behaviours.

```
interface Quest { fun embark() }
```

```
class RescueDamselQuest: Quest {
    override fun embark() {
        println("Rescuing the damsel")
    }
}
```

```
class BraveKnight(private val quest: Quest) : Knight {
    override fun embarkOnQuest() {
        quest.embark()
    }
}
```

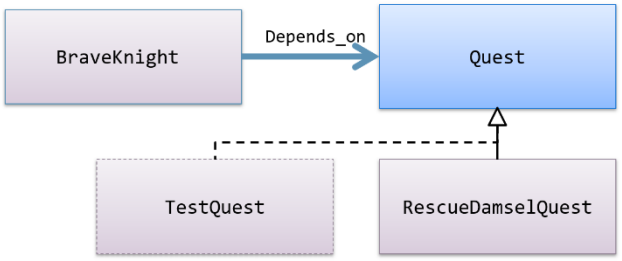
The interface we have introduced separates two planes (**Inversion of Control**):

- **What** must be done (concrete classes that implement it take care of this)
- **When** it must be executed (this is a business of the classes that use the interface)

In more general terms, this kind of inversion is made moving from a direct invocation on an object of your own choice, to an **indirect invocation** on an object supplied by somebody else.

- Typical of **event pattern**: who rises an event is independent of those who consume it
- Typical of **plug-in pattern**: who hosts a plug-in and supports its life-cycle is independent of who supplies the plug-in

This rises a problem for the piece of code in charge of instantiating the first class. A suitable parameter needs to be passed to the constructor. If the project contains a lot of objects and they are deeply nested, it may become very difficult to find the correct instantiation order.



Dependency Injection

The **Dependency Injection** is a programming pattern in charge of **wiring objects at run time** instead of compile time. A dedicated software component is introduced in the project, with the role of creating and maintaining the **objects' dependency graph**. From code analysis point of view, a client object (the one that uses a service) knows only the interface provided by the server one, and not its actual class. Objects are dynamically wired on the basis of declarative pieces of information (like name, type, annotations) from a third party (supplied by the framework).

All dependencies are created and kept in a **software container**. Later on, they are injected in the code in terms of object properties. The program, thus, becomes able to make its own computation. This is in contrast with the naïve approach to application configuration. Usually, the program entry point is explicitly in charge of declaring and instantiating programmatically object dependencies.

There are several ways in which application components can be wired in a Spring project:

- **Explicit** configuration via an XML file
- **Explicit** configuration based on data type and constructor parameters
- **Implicit** bean discovery and autowiring

These three configuration styles can be freely mixed in a project.

Aspect Oriented Programming

There are often behaviours (like logging and security) that are spread and tangled across all the code base since the purpose they support may apply to totally unrelated portion of code. Aspect Oriented Programming is a **programming paradigm** that aims at improving **code modularity** by separating cross-cutting concerns and applying them declaratively wherever they are needed. We call **cross-cutting concern**, any cohesive functionality that impacts horizontally on the system (thus crossing several "functional" modules).

AOP allows the programmer to express a given functionality in a single place without having to manually manipulate all the places where it pertains to. Cross-cutting concerns can be expressed in special classes, named **advices**. These can be applied declaratively to all methods that need them. AOP introduces two benefits:

- The aspect logic is written in a **single place**
- Classes impacted by the logic are kept **clean**, since they just contain their primary logic

Aspect logic is expressed in terms of advice. It defines both what and when an aspect needs to be applied.

Spring aspects support 5 types of advice:

- Before
- After
- After-returning
- After-throwing
- Around (before+after)

To use AOP, a dependency must be added: *spring-boot-starter-aop*.

From a practical point of view, in AOP, the classes written by the programmer are automatically modified at load time by inserting, in the byte code, suitable instructions or dynamically extending them in order to overwrite those methods that are influenced by one or more advices. Spring supports both methods.

Application Context

In a Spring application, objects are created and wired together by a software container that takes care of DI and aspect weaving. All objects kept inside the container have their lifecycle managed by it. Spring offers several types of container implementations which can be broadly categorized as:

- **BeanFactory** (useful in simple scenarios)
- **ApplicationContext** (which can support scenarios of any complexity)

BeanFactory

BeanFactory is an interface providing **basic bean life-cycle management**. It:

- Supports bean **creation** (with constructor resolution), **populating** properties, **wiring** (including autowiring)
- Handles **runtime** bean **references**, resolves managed collections, calls initialization methods
- Supports **autowiring constructors**, properties by name, and properties by type

Class *XmlBeanFactory* is one of its concrete implementations. It reads bean definitions from an XML file.

ApplicationContext

ApplicationContext is an extension of the BeanFactory interface that, to simple bean life-cycle management, adds support also for:

- **Dynamic resource loading**
- An **extended version** of the "observer" pattern supporting publish/subscribe of application events
- Internationalization (i18n) support
- Nested context handling

It is implemented by several concrete classes. For example, *AnnotationConfigWebApplicationContext* provides support for web applications.

Defining beans in Spring

All classes labelled with *@Component* are considered beans by Spring. They are automatically discovered, instantiated, wired, and registered inside the ApplicationContext, provided they are part of a scanned package. The same applies for classes labelled with annotations that include @Component in their definition (like *@Controller, @Service, @Repository, @Configuration*, ...)

Classes annotated with *@Configuration* are further inspected. Any **method** labelled with @Bean is invoked and its result is stored as a bean in the ApplicationContext.

Autoconfiguration

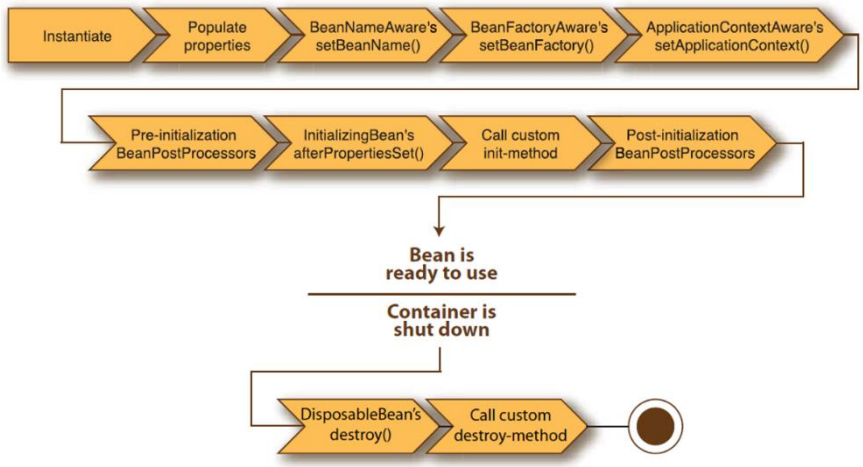
Beyond all beans explicitly declared by the programmer, Spring Boot offers a selection of more than 300 other potential beans classes defined in then *spring-boot-autoconfigure.jar* library. Each of these classes introduces one or more beans suitably configured to support a specific functionality (i.e., e-mail client, interfacing to message brokers, embedded web servers, DBMS drivers, …)

These beans are annotated with **conditional expressions** that limit their usage to those situations in which the programmer has/has not explicitly provided hints (i.e., by defining a custom bean implementing a given interface, by setting environment properties, by adding some classes to the class path, …)

Life-cycle management

When not using a framework, object life-cycle is usually simple: the object is explicitly created by the programmer invoking one of its constructors and when no more references to the object exist, it becomes a candidate for garbage collection. Spring bean life cycle is much more elaborated: the framework oversees **instantiating** and making **available** bean instances. If a bean class is annotated or if it implements some interfaces, it will undergo further processing steps.

At start-up, the ApplicationContext locates bean definitions. Whenever a given bean is needed for the first time, it will be instantiated. Beans are populated with their dependencies and if one of the properties references another bean, the latter is built and initialized first. Depending on how a bean was declared, special cases may apply. Usually beans are **singleton**, but they can be labelled with the @Scope annotation and marked differently. If the bean class implements some given interfaces or is labelled in a particular way, the proper methods will be invoked consequently. For example, if there exist a bean implementing the *BeanPostProcessor* interface, its *postProcessBeforeInitialization(…)* method will be invoked, with a reference to the bean which is going to be configured. When the bean life-cycle terminates, further processing may happen: e.g., if it implements the *DisposableBean* interface, its *destroy()* method is invoked.



When a bean is defined, it is possible to constrain its life-cycle via the *@Scope* annotation (e.g., if marked as **Singleton**: a single instance of the bean is created, and it is going to exist as long as the ApplicationContext exists (default)).

If a bean offers a method named *init(),* this will be automatically invoked (before the bean is made visible to the environment). Analogously, the cleanUp() method act as a destructor. Alternatively, a bean may implement interfaces *InitializingBean* and/or *DisposableBean.* Method *afterPropertiesSet()* will be invoked at the end of the initialization and method *destroy()* will be invoked at destruction time.

Autowiring

Autowiring is a Spring feature that enables to **implicitly inject object dependencies** by searching the ApplicationContext for the requested beans. Several kinds of autowiring exist: by **constructor**, by **type** and by **name**.

Autowire by constructor

If a bean requires one or more parameters in its constructor, the ApplicationContext will try to provide them by searching for beans conforming to the declared parameter type. If more than one bean matches, an exception is thrown. To avoid ambiguities, beans can have extra annotations:

- *@Primary* indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency.
- *@Profile(profileName)* indicates that a bean is eligible for registration when one or more specified profiles are active.

Autowire by type

Inside a bean class, *@Autowired* can be used to label lateinit var. The ApplicationContext will populate those properly, selecting a target bean according to its type. In case of an array, Collection, or Map dependency type, the ApplicationContext auto-wires all beans matching the declared value type. For such purposes, the map keys must be declared as type String which will be resolved to the corresponding bean names.

Autowire by name

Using *@Qualifier("...")* together with *@Autowired*, it is possible to ask for a bean whose name and type match the requested ones.

SpringBootApplication

It is an annotation applied to the main project class. @SpringBootApplication corresponds to the following three annotations:

- @**Configuration** – means that methods of the main class can be used to declare beans
- @**ComponentScan** – sets the current package as the root of packages to scan for annotated components
- @**EnableAutoConfiguration** – attempting to guess and configure beans that are likely to be needed based on classes in classpath and application properties.

Function *runApplication<MainClass>(…)* initializes Spring, runs the Spring application, creating, populating, and returning a new ApplicationContext.

Application properties

Each Spring Boot project comes with a configuration file named *application.properties*, and located in the *src/main/resources* folder. It is implicitly loaded at start-up, and it can be used to provide configuration data. Initially, it is empty, but it is possible to add rows with the syntax *<key>=<value>*. Valid keys and values depend on modules added to the project.

At run-time, Spring will look for the property file in the following possible locations (in order of priority): in the file system, inside the "./config" folder (being "." the folder where the jar file is located) and the in the "." folder; otherwise inside the jar file, in "/config" and then inside the jar file, in "/".

The chosen order makes it easy to replace the content of the *application.properties* file without repacking the application. Values set in the property file can be assigned to properties of any classed labelled with the Component annotation by labelling the corresponding property with *@Value(…).*

Spring Web MVC

The Spring framework supports web application in two different flavours:

- Traditional, **blocking code**, based on the servlet abstraction – supported by the Spring MVC library
- Modern, **non-blocking code**, based on the reactive streams abstraction – supported by the Spring WebFlux library

The former allows writing code using **imperative logic**, so it is simple to write and debug while the latter requires writing code using **functional programming logic**. It can largely reduce the application footprint and allows for shorter latencies and higher elasticity.

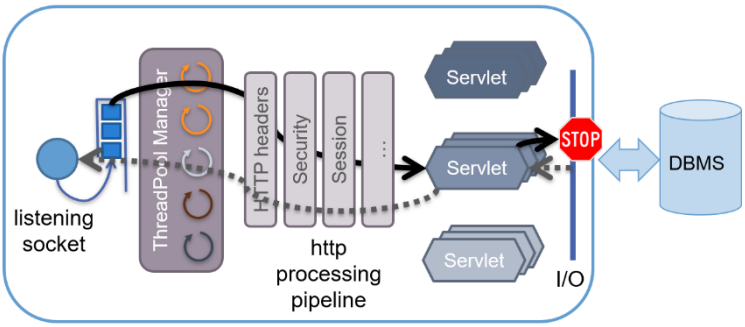
Managing connections

Both approaches provide a similar behaviour from the client point of view:

- They leverage on a **server component** to implement an HTTP endpoint and the corresponding stack (processing headers, coding/decoding data, handling encryption, …), concurrently accepting incoming requests and returning responses
- They both **delegate the task of creating a proper response** for a given request to some programmer supplied pieces of code

However, thread management completely differs in the two flavours.

The servlet stack



It is the traditional mode of creating Java (and Java-derived languages) web applications. Owes its name from the fact that the **code responsible to handle a given HTTP request is packaged in classes implementing the javax.servlet.Servlet interface**. It assumes the existence of a software module (named **container**) that manages the life cycle of Servlet instances and that defines the concurrency policy to be adopted for processing incoming requests.

Concurrency is based on a **synchronous blocking I/O paradigm** using a **one-request-per-thread** model. Whenever a connection is received, the container selects a thread in a large (150+) thread pool and assigns it the job to fully handle it. This will invoke the servlet code which can access remote resources to generate the proper response. If any I/O is performed (e.g., accessing a DBMS), the thread will be **blocked until** the operation completes. If the operation is slow, the thread remains blocked for most of the HTTP request elapsed time. For this reason, the thread pool **must contain many threads**, in order to be able to process multiple incoming requests.

Spring MVC

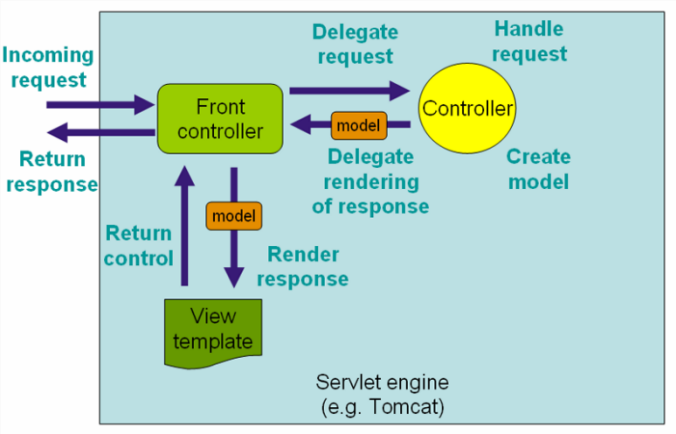
MVC is a **design pattern** which provides a solution for layering an application by separating:

- **Domain data** (*Model*)
- **Presentation** (*View*)
- **Control Flow** (*Controller*)

Spring Boot offers full support for it, providing an **opinionated**, **auto-configured stack** by including the *spring-boot-starter-web* dependency. This will instantiate an embedded version of the Tomcat servlet container, as well as a single Servlet that will manage all the incoming requests.

By default, the container listens to port 8080. This can be changed by setting the *server.port* key in the application.properties file. If classpath contains a different container (like Jetty) this will be used in place of Tomcat. As usual, Spring Boot will manage all defined beans: both those explicitly created by the programmer, via @Component or any derived annotations and those coming from the auto-configuration process. Beans annotated with @Controller play a special role since they are **selected as potential handler** of incoming requests.

Framework module **Web MVC** provides the *Model-View-Controller* architecture that powers the application.



Classes annotated with @Controller are **automatically instantiated** (since it implies the @Component annotation). Spring MVC looks for any method annotated with @RequestMapping (or @GetMapping, @PostMapping, ...) to locate which URL they manage. Each time that a request is received for a given verb/URL pair, the corresponding method is invoked. Its return value is used drive the process of creating a view that will eventually be returned to the invoking client.

Showing HTML content

The transformation of the value returned by a controller into a view is managed by a specific bean, named *viewResolver*. If the classpath contains the *thymeleaf* starter pack, this engine will be used to synthesize the view. Strings returned by controller methods are interpreted as template file names. These are searched in the project '*resources/templates/*' folder (potentially adding ".html" as a suffix).

```kotlin
@Controller
class HomeController {
  @GetMapping("/")
  fun home(): ModelAndView {
    return ModelAndView("home",
        mapOf( "date" to LocalDateTime.now() )
    )
  }
}
```
HomeController.kt

