Java 201 - Software Architecture

Loïc Ledoyen

Java 201

Introduction

> Architecture is about the important stuff.
> Whatever that is.
>
> — Ralph Johnson

> It means that the heart of thinking architecturally about software is to decide what is important, (i.e. what is architectural), and then expend energy on keeping those architectural elements in good condition.
> For a developer to become an architect, they need to be able to recognize what elements are important, recognizing what elements are likely to result in serious problems should they not be controlled.
>
> — Martin Fowler

A piece of software tries to achieve several objectives depending on the context.
Among these objectives, we encounter:

- Maintainability
- Testability
- Reliability / Resilience
- Scalability
- Security
- Performance
- etc.

Software architecture is the technical means used to meet the objectives set for a particular project while complying with constraints.

As all the objectives cannot be met in one development iteration, software architecture is the ideal compromise in regard to project priorities.

Constraints can be:

- Short term realisation / deadline
- Limited funding
- Security compliance
- Connection to a third party software using specific technology / protocol
- etc.

Priorities must be decided by all stakeholders, and periodically adjusted to follow the project evolution.

It is a partnership.

The development team uses architecture principles to reach projects objectives in the best way.

1. Tools recap

1.1. Git

Git is a decentralized SCM (**S**ource **C**ode **M**anagement tool).

Decentralized means that they exists multiple remotes instances of the same repository on different servers.
Typically, in the open-source community, when someone external to a project wants to contribute, this person starts by creating a copy of the repository (fork), work on this instance, then offer a submission (pull or merge request) to the project's owners.

Git is broadly used today, and succeed to other SCM tools (CVS, SVN, Mercurial, etc.).
These various tools offer the same ability to historize differentials, to allow to restore a previous version, or work on a new version in parallel, which will be later on merged on the main branch.

Here is the vocabulary in use:

- **commit** : a revision / version containing code modifications
- **branch** : un thread of commits
- **tag** : alias for a specific commit, ofter used to mark an *applicative version* (1.0.25 for example)
- **merge** : merge of one branch onto another
- **checkout** : retrieve locally the code from a remote server in a specific version

1.1.1. Some useful commands

- Initialize a repository
  - `git clone <url>` : copy locally a remote repository
  - Ou `git init` : turn the current directory into a local repository. A default remote repository can later be attached using `git remote add origin <url>`.
- Update

    - `git fetch --all --prune` : retrieve the Git database latest version
    - `git pull` : on the current branch, retrieve the remote changes (merge or rebase, depending on the configuration)
    - `git rebase origin/<current-branch>` : on the current branch, move local commits after those having been pushed on the remote repository
- Switch branches
    - `git checkout <branch>` : set the working directory files in the latest version of `<branch>`
    - `git branch -b <branch>` : create a new branch named `<branch>` with the starting point being the latest commit
- Display changes
    - `git status` : display indexed (green) and not-indexed (red) local changes along with commit differences with the default remote
    - `git log --oneline -n 15` : display the last 15 commits of the current branch (including their hashes)
    - `git diff --stat` : display a summary of local changes
    - `git diff --word-diff=color <file>` : display the the local changes of the file `<file>`
- Introduce changes
    - `git add <file>` : add the file `<file>` to Git index
    - `git add .` : add all (recursively) changed files to Git index
    - `git reset <file>` : remove file `<file>` from Git index
    - `git commit -m "<title>"` : create a new commit with all indexed changes with the title `<title>`
    - `git commit --amend --no-edit` : include all indexed changes in the latest commit
    - `git commit --fixup <hash>` : create a new commit with all indexed changes, "tagged" as *fix*, targeting an existing commit of ID `<hash>`
    - `git rebase -i --autosquash <hash>` : start an interactive rebase until the commit of ID `<hash>` excluded, and moves *fixup* commits right after targetted commits, and mark them for merge (fixup)

Source : https://git-scm.com/docs

1.1.2. Windows users

Git is aware of files execution bit (`chmod +x`).
As **Windows** does not handle file permission Unix does, it is recommended to disable this awareness with `git config core.fileMode false`.
To explicitly set a file as executable from Git perspective: `git update-index --chmod=+x <file>`

1.2. Git concepts

Remotes are git repositories located in remote servers.