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head></head>
  <body>
    <h1>Home Page</h1>
    <p th:text="'Now is ' + ${date}"></p>
  </body>
</html>
```
resources/templates/home.html

Processing input data

A controller can receive input data in three ways:

- Via URL text segments
- Via query string parameters
- Via request body (when method is POST or PUT)

In the first case, a formal parameter per each URL segment must be added to the method signature

- It will be annotated with @PathVariable
- The method mapping URL will have the corresponding segment enclosed in braces

Data sent in the query string can be accessed by declaring them as parameters of the controller method, as well. Each parameter will be labelled by *@RequestParam(<paramName>)*. They can be of type String or any other basic type: in this case it will be automatically converted, and an exception will be thrown if the conversion is not possible. Methods annotated with *@RequestParam* are not selected if the corresponding parameter is not contained in the request unless the annotation contains *required=false*.

In most situations, input data must be checked against a set of **formal constraints**. While it is possible to make these controls inside the controller method, Spring offers – for many cases – a more general

mechanism. DTO properties can be **labelled** with: *@NotNull, @Min(<val>), @Max(<val>), @Size(min=<val>, max=<val>), ...* If the DTO parameter is labelled with *@Valid* and an extra parameter of type *BindingResult* is added to the method signature, Spring will automatically enforce the constraints. Results will be stored in the *bindingResult* param. Dependency "*org.hibernate.validator:hibernate-validator*" need to be added.

Processing form data

Data sent as part of an html form can be passed to the controller as an instance of a data class. Spring automatically instantiates and populate the given object, by matching the form input names with the corresponding class properties. Errors will be thrown if a conversion to a simple type is requested but it fails.

Customizing error messages

Class *BindingResult* carries the result of the validation of the fields of the DTO object:

- Its hasErrors() methods returns true if any error is present
- Method getFieldErrors(<fieldName>) returns the list of all the errors related to a given field

Using the thymeleaf view engine, it is possible to decorate the view elements with their corresponding error message as well as localizing them according to the client or server locale. For each violated constraint, an error message is added to the model with the following key structure *<violatedConstraint>.<DTOname>.<fieldName>*. File "resources/messages.properties" can be used to replace those keys with more specific error descriptions.

Serving REST content

Instead of returning HTML content, a controller may directly return **structured data content**, in accordance with the REST architecture. The format of the returned resources can be negotiated from the client side using URL naming or via the "Accept" header. Annotation *@RestController* is used to label controller classes whose methods return values are used as the body of the response. Valid return types are *Unit*, *elementary types*, *POKOs* (*Plain Old Kotlin Objects*).

SKIPPED SLIDES OF CAP 06 FROM 27 TO 31

Accessing the lower layers

Controllers and handler function only manage the **presentation layer**. Although it is possible to directly access the data tier inside those components, it is not convenient to do so. By relying on Dependency Injection an Inversion of Control, it is possible to introduce **further components** Which are responsible of the lower layers and that will be auto wired where necessary.

The service layer

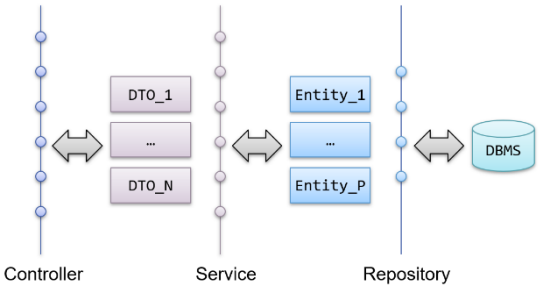
Independently from the way in which a given view is created, an application is characterized by the functionalities it offers. From a programmatical point of view, functionalities take the form of a set of APIs. In object-oriented terms, APIs can be described as software interfaces, whose methods represent the logic of the application. Data that are received as parameters and/or returned as results represent the abstractions of the application domain.

The definition of the service layer represents the key aspect of server application design. It mainly **stems from the application requirements**, and it is largely independent of the overall application flavour (single page application + REST API or traditional web application).

Given the service layer definition, it is possible to create a set of tests, to verify the compliance of the implementation to the functional requirements but also a mock implementation that allows splitting server implementation among different developers, reducing temporal constraints.

The service layer is specified as an **interface** and its implementation is provided by a class labelled with ***@Service***. This makes it easy to wrap service implementation inside dynamic proxies in order to enforce security constraints, enable transaction management or simply logging request. All these cross-cutting concerns will be implemented via further annotations.

The spring-test module offers a set of powerful functionalities to define tests and support their execution both at unit testing level and to integration testing level. Dependency injection and AOP are the main ingredients to support the implementation of these features.



DTO – Data Transfer Object

Model information offered/consumed by the service:

- The **controller is responsible to translate incoming data into one or more DTOs** as requested by the service
- The controller may use DTOs received from the service to populate a view, or it can just return their serialization (as JSON objects or XML documents)
- The service is in charge of transforming DTOs into the data representation requested by the data access layer (entities, documents, graphs, …)

They are often implemented as **Kotlin data classes** and they benefit from automatic generation of relevant member functions (*equals(…), hashCode(), copy(…)*). Mapping from DTOs to entities (and vice-versa) is usually provided via Kotlin extension function (i.e., *User.toUserDTO()*).

The data access layer

Services delegate the task of persisting and retrieving domain objects to the **data access layer**. This may be implemented leveraging directly on **DBMS APIs** (most notably, the JDBC interface) or using some **higher-level** libraries. The former approach gives 100% fine-grained control over what is happening but leaves a lot of responsibilities on the programmer's shoulders (creation of entities from queries, navigation between entities, lazy loading, caching, …).

Spring Data

The Spring Data framework provides rich abstractions to access the data tier. It works with both relational and non-relational databases, sharing – wherever possible – a common logic for CRUD operation support and for transaction management. The main features are:

- Based on a **common set of abstractions** like Entity and Repository
- **Machine generated queries** derived from conventionally named interface methods
- **Automatic auditing**
- Supporting many different DBMS via self-contained, autoconfigured starter library

Access to RDBMS is provided by two alternative (blocking) stacks

- Module **spring-boot-starter-data-jpa** relies on a JPA provider (Hibernate, by default) to implement a **full-fledged ORM** (Object-to-Relational Mapping)
- Module **spring-boot-starter-data-jdbc** provides a basic ORM with less abstractions (and functionalities), but it may be simpler to use

Blocking access to NoSQL databases is supported as well. Spring Boot provides auto-configuration for MongoDB, Neo4j, and several others. MongoDB is an open-source NoSQL document database that uses a

JSON-like schema instead of traditional table-based relational data and requires the *spring-boot-starter-data-mongodb* dependency. Neo4j is an open-source NoSQL graph database based on a rich data model of nodes connected by first class relationships. It requires the *spring-boot-starter-data-neo4j* dependency.

Reactive Spring Data

Spring Data also supports accessing DBMS in a non-blocking way. This, in turn, implies the usage of the **WebFlux reactive stack**. RDBMSs can be accessed via the *spring-boot-starter-data-r2dbc* which is a simple, limited, opinionated object mapper easy to be used, but lacking most advanced features of ORMs. Mongo can be accessed via **spring-boot-starter-data-mongodb-reactive** while reactive access to Neo4J is not available, yet.

ORM – Basic principles

Relational DBMSs share a common conceptual model:

- Data is organized in **tables**, with columns and rows
- Each column in a table stores **one** (elementary) **type** of data
- The data for a single "instance" of table data is stored as a **row**
- Each row has a **unique key** that can be used as a reference to create relationships between different rows

This contrasts with typical OOP, where objects graphs are used to model domain abstractions. The main idea is that an object may roughly correspond to a database row while a class corresponds to a table. A semantic gap exists, however:

- **Objects can be polymorphic, rows cannot**
- DBMS data representation is **normalized**, objects may contain **duplicates** and/or complex data structures
- Rows have **stable keys,** objects are referenced by their **ephemeral memory address**
- DBMSs **use foreign keys in rows** to create relationships, which can be navigated in both ways, objects have **pointers** that are **uni-directional**

Software libraries offer a conceptual framework to model, queries, and process persisted data filling the gap between the object world and the relational one and by providing transparent conversions and getting rid of (most) boiler-plate code. It is **fact-oriented**:

- **Mapping** between classes and tables and relationships among tables are made **explicit** via meta-data (annotations)
- This allows the library to dynamically generate the necessary code, providing efficiency, scalability, maintainability

Entities

Classes that map to DBMS rows are labelled with the *@Entity* annotation. Each entity class maps to corresponding table and entity attributes map to table columns. Annotations (@Table, @Column) can be used to prevent default name mapping. Each entity class must have a *@ID* labelled property. Its value represents the **DBMS primary key** for the corresponding record. Entities must have a public, no-args constructor. It can be added automatically, including the *org.jetbrains.kotlin.plugin.jpa* gradle plug-in.

Schema definition

Depending on how JPA is configured, DBMS schema can be either:

- **Validated** against the entities, throwing an exception in case of mismatch
- **Created** from the entity definitions, deleting any previous data
- **Updated**, according to the current entity definitions
- **Left untouched**, without pointing out nor solving potential conflicts

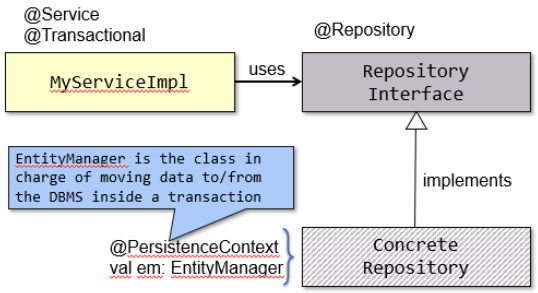
To set this behaviour property *spring.jpa.hibernate.ddl-auto* can be set. Since JPA is **not designed to work with immutable classes**, entities in Kotlin are based on **plain classes** and not data classes:

- Properties must be **mutable** (var)
- Equals implementation should rely on a "**natural key**"
- Because of some low-level interactions between Kotlin and Java, it is recommended to always use **entities with generated IDs in Kotlin**

Conversely, most other Spring Data flavours do support data classes with immutable properties and generated methods: Spring Data MongoDB, Spring Data JDBC, etc.

Repositories

Term used in Spring to define a class that **moves Entities to and from the DBMS**. A repository offers CRUD methods on a single table/collection of the underlying data base. A repository is introduced in the project, defining an interface extending org.springsource.data.repository.Repository<T, Id> or one of the interfaces that derives from it. T represents the class that models the record to be manipulated



- When using JPA, class T must be labelled with @Entity
- When using MongoDB or ElasticSearch, class T must be labelled with @Document

while *Id* is the type of T's primary key that **must be serializable** and class T must contain a property of the **same type**, labelled with @Id.

Spring Data **automatically generates an implementation** for any interface labelled with @Repository, marking it as a bean. When using JPA, the implementation class will delegate standard methods to an instance of class SimpleJpaRepository<T, Id>. The custom repository interface may contain user defined methods and they will be implemented according to a convention based on method name, parameters, return type, and annotations.

- **CrudRepository** offers all CRUD functionalities: if they are not needed, it can be more convenient to extend the Repository interface.
- **PagingAndSortingRepository** offers mechanisms to split larger result sets in pages transferring a given page inside a single connection to the DBMS. It is convenient when data sets are expected to be large.

If an interface labelled with *@Repository* declares methods which do not map on those of the base implementation, these are considered custom queries. The actual query will be derived from the method name and parameters. Result value will be adapted to match the declared return type. By labelling a custom method with *@Query(…)*, it is possible to define the actual query to submit to the DBMS. When using JPA, the query will be written in JPQL (Java Persistence Query Language).

When *spring-data-starter-jpa* is part of the classpath, several autoconfigured beans are injected in the context and they can be customized and replaced at programmers' will.

- javax.persistence.**EntityManagerFactory**: a factory for **creating instances of the persistence** context
- javax.sql.**DataSource**: a **factory for connections** to the physical data source
- org.springframework.transaction.**TransactionManager**: the **access point of Spring's transaction** infrastructure

To **get direct ORM access**, it is possible to access bean of class *EntityManagerFactory* which is automatically instantiated whenever dependency *spring-boot-start-data-jpa* is in the classpath. Whenever a persistency operation must be performed, an *EntityManager* object must be derived from it. It will be in charge of managing the ongoing session.

Repositories and *EntityManagers* refers to the underlying DBMS using a bean implementing interface: *javax.sql.DataSource*.

Spring offers three alternative implementations:

- **Direct connection** to the DBMS driver
- **Connection pooling** implementation – produces a Connection object that will automatically participate in connection pooling
- **Distributed transaction implementation** -- produces a Connection object that may be used for distributed transactions and almost always participates in connection pooling

To allow remote management of the deployed application, it is useful to collect information about its working. Module *spring-boot-starter-actuator* makes this task trivial, automatically exposing a set of URLs (/errors, /environment, /health, /beans, /info, /metrics, /trace, /configprops, /dump, /shutdown). Most of these URLs are subject to a simple access control scheme based on HTTP basic authentication. This can be disabled by adding, in the application properties file, the entry *management.security.enabled=false*.

Spring Data in depth

Spring Data is a **general framework** for persistence which is suitable for both SQL and NoSQL databases and it is based on the *@Repository* concept which is an adapter that takes domain objects (*@Entity, @Document,* …) and makes specific calls to the persistence layer, implementing CRUD operations.

Repositories are defined as interfaces the implementation of which is automatically generated by Spring Data. This avoids copying and pasting the same code over and over, possibly introducing subtle errors.

Spring Data assumes the presence of an underlying data persistence layer, to which actual operations are delegated. In the case of RDBMS, there are three alternatives:

- Spring Data JDBC
- Spring Data JPA
- Spring Data R2DBC

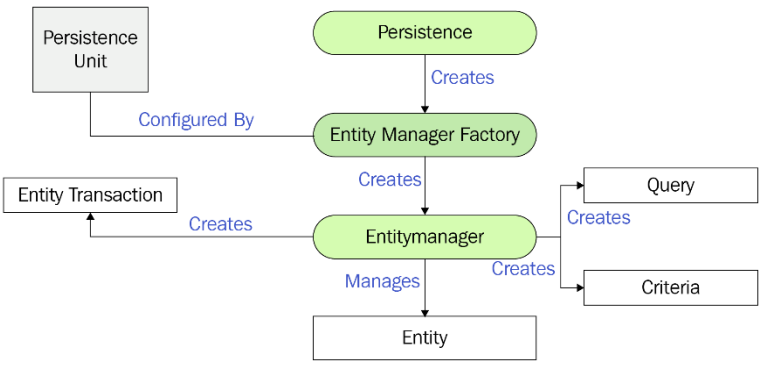
Spring Data JDBC

This is the **simplest approach** to persist data on a relational DBMS:

- When an entity is loaded, the corresponding SQL statement is run - **no lazy loading, no caching**.
- When an entity is explicitly saved, it is persisted to the DBMS – if no explicit save is performed, data changes are lost: **no dirty tracking, no sessions**.
- Entities are mapped to tables using **simple rules** – annotations provide limited support to customize this behaviour

Spring Data JDBC is strongly influenced by the approach named **DDD – Domain Driven Design** which is a software design approach aimed at supporting the development of complex applications. It focuses the designer's attentions on aggregates entities that exist inside a well-defined "bounded context". An application area where specific business processes are implemented, and a set of terms used by expert makes sense to them. Inside each of these areas, entities tend to form aggregates that need to be manipulated.

Spring Data JPA



Leverages on **JPA-compliant ORMs** to make it easier to build Spring-powered applications accessing data. The main goal is to **bridge the semantic gap between the object-oriented representation of data and the relational database**. An ORM (Object-to-Relational-Mapping) is a software layer that (automatically) **converts tables rows into object instances and vice-versa** by keeping the two representations in sync. Default ORM implementation in Spring Data JPA is provided by Hibernate.

It is a generic architecture that provides developers with all the required operations and techniques for mapping objects from and into a relational database. It specifies a set of behaviours in terms of interfaces and requires a concrete implementation, providing the actual classes supporting the interfaces.

Hibernate is a concrete implementation of JPA, but it also offers a set of additional features, based on a proprietary API. These may matter a lot when high performances are a must in the application development.

JPA interfaces

- **Persistence provider**: a concrete software module conforming to the JPA specification
- **Entity Manager Factory**: a singleton object responsible of creating Entity Managers and configuring them to communicate with a specific database, as described by the persistence unit
- **Entity Manager**: an object that encapsulates a connection to a database, offering the actual methods for persisting and retrieving the entities mapped on that DBMS
- **Entities:** a class that represent a domain object in the application. Each entity is represented in the DBMS with a table and a specific instance of an entity corresponds to a record in that table.
- **Entity transaction:** a database transaction that can be either committed or rolled back according to the application state. Any operations (query, insert, update, delete) should be performed within the boundaries of an entity transaction
- **Query:** an object that encapsulate a custom query. JPA provides a custom query language (named JPQL) that can used to perform queries with object-oriented concepts
- **Persistence unit:** defines a persistence context, describing connection information, involved entities, and other useful configuration information