The default remote is called **origin**.
When a local repository has been cloned (hence, not initialized), **origin** targets the URL used at the time of the clone operation.

Git stores its data in a specific directory (**.git**), which contains the graph of all revisions of all branches, local and remote.
Remote branches are available through their local name: <**remote_name**>/<**branch_name**>.
For example, **origin/main** is the branch **main** as seen by the **origin** server when the last synchronisation (`git fetch`) was done.
It is quite usual to have one (and one only) local version of a branch, and one more remote branches with different versions.
It is only when a push will be done to a specific remote that these versions will match.
Deleting the **.git** directory will permanently erase all revisions that have not been pushed to a remote.

The **working copy** is all the files in the local repository (expect for the **.git** directory).
Modification, creation or deletion of these files can be done without alter revisions known to Git.
In any case, Git can restore the previous state of the working copy as long as this state is recorded as a commit.

In order to *commit* modifications of the **working copy**, they must be indexed.
The **index** allows to select only some files (and not others) when creating a commit.
Files can be added, with the `add` command, or removed, with the `reset` command.
The `status` command displays in different colors current changes, in green the indexed files, in red, the others.
Performing a commit will consider all files displayed in green.

1.2.1. Rebase

The rebase is one of the feature that place Git way ahead of its predecessors.

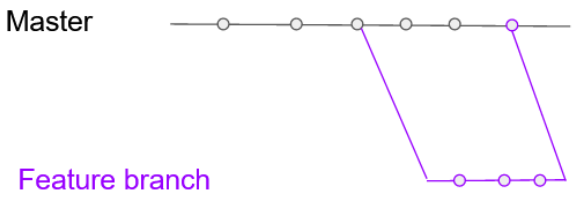
It allows to update a branch with its remote state without complicated merge manual actions, as long as there no conflict requiring human oversight.

```
git fetch --all --prune ①
git log --one-line -n 10 ②
git rebase origin/main ③
```

① Get the latest version of all branches of the default remote (**origin**)

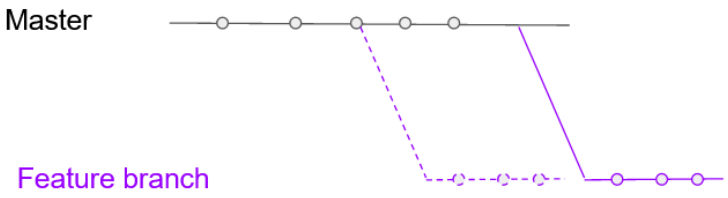
② Display the last 10 commits of the current branch

③ Modify the history of the current branch by moving commits done after the base after the last ones pushed on the `main` branch of the remote **origin**



Merge :

- conflict resolution at the last minute
- additional merge commit



Rebase :

- conflict resolution as they arise
- optional merge (when the branch is in front of the remote one)

The **rebase** command can also be used interactively to alter the local history of the current branch:

- Add changes in an exiting commit
- Change the name or description of a commit
- Merge several commits into one
- Delete commits (discarding changes)
- Change commits order

ADVICE	Do not use rebase on a branch shared by more than one person. Especially the `main` branch.

1.3. Maven

Maven is a build automation tool, for JVM-based projects.

It handles, among other functionalities:

- Dependency management
- Source code compilation
- Tests execution
- Documentation generation
- Packaging of binaries

It's plugin-based system allows to adapt to various languages (Java, Scala, Kotlin, etc.) as well as different contexts (Continuous Integration, code generation, deployment, etc.).

1.3.1. Project structure

A Maven project is organized by convention (over configuration) to avoid re-define standard parts, such as: source code, tests, etc.

A the root of a Maven project, we found:

- A pom.xml file which contains all information needed for Maven to build the project. Its minimal content is as follow:

Listing 1. File pom.xml

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId> ①
    <artifactId>my-app</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <properties> ②
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

① The tuple `groupId`, `artifactId` et `version` are the coordinates which identifies one Maven project and allows referencing it from others.