Schema ownership

The duality in data representation (Entities vs tables) allows each of the two sides to push changes to the opposite one. While JPA allows for many different solutions, in real applications the schema does belong to the DBMS. Thus, the database schema should not be updated as a consequence of (possibly involuntary) changes in the code base, but always checked against the entities structure to verify any inconsistences.

This is achieved by setting the following property: *spring.jpa.hibernate.ddl-auto=validate*

The DBMS schema must be part of the source code. Any variation of it should be included in the Version Control System repository and committed.

Entity configuration

Each class defining an entity is annotated with *@Entity*. By default, it is matched to a table having a corresponding (lower-case) name in the DBMS. This may be changed, using annotation *@Table(name="table_name")*. The properties of an entity are mapped to table columns, based on their type and annotations. Attribute *@Column(…)* can be used to provide custom details on the column naming as well as to express extra constraints, like nullability and uniqueness.

An entity has some constraints:

- An entity must be annotated with the @Entity annotation.
- It must have a **public or protected no-arg constructor**. Other constructors are allowed
- It should not be final, nor its method/properties should be final
- It may extend a non-entity class
- If this is labelled with @MappedSuperclass, its properties become part of the current entity
- It must provide an @Id-labelled property
- It must provide useful implementations for equals(…) and hashCode()

The need of having the entity class open, makes it difficult to use data classes. The jpa Gradle plugin provides a no-args constructor for all classes labelled with @Entity, @Embeddable, or @MappedSuperclass. This allows Hibernate to instantiate classes and use Kotlin non-nullable properties with JPA. The allopen Gradle plugin helps meet the non-final constraints on methods and classes. This allows Hibernate to generate proxies for lazy-loading the entity.

In order to improve performances, Hibernate can lazy load instances via runtime proxies (synthesized classes that extends the original entity). The first time any of their method/properties is invoked, the actual entity is

fetched, thus avoiding round-trips with the DBMS if their content is never accessed. This happens when an entity is connected to other entities via relationships. The relationship annotation may contain a fetch attribute, specifying the policy to be adopted.

Identifying entities

Each entity must have one property labelled with @Id, representing the primary key. The corresponding column in the DBMS must be both **unique and non-null**. The primary key can have a meaning in the domain (This is called a natural ID) or it can be generated synthetically (this is called a surrogate ID). The @Id property can be temporarily null, if it is a surrogate as long as the object has not yet been stored and provided that the property is annotated with *@GeneratedValue(strategy = …)*.

Natural keys may be composed of **multiple columns** either the corresponding columns are all labelled with *@Id* or a single property labelled with *@EmbeddableId* is introduced. In the latter case, the property should refer to a class marked with @Embeddable, listing all the columns that are part of the natural key. Compound keys might incur in performance penalties since they require multi-column joins. When working in Kotlin, always prefer a **surrogate key** to a natural one. If using natural keys, it is necessary to specify that is mandatory and immutable.

SKIPPED SLIDES OF CAP 07 FROM 35 TO 39

Implementing equals and hashCode

When an entity has xToMany relationships with another entity, it is convenient to declare the corresponding property as a (Mutable)Set<TargetEntity>. Unfortunately, **Hibernate guarantees equivalence of persistent identity (database row) and Java identity only inside a particular session scope**. So, as soon as instances retrieved in different sessions are mixed, a proper implementation of *equals(…)* and *hashCode()* must be provided to have meaningful semantics for Sets. Although the @Id field looks like a valid solution for implementing equality test and hashCode, problems **may arise** especially if the id is a @GeneratedValue.

While an **entity has not yet been persisted**, it is null, and **comparison should be based on address equality**. Usually, equality should obey to the following constraints, in priority order:

- If the other object is null, return false
- It the other object is identical to this, return true
- If the other object has a different class, return false
- If the current id is null, return false
- Otherwise return the result of comparing the two ids

The existence of proxies may make the comparison more difficult. We need to extract the actual class of the other element, via ProxyUtils.getUserClass(other).

As per Java (and Kotlin) specifications, equals and hashCode should be coherent: if two objects are equal, they should return the same value from hashCode(), **while the converse does not need to be true**. The value returned by hashCode should not change if the content of the object does not change but with generated ids, this may be a problem: when an object is persisted, it gets its own id (that was previously null), but it does not change from the perspective of the application domain. The problem may be solved having hashCode() returning a **constant value**. This impact on the performance of hash tables (and hash sets) whose retrieve complexity moves from O(1) to O(N).

The many constraints that an entity must satisfy can be approached by designing a **generic abstract class** that encapsulate the *@Id* property that will be labelled with *@MappedSuperclass.* It is responsible to implement equals(…) and hashCode() (and, possibly, toString()) and having the **actual entities derive from such a class** thus benefitting from the shared implementation of the helper methods, preventing a lot of boiler plate code.

The *EntityBase* class is generic. It allows subclasses to specify the type of the primary key to be used. The class is labelled with *@MappedSuperclass* thus, its only property (id) becomes part of the subclasses. Method getId() allows subclasses to access the value of the id, but not to change it.

Mapping object properties

Per each property in the Entity class, the following items can be specified via the @Column annotation:

- The name of the DBMS column onto which the value will be mapped, via *@Column (default: the property name)*
- The SQL type to choose (via columnDefinition)
- Further constraints (insertable, updatable, nullable, unique, length, precision, scale)

Property type is used to understand how the property will map onto the table:

- **Elementary** types (Boolean, Int and its derivatives, Float, Double, BigInteger, BigDecimal, String, Enums, Date, Time, Timestamp and derivatives) are directly mapped to their column and managed as value types
- Types marked with the **@Embeddable** annotation are exploded in their elementary properties and added to the current table
- Types marked with **@Entity** or that extend **Collection<T>** (where class T is marked with @Entity) denotes relationships: this must be marked with some special annotation to express their cardinality: **@OneToOne, @ManyToOne, @OneToMany, @ManyToMany**

Date and time mapping

In a typical relational database, there are three basic data types for columns: date, time, timestamp. All of these three data types are represented as instances of java.util.Date object:

- @Temporal annotation is used to disambiguate it
- TemporalType.DATE: represents date-only values
- TemporalType.TIME: represents time-only values
- TemporalType.TIMESTAMP: represents date with time values

Internally, a Date only stores a long number, which is the number of milliseconds from the Epoch (1970-01-01 00:0:00.000 UTC). To represent it in local date and time, a time zone (offset) is needed: this must be stored in a different field.

Relationships

In a DBMS, relationships are implemented by referencing the primary key, via foreign keys. Given the schema and a record, it is possible to reach all other records related to it. Navigation from entity to entity may happen in three ways:

- By directly selecting the corresponding row, if the given record contains a foreign key (1-to-1, many-to-1)
- By selecting all rows of the target table that refer to the primary key of the given record (1-to-many)
- By performing a inner join on a secondary table, in order to select those rows that are related to the primary key of the given record (many-to-many)

In the application code, relationships are modelled via pointers: these are intrinsically unidirectional. When the application logic requires navigating a relationship along both directions, it must be defined in both entities, possibly referring to an intermediate join table. Whenever the relationship changes because of the application code execution, changes must be propagated on the opposite site (if this is mapped).

- **@OneToOne**: Each instance of the first entity is related to (exactly/at most) one instance of the second entity
- **@OneToMany**: Each instance refers to a collection of entities stored in a (Mutable)Set. Referenced entities must have a working implementation of equals and hashCode
- **@ManyToOne**: The current entity is related to (at most/exactly) one other entity. A join column is added to the table, whose name defaults to the <field_name>_<other_id_field>. This can be overridden via the @JoinColumn annotation
- **@ManyToMany**: these relationships require a join table where pairs of related keys are stored. Both sides may offer convenience functions to manipulate the relationship by owning side references to the table, the owned one refers to the other side via the **mappedBy** attribute.

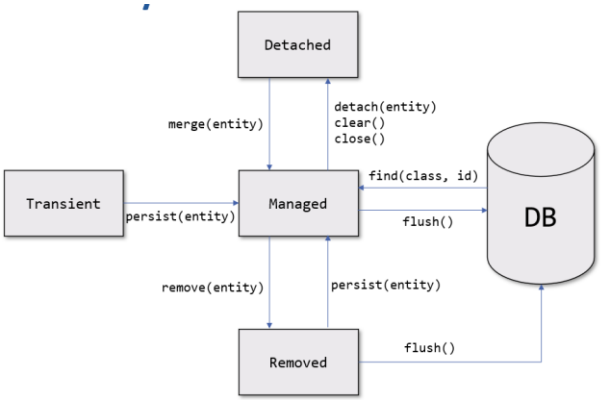
Updating relationships

A special situation arises when one entity is created, updated, or destroyed. Depending on application logic, this might require creating/updating, destroying other entities as well. This is called "**cascading**" and can be controlled by the cascade attribute of the various relationship annotations. It expresses the transitivity constraints that the relationship enforces.

JPA operations

In JPA, the *EntityManager* is the class responsible of keeping in sync entities and the database: this class is obtained from the EntityManagerFactory, via the createEntityManager() method. The EntityManager will do its job of keeping entities in sync since its creation until it gets closed. In this timeframe, several DBMS transactions can be performed. The current transaction is referred to by the transaction property of the EntityManager which offers methods begin(), commit() and rollback() to delimit the set of operations that must be executed with ACID properties.

Every entity has a state at each point, during runtime:



- **Managed** – A managed entity is one that is synchronized with the database: any changes in its properties will be reflected in the database, as long as it is in this state
- **Detached** – A detached entity is one that is not synchronized with the database: this happens when the EntityManager is closed, or cleared (to discard changes)
- **Removed** – If a remove operation is performed, the mapped database record doesn't get removed immediately: when the next flush() operation will be issued, the SQL statement to delete it will be executed
- **Transient** – When an object is created, the JPA provider has no notion of its existence, so no action can be taken to align it to the database

Changes in the state of an entity occurs invoking methods of the EntityManager

- find(…), persist(…), merge(…) cause entities to enter the Managed state
- detach(…), clear(), close() cause entities to enter the Detached state
- remove(…) cause an entity to enter the Removed state

Actual changes in the DBMS only occur when the flush() method is invoked.

SKIPPED SLIDES OF CAP 07 FROM 65 TO 74

JPA Repositories

When using Spring Data JPA, repositories instances are implemented as proxy of class *SimpleJpaRepository*. It **encapsulates** a reference to an EntityManager that will track objects of the given type. A repository offers a more sophisticated interface than the plain EntityManager. Saving and updating entities is achieved via the **save()** method. This will invoke either em.persist(…) or em.merge(…) depending if the entity is new or not. State detection is based on version and id property value.

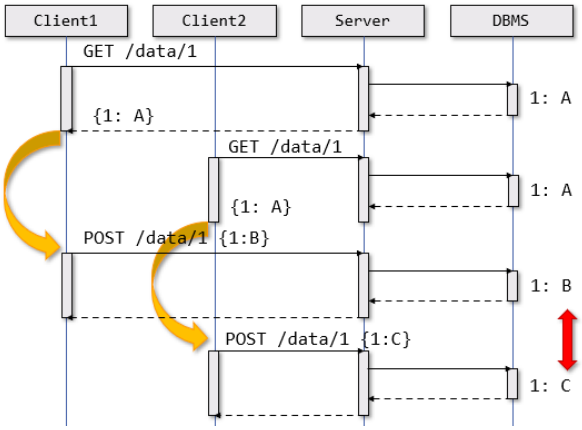
Repositories offer a bunch of **ready-made query methods**, covering general cases. More can be added, specifying the query manually, via the @Query annotation, or having it being derived from the method name. @Query annotations can also be used for introducing custom update and delete methods but the extra annotation *@Modifying* needs to be added.

Transactions

By default, provided CRUD methods on repository instances are **transactional**. For read operations, the transaction configuration readOnly flag is set to true, thus informing the transactionManager that no change will take place. Custom methods should be explicitly marked with *@Transactional* possibly specifying the attribute *readOnly=true*, if it is a read operation. If the transaction needs to span across several methods or repositories, a @Service method labelled with @Transactional must be introduced.

Locking

If the application has concurrent writers to the same objects, then a locking strategy is critical in order to prevent data corruption. Locking assumes a co-operative approach to maintain data integrity and it may involve application-level changes and ensuring other applications accessing the database also do so correctly for the locking policy being used. Setting the transaction isolation strategy may not be enough for a typical web application.



JPA offers two locking strategy to manage concurrency: *optimistic locking* and *pessimistic locking*.

Optimistic locking relies on entities having a numerical or timestamp field labelled with *@Version*. Its value will be automatically managed by the repository implementation and when a read operation is performed, the field is populated from the database. When a write operation is performed, an extra check is added, in the transaction commit code, to verify if the version has changed. In case of mismatch, an *OptimisticLockException* will be thrown otherwise the transaction commits and increments the version.

In case of **pessimistic locking**, JPA creates a transaction that obtains a lock on the data until the transaction is completed. This prevents other transactions from making any updates to the entity until the lock is released but introduces potentially long delays in concurrent transactions, thus lowering the overall scalability of the system. If a lock cannot be obtained (because an other is in place), different exceptions can be thrown

- PessimisticLockException causes a transaction-level rollback
- LockTimeoutException causes a statement-level rollback
- PersistanceException causes a transaction-level rollback

Pessimistic locks will be automatically released when the transaction commits or roll-backs. To specify the lock mode to be used, the @Lock(LockModeType) annotation is added on repository query methods.

In order to deal with concurrency conflicts a proper compromise must be achieved between overall data-coherence and system usability. If optimistic locking is used, versions must be propagated to clients along

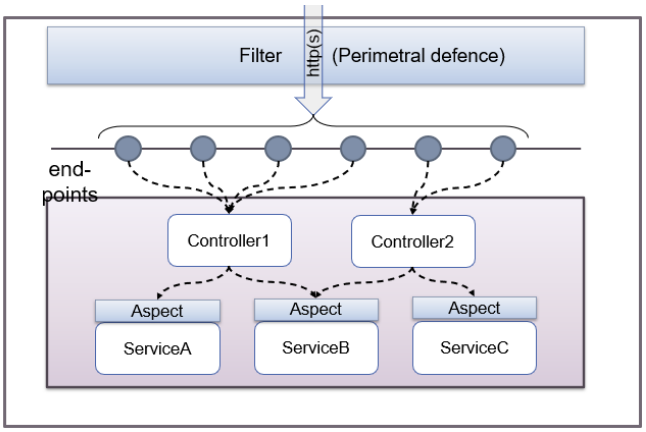
with data so the client will send it back to the server that will check the version and commit the transaction. In case of OptimisticLockException, the conflict should be reported back to the user, who is the only one who can safely decide what to do.

Auditing

JPA supports auditing by inserting extra properties and annotations on entities. If a temporal property is labelled with @CreatedDate or @LastModifiedDate, its value will be automatically populated whenever the corresponding event takes place. If the Spring Security infrastructure is in place and a security context storing the currently authenticated user is available, properties annotated with @CreatedBy and @LastModifiedBy will be populated with the corresponding data fetched from the Principal.

Spring Security

Although it is possible to set security constraints at application level, it is not a good idea doing that. Application security must be implemented in a **declarative way**, leaving to the execution environment the task of translating those constraints into concrete implementations, in all places that can be affected. Project Spring Security implements **web application security** in a **declarative way** by using "**filters**" for the HTTP protocol and by using dynamic "**aspects**" around internal methods, in order to grant proper access only to those who are entailed to do so.



Access control

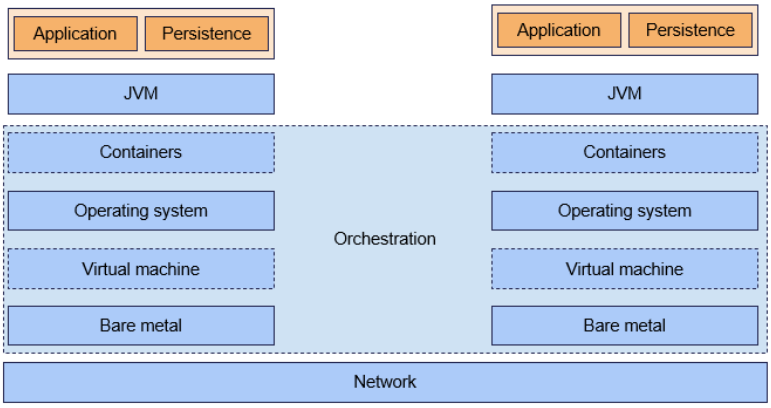
An application that controls and limits the accesses to its own functionalities requires a rich configuration in order to allow:

- **Proper access** from all those users who are entailed to do so
- The deployment of a set of security **countermeasures** needed to prevent any unauthorized usage

This implies managing a **user database** supporting user registration, user authentication, updating profiles, recovering lost credentials and the presence of an authorization mechanism that grants the proper level of access once authenticated. It also imply managing a set of **secrets** which provide evidence that someone really is who she pretends to be or that allow to create signatures and encryption in order to validate and protect exchanged messages.

Spring Security

Spring Security is a framework that lets you build application-level security by **declaratively introducing components**, which can be configured as beans, via annotations and the Spring Expression Language (SpEL) in order to implement predefined functionalities and avoid writing boilerplate code. However, it is up to the developer to understand and use it properly. **By itself, it does not secure an application or sensitive data at rest or in flight**. In a layered system, every layer and all their communication is at risk and must be properly hardened.



Some common web applications vulnerabilities are: Broken authentication, Cross-site scripting (XSS), Cross-site request forgery (CSRF) Injections, etc..

Terminology

- **Principal** – Any user, device, or system that need to interact with the web application
- **Authentication** – the process by which the application verifies that the principal really is the one who it pretends to be
- **Credentials** – Information provided by principal as an evidence of its own identity
- **Authorization** – Procedure that checks whether a given principal can access a specific service / information inside the application
- **Secured resource** – Part of the application the access to which requires the principal to have successfully completed both the authentication and the authorization
- **GrantedAuthority** – Spring interface that describes a fine-grained access right owned by a principal when accessing the secured resource
- **SecurityContext** – Spring interface that keeps the details of the authentication of a principal

Registering user information

In order to create a system that fully supports registering interactive users, the following pieces of information need to be collected and archived:

- **UserID** – it can be chosen directly by each user, but it must be unique in the system
- **Password** – it must satisfy complexity constraints in order not be guessable too easily
- **E-mail** – it allows the user to be contacted and permits the password recovery schema
- **Security questions and corresponding answers** – they provide a weak form of two-factors identification
- **Captcha** – it prevents the end point to undergo massive attacks

The registration process has many use cases, each with its own work-flow, sometimes requiring the intervention of an administrator:

- e-mail verification
- Password forgotten
- Disable account
- Role and privilege management

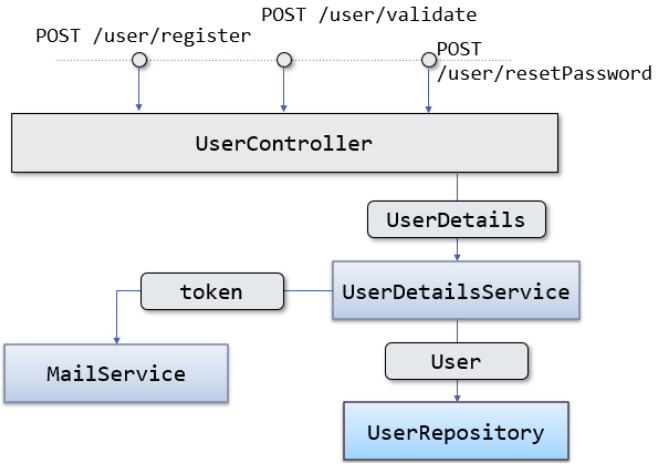
The registration service can be implemented by a controller exposing suitable end-points

- /user/register (GET, POST)
- /user/validate (POST)
- /user/resetPassword (POST)

The controller requires two services:

- One for managing users (UserService)
- One for sending e-mails (MailService)

Information are persisted by a corresponding UserRepository.

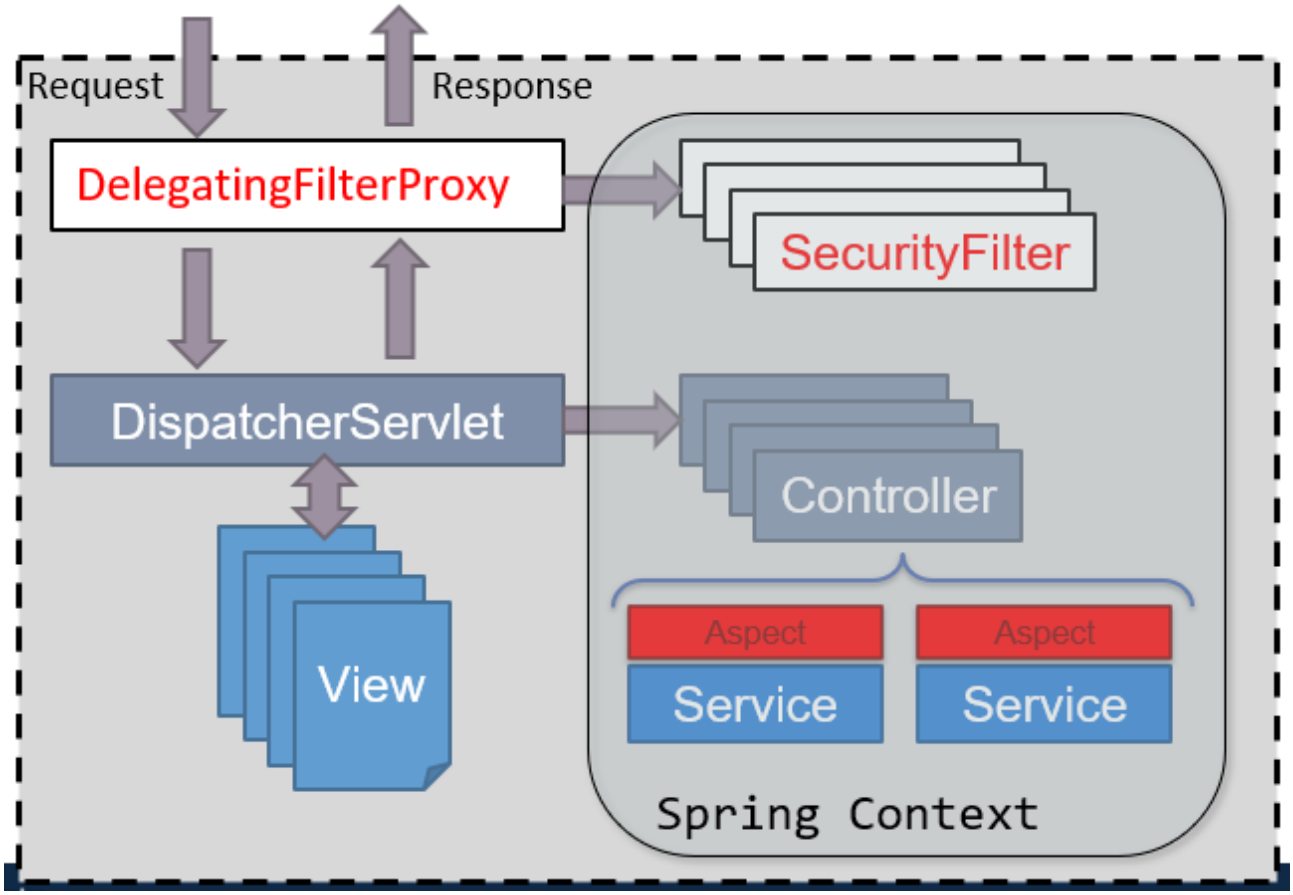


Registration end-points are not subject to access restrictions. However, submitted data need to undergo several correctness checks. In case of failure of any check, a suitable error message must be generated and displayed. If the application is a multi-page one, error messages can be shown directly in the registration form while for single-page applications, the returned JSON object should convey the detected errors or report success.

Access control

Having list of users and their roles is the **first step** to setup access control. The Spring framework support this via the Spring Security dependency. This module creates a configurable infrastructure that allows declaratively managing who can access what part of the system by setting **rules** and **mechanisms** for **authentication** (who is accessing the system) and **authorization** (who can do what with the system)

Spring supports many authentication mechanisms: username/password, SAML, OAuth/OIDC, LDAP and custom ones can be setup, as well. Access restrictions can be expressed in two ways: at URL-level and at bean method-level (typically in services).



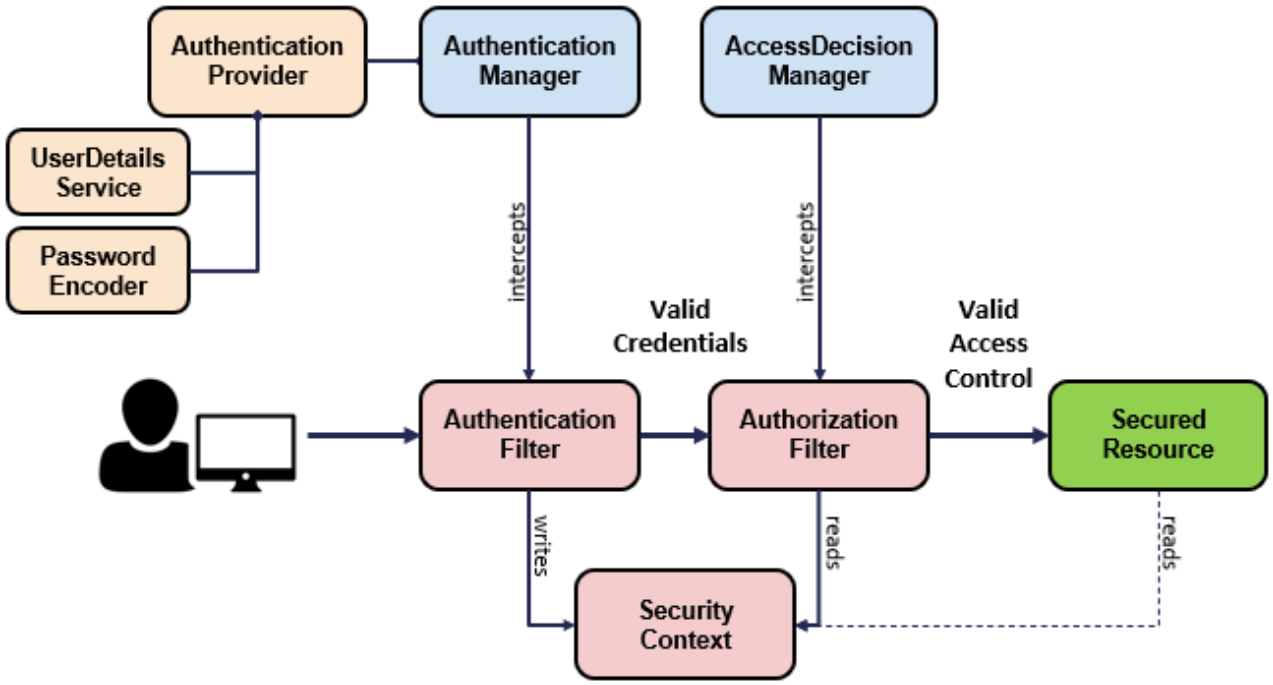
By adding the starter pack, the **autoconfiguration policies are enforced**. If in the application configuration file the following keys are missing, a random password is created and printed in the log file, to be used with the 'user' login:

- spring.security.user.name
- spring.security.user.password

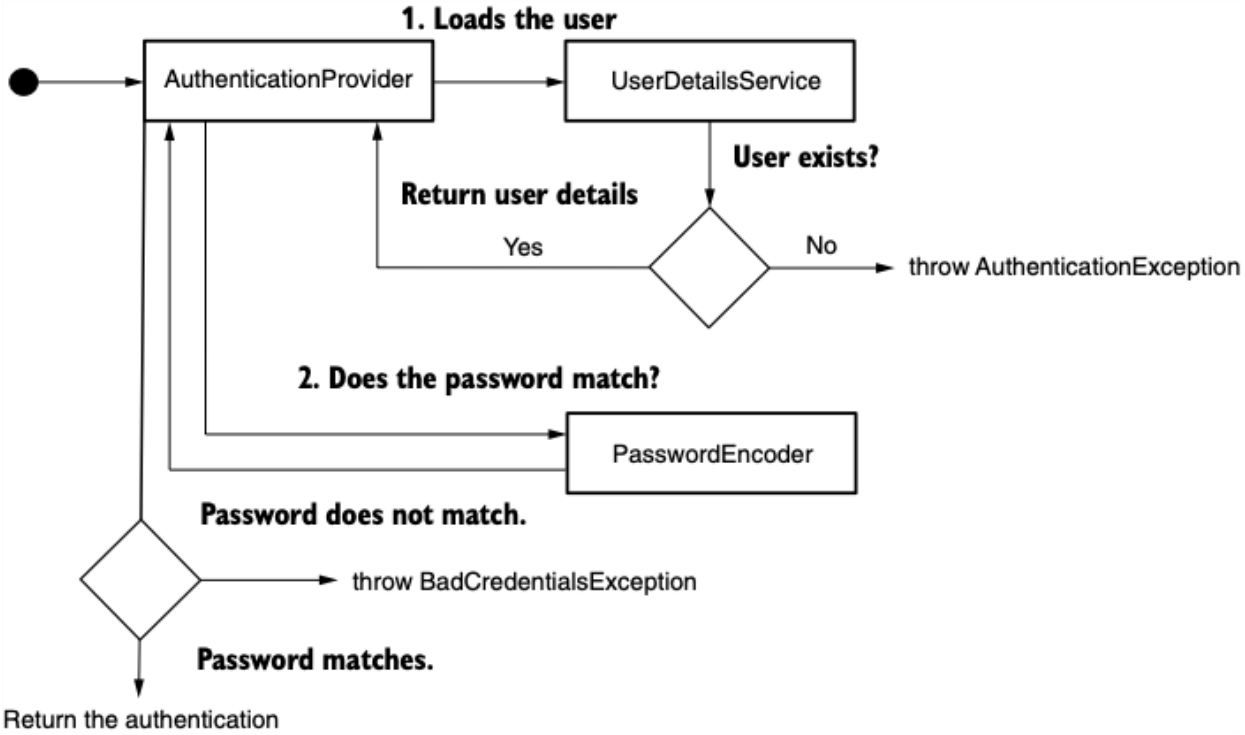
The set of controls to be enforced and their options are configured via a bean that extends the class *org.springframework.security.config.annotation.web.configuration.**WebSecurityConfigurerAdapter*.**

Applications not based on Spring Boot can configure the security filter adding a class that extends *org.springframework.security.web.context.**AbstractSecurityWebApplicationInitializer***. The constructor must pass to its superclass the name of the class where the security configuration is implemented.

Request flow



- The **AuthenticationFilter** delegates the authentication request to the authentication manager and, based on the response, configures the security context
- The **AuthenticationManager** uses the authentication provider to process authentication
- The **AuthenticationProvider** implements the authentication logic
- The **UserDetailsService** implements user management responsibility
- The **PasswordEncoder** implements password management
- The **SecurityContext** keeps the authentication data after the authentication process



Identifying users

AuthenticationManagerBuilder allows to configure the behaviour of the AuthenticationManager. Such an object is responsible of assessing whether a given credential is valid or not. Spring supports several types of AuthenticationManager:

- In **memory**: users, passwords, and roles are defined directly in the configuration class
- In a **relational DBMS**: two tables must exist – one containing username, password, and the enabled flag; the other with the roles associated to each user
- In a **service** that implements the **UserDetailsService** interface, which has a single method
    - fun loadUserByUsername(name: String): UserDetails

In non dummy projects, users are managed via a *UserDetailsService* which is set via *auth.userDetailsService(userDetailsService)*. Password should **never** be exposed in source code, nor in DBMS The Spring CLI provides a convenient support to determine a proper (salted) encoding for a given password.

Protecting URLs

Class *HttpSecurity* offers a fluent API to define in a compact way URLs that must be protected and what rules to apply. It also allows to define how to login (html for, basic authentication, OAuth2, X509, …) and logout.

Performing logout

URL /logout is usually used to finish the current session. Must be contacted with method POST, including the _csrf.token in the request. When the URL is contacted via GET, the default controller renders a html page containing a form that posts to /logout.

Accessing the principal

Any controller can retrieve user identity via the static method

$$val\ userDetails\ =\ SecurityContextHolder.getContext().authentication.principal$$

provided it operates on the the same thread (context, by default, is threadlocal) or it can declare a parameter of type Principal or Authentication, the value of which will be automatically injected when the method is invoked. Principal implementation is left to the programmer. It is convenient to implement it as an object of type UserDetails.

Authentication and sessions

By default, Spring relies on **HTTP session** to keep access information. The servlet filter which is invoked before the *DispatcherServlet* is responsible to verify whether the current request has already been authenticated and, in case, inject the user identity in the security context. If the request is not part of a session having authentication information, an AuthenticationException is thrown. It is possible to set up an *AuthenticationEntryPoint* bean which is responsible to manage such a situation.

Handling application security

The functionalities offered by Spring Security can be exploited at **Service level**. This level is – in fact - the key point of the application logic and can be used in different scenarios. The SpEL scripting language can be used to express application-level constraints. To leverage on this, it is necessary to annotate a **@Configuration** class with the extra annotation

$$@EnableGlobalMethodSecurity(prePostEnabled = true, secured = true)$$

Adding security at Service-level

Service methods can be labelled with security annotations. These will be processed using Aspect Oriented techniques in order to enforce the given constraints. Among the supported annotations, there are:

- **@PreAuthorize** – evaluates the SpEL expression before invoking the method
- **@PostAuthorize** – evaluates the SpEL expression after having invoked the method, allowing to control the returned value
- **@PreFilter** – Applies the SpEL expression to all elements in parameters having Collection type, removing those that evaluate to false
- **@PostFilter** – Applies the expression to the elements returned by the method (that are wrapped in a Collection), removing those that evaluate to false
- **@Secured** - Check that principal has the specified role

Protecting a REST API

In REST services, **unauthenticated requests must not be redirected to a login page**. Conversely after a successful login, it is necessary to return a **credential** that let the requester access the allowed end-points. By default, Spring Security binds authentication information to the http session. This can be disabled and replaced by a solution based on **HTTP basic authentication** or on a **JsonWebToken** that will be provided in each following request.

Implementing login responses

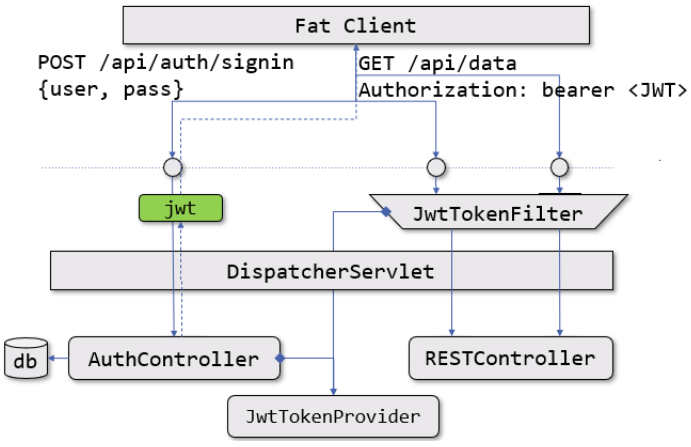
If the authentication process succeeds, the *AuthenticationSuccesHandler* will be invoked. It will receive the current HttpRequest, HttpResponse, and the computed Authentication in order to generate a suitable response. This will set a success status code and will convey in the response any information that the client will need to access authenticated endpoints. If the check fails, the *AuthenticationFailureHandler* object will be notified. This will receive the HttpRequest, HttpResponse, and the AuthenticationException thrown by the validation process. A typical implementation will report the SC_UNAUTHORIZED status code.

JWT – JSON Web Token

It is a mechanism for authorizing access to web APIs. Assumes that the client send, in its request, a header made of three parts separated by '.' **<header>.<payload>.<signature>**.

- <header> contains metadata about the token type
- <payload> contains details about the user the permissions granted to him – its content is arbitrary
- <signature> contains a signature of the parts above

This can be based on asymmetrical algorithms, or on a shared secret between the signer and the verifier. Typically, the three parts of a JWT are Base64-coded. This makes it easier to manipulate it.



Spring WebFlux

When processing incoming requests, some operations may block the execution for an unknown amount of time

- Anything concerning I/O and networking
- Delay functions
- Synchronizations

Keeping a thread blocked while waiting for their outcome, makes the **code easier to be written** and maintained. Since it is possible to adopt an imperative coding approach, based on the fact that the current computation remains "stable" (i.e., variables keeps their values, the history of function invocation does not change, …) while waiting takes place.

In order to support many concurrent requests, **many threads need to be pre-allocated**. Whenever a new request arrives, a thread is selected to deal with it. If the thread blocks, other incoming requests will not be affected, since they will be managed by a different thread. Once the request is done, the thread returns to the pool, and can be selected for managing **other requests**. Unfortunately, this approach requires a lot of **resources**. Each thread need to be pre-allocated together with its execution stack.

It is possible to evolve such paradigm, if the underlying system provides **asynchronous API**. These are typically built using call-backs. Computation is broken in two parts: what that happens before the blocking call, and what happens after. Apparently, it is an easy solution, but which thread is responsible to invoke the call-back?

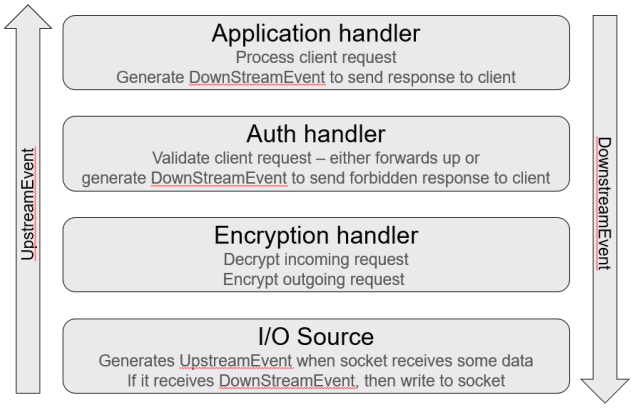
A reactive stack

By adopting a non-blocking approach to computation, a significant number of threads (and thus of computational resources) can be **dismissed** while gaining improved performances. The call-back mechanism makes it difficult to handle most asynchronous corner cases

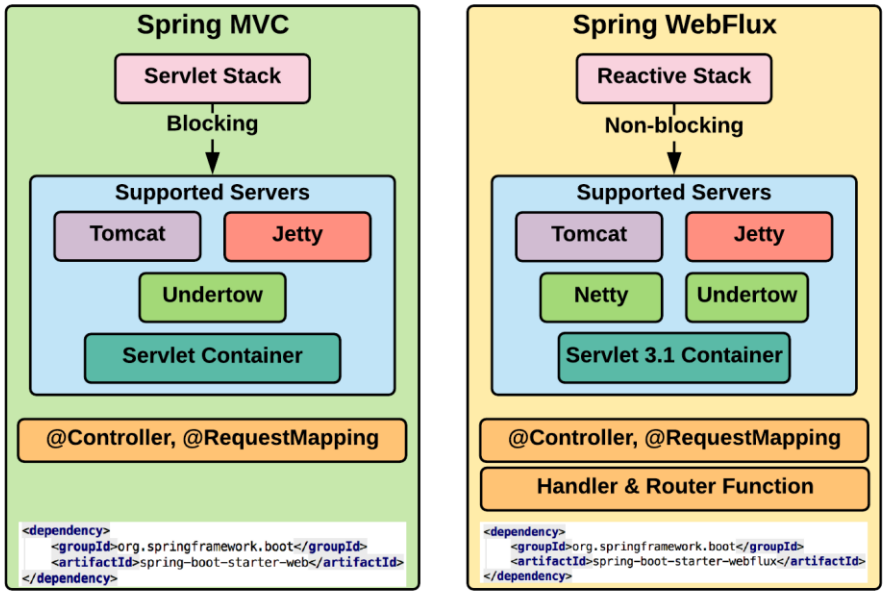
- How to deal with exceptions?
- How to deal with cancellation?
- How to compose multiple asynchronous computations?

In order to deal with these problems different solution have been proposed

- **Reactive libraries** (like RxJava or Reactor) require the programmer to describe the computation to be performed in terms of a chain of functional operators to be applied to an observable stream of events
- **Coroutines** are based on the idea of suspendable computations and leverage on the compiler capability to transform a mostly-imperative code into a call-back-based state machine



Both require new programming styles, but the second option tends to have a lower entry step.



Improving the overall performance

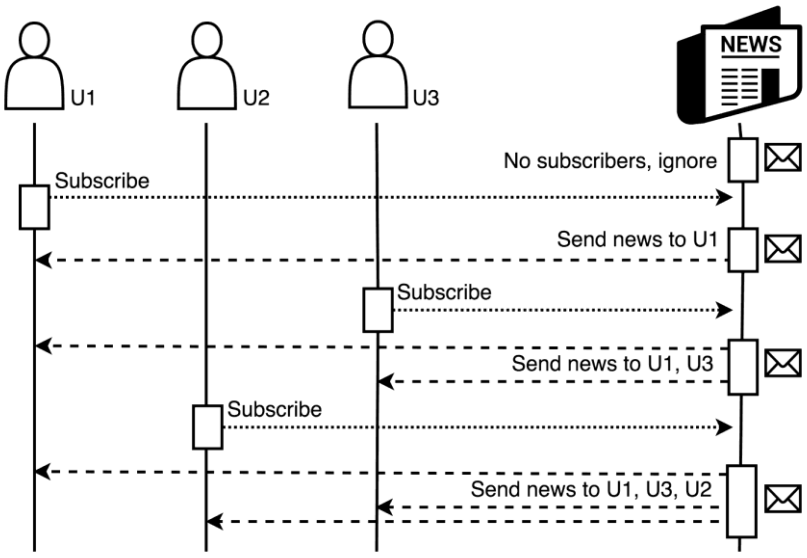
In order to improve scalability and reduce the quantity of resources needed to power a system, given a load level, a different programming model has been investigated. This is based on using **non blocking functions** for accessing remote resources.

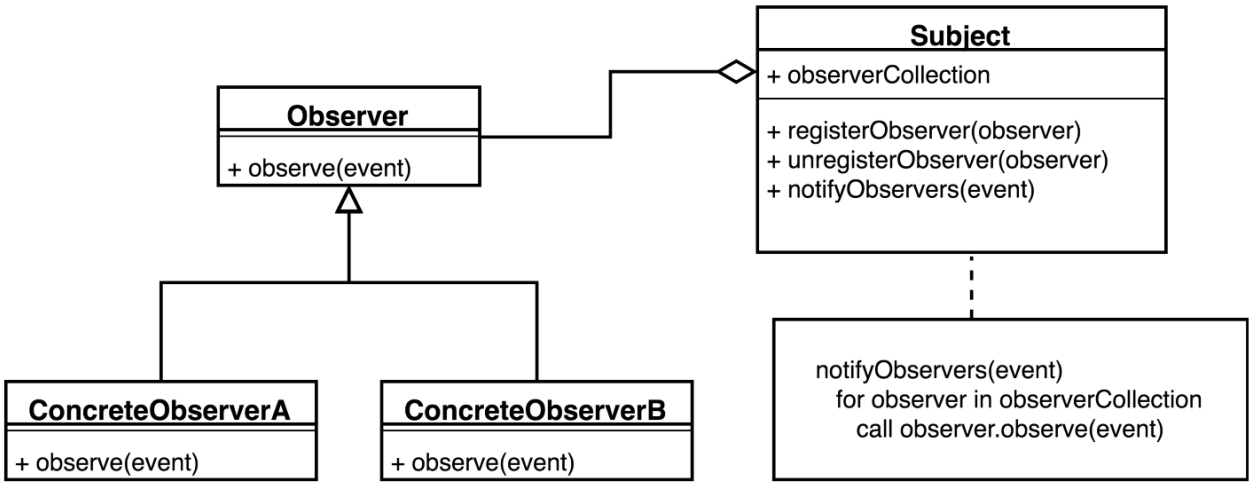
Since the asynchronous approach tends to lead the so called **call-back hell**, functional programming paradigms were explored to create a better modelling of the computational pipeline capable of accounting for all programming corner cases (partial failures, termination, cancellation). This was called **reactive programming**.

Reactive Programming

In **imperative programming**, an expression is evaluated once and the value is assigned to a variable. On the other hand, **reactive programming** is all about responding to value changes.

The Observer pattern





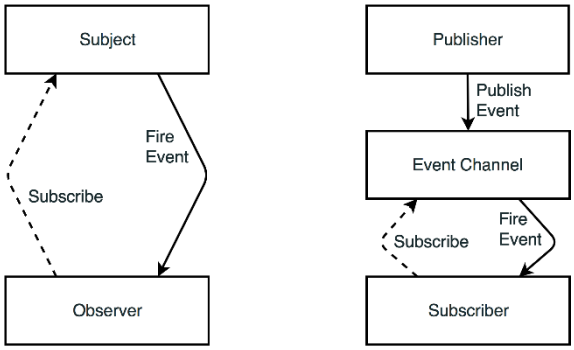
Limitations of the Observer pattern

The observer collections must be **thread-safe** otherwise, it may break unexpectedly. If notifications are sent synchronously a slow observer will slow down all the remaining ones. Some improvement may be introduced by delegating the actual notification to a thread pool:

- If the collection grows too much, a lot of thread will be used
- If an observer is slow, it might receive overlapping notifications
- Push based: no way to exert back pressure
- Cancellation is difficult

Publish-Subscribe pattern

The Spring framework offers a **generalization of the Observer pattern** that decouples the sender of the event from its receivers. The **Event Channel abstraction** is introduced with the aim to separate publishers from subscribers.



The Event Channel may filter incoming messages and distribute them between subscribers. The filtering and routing may happen based on the message content, message topic. Subscribers will receive all messages published to the topics of interest

By labelling a component method with *@EventListener*, any message matching the method argument type, will be dispatched to it. Further restrictions on message routing can be added via the condition attribute. Spring offers the generic *ApplicationEventPublisher* bean to submit new events of a given type to the system bus.

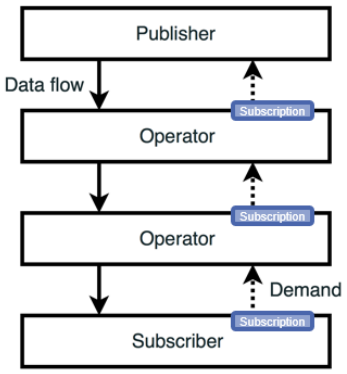
Since publishers and subscribers are decoupled, it is difficult to foresee how the system will behave and what is the overall workflow. Moreover, there is no support for corner cases, such as dealing with failures, termination and cancellation. When the load increases, the implementation of the Event Channel may be significantly stressed possibly making this component the system **bottleneck**. In order to prevent blocking subscribers from slowing down the dispatching process, events must be scheduled in different threads which increase system resources consumptions.

Reactive frameworks

To deal with these problems and provide a general solution supporting the development of libraries, **Reactive eXtensions (Rx)** were introduced. Package *org.reactivestreams* defines the java/kotlin abstractions for Reactive Extensions. It is a set of tools that allows **imperative languages** to work with both **synchronous and asynchronous** data streams. It offers a combination of the **Observer pattern**, the **Iterator pattern**, and **functional programming.** Although Iterator is often used to inspect a data collection that extends in space (at the same time), it can be conceived as a way to iterate across time, waiting for the next event.

It provides the following classes and interfaces:

- **org.reactivestreams.Subscriber<T>**
- **org.reactivestreams.Publisher<T>**
- **org.reactivestreams.Subscription**
- **org.reactivestreams.Processor<T,R>:** represents a transformation logic between a publisher and a subscriber typically implemented as an operator method (map(…), filter(…), reduce(…), …)



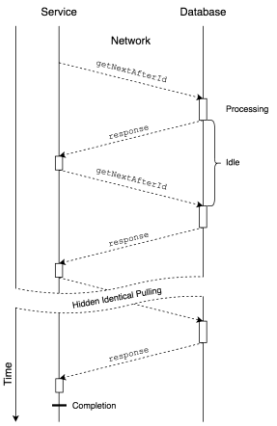
Base concepts

When a Subscriber connect to a Publisher (via the subscribe(…) method), an asynchronous process of event generation is triggered. Subscription acts as an intermediary that provides control over the amount of produced and consumed events.

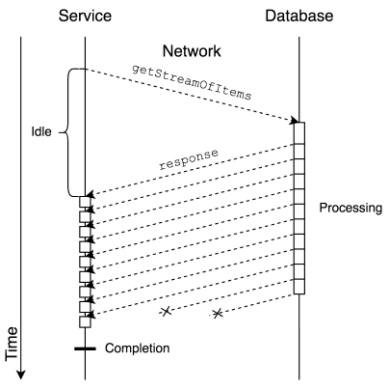
No data is actually transmitted between the two parties, until the receiver **signals** its will to accept some data via *subscription.request(…).* This avoids the need of introducing queues between the publisher and the subscriber, making the overall system **simpler** and more capable of dynamically adapting to changing work-loads.

Push- vs. pull-based architectures

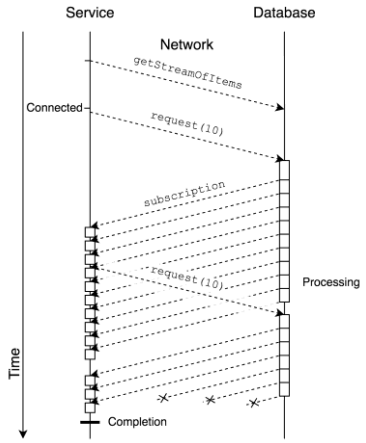
A **pure pull-based architecture** is ineffective when working at network boundaries. Large communication times sum up when requesting many elements one at a time. Batching request improves only slightly.



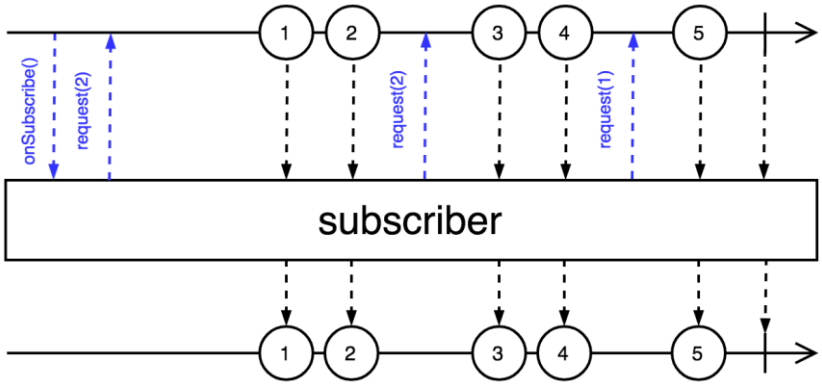
By letting the remote source **push items freely**, a large speed up can be achieved. Flow control should allow stopping the sender from overwhelming the receiver.



Hybrid push/pull



Backpressure via subscription



Implementing reactive extensions

Two alternatives' implementations exist:

- **RxJava** (and RxKotlin)
- **Project Reactor**

Both schedule the events to be sent according to a **well-defined state machine** linking Publisher to Subscriber providing **back pressure and cancellation**. Error and termination corner case are inherently handled by the Subscriber interface. The two libraries differ in the concrete classes that implements the reactive abstractions and in the set of extension operators that allow to describe the processing pipeline.

Core reactive types

Project Reactor provides two alternative implementations of the Publisher<T> interface

- abstract class **Flux**<T> { … }
- abstract class **Mono**<T> { … }

Flux defines a reactive stream that can produce **zero, one, or many** elements, potentially an infinite amount of them. **Mono** defines a stream that can produce, at most, **one** element. This allows a more efficient internal implementation, skipping buffers and synchronizers.

Creating Flux and Mono sequences

Flux and Mono provide many factory methods to create instances based on available data. Mono has similar methods, limited to emitting at most one element (null may be interpreted as an empty content). Sequence of numbers may be created with Flux.range(…) static method. A Mono object can be created using the corresponding factory methods from callables, runnables, suppliers, futures, …

Consuming sequences

To access the elements wrapped into Flux and Mono objects, it is necessary to **subscribe** to them. Several overloaded versions of the subscribe() method exists all return a Disposable object, which can be used to cancel the underlying Subscription.

If a subscription is cancelled by the subscriber, the onComplete() signal will not be triggered. onError(…) and onComplete() are only invoked if the (abnormal) termination is determined by something in the producer context. Both Publishers and Subscribers should strictly respect the state machine rules defined in the Reactive Streams Technology Compatibility Kit (TCK). Class BaseSubscriber provides a compliant Subscriber implementation that can be safely subclassed.

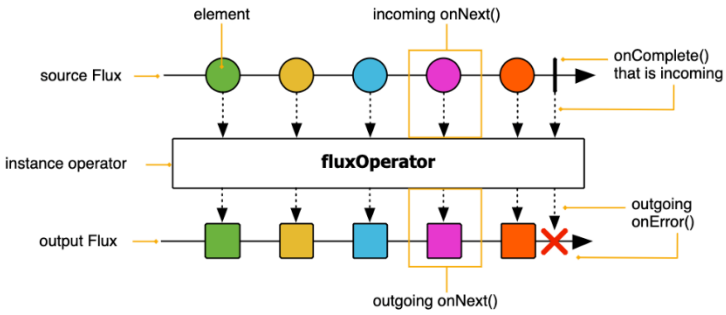
Operators

Flux and Mono offers an extremely rich set of operators, which can be roughly divided into the following areas:

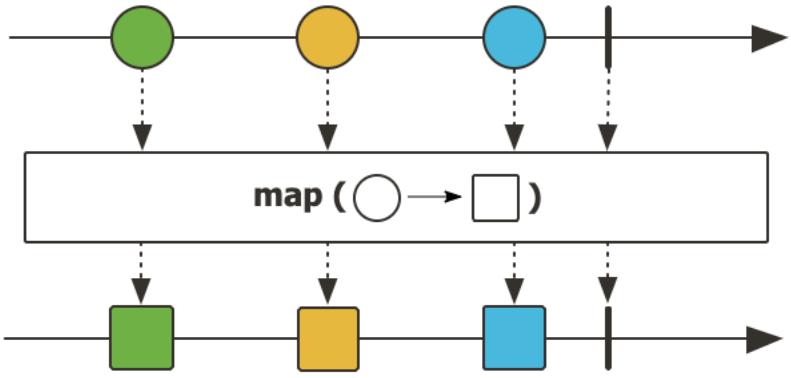
- Transforming existing sequences
- Peeking data as they are processed
- Splitting and joining sequences
- Returning data synchronously

Marble diagrams

It is a graphical representation that depicts the time behavior of an operator.