② Optional section, here fixing the encoding and Java version to avoid conflicts later on

- A **src** directory which will contains all files we want to keep in the SCM
  - In **src**, there is standardly two directories: **main** et **test** which respectively contain production code, and test code (code not included in binaries produced during the ***packaging*** phase)
    - In each of these two directories, a directory named after the language used exists, in this exemple: **java**, but can be **groovy**, **kotlin**, etc.
      - Finally, in these directories, we put the code. This code is organized in packages, being themselves composed of directories

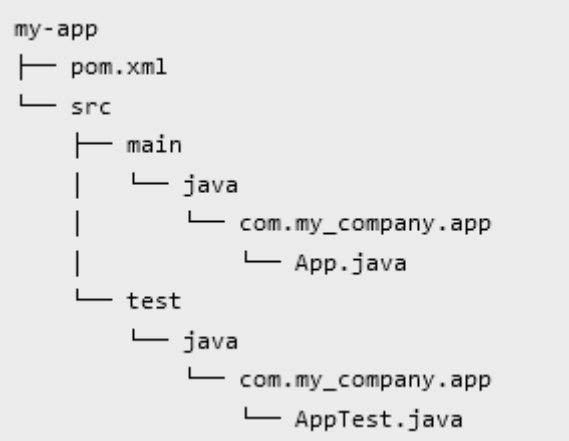


Figure 1. Structure of a Maven project

1.3.2. Lifecycle of a Maven project

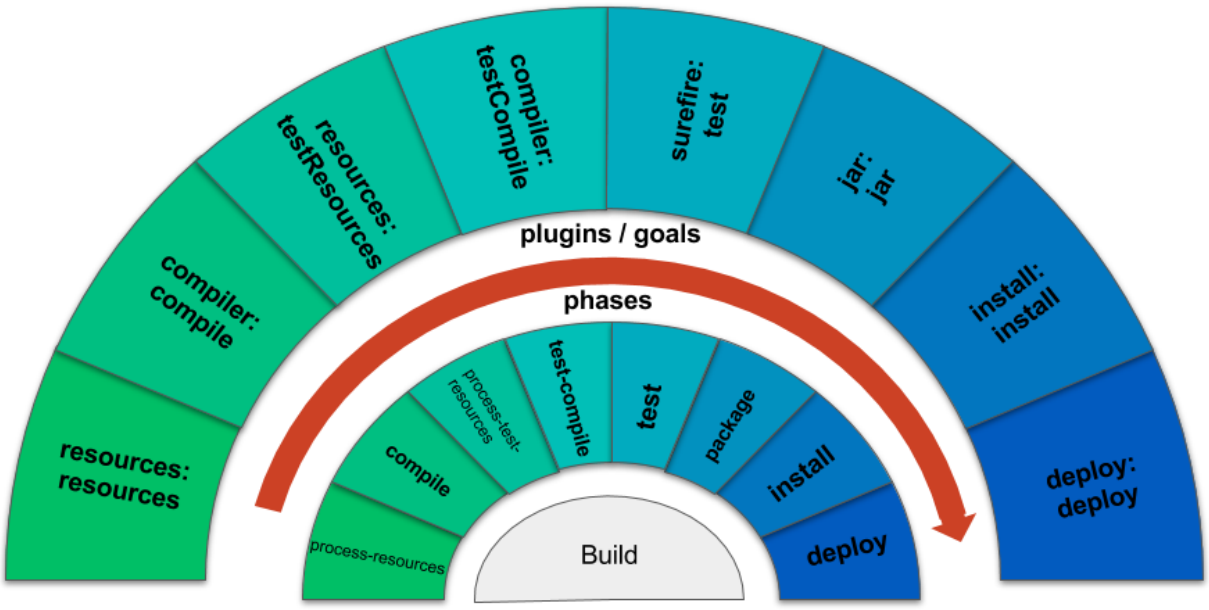
By default, Maven uses a lifecycle which allows a majority of projects to be built with minimal configuration.

The main **phases** are:

- **clean** : delete all created / generated files
- **compile** : compile *main* sources
- **test-compile** : compile *test* sources
- **test** : run tests
- **package** : create the binary archive (**jar** by default)
- **install** : copy the binary archive in the local Maven repository
- **deploy** : copy the binary archive in a distant Maven repository
- **site** : generate the documentation

Each phase can be associated to one or more **plugins**, which makes Maven very extensive.

The default associations are:



Some plugins are supplied by the Maven team, as the **maven-clean-plugin**, which deletes all generated files.

Others are created by the community, and do not require evolution in Maven itself. For example:

- **cukedoctor-maven-plugin** : create an HTML report of Cucumber tests execution
- **checkstyle-maven-plugin** : statically analyze the code, and fail the build in case of a rule violation

Source : https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

1.3.3. Configuration tags

All tags must be contained in the `<project>` bloc.

Among the most used tags, there is:

- `properties`: this tag contains key-value pairs, which can be used later on, either by convention by plugins, or explicitly using a language expression `${my-property}`

```
<properties>
    <my-test-lib.version>1.2</my-test-lib.version>
</properties>
```

- `dependencies`: this tag contains all dependencies used tin the project. A dependency can be a sibling module or and external library available from a remote repository.

```
<dependencies>
    <dependency> ①
```

```
        <groupId>com.mycompany</groupId>
        <artifactId>my-lib</artifactId>
        <version>1.45.3</version>
    </dependency>
    <dependency>
        <groupId>com.mycompany</groupId>
        <artifactId>my-test-lib</artifactId>
        <version>${my-test-lib.version}</version> ②
        <scope>test</scope> ③
    </dependency>
</dependencies>
```

① The `dependencies` bloc is composed of `dependency` (singular) blocs, each containing the coordinates of one dependency

② The version value references the `my-test-lib.version` property, therefore `1.2`

③ This second dependency has the `test` ***scope***, hence this dependency is only available for test code

- `build/plugins`: this tag contains all plugins used by the project as well as their configurations

```
<build>
    <plugins>
        <plugin> ①
            <groupId>org.apache.maven.plugins</groupId> ②
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
            <configuration> ③
                <failIfNoTests>true</failIfNoTests>
            </configuration>
        </plugin>
    </plugins>
</build>
```

① As for the `dependencies` tag, the `plugins` one contains `plugin` (singular) blocs

② Here the `maven-surefire-plugin` plugin is used. It is the default plugin for executing tests. By adding this bloc, the version is fixed and the configuration overridden. A plugin is a Maven project, and as such, has coordinates like dependencies

③ The `configuration` tag allows to change the default behavior of the plugin. Here the project build will fail if no test is found

- `profiles`: this tag allow to add configuration fragments that can be activated/ deactivated at will, without changing the project. A profile may add `properties`, `dependencies`, `plugins` and even `modules` (used in multi-modules project)

```
<profiles>
    <profile>
        <id>disable-tests</id> ①
```

```
        <properties>
            <maven.test.skip>true</maven.test.skip>
        </properties>
    </profile>
</profiles>
```

① Mandatory tag, a profile must have an ID, used to activate it from the command line. For example: `mvn install -P disable-tests`

1.4. JUnit

Java does not supply a built-in tool or API to describe or run tests.

Maven supply a dedicated directory, phases (compilation & execution), and a scope to handle this code which is not going into production.
However, Maven does not supply either tool or API to describe and execute these tests.

This is where *test frameworks* come in.
There are several of them, and JUnit is the most used today in the Java world.

1.4.1. Utilisation of JUnit-Jupiter API

```
class CalculatorTest {

    private final Calculator calculator = new Calculator();

    @Test ①
    void simple_division() {
        int result = calculator.divide(8).by(2); ②

        Assertions.assertThat(result) ③
            .as("division of 8 by 2")
            .isEqualTo(4); ④
    }

    @Test
    void division_by_zero_should_throw() {
        Assertions.assertThatExceptionOfType(IllegalArgumentException.class) ⑤
            .isThrownBy(() -> calculator.divide(3).by(0)) ⑥
            .withMessage("Cannot divide by zero"); ⑦
    }

    @ParameterizedTest ⑧
    @CsvSource({
        "0, 3, 3",
        "3, 4, 7"
    }) ⑨
    void addition_cases(int a, int b, int expectedResult) { ⑩
        int result = calculator.add(a).and(b);
```

```
        Assertions.assertThat(result) ③
            .as("addition of " + a + " and " + b)
            .isEqualTo(expectedResult);
    }
}
```