Method map(…) transform the items emitted by a sequence by applying a synchronous function to each item



The index() operator wraps each element in a tuple with the sequence number. Method flatMap(…) applies to each element a function that returns another sequence of elements, which is flattened in the result, and so on..

Peeking data as they flow

Methods *doOnXXX(…)* allows to **add behavior**, which is triggered when the given condition arises. It is useful for **side-effects**. Methods log(…) observe all Reactive Streams signals and trace them using the Logger system support

Filtering sequences

The filter(…) operator only emits those elements that satisfy the condition. The ignoreElements() operator filters out all elements and returns Mono<T>. The resulting sequence ends only after the original ends or fails if the original fails. It is possible to limit the number of taken elements using the take(…) method

- takeLast() returns only the last element of the stream
- takeUntil(…) forwards incoming elements, until the given condition is satisfied

SKIPPED CAP 09 FROM SLIDE 45 TO 48

Returning data synchronously

- Method *toIterable*() transforms a Flux into a blocking Iterable
- The *toStream*() method transforms the sequence into a lazy, blocking Stream
- Method *blockFirst*() blocks the current thread until the upstream signals its first value or completes

Hot and cold publishers

Publishers are divided into two different categories:

- **Cold** publishers operate **lazily**: no data is generated without a subscriber and if two different subscribers subscribe, a new data sequence is generated
- **Hot** publishers operate **eagerly**: they encapsulate data sources that exist without the application having requested them (like incoming network connections, event streams, …). When a subscriber connects, it will not receive past values

A cold publisher can be transformed into a ConnectableFlux<T> via the publish() method. It allows several consumers to subscribe to it. Elements are requested from the original publisher only after the connect() or refCount() method have been invoked.

Method share() transforms a cold publisher into a hot one. If there is at least one subscriber, this Flux will be subscribed and emitting data. When all subscribers have cancelled, it will cancel the source Flux. This is an alias for publish().refCount().

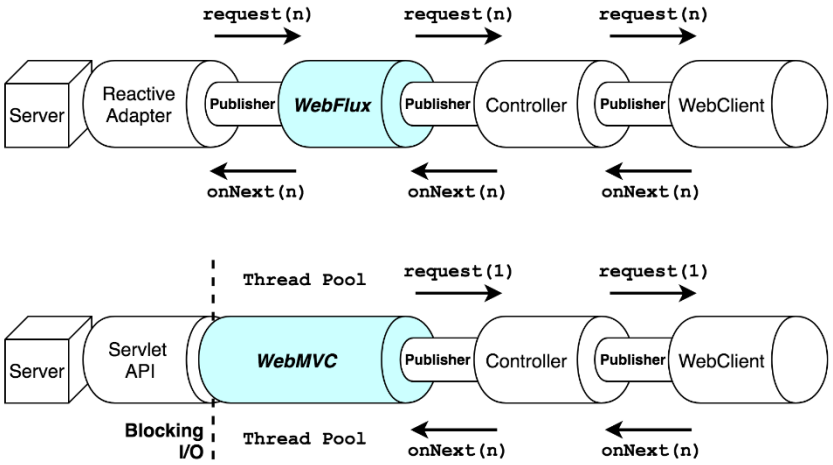
Reactive I/O in Spring

To support reactive abstractions, Spring Frameworks has introduced several abstractions. Interface **DataBuffer** abstracts the concept of a buffer of bytes from the actual strategy used to allocate and access its content, making it easier to interoperate between different non-blocking implementations (java.nio, io.netty.buffer). Class **DataBufferUtils** offers a bunch of methods to read and write files in a reactive way: its methods mostly return Flux<DataBuffer>.

Generic interfaces Encoder<T> and Decoder<T> have been extended to support encoding/decoding streams of data thus offering a non-blocking way to convert serialized data to Kotlin objects and vice-versa. The spring-core module provides byte[], ByteBuffer, DataBuffer, Resource, and String encoder and decoder implementations. The spring-web module adds JSON, binary JSON (Smile), JAXB2, Protocol Buffers and other encoders and decoders.

Reactive web

Spring module *WebFlux* provides a fully **re-written support for web applications** based on reactive streams. It provides **integration** with several server engines, both intrinsically non-blocking (like Netty and Undertow) and capable of asynchronous operations (i.e., based on Servlet API 3.1). A **reactive adapter** replaces the blocking Servlet API, allowing seamless data streaming. Moreover, the non-blocking **WebClient** has replaced the previous RestTemplate class, allowing to perform streaming access to remote web resources.

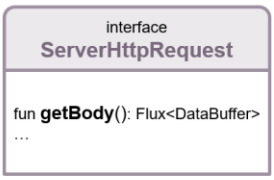


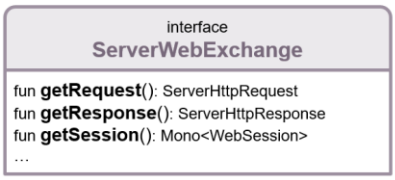
Reactive Spring Data

Several new abstractions have been introduced in the Spring Data project to support reactive streams. Interface **ReactiveCrudRepository<T,ID>** offers the entry point for **non-blocking access to the data layer**, rewriting the typical CRUD operations in terms of Flux<T> and Mono<T>. To properly support asynchrony and back-pressure across process boundaries a set of specific database drivers has been implemented.

WebFlux main abstractions

To support asynchronous management of the HTTP protocol, the main abstractions have been redefined.





ServerHttpRequest models incoming requests. It offers usual methods to access method, URI, headers, cookies, query params, peer details, … as well as a streaming access to the request body (if present).

ServerHttpResponse builds outgoing responses. Response is built filling the body from the supplied Publisher. No data is sent unless the returned Mono<Void> is subscribed. The receiving party may exert back-pressure using the transport protocol control flow.

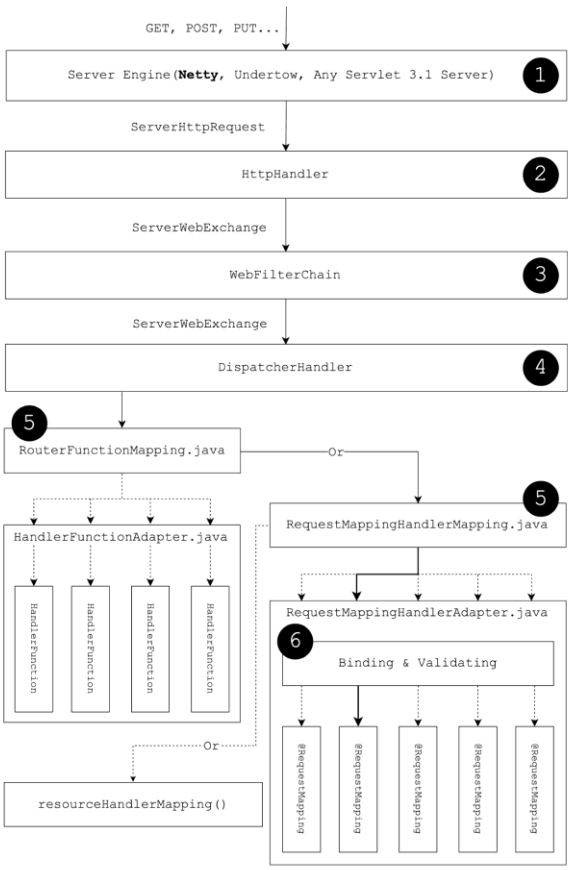
Interface **ServerWebExchange** models an HTTP **request-response interaction**. Beyond giving access to the HTTP request and response, it also exposes additional server-side processing related properties, like session management and attributes.

To provide a global framework that mimics the functionalities offered by the blocking servlet API, three more interfaces are defined: *WebHandler, WebFilter and WebFilterChain*.

The WebFlux architecture

The incoming request is handled by the underlying server engine (Netty, Undertow, …). Each engine has its own reactive adapter, mapping requests and responses to Spring abstractions. The framework encapsulates this information into a **ServerWebExchange** instance.

This goes through the **WebFilterChain** and, if not filtered out, it is eventually passed to a **WebHandler** instance which is responsible to locate the proper Adapter that will process it. The process mirrors the behaviour of Spring WebMVC thus making it easy to understand what is going on.



Router DSL

Spring idiomatic Kotlin extension allows a different way to specify which verb/URLs pairs are managed by the application and what handler function must be invoked. This is achieved using a **Domain Special Language** (DSL) implemented via **router function**. Router allows the programmer to express complex, programmatic expressions that map requests to HandlerFunctions. The HandlerFunction represents a **function that generates responses for requests routed to them**.

The **advantages** of router functions are:

- All routing configuration is kept in **one place**
- **No loss in expressivity** or flexibility regarding the annotation-based approach. It is possible to access incoming request parameter, path variables, and other important components of the request
- Since a reduced number of annotated classes is needed, application bootstrap times can decrease significantly

Remote access via WebClient

WebClient is the reactive replacement for Spring RestTemplate implementation. It provides the ability to perform non-blocking requests to other web services. Instances of this class are created via the **create()** static method or the builder pattern.

Reactive database access

The **ReactiveCrudRepository** interface provides a contract that allows to save, update, or delete entities by consuming not only Entities but also by consuming a reactive **Publisher<Entity>**. At the same time it offers methods to query the corresponding table, returning **Mono<Entity>** or **Flux<Entity>**. This provides effective thread management since no thread is required to ever block on IO operations. Latency of first result is reduced which is convenient for supporting interactive UI components. Allows **backpressure propagation** which allows server and DBMS to adapt to each other's pace.

The *DatabaseClient* provided by the **Spring Data R2DBC** module exposes a reactive API to access SQL databases. It offers a simple, limited, opinionated object mapper. The project was designed valuing ease of use and understanding. Consequently, it **does NOT** offer caching, lazy loading, write behind or many other features of ORM frameworks. Spring Data R2DBC allows a functional approach to interact with your database.

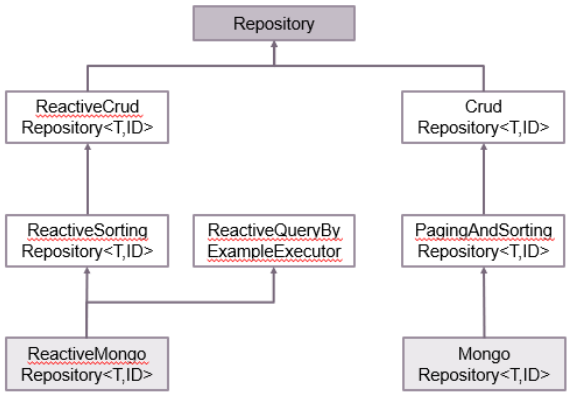
DatabaseClient features are:

- Execution of generic SQL and consumption of update count/row results
- Generic SELECT with paging and ordering
- SELECT of mapped objects with paging and ordering
- Generic INSERT with parameter binding
- INSERT of mapped objects
- Parameter binding using the native syntax
- Result consumption: update count, unmapped (Map<String, Object>), mapped to entities, extraction function
- Reactive repositories using @Query annotated methods
- Transaction Management

Reactive repositories

Spring Data offers several opportunities to access SQL and NoSQL databases in a reactive way. MongoDB is one of the supported options. Reactive Spring Data Repositories use the same annotations and support the majority of synchronously provided features:

- Method name conventions
- @Query annotations to express custom queries
- @Meta annotations to fine-tune runtime parameters



ReactiveCrudRepository<T,ID>

It is an interface that declares methods for saving, finding, and deleting entities or documents.

- fun save(e: T): Mono<T> persists an element and returns its updated representation
- fun findById(id: ID): Mono<T> looks for the given element given its primary key
- fun findAllById(p: Publisher<ID>): Flux<T> accepts any publisher (Reactor or RxJava) of ID and returns all the matching elements: order is not guaranteed
- No direct support for paging

Mapping types

Nested object can be mapped to database tables using data converters. They are responsible for mapping complex domains objects onto the corresponding columns. Two kinds of converters exists:

- @ReadingConverter – implementing the Converter<Row, Entity> interface
- @WritingConverter - implementing the Converter<Entity, OutboundRow> interface

In order to support conversions, a proper @Configuration class should be declared overriding the getCustomConverters() method.

Managing relationships

Properly handling relationships is cumbersome in R2DBC, according to the authors of library. **One-to-one** and **many-to-one** relationships can be managed using a @ReadingConverter and adding a custom query to the repository performing a JOIN SQL operation, to read data. Writing must be accomplished in two separate steps, taking care to properly wait for the first one to finish, to get the key to store in the second one.

One-to-many and many-to-many relationships **cannot be managed** via @ReadingConverters. Explicit query methods need to be added to the repository to fetch related elements and return them as a Flux. This, in turn, requires proper mapping – at the SQL level – of the foreign key constraints, specifying cascading operations to be applied when the referenced key is deleted or updated. Further optimizations can be added in the case of many-to-many to handle fast delivery of aggregated data into the return flux as soon as each object is completed.

Coroutines and flows

In Spring, Reactor is used for 2 different purposes:

- It is the Reactive Streams implementation
- It is also the default reactive public API exposed

Spring Framework 5.2 introduced a new major feature: **Kotlin coroutines can be used to leverage Spring reactive stack in a more imperative way.** A **coroutine** is an approach to write asynchronous code, based on **suspendable computations.** A coroutine may suspend itself, allowing the current thread to perform other useful actions, and later be resumed – possibly in a different thread, when conditions to continue will be satisfied (e.g., a requested response has been received). The programming model adopted by a programmer is almost the same that would be used in case of synchronous code even if the execution happens asynchronously.

Computations

A computation is the execution of an algorithm defined as a function to be invoked. The outcome of the algorithm can be the **result** of the invoked function, or an **exception**, if the computation fails for any reason. The computation takes place in steps, progressively reducing the function, by invoking sub-functions referred to in the function's body. While the computation takes place, a support structure – the **stack** – is used to maintain the necessary information about the computation state.

Computations are carried out in the context of a thread, which is responsible to execute it. If many computations are to be carried out at the same time, a corresponding number of threads must be allocated. If the computation requires some operation that requires waiting for some external event (I/O, synchronization, delay) the thread is blocked until the event happens. Threads may be dedicated to executing a single computation or be part of a generic set – a thread pool – and be temporarily allocated to perform a given computation, until it finishes, then becoming available for executing another computation.

Suspendable computation

Suspendable computation is a computation that may request to freeze its state (the stack) into an external object and temporarily suspend the execution in change of a callback that will be used to resume it. When the external event will happen, some thread will be able to invoke such a callback and continue the computation.

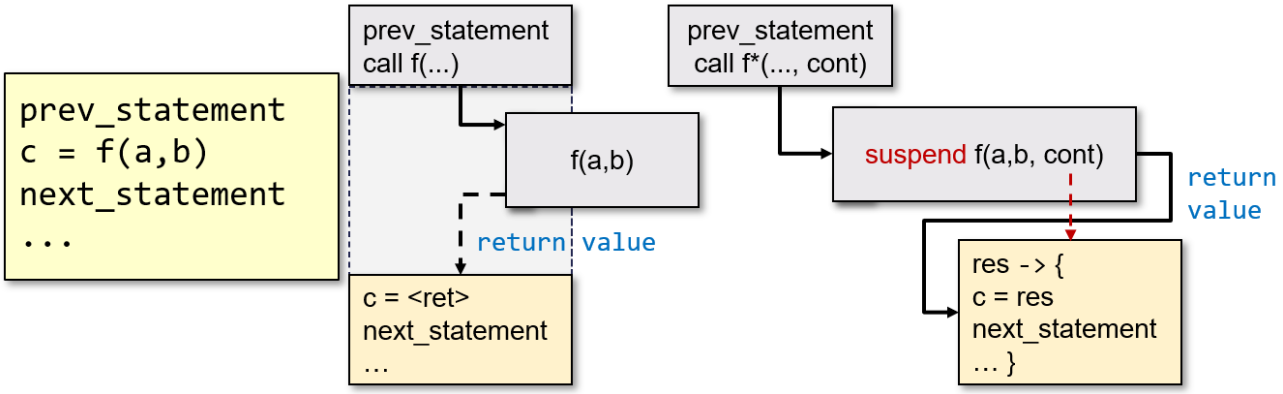
When a computation suspends, the threads that is executing it is freed of the current job. If it were part of a thread pool, it becomes eligible to execute other tasks. If it did operate as a Looper (looping on a message queue for tasks to be executed), it fetches the next task (typical of event threads in GUIs). If it were a dedicated thread, it would become useless (but Kotlin prevents this case).

Saving the computation state into an object can be expensive. A naïve solution would require duplicating the stack into a heap-allocated object (and restoring it when the computation resumes). To reduce this cost, a different function invocation style is introduced by the compiler: this is called **Continuation Passing Style** (**CPS**). In Kotlin, function that contains instructions that may suspend the execution must be marked with the **suspend** keyword. The compiler then uses CPS to invoke those functions.

Function invocation styles

When a piece of code is compiled, two approaches can be used by the compiler to implement function invocation:

- Direct passing style
- Continuation passing style



Continuation-passing style

Using this style, the code **AFTER** the function is encapsulated into another function, named **continuation**. The continuation is passed as an extra parameter to the invoked function. The invoked function does not RETURN its own result. When it reaches its end, it invokes the continuation, passing its result value as an argument.

CPS is enabled in Kotlin by prefixing a function with the keyword **suspend.** The continuation encloses both the state of the execution inside the function and the callback. The code in the function can control when and how the continuation is invoked:

- Synchronously, in the same thread
- Asynchronously, in another thread

A suspend function may invoke, in its body, both regular and suspend functions. The latter will introduce a suspension point, while a regular function **cannot invoke** a suspending one. The compiler rewrites the body of a suspend function, transforming it into a **Finite State Machine**. If there are N suspension points, the FSM will have **N+1** states (the beginning of the function, the end of the function).

The continuation is actually a closure that contains:

- The current state of the FSM
- The set of local variables
- The result of the previous computation
- The callback to invoke to return the value
- The coroutine context that defines
    - the **Dispatcher** responsible to invoke the continuation,
    - the **Job** that conveys information about how the current coroutine relates to other (parent, siblings, children) coroutines
    - the **ExceptionHandler** that deals with failures

Coroutines

A coroutine is an instance of a suspendable computation that is, a computation that can suspend at a given point, freeze its own state in a context object, and later resume, possibly in another thread. Coroutines are designed to support:

- Asynchronous computations
- Futures
- Generators
- Channels
- Flows

Like normal functions, coroutines are executed by a thread. When they reach a suspension point, the state is frozen in the the context and the function returns thus the thread is detached from the coroutine, and it can perform other computations. If a suspended coroutine must be resumed, a thread will be picked to invoke it again. The coroutine context contains the information necessary to pick such a thread and jump to the proper code label.

A suspending function makes it easier to switch from a thread to another and back to the previous one. The suspending function may be seen as a four gate block.

The Suspend operation captures the current continuation (with all local variables) and makes it available as the only argument to a call-back routine. This has the opportunity to arrange the execution of the blocking operation in a different thread, which will – later on – invoke the continuation. When the call-back returns, the computation is actually suspended (the invoking function returns), freeing the current thread.

When a system thread that monitors coroutines will detect that the awaited condition is ready, it will submit the continuation to the corresponding dispatcher, letting the suspend function resume and restoring the same state it had when it was suspended.

Invoking coroutines

Because of their particular code structure, a suspend fun can only be invoked from another suspend fun or from a **coroutine builder function**. Android provides several coroutine builder functions as extension of a CoroutineScope:

- fun **launch**(…): launches a new coroutine without blocking the current thread and returns a reference to the coroutine as a Job
- suspend fun **async**(…): creates a coroutine and returns its future result as a Deferred object

A special coroutine builder function is available as a regular function: **fun runBlocking(...)**. It launches a new coroutine, blocking the current thread until it and all its sub-coroutines finish.

CoroutineScope

It is an interface that encapsulates a **CoroutineContext** and defines a set of extension functions useful for managing (creating, destroying) a group of related coroutines. Coroutines execution is confined inside the scope in which they were created. A specific scope is typically bound to those entities that have a well-defined lifecycle, so that when the entity is destroyed, the scope is cancelled, actually terminating all contained coroutines that are still alive. Function **CoroutineScope**(...) creates a new scope, allowing to customize the encapsulated coroutine context. Kotlin defines also a couple of ready made object scopes, namely **MainScope** and **GlobalScope**, which have some restrictions on their usage.

Coroutine usages

It is possible to **spawn cancellable tasks** (a.k.a. Fire and forget). A coroutine can be created to execute a subtask, without waiting for its completion, somewhat similarly to what can be done spawning a new thread, dedicated to execute the task. However, differently from threads, **coroutines can be easily cancelled**.