① Method mark as a test one, because annotated with `org.junit.jupiter.api.Test`

② Trigger event, some *main* code is executed

③ The result of the main code call is checked (here with the **AssertJ** library)

④ These three lines compose a single expression, as the compiler ignores line breaks. This kind of writing is called **fluent interface** and is built on consecutive method calls in such as way that they compose a meaningful sentence. Here, literally: check that the `result` variable as the "division of 8 by 2" is equal to 4

⑤ Check that an error is produced. This test will break if no error is produced or if the type of the error is different from the one expected

⑥ A function is given to the assertion API, it will be executed by the library, in a `try / catch` bloc

⑦ Check of the error message, if the message does not match the expected one, the test will fail

⑧ Method marked as a parameterized test, it will be executed as many times as there are test cases. In this example, the method will be executed twice

⑨ The dataset, here supplied as inlined CSV (Comma Separated Values), other sources may be used

⑩ Method parameters must match those in a test case, and follow the same order.

1.4.2. How JUnit works with Maven

JUnit-Jupiter is composed of several parts:

- An API to describe test methods (`@Test`, etc.)
- An execution engine which is able to detect and run test methods

JUnit also supplies a launcher of execution engine: **junit-platform-launcher**

Finally, the **maven-surefire-plugin** plugin "knows" how to connect (among other tools) to this *launcher* (since version 2.22.0).

NOTE

To summarize :

- The **test** phase of Maven is associated to the **maven-surefire-plugin** plugin
- This plugin can run **junit-platform-launcher** (if this library is available in the *classpath*)
- This *launcher* can run execution engines built with the **junit-platform-engine** API, **JUnit-Jupiter** being one of them
- **JUnit-Jupiter** can detect and run tests described with its API

1.4.3. A story about JUnit

JUnit is an old framework (1997) and has evolved a lot through Java versions.
The version 4, released in 2006 (after Java 1.5), has been broadly adopted and used during a long time, due to its simplicity using annotations (`@Test`).
In 2015, a crowdfunding campaign is launched to create JUnit5, a complete rewriting of the framework.

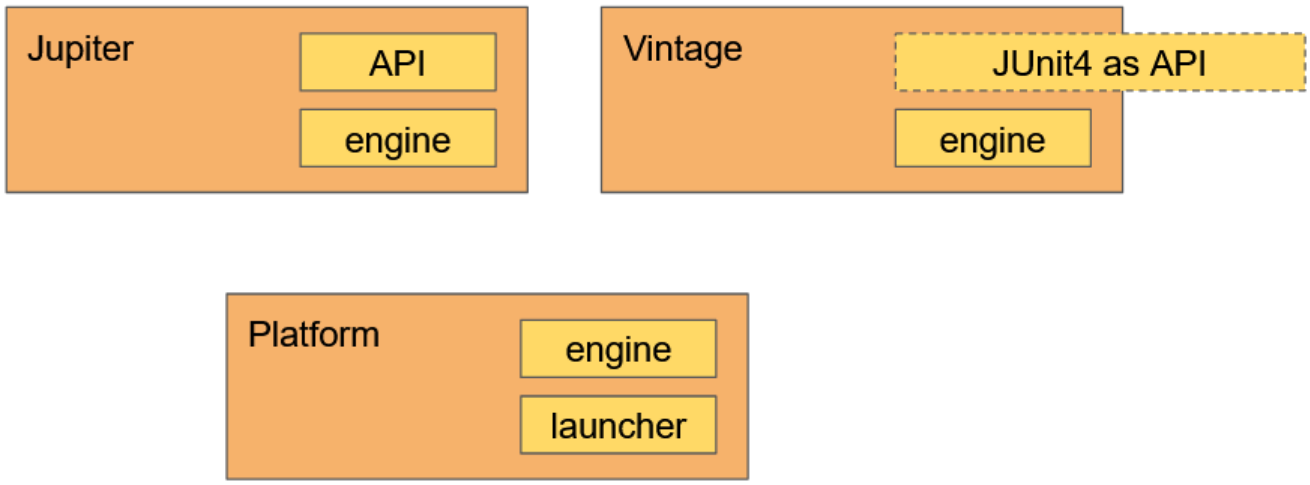
Right from the start, the team choose to create modular and dedicated APIs to avoid the same slides in the previous API, which was both very permissive and very complicated.

This was due to the initial API being designed to mark test methods, but evolved to allow plugins to describe tests arbitrarily (Cucumber, etc.) where a test can be a paragraph in a text file.

Furthermore, even if there was several extension points in version 4, the most used was `Runner`, which could not be composed.
This lead plugin teams to supply tools available from several extension points (`Runner`, `Rule`, initialization in `setUp` method, etc.) to bypass this flaw.

JUnit5 team stated that each paradigm should have its own API and execution engine, so that the code can be more specific, and much simple.

The resulting architecture is:



JUnit-platform is a framework to build and run execution engines.

JUnit-Vintage is an execution engine compatible with JUnit4 API.

JUnit-Jupiter is an execution engine with a new API which offers multiple and composable extension points.

2. Low coupling, a target at each scale

> I answered their issue by offering to add a new layer of indirection.
>
> — some architect passing by

In most situations, the most important objective is **maintainability**, as software is built to last for a long time.
In case of an issue, link to performance, security, or normal operation, the solution must aim to have the minimum impact.
Bigger is the code change, bigger is the risk to broke something else and longer is the time before deploying the change.
A fix or an evolution, must have the least risk possible.

Reducing the time between the design of a new feature and its delivery to production, is increasing the mastery and confidence of the team producing code the code.

2.1. At a Java class scale

2.1.1. Example 1: Internal state vs public contract

Consider this class:

Listing 2. File TrafficLight.java

```
class TrafficLight {

    private int color;

    public void setColor(int newColor) {
        this.color = newColor;
    }

    public int getColor() { ①
        return color;
    }
}
```

① Changing the type of internal state representation (`int`) will need to adapt the code using this class

The class could be rewritten to uncouple:

- the internal state, represented by an `int`
- the `public` contract (API) callable by other classes

Listing 3. File TrafficLight.java

```
class TrafficLight {

    private int color;

    public Color nextState() {
        color = (color + 1) % 3;
        return Color.values()[color];
    }

    public enum Color {
        GREEN,
        ORANGE,
        RED,
    }
}
```

2.1.2. Example 2: Positional coupling of parameters

Consider this interface:

Listing 4. File PersonRepository.java

```
interface PersonRepository {

    void savePerson(String firstname, String lastname, int birthYear, int birthMonth);
}
```

When calling this method, one may invert one parameter with another of the same type without noticing it.
Here the compilation will not help detect a bug, where month and year are inverted for example.
This is called *positional coupling*.
A better design could be:

Listing 5. File PersonRepository.java

```
interface PersonRepository {

    Person savePerson(Person person);

    @RecordBuilder ①
    record Person(String firstname, String lastname, YearMonth birthMonth) {}
}
```

① The **io.soabase.record-builder:record-builder** library is used to generate the matching *builder*

It is now harder to make a mistake:

```
Person person = PersonBuilder.builder()
    .firstname("Bobby")
    .lastname("Singer")
    .birthMonth(YearMonth.of(1962, Month.DECEMBER))
    .build();

repository.savePerson(person);
```

2.1.3. Example 3: Public contract extracted in an interface

The point of using interfaces is to uncouple the public contract from the concrete implementation.
This allows to:

- Substitute one type by another (implementing the same interface) without modifying the calling code
- Hide the implementation (methods and fields missing from the interface) from the calling code

The `Logger` concept, widely used in IT, is an abstraction to send events *somewhere*.
From the perspective of the business code, it does not matter where events are sent.

This abstraction is an interface, for example:

Listing 6. File Logger.java

```
interface Logger {

    void log(Level level, String message);

    enum Level {
        INFO,
        WARNING,
        ERROR,
        ;
    }
}
```

And can be used this way:

```
record CoffeeShop(CoffeeMaker coffeeMaker, Logger logger) {

    public Cup makeCoffee(String firstname) {
        if(!coffeeMaker.isReady()) {
            logger.log(Level.WARN, "Tried to make some coffee, but the coffee maker is
not ready yet");
            return Cup.EMPTY;
        }
        Cup cup = new Cup(firstname);
        coffeeMaker.pourIn(cup);
```

```
        logger.log(Level.INFO, "Made coffee for " + firstname + ", careful it's hot
!");
        return cup;
    }
}
```

Using an implementation of `Logger` which writes in

- The standard output
- A file
- A database
- A message broker
- A composition of all of the above

will not change the code of the `CoffeeShop` class.