```
val scope = CoroutineScope(Job())
scope.launch {
    while(true) {
        delay(1000)
        println("Coroutine - tick")
    }
}
//…do other work
scope.cancel()
```

```
val t = Thread {
    try {
        while(!Thread.currentThread()
                    .isInterrupted)
{
            Thread.sleep(1000)
            println("Thread - tick")
        }
    } catch
(e:InterruptedException){}
}.apply { start() }
//…do other work
t.interrupt()
t.join()
```

It is also possible to **launch multiple computations in parallel and collect their results**. The suspending nature of coroutines acts as a guarantee for accessing the result without having to resort to shared state and synchronization.

```
val res = runBlocking {
    val d1 = async { task1() }
    val d2 = async { task2() }
    d1.await() + d2.await()
}
```

```
var r1 = 0
var r2 = 0
val t1 = Thread { r1 = task1() }
             .apply { start() }
val t2 = Thread { r2 = task2() }
             .apply { start() }
t1.join()
t2.join()
val res = r1 + r2
```

It is also possible to **easily implement timeouts**, even in case of multiple concurrent tasks. To implement it with standard threads it is necessary to resort to class *CompletableFuture*.

Execution flow

Not every suspend fun invocation inside a coroutine will actually suspend. The compiler takes care of that by introducing a special return value to mark a suspension. Two typical cases where suspension occurs:

- **delay(milliseconds: Int)** – causes the current thread to be suspended and resumed in milliseconds ms
- **withContext(ctx: CoroutineContext) { λ }** – suspends the current thread, delegating the execution to any thread inside the given context; current thread will be resumed with the value/exception returned by the provided lambda

Coroutine contexts

Coroutines require a context in order to be executed, since it defines:

- **Dispatcher** – the set of threads used to execute it
- **Job** – the state of the computation of the coroutine
- **CoroutineExceptionHandler** – defines how exceptions thrown inside the coroutine are handled
- **CoroutineName** – the name of the coroutine for debugging purposes

Contexts are typically created adding the elements they are made of:

$$val\ c: CoroutineContext\ =\ Dispatchers.IO\ +\ SupervisorJob()$$

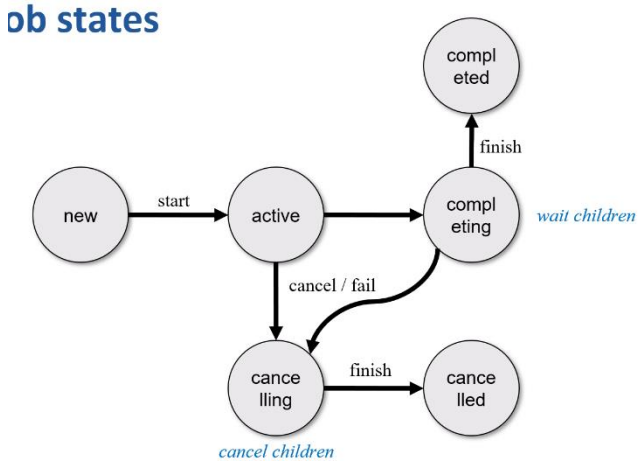
Dispatchers

They are objects that define the thread pool used to invoke the coroutine. A custom dispatcher may be created from an ExecutorService. Class **Dispatchers** provide several ready made ones:

- **Dispatcher.Main** which contains only the main thread and it is only available in Android and for GUI applications
- **Dispatchers.IO** which contains a large thread pool, optimized for disk and network IO and used for reading/writing files, accessing databases, networking
- **Dispatchers.Default** which is optimized for CPU intensive work off the main thread. It is used for task like sorting, parsing JSON, encrypting and decrypting, …

Jobs

Jobs are **representations** of background tasks carried out by coroutines. A job may have six states, expressed as a combination of three boolean properties: **isActive, isCompleted, isCancelled**. Jobs may be organized in an hierarchy and each job may have 0 or more children. If the computation represented by a job fails, failure is propagated to sibling jobs in the same scope as well as to the parent job. If the job is cancelled, only sibling jobs are cancelled. A SupervisorJob is Job subclass that allows children to fail independently of each other.



Jobs offer several methods to control their state:

- **fun start()** – start the coroutine related to the job if not started yet
- **fun cancel(cause**) – cancel the job with an optional cancellation cause
- **suspend fun join()** – suspends the invoking coroutine until the job is complete

Handling exceptions in coroutine

The fire and forget nature of the **launch** builder function does not require explicit handling of exceptions thrown by the coroutines it build. However, in order to avoid have them swallowed by the system and be treated as uncaught exceptions, it is possible to install, in the coroutine context a **CoroutineExceptionHandler**. Children coroutines usually delegate exception handling to their parent coroutine, unless their context is populated with a SupervisorJob(). Coroutines created with the **async** builder function always catches all their exceptions and throw them when await() is invoked on the deferred result.

Coroutine cancellation

All suspending functions from *kotlinx.coroutines* package are cancellable. User provided suspend functions must be cooperative with cancellation. If a function start executing a CPU intensive task that does not invoke any other suspend functions, a periodic check to the isActive property or an invocation of the **yield**() method should be performed, in order to quit in case of cancellation.

The **job object** set in the scope context determines how cancellation will be propagated among sibling coroutines. If the context contains an instance of the Job class, cancellation of a single coroutine will propagate to siblings causing their cancellation as well. Conversely, if it contains an instance of the SupervisorJob class, cancellation will not propagate to sibling coroutines

The Flow abstraction

The Kotlin coroutine library introduces a new interface named Flow<T>. An asynchronous data stream that sequentially emits values (i.e., inside the same coroutine). It can either complete normally or with an exception. A bunch of extension functions (map, filter, take, zip, …) provide intermediate operators implementation. They are applied on upstream flow(s) and return a downstream one. They just set up a set of operations for future execution and quickly return.

Terminal operators are either suspending functions (collect, single, reduce, toList,…) or operators that trigger execution of further processing steps. They complete normally or exceptionally depending on successful or failed execution of all the flow operations upstream. Most flow operations are executed sequentially in the same coroutine and only few operations (buffer, flatMapMerge) are designed to introduce concurrency into flow execution. Moreover, flows can be **cold** or **hot** so a set of operators are provided to convert a cold flow into a hot one or into hot channel.

All implementations of the Flow interface **must adhere to two key properties:**

- Context preservation
- Exception transparency

These properties ensure the **ability to perform local reasoning** about the code with flows . They modularize the code in such a way that upstream flow emitters can be developed separately from downstream flow collectors. A user of a flow does not need to be aware of implementation details of the upstream flows it uses.

Context preservation

Flows preserve execution context. They encapsulate their execution context (dispatcher, coroutine, …) and never propagate it downstream. A flow should only emit values from the same coroutine. It cannot start a new one, nor switch thread or coroutine. Operator flowOn(d: Dispatcher) make the upstream flow to be executed on the given dispatcher. ChannelFlow encapsulates all the context preservation work, allowing collection and emission to happen in different coroutines.

Exception transparency

Flows are **transparent** to exceptions. A downstream exception must always be propagated to the collector. No value should ever be emitted from try/catch blocks. Flow can use the catch(...) operator to handle upstream exceptions. Terminal operators throw any unhandled exception that occurs in their code or in upstream flows. The **onCompletion**(…) operator replaces the finally block.

Backpressure

Kotlin Flow backpressure is based on suspension. When downstream is suspended, no collection is happening, thus suspending upstream as well. When onFlow(…) is applied to a flow, upstream and downstream computations are separated by a channel. This acts as a synchronization device, thus slowing down the producer, as long as the consumer is slower. Flow backpressure is not 100% equivalent to reactive streams backpressure, especially at process borders.

Flows in Spring

Spring WebFlux, Spring Data, Spring RSocket conveniently supports suspending functions by providing automatic adapters where a function returning a Publisher was expected. Methods annotated with @RequestMapping and derived can be implemented as suspending functions, as shown in the picture.

- `fun f(): Mono<Void> → suspend fun f()`
- `fun f(): Mono<T> → suspend fun f(): T`
- `fun f(): Mono<T> → suspend fun f(): T?`
- `fun f(): Flux<T> → fun f(): Flow<T>`

Spring Data R2DBC repositories

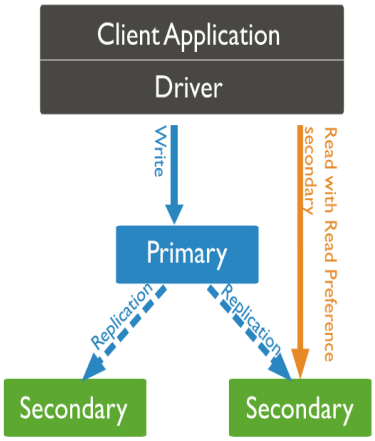
By using the R2DBC Spring starter pack and a suitable non-blocking database driver, it is possible to leverage on the Repository abstractions and access relational data in a non-blocking way. Domain classes need to have a field labelled with @Id to identify the corresponding primary key. **@Table** and **@Column** annotations can be used to properly constrain instruct the object mapper when fetching and storing data form/to tables.

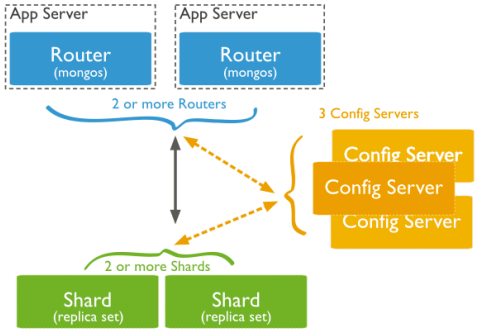
Repositories can be created extending a coroutine based interface either **CoroutineCrudRepostitory** or **CoroutineSortingRepository**. They offer a set of fun / suspend fun supporting the standard repository operations. Custom method can be added preceding them with the *@Query(sql_string)* annotation, which is part of the *org.springframework.data.r2dbc.repository* package.

A transactional operator was introduced to define transaction boundaries using a reactive stack. A corresponding extension for flows exists. This is necessary to overcome limitations of traditional implementation of transactions which relies on a ThreadLocal variable where the transaction context is stored.

MongoDB

Schema-less document-oriented database designed to handle a massive amount of data and ensure high availability. Available both as an open source tool and as a fully-managed cloud-based platform. A MongoDB installation can be as small as a single process (or a Docker container) hosted in the developer machine or be as large as hundreds of dedicated hosts, sharding and replicating the data set.





Documents

The unit of storage in MongoDB is a document, which is a binary encoded JSON object, consisting of multiple key/value pairs, where keys are strings. Values can have one of the following types: Null, Boolean, Numerical (32-bit int, 64-bit long, 64-bit double, 128-bit decimal), String, Temporal (date, timestamp), Binary data, ObjectID, Objects, recursively containing zero or more key/value pairs, Arrays of all the above types.

Each document has a distinctive key, named "_id" containing a unique value acting as the document primary key. Document size cannot exceed the hard limit of **16Mbytes**.

Collections

MongoDB stores documents in **collections**. Collections has some analogy to tables in relational databases: like a table, a collection groups a set of semantically-related documents. Differently from tables, collection are not homogeneous: each document in a collection can have a different structure. Collection have a name which is used to designate it in queries. A collection is automatically created as soon as the first document is saved into it or an index is added.

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. They work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

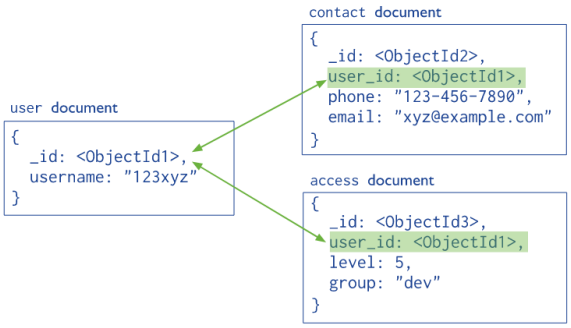
Database

Collections are logically grouped in a **database**. A database stores one or more collections of documents. If a database does not exist, MongoDB creates the database when the first data is stored in it. A MongoDB installation can consist of **zero or more databases**, each having zero or more collections, each having zero or more documents.

Data modelling

The document structure consents different approaches when storing data in MongoDB:

- A document can refer to another document by storing its primary key (_id) in one of its fields (**normalized** data model).
- The target document can be embedded in the source one: if it must be referenced by more than one source, many copies of it will exist (**embedded** or **denormalized** data model)



Embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. It also makes it possible to update related data in a **single atomic write operation**. If the same data is referenced by two or more documents, **multiple copies** of it exist: this may be problematic in case of update. Embedding should not violate the size limit of BSON documents.

Normalized data models allow to represent more complex relationships as well as large hierarchical data set. A special kind of query, based on the aggregation pipeline need to be created to perform data joins across collections of the same database. A recursive search on a collection, with options for restricting the search is provided, to navigate trees and graph structures. Join performances are **not as high as** in RDBMS, if the field storing the foreign key is not indexed

CRUD operations

Write operations target documents in a **single collection**. If the collection does not exists and the operation is an insertion, the collection is automatically created. All write operations are atomic on the level of a **single document.** Update and delete operations use filters that specify which documents need to be updated or deleted.

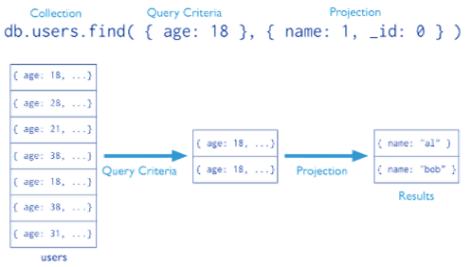
Read operations retrieves documents from a collection. The basic operation is find(…) and it operates on a single collection.

Queries that span over several collections are possible via the **aggregation framework**. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

SKIPPED SLIDES FROM 15 TO 22

Projections

Determines which fields are returned in the matched documents. Reduces the overhead due to transferring useless data. Field *_id* is included in the result by default. Fields are included/excluded setting their value to 1 or 0 (true or false)



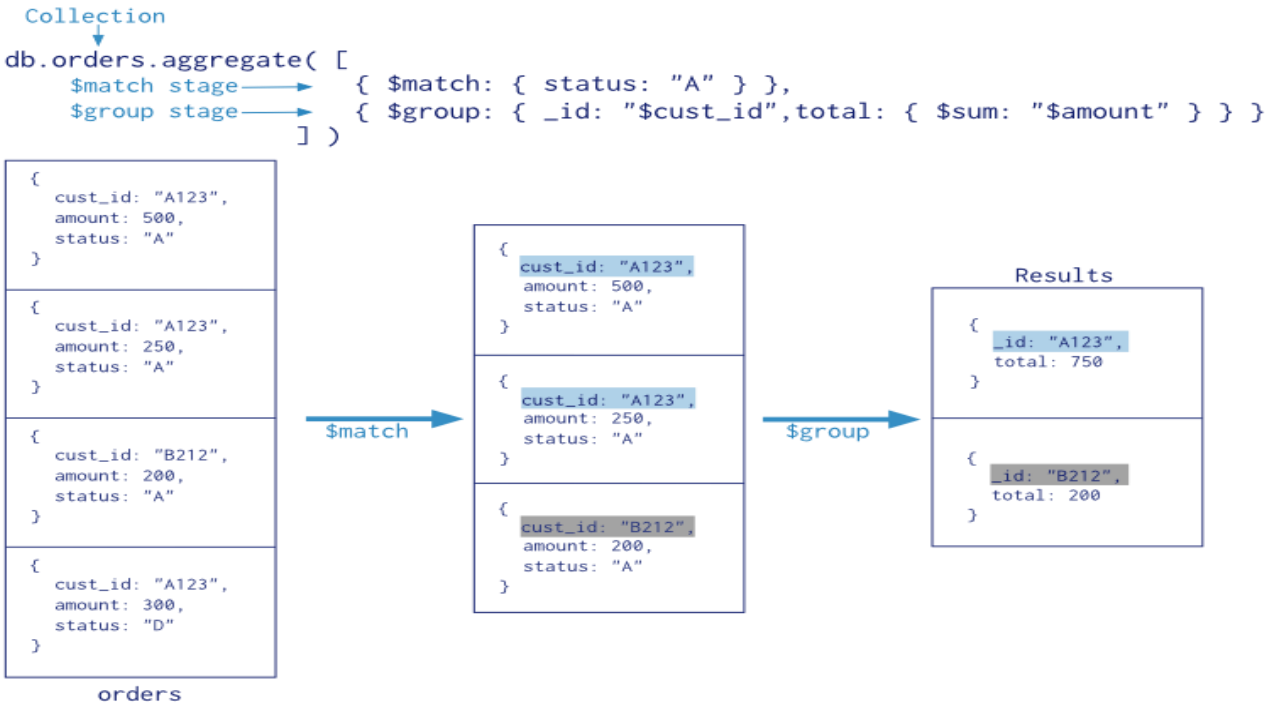
Aggregations

Aggregation operations process data records grouping values from multiple documents together. Several operators are provided to transform selected documents into an aggregated result. MongoDB provides three ways to perform aggregations:

- The aggregation pipeline
- Single purpose aggregation methods

Aggregation pipelines are modelled as data processing pipelines:

- Documents enter a multi-stage pipeline that transforms the documents into an aggregated result
- Each stage can produce zero or more output documents per each input one, which are fed to the next stage or returned if it is the last one
- A given operation can appear more than once in the pipeline



Indices

MongoDB offers several index types optimized for various query lookups and data types:

- **Single Field Indexes** are used to an index single field in a document
- **Compound Field Indexes** are used to index multiple fields in a document
- **Multikey Indexes** are used to index the content stored in arrays
- **Geospatial Indexes** are used to efficiently index geospatial coordinate data
- **Text Indexes** are used to efficiently index string content in a collection
- **Hashed Indexes** are used to index the hash values of specific fields to support hash-based sharding

Indices exchange query speed up with memory and write penalties especially on **sharded clusters**, inserting an indexed document can become expensive.

MongoDB in Spring

Spring provides integration with MongoDB via the **Spring Data MongoDB project**. Spring Boot provides both the *spring-boot-starter-data-mongodb* dependency based on a **blocking** client and the *spring-boot-starter-data-mongodb-reactive* one to support **web-flux-based** implementations.

Major abstractions are provided respectively by the *MongoTemplate* and *ReactiveMongoTemplate* helper classes. They make it easy to perform common operations, leaving application code to provide BSON documents and extract results. Libraries also provide automatic implementation of Repository interfaces including support for custom query methods, geospatial queries, aggregation framework and map/reduce.

Defining documents

Domain objects are named **documents**. They can be modelled by immutable data classes prefixed with the **@Document annotation**. **@Id** is used to mark the distinguished identifier: supported types are String, ObjectId, and BigInteger.

Repositories

Interface **MongoRepository<T, ID>** extends PagingAndSortingRepository<T, ID>. It is used as a base interface for defining custom repository based on MongoDB. It is supported by class SimpleMongoRepository. Custom methods can be defined using the standard naming conventions used by Spring Data. The **@Query** annotation can be used to provide a custom query: attribute value represents the matching condition; attribute fields is used to convey the projection. Query expressions can be combined with SpEL expressions to create dynamic queries at runtime.

Aggregations

The repository layer offers means to interact with the aggregation framework via annotated repository query methods. The aggregation pipeline is expressed via the **@Aggregation** annotation in front of custom methods. **@Meta** annotation can be used to express additional options, such as a maximum run time, additional log comments, or the permission to temporarily write data to disk.

Transactions

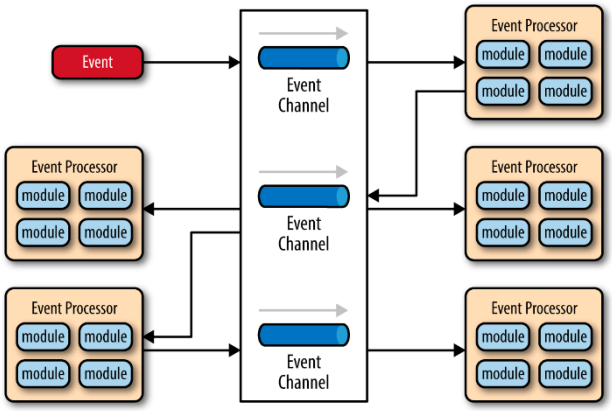
All write operations in MongoDB are **atomic** on the level of a **single document**, even if the operation modifies multiple embedded documents within it. When a single write operation modifies more than one documents, each single modification is atomic, but **the operation as a whole is not**. Starting from version 4.2, MongoDB fully supports multi-document transactions, even on replicated and sharded clusters.

Message Brokers

Event driven architectures

The HTTP protocol mostly assumes a **synchronous interaction:**

- The request sender (client) expects a response to be generated by the server in the short time
- This creates a strong (temporal) coupling between the two roles



Choosing an asynchronous web framework **does not mutate this paradigm**. The fact that the client and/or the server is based on a reactive implementation is just a detail and still requires that a response is to be produced.

An alternative approach to communications is provided by **event driven architectures.** They are based on systems that interact by **producing** and **consuming events**, stored in a **channel** operated by a broker. This **removes the temporal dependency**, allowing one system to be temporarily down and yet be able to later process received events notifications and react accordingly.

Events

Events are **facts that happen** in the system: an incoming request, a state change, an amount of time that has elapsed, and so on. They are represented by **data structures** which may be **transferred** from one sub-system into another one. These data representations are named **messages.** A sub-system may operate as a **message producer** or a **message consumer** (oftentimes, it operates in **both ways**).

- A **producer** is a component detecting an event and sending the corresponding notification
- A **consumer** reacts to the notification, triggering further processing

Events vs commands

A message is the **unit of communication** for event-driven architectures. However, special attention must be paid not to confuse events with commands. While both can be represented with messages, their semantics is different.

Event	Command
Communicate that something happened (in the past!)	Request something to happen (in the future!)
Has one logical owner	Has one logical owner
Should be published by the logical owner	Should be sent to the logical owner
Cannot be sent	Cannot be published
Can be subscribed to and unsubscribed from	Cannot be subscribed to or unsubscribed from

Message brokers

A message broker is an **infrastructural component** responsible of **validating**, **transforming**, and **routing messages** among applications. Applications interact with a broker sending and receiving messages to/from **message queues**. The broker is responsible to **store the messages** and **deliver them** to connected consumers. If no consumer is currently available, messages are **retained and delivered upon connection** of a consumer.

Brokers may support **two approaches** to message delivery

- **Publish/Subscribe**: messages are sent to a specific channel by the producer; many consumers may subscribe to the channels, and they all get a copy of the incoming messages, which are then discarded: events cannot be replayed after having been received and novel subscribers will not get past events.
- **Event Streaming**: messages are appended to a log, in strict time order; consumers can read from any part of the stream, thus they may replay past events.

Brokers can operate with a **push- or pull-strategy:**

- **Push-based** brokers actively forwards incoming messages to subscribers, forwarding them as fast as possible (*RabbitMQ*)
- **Pull-based** ones let subscribers pull for new messages and fetch them at their own rate (*Apache Kafka*)

Consuming strategies

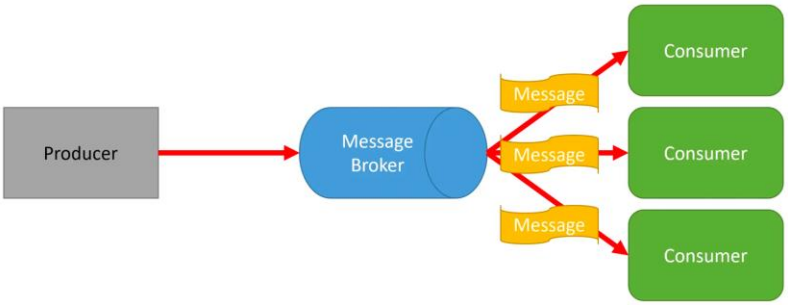
Brokers offers two types of **consumer policies:**

- **cooperating** – each message is delivered to all connected consumers, thus allowing collaboration among subscribers letting them share a common view of the world
- **point-to-point** – each message is delivered to only one subscriber, even if more than one is connected, thus favoring competition among consumers

Commands and events in message brokers

A **command** will have exactly **one consumer**: it is responsible of executing the requested action. In order to deal with command messages, a message broker must be **instructed** to send them to only one consumer.

An event will have **zero or more consumers:** when a producer publishes an event to a message broker, it will deliver it to all subscribed consumers. The publisher is unaware of the number and identity of the currently subscribed consumers.



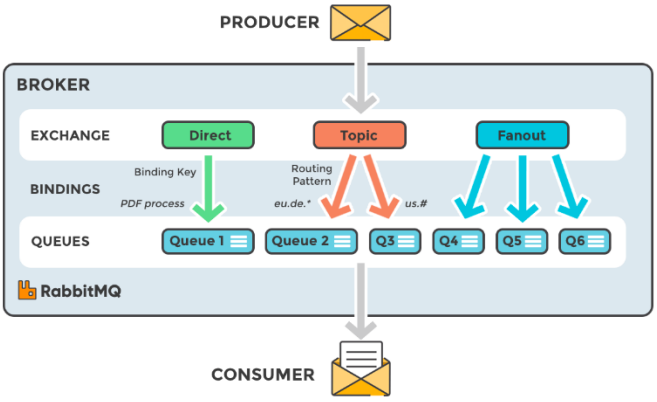
RabbitMQ

Popular open-source message broker based on the **Advanced Message Queuing Protocol** (AMQP). It can also support other protocols like STOMP, MQTT and WebSockets. Internally it is based on **exchanges** and **queues:**

- Producers send messages to an exchange, possibly adding a routing key
- The exchange applies routing rules and determines which queue should receive a copy of the incoming messages
- Messages may contain metadata (headers) that the broker can use to direct the routing algorithm

When a message is delivered to a consumer, this can **acknowledge** its reception and the broker will then **drop** the message.

Several types of **exchanges** are supported:



- **Direct exchange** - it forwards messages to the queues whose binding keys exactly matches the routing key of the message
- **Fanout exchange** - it broadcasts the message to all the matching queues
- **Topic exchange** - this exchange is connected to a set of queues via binding keys that must be a list of dot-delimited words: two special wild-chars are allowed ('*' and '#'), matching, respectively, a single word or a sequence of zero or more words
- **Header's exchange** - it bases its routing decisions on message headers rather than routing key

Using RabbitMQ in Spring Boot

Spring Boot fully supports RabbitMQ via the Spring AMQP project. It facilitates the management of AMQP resources providing the AmqpTemplate bean as a high-level abstraction for sending and receiving messages. Moreover, it supports several annotations to decorate components with extra configuration information. By default, it tries to connect to a broker running on localhost:5672. Default connection parameters are guest/guest. A **CachingConnectionFactory** bean can be defined to customize this behaviour.

Queues may be defined declaring beans of class org.springframework.amqp.core.Queue. The constructor accepts up to 4 parameters:

- **name** - a string defining the name of the queue (mandatory)
- **durable** - a Boolean stating whether the queue must survive server restarts
- **exclusive** - a Boolean stating whether the queue is for exclusive use of the current connection
- **autoDelete** - a Boolean stating whether the broker should delete the queue when it is no longer in use

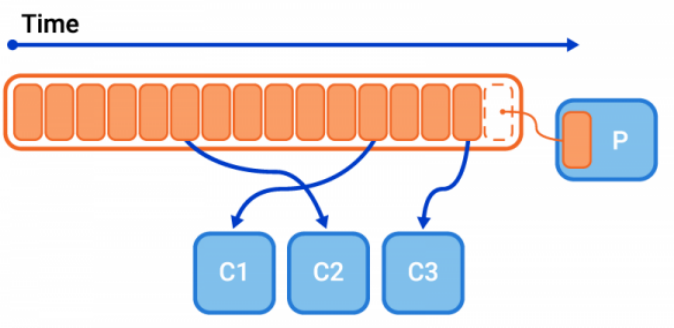
Interface AmqpTemplate describes basic operations for interacting with the broker. It provides synchronous send and receive methods

Apache Kafka

A highly **scalable and fault tolerant event streaming** platform based on a cluster of servers. It is designed to support mission critical applications and capable to be run on bare-metal hardware, VMs, containers, both on-premises and in a cloud. It logically **manages a set of topics**, which organize and durably **store events** sent by applications. Each topic can have **zero or more producers** writing events to it, and **zero or more consumers** that subscribe to these events. Events are appended to a **topic** and can be read **as often as needed**. Event publication occurs inside a transaction.

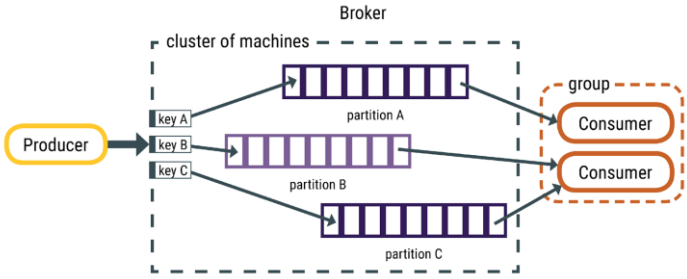
Topics

A Topic is a **category/feed name** to which messages are **stored** and **published.** It is organized as a **log file**, making the consumers responsible for tracking their position in the log.



Topics are divided into a number of **partitions**. When a message is sent to a topic, it is stored in **exactly one partition**. **Messages** are the elementary objects stored in a topic.

Each message has a **key** (a string or a byte[]), a **value** (a string or a byte[]), a **timestamp** and can be directed to a specific partition. Messages with the same key are stored in the same partition, keeping the arrival order.



Using Kafka in Spring Boot

Spring Boot supports Kafka integration, applying core Spring concepts to the development of a Kafka-based messaging solution. Class **KafkaTemplate** provides a high-level abstraction for sending messages. The *@KafkaListener* annotation can be used to label **message-driven components** that act as receivers of messages published to a given topic. **Custom message content** is supported, by serializing it to JSON format. Custom KafkaTemplate instances must be created in order to support serialization to a given data class.

Message brokers comparison

Tool	Apache Kafka	RabbitMQ
Message Ordering	Provides message ordering because of its partitions. Messages are sent to topics by message key.	Not supported.
Message Lifetime	Kafka is a log, which means that messages are always there. You can manage this by specifying a message retention policy.	RabbitMQ is a queue, so messages are done away with once consumed, and acknowledgment is provided.
Delivery Guarantees	Retains order only inside a partition. In a partition, Kafka guarantees that the whole batch of messages either fails or passes.	Doesn't guarantee atomicity, even in relation to transactions involving a single queue.
Message Priorities	N/A	In RabbitMQ, you can specify message priorities, and consumed message with high priority at the onset.

Microservices

Monolithic applications

It is the "Standard" web applications based on **three tiers**: front-end, back-end, persistence. It provides all functionalities in one single deployable unit. It is easy to design and to implement. Frameworks offer a lot of support, providing readymade solutions for most problems. Having all logic inside **one process** makes it easy to access relevant information whenever necessary and as long as requirements do not change, everything is nice.

Handling increasing complexity

If the application is successful, more features will probably be required. This will add more modules, more dependencies, …, more stress. Testing will become an issue, because of the entangled nature of the modules causing more **data path** and **business logic to be verified.** Deployment will suffer as well: each new feature causes the entire system to be stopped and re-deployed.

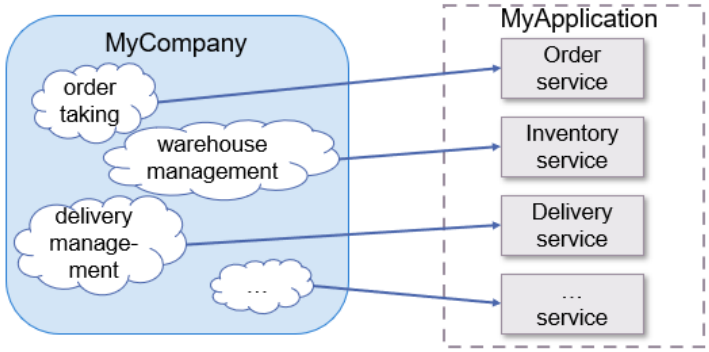
As time passes by, two forces will exert a strong pressure on the development team:

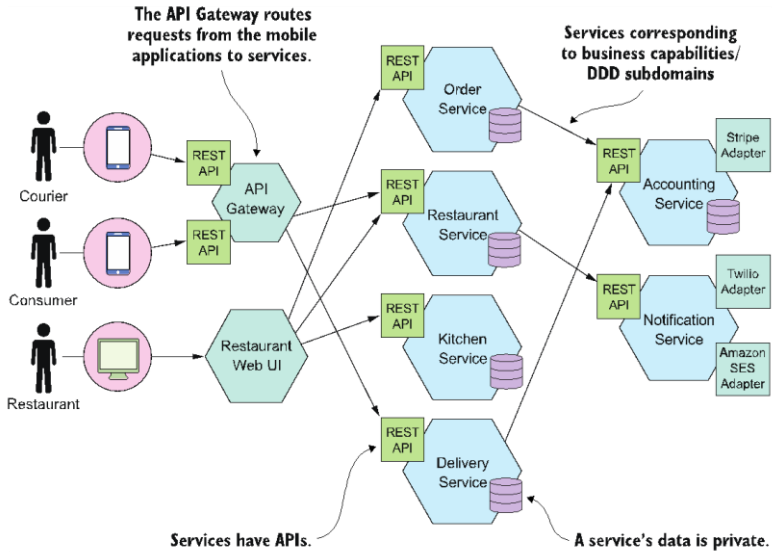
- **Obsolescence of the technology stack** (and growing maturity of the system designers) will call for a rewrite of the system – but this will hardly be possible, since it would probably require too many resources
- Market will ask for significant innovations in the product, to be delivered **rapidly, frequently, and reliably**

A possible solution to the problem is evolving the system into a **microservice-based** one. Beware, however, that this is going to introduce a high degree of complexity: do it only if there is no simpler alternative.

Microservice architecture

A monolithic application is decomposed into a set of loosely coupled services, organized around business capabilities, i.e., those areas where a given business delivers value to its users.



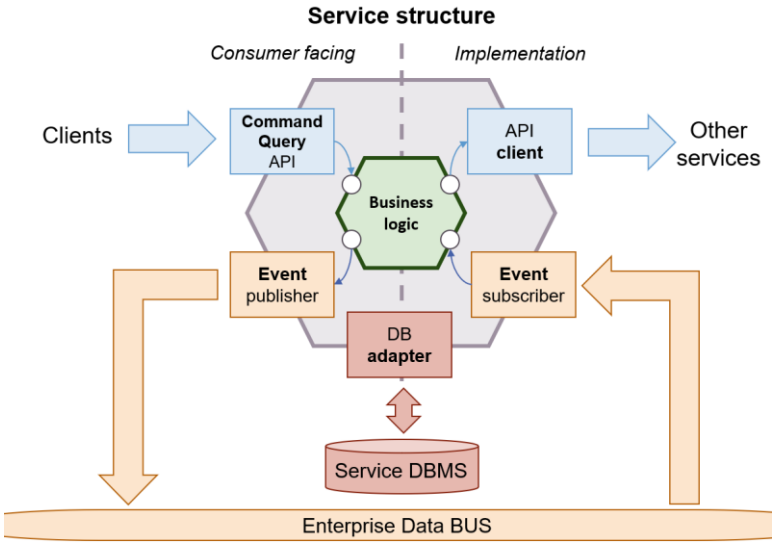


A service is an **independently deployable component** which offers a well-defined client facing interface (end-points, methods, data types, relationships, …) and conforms to a given Service Level Agreement (availability, scalability, resilience, …). A service has its own database for persisting data owned by the service and replicate data of other services, if needed. A service **may act as a client of other services** and can be connected to a **shared communication bus** where relevant events are published and subscribed.

Service features

- **Highly maintainable and testable**
  - Fast development cycles and easy deployment
- **Loosely coupled with other services**
  - Teams can work independently the majority of time on their service, without impacting other services or being impacted by their changes
- **Independently deployable**
  - Enables a team to deploy their service without having to coordinate with other teams
- **Capable of being developed by a small team**
  - Essential for high productivity by avoiding the high communication overhead of large teams

The hexagonal architecture



Service structure

A service exposes an API, made of operations and a set of published events. This is the contract that the service offers to the external world, and it should be kept as stable as possible.

The API can be implemented

- **Synchronously**, via REST, GraphQL, gRPC: better for external clients
- **Asynchronously**, based on message queues: better for internal clients

Services may **publish** and **consume** events. This informs other services that relevant action took place in the service. A **message broker** (Apache Kafka, RabbitMQ, ActiveMQ, …) is needed to provide message queues where event are published to/read from. The **business logic kernel** is the heart of the service: this is the layer that implements the API's operations and that publishes events. It consumes the APIs of other services and may subscribe to their events. The kernel **must be kept independent of any dependencies**: adapters are provided to handle connections with the external world (DBMS, message brokers, APIs, …).

Services have **their own database**: data contained therein is **private**. If two services need to collaborate, they **never access each other's database**, but connect via the provided APIs. This improves **runtime isolation**, preventing a service to hold a lock that would block other services.

Even if databases (i.e., schemas) are different, **there can be a single DBMS hosting them all,** provided all services can benefit from a single persistence technology

Multi services transactions require special attention: specific patterns (sagas) exist to support this kind of operations.

In spite of the name (micro-services), the size of the service does not matter. What matters is the fact that the service is an **independently deployable unit** and that it can be **scaled horizontally**, manually or automatically. Each service will have its code base with a source code repository and an **(automated) deployment pipeline**. Teams responsible for each service should be able to work independently most of the time.

In order to act as autonomous software component, a service must fulfil the following criteria:

- **Shared-nothing architecture**: services do not share data in databases with each other
- Communication happens only across **well-defined interfaces**: these may be synchronous (REST, GraphQL, gRPC, …) or asynchronous (message based)
- Message formats are **stable**, **well-documented** and **versioned**
- Deployed as **separate runtime processes**, preferably in a container
- **Stateless**: incoming requests can be handled by any of the microservice instances

Microservice rationale

Before being a technical choice, a microservice architecture is an **organizational matter**. The objective is to support business needs, moving fast and not breaking things.

	Top performers	Medium performers	Low performers
Deployment frequency how often is code deployed?	On demand	Once per week	Once per month
Lead time for changes how long does it take to go from commit to production?	< 1h	1d to 1w	> 1w
Mean time to recover how long does it take to restore a service?	< 1h	1h to 1d	1d to 1w
Change failure rate What percentage of changes require remediation?	0-15%	0-15%	15-45%

2017 state of dev-ops report

Microservices issues

- How to decompose an application into services?
- How to deploy an application service?
- How do services communicate?
- How to handle cross-cutting concerns?
- How to discover the address of a service?
- How to debug an application?
- How to support testing?
- How to implement a UI that displays data from multiple services?

Many small components that use synchronous communication can cause a chain of failure and **high loads** makes this problem worse. The configuration of system must be kept up to date, but the problem increases with the number of services and manual configuration can be daunting. Tracking requests across services is very complex especially if no shared log infrastructure exists.

Design patterns for microservices

A design pattern is a reusable **solution** to a **problem** occurring in a particular **context**. The structure of a pattern encourages objectivity. Each pattern is described putting in evidence

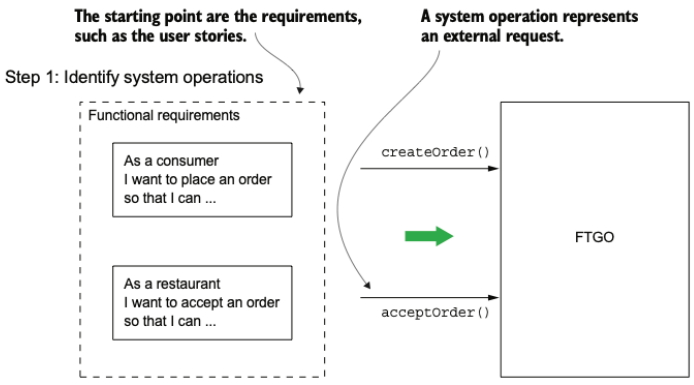
- The problem
- A solution
- Requirements for the solution (the context)

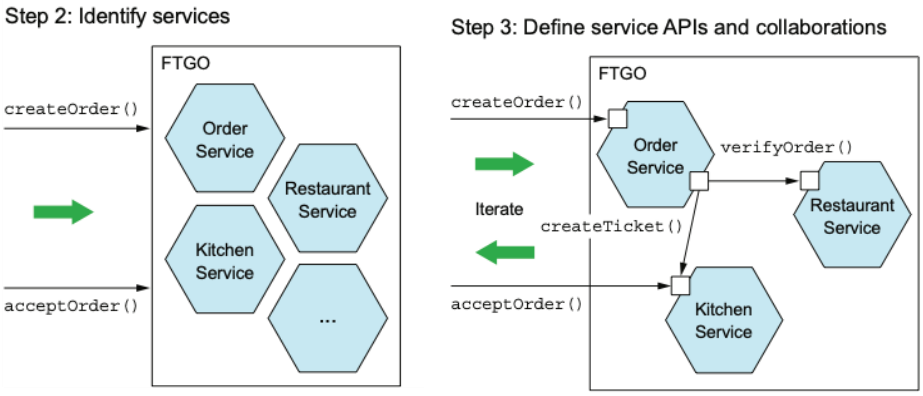
Patterns assume a landscape of cooperating microservices where communication can be both synchronous (via HTTP) or asynchronous (via a message broker).

Decomposing an application

The process that leads to defining a microservice infrastructure stems from the application requirements

- The first step requires **identifying system operations**: these are commonly depicted as user stories that narrates how the different users of the system interact with it issuing commands or performing queries
- The second step **decomposes the system into services**: these are organized around business concepts rather than technical ones
- The third step **determines each service's API**: a service may implement an operation by itself, or it might require support from other services





API composition

The consequences of the decomposition are that a given request coming from the client may translate into **several sub-requests directed to other services** that need to be recomposed into a single response. This introduces various operational risks:

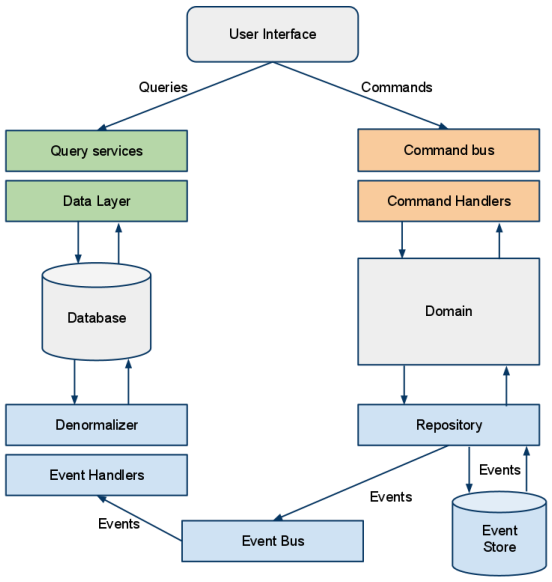
- **Network latency** - round trip delays add up and may result in unacceptable response time
- **Reduced availability** - synchronous communication requires both parties to be alive at the same time
- **Consistency problems** - often, data managed by different services need to be updated transactionally as a consequence of a single request
- Getting consistent views of data spanning across several services contrasts decomposition

The CQRS pattern

Sometimes, system have very different usage patterns: few updates that need to provide transactional guarantees (ACID) and large number of read operations. One possible approach is based on partitioning commands and queries:

- **CQRS – Command Query Resource Segregation**
  - All commands (write operations) are directed to a system that implements the business logic and updates the master data copy
  - All reads are directed to a system that operates on a slave copy of the data

Master and slave data set are **kept in sync by an asynchronous process** based on a **message broker.**



Master and slave data representation can be different:

- Materialized views (with duplicated data) can be used in place of the normalized representation adopted in the master DBMS, if this can be convenient in providing faster responses to the clients

Change propagation needs to be **transactional**

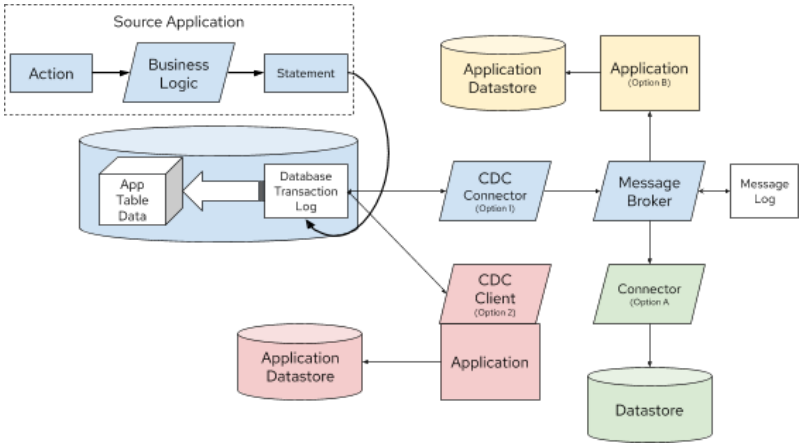
- Since master and slaves can fail independently of each other and of any infrastructural component, care must be taken to guarantee that all changes occurring to the master copy are properly (and sequentially) propagated to all the slave ones
- Message exchange must guarantee **idempotency** (duplicate reception of the same message should cause no harm to the system)

Approaches to transactionality

How to reliably/atomically update the database and send messages/events? If the database transaction commits messages must be sent. Conversely, if the database rolls back, the messages must not be sent. One approach is based on the **2-phase commit protocol:**

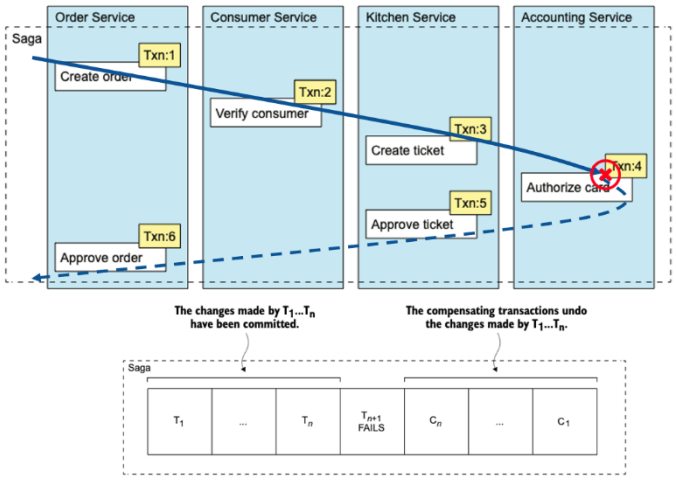
- Two messages are sent: one before writing to the DBMS, the other after it commits
- The receiver only reacts to the second one
- If the sender fails, when in resumes, it must read back sent messages, looking for begin messages not followed by the corresponding commit and re-execute the DB operation, publishing the corresponding commit event

Every change to the state of an application is captured in an **event object** describing it. Events are **securely stored in sequence**. Most DBMS already offer a **stream of all write operations** performed to their data via the **Change Data Capture** (CDC) hook. By observing the Write-ahead transaction journal it is possible to replicate the state of the DB in several ways



Sagas

When a command causes a change to happen in several services, guaranteeing a transactional behavior, a saga can be implemented. It consists of a **sequence of local transactions**. Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries. The completion of a local transaction triggers the execution of the next local transaction. In case of failure of a local transaction, the **saga is interrupted,** and a set of **compensating transactions** need to be executed to revert the transactions committed so far.



Implementing sagas

Sagas can be implemented using two alternative approaches:

- **Choreography** - distributes the decision making and sequencing among the saga participants that communicate by exchanging events
- **Orchestration** - one service is responsible to act as a coordinator, sending commands to saga participants and collecting their responses

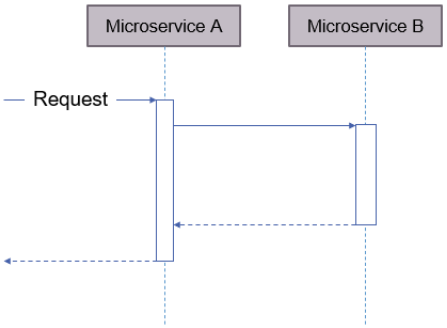
Whichever the approach, sagas requires that operations happen in a **transactional way**:

- Care must be taken to update the local database and publish the resulting event atomically
- Failure to do so will cause data loss or incoherences if a service fails abruptly when managing a message
- The transactional outbox pattern provided by Debezium helps in this case, too

Spring Cloud

Microservice architectures

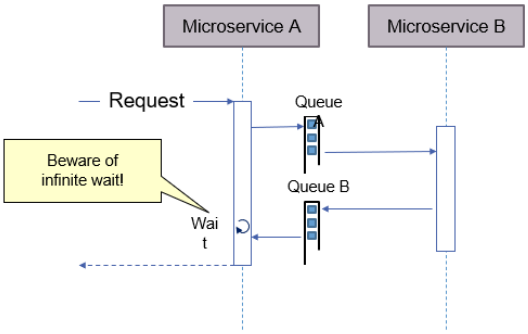
A microservice system can be implemented according two different approaches: **synchronous** or **asynchronous**. A microservice is **synchronous** if it makes a request to other microservices while processing requests and waits for the result in order to provide a full response. It is **asynchronous**, if the incoming request only returns an indication that it was accepted/rejected, and providing a way of later informing the requester of the outcome of the operation (via events or polling)



Synchronous microservices

A synchronous microservice system may rely on both synchronous and asynchronous protocols to implement its behaviour:

- REST/JSON HTTP and gRPC are obvious candidates when synchronous protocols are used
- Synchronous communication with an asynchronous protocol occurs when one system sends a message to another system and then waits to receive a response, in the form of another message



Advantages

Synchronous communication is a **natural approach** if the system is to offer an API. Any microservice can implement part of the API. It can be easier to migrate into such an architecture. In a **monolith**, communications are already synchronous, and developers are familiar with this model.

Challenges

- **Low performance**: latencies due to network communication and processing times add up
- **Failure propagation:**if the inner microservice fails, the calling one can fail as well, making the system very vulnerable
- **Higher level of dependency**: while asynchronous architecture is based on events and on reactions to them, synchronous architectures need to define what a microservice must do per each request, possibly invoking other microservices

To cope with these challenges, a set of architectural concerns must be addressed

Service discovery

In a distributed architecture, services must be able to communicate with each other. This requires having a **known protocol** (e.g., a set of APIs) and an **end point that implements that protocol**. The former is defined when the system is designed, the latter depends on run-time configuration.

Service discovery serves the purpose of letting a service find the port and IP address of another service, given its name. This should be dynamic, since services may be scaled up (new instances may become available) and can fail.

Resilience

When communication is synchronous, microservices must be prepared for the failure of other microservices. It has to be prevented that the calling microservice fails as well. Otherwise, there will be a failure cascade:

- First one microservice fails
- Then, other microservices call this microservice and fail as well
- In the end, the entire system will be down

Therefore, there must be a technical solution to achieve **resilience.**

Load balancing and routing

Each microservice should be **scalable independently** of the other microservices. Load must be distributed between microservices. This does not only pertain to access from the outside, but also to internal communication. Therefore, there has to be a **load balancing** for each microservice. Every access from the outside should be forwarded to the responsible microservice and this requires routing.

API gateways

Front services hiding the complexity of the microservices forest, offering a centralized solution for a set of features:

- User authentication
- Centralized logging
- Caching
- Monitoring
- Documentation
- Mocking

Spring Cloud

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems. Distributed/versioned configuration, service registration and discovery, routing, service-to-service calls, load balancing, circuit breakers, global locks, leadership election and cluster state, distributed messaging. It consists of more than 20 subprojects, addressing specific functionalities and patterns and it is fully integrated with Spring Boot.

Eureka service discovery

Although it is possible to rely on DNS for resolving logical names to IP addresses, several practical limitations exist:

- DNS clients cache the resolved IP addresses and hangs to the first result
- Neither the DNS server nor the DNS protocol is well-suited for handling volatile microservices instances that come and go
- Some microservice instances might take some time to start up, and traffic should not route to them until they are fully running
- Unintended network partitioning can occur at any time

Spring Cloud offers (among other solutions) **Netflix Eureka** as a discovery service. This is a provided as an annotation (*@EnableEurekaServer*) to be added to an otherwise empty Spring Boot module. A dashboard is published on the host running the server at the URL *http//eureka_host:8761*. A proper configuration file must be provided.

Registering services with Eureka

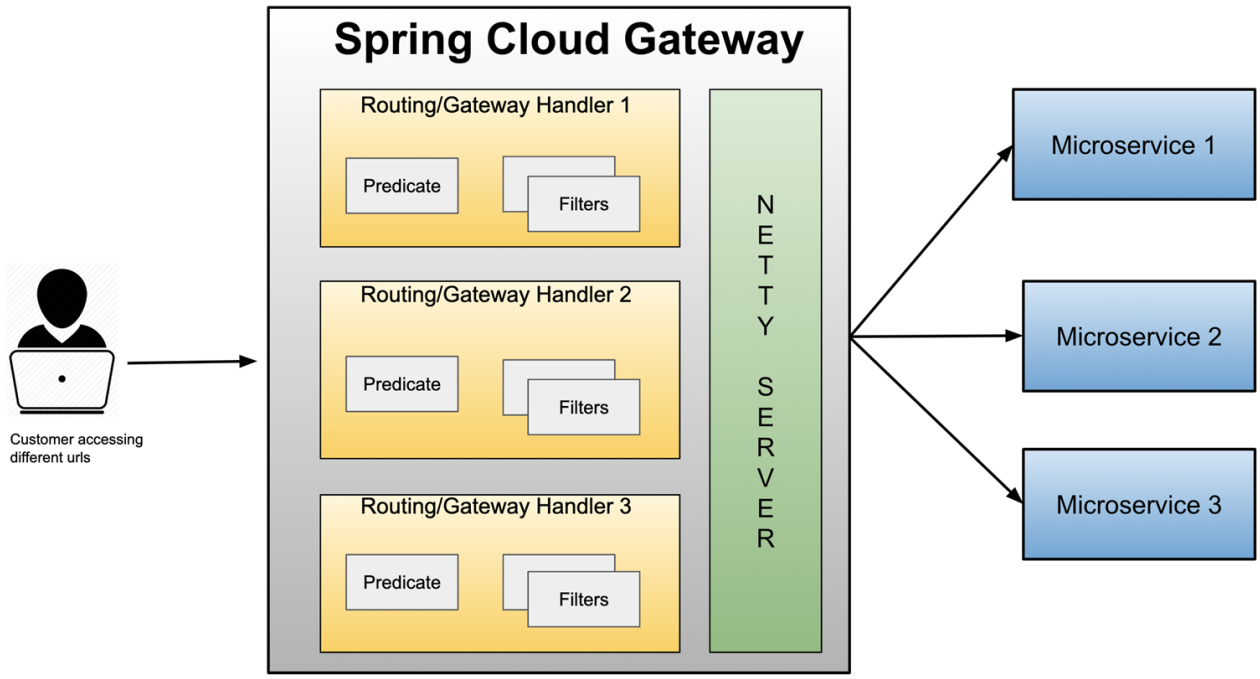
Microservices can automatically register with the Eureka server by including the @EnableEurekaClient annotations. Each microservice should specify in its configuration file a distinctive value for the spring.application.name key: this will be used to locate it in the registry.

- There are parameters for the Eureka server
  - Prefixed with eureka.server
- There are parameters for Eureka clients
  - Prefixed with eureka.client
  - This is for clients who want to communicate with a Eureka server.
- There are parameters for Eureka instances
  - Prefixed with eureka.instance
  - This is for the microservices instances that want to register themselves in the Eureka server

Microservices not based on Spring Boot must use a library to access Eureka. Basically, access is possible in every programming language since Eureka offers a REST interface to access its functionalities.

Spring Cloud Gateway

The Spring Cloud Gateway uses routes to forward requests to downstream services. It provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency. It is based on **WebFlux**: as a consequence, any access to a database or to security functionalities need to be based on the corresponding asynchronous versions of the Spring library.



The basic building block of a gateway is the **Route**: it consists of an ID, a destination URI, a collection of predicates and a collection of filters. A route is matched if the aggregate predicate is **true**.

A **predicate** is a **function** that maps a ServerWebExchange object to a boolean value: this allows to match on anything from the HTTP request, such as headers or parameters. A filter is an instance of GatewayFilter and it allows to modify requests and responses before or after sending the downstream request.

When a client makes a request to a Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the filters, that may manipulate it. Routes can be created in two ways:

- **Programmatically**, customizing a RouteLocator bean
- Via **configuration**, in the application.yml file

Routes can be selected based on almost all request contents:

- Request time (after a given date, before a given date, in a given interval)
- Cookies (matching the name as a string and the value as a regex)
- Headers (as cookie)
- Host name (expressed as a set of alternative regexs)
- Method
- Path
- Query string
- Remote address
- Random resolution to a set of destinations given a weight per each of them

Many kinds of request manipulation are provided by filters:

- Adding/removing/transforming request/response headers
- Adding/removing request parameters
- Removing duplicate response headers
- Adding/removing a prefix path to the forwarded request
- Redirect to a different URI or manipulating a redirection from the invoked service
- Modify the request/ response body by applying a function

When integrated with Resilience4j the following actions are supported, too:

- Falling back to a different URI if the invocation fails or a given status code is returned
- Limiting the request rate

Resilience4j

It is the implementation of a circuit breaker aimed at providing resilience patterns:

- **Circuit Breaker** – causing request not to be forwarded any more in case of failure of a given service
- **Retry** – automatically retries the connection, inserting a given delay, up to a maximum number of attempts
- **Threadpool Bulkhead** – limiting the number of concurrent invocations via a thread-pool
- **Semaphore Bulkhead** – similar to above, but based on a semaphore
- **TimeLimiter** – setting a maximum time to complete calls
- **RateLimiter** – delaying further incoming requests if the number of pending ones is larger than a threshold in a given period

Configuring Resilience4j

Two alternative implementations are available (reactive and blocking):

- org.springframework.cloud: spring-cloud-starter-circuitbreaker-reactor-resilience4j
- org.springframework.cloud: spring-cloud-starter-circuitbreaker-resilience4j

In order to operate, a configuration bean must be provided that Implements the Customizer<{Reactive}Resilience4JCircuitBreakerFactory> class.

GraphQL

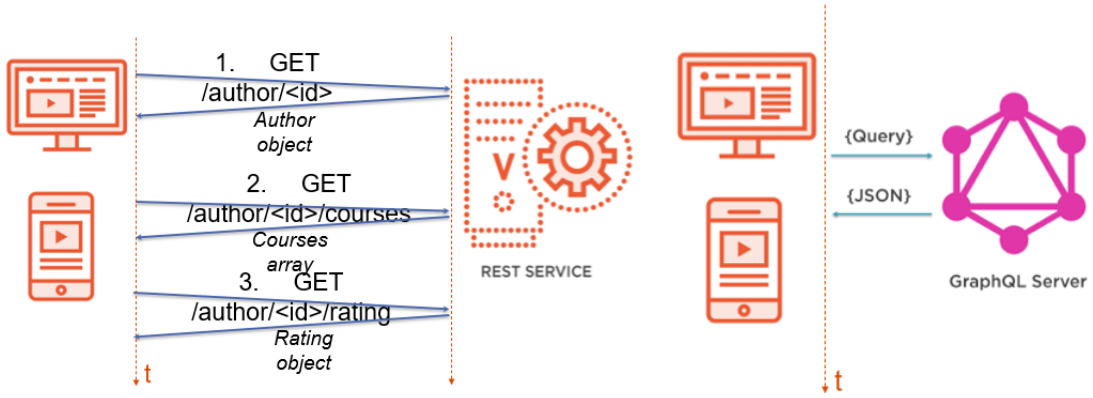
It is a query language for APIs as well as a runtime for fulfilling those queries with actual data. It is based on an explicit schema declaration, that defines the set of types that can be exchanged:

- Used, client side, to customize a request
- Used, server side, to validate the request and implement the response generation

It has been created by Facebook in 2012 in order to get more flexibility and efficiency in client-server communications. Client and server implementations are available in many programming languages.

Features:

- The client can ask **exactly the data it needs** and nothing more
- A single GraphQL query can return **all data** required by a given view
- **Language agnostic**: many client and server libraries are available for almost any language



GraphQL Query

```
{
  author (id: 2100) {
    name
    courses {
      title
    }
    rating
    topics (last: 3){
      name
    }
  }
}
```

JSON Response

```
{
  "data": {
    "author": {
      "name": "John Smith",
      "courses": [
        {"title": "GraphQL: The Big Picture"},
        {"title": "ReactNative: The Big Picture"}
      ],
      "rating": 4.5,
      "topics": [
        {"name": "GraphQL"},
        {"name": "React Native"}
      ]
    }
  }
}
```

REST	GraphQL

• Architecture
  ◦ Server driven
• Operations
  ◦ GET, POST, PUT, DELETE
• Structure
  ◦ End-points
	• Architecture
	  ◦ Client driven
	• Operations
	  ◦ Query, Mutation, Subscription
	• Structure
	  ◦ Schema and Type System

REST

- Multiple round trips to collect information stored in different resources
- Over Fetching and Under Fetching
- Frontend teams depend on backend teams manage APIs
- Resource caching automatically provided by the HTTP layer

GraphQL

- One single request to collect all aggregated data
- Only needed fields are fetched
- Frontend and backend teams can work independently
- Client libraries need to explicitly manage data caching

The strength points of GraphQL:

Increases team productivity

Rapid development

Improves web and mobile performances

Lower costs of both coding and testing

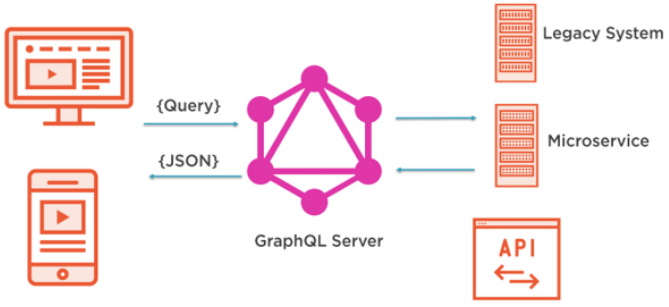
GraphQL weakness points:

Steep learning curve

Does not support file uploading

Smaller development community

Integration with existing APIs



GraphQL is strongly typed

GraphQL is a strongly typed language. Its schema defines the types of the objects that are exchanged and the operations that can be invoked. The schema is used to define a collection of types and the relationships between them:

- The GraphQL server uses the schema to define and describe the shape of the data graph
- The schema establishes the hierarchy of types with fields, exposing the functionality available for the client applications to use

The GraphQL schema language, is used to define a schema and types.

GraphQL schema

The schema is a model of the data that can be retrieved through the GraphQL server. It specifies:

- what operations clients are allowed to make
- what types (scalar, object, enum) of data can be fetched from the server
- what are the relationships between these types

GraphQL operations can be divided into three categories:

- **Queries** are used for data fetching
- **Mutations** are used to modify server-side data (create, delete, update)
- **Subscriptions** are used to observe server-side data and get an updated version of it whenever it changes

Each operation returns a result that has a specified data type.

GraphQL types

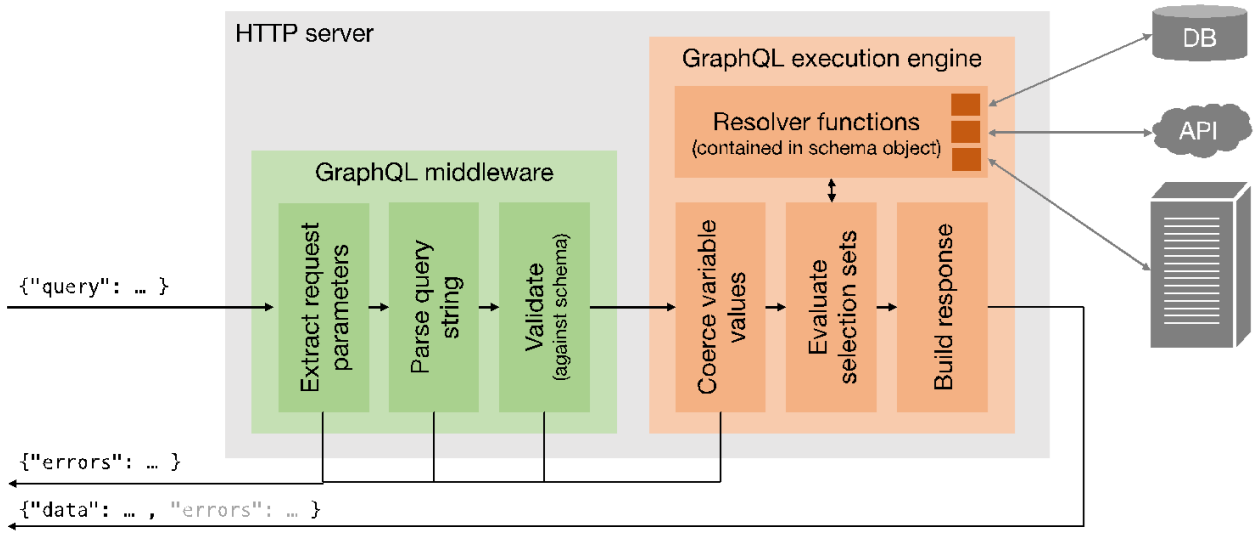
A GraphQL schema can define/use the following kinds of types:

- Scalars: represent common primitive types and do not require any definition.
- Objects: An object type describe something that can be fetched, listing all the fields it provides with their constraints / variations
    - Fields have a name (unique in the context of the type) and a type
    - Required fields have their type followed by "!" (they cannot be null)
    - When a field contains an array of values, the value type is enclosed in [ ]
- Enumerations: a type of scalar restricted to a set of pre-defined values and introduced by the enum keyword
- Interfaces: abstract type that includes a set of fields that must be implemented by another type
    - Interface types are used when returning an object that can be one of several types (duck typing)
- Unions: similarly to interface types, they can be used to model alternative results. However, union types do not specify common fields
- **Queries:** the query type defines the queries (read operations) exposed by a given GraphQL service.
    - Queries consist of a name, that selects the operation to be performed, an optional list of parameters, that provides context to the operation, and a result type. Parameters have a type that can be a scalar, enum, or Input type.
- **Mutation:** they define an optional set of operations that cause a modification on the data server-side. Like queries, mutations have a parameter list and a return type.
- **Subscriptions**

Fragments

A fragment is a reusable set of fields that can be defined and referenced by name in a query. Fragments **avoid** repetition in queries by providing a way to reference a set of fields by name. To apply a fragment inside a query, the fragment spread operator (...) is used inside the selection set.

Implementing GraphQL



GraphQL in Spring Boot

Spring for GraphQL provides support for Spring applications built on GraphQL Java. By providing a suitable HTTP adapter, it works both with WebMVC and with WebFlux projects. By default, it provides a */graphql* POST endpoint that always return a successful "*application/graphql+json*" response:

- Possibly encapsulating any error condition in the content body, as specified by the proposed GraphQL over HTTP specification
- Requests and responses may also be exchanged via WebSockets and RSockets

Defining the schema

By default, files located in the resources/graphql folder having ".gqls" extension are considered as parts of a federated schema. Spring Boot automatically loads and parses them, deriving a set of constraints and automatic implementations of the GraphQL endpoint.

This can be overridden by properly defining and configuring a **GraphQlSource** bean. This control the behaviour related to request execution and it provides a builder API that simplifies its creation.

Spring Boot provides the following default features:

- Loading schema files from a configurable location
- Exposing properties that apply to GraphQlSource.Builder
- Detecting RuntimeWiringConfigurer beans
- Detecting Instrumentation beans for GraphQL metrics
- Detecting DataFetcherExceptionResolver beans for exception resolution

Implementing annotated controllers

A Spring @Controller component can have its methods annotated with @QueryMapping:

- The label a method as an implementation of the corresponding field of the Query type as expressed in the GraphQL schema
- The returned value should match the type defined in the schema
- Internally, the annotation marks the method as a graphql.schema.DataFetcher for the given query

Methods annotated with @MutationMapping and @SubscriptionMapping are automatically used to resolve mutations and subscriptions: @SchemaMapping is the most general annotation that can be applied to a specific typeName.

Controller methods accept parameters with a flexible signature:

- @Argument allows to map an argument of the schema into a parameter injected into the method; it can label a scalar value, a Map<String,Any> or a List<Any>
- @Arguments