2.1.4. Various forms of coupling

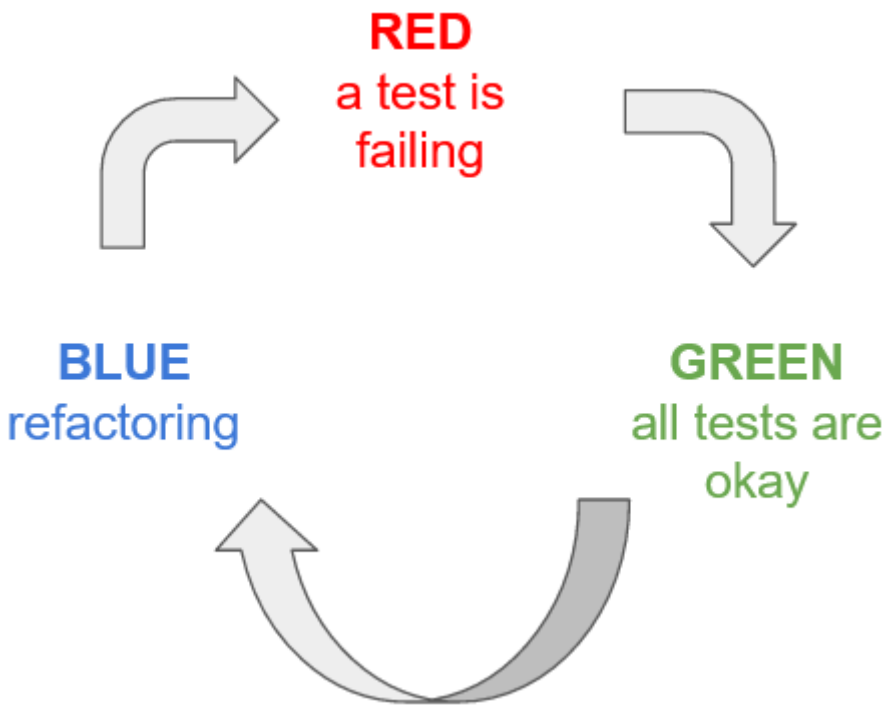
Various forms of coupling can be found here: https://connascence.io/

Most of couplings can be avoided by using the code at the moment of its creation.
The simplest technic is follow the **TDD** (Test Driven Development) principles.

Practicing **TDD** involves writing a *minimalist* test before writing the *minimalist* production code that makes it pass.
Code written this way, strictly meeting test requirements, is by design easily usable (as already used in tests).

> 1) Write NO production code except to pass a failing test
> 2) Write only enough of a test to demonstrate a failure
> 3) Write only enough production code to pass the test
>
> — Robert "Uncle Bob" Martin, Three laws of TDD

Writing *minimalist* tests is very important, as we want, when practicing **TDD**, shorten to the maximum the feedback loop between phases:



2.2. At an application scale

2.2.1. Business low coupling

An application uses most of the time distinct business domains.

For example, in an e-commerce application, we find catalog management, product sheet, cart detail, payment, invoicing, etc.

These domains are connected, but can evolve separately from each others.
Code must express these connections, but also the self-sufficiency.
A modification in the cart detail code will not (or should not) need another modification in the payment code.

Maintaining a low coupling between applicative components reduce the risk of introducing a bug in case of a change.

2.2.2. Technical low coupling

The same way, splitting business code (containing business rules) and technical code is a good approach.
What is meant by *technical code* is code needed to connect to the outside world, typically by way of an

- API
- Graphical user interface
- Connection to a message broker
- Connection to a database
- etc.

The core code of an application can be built without any framework or library in order to simplify tests and reduce the business code to its simplest form.

Connectors can later on be wired to it, to bridge business rules to other systems of users.

One of these approach is hexagonal architecture, which will be detailed in the next chapters.

2.3. At an information system scale

In an information system, multiple applications, maintained by distinct teams must exchange data.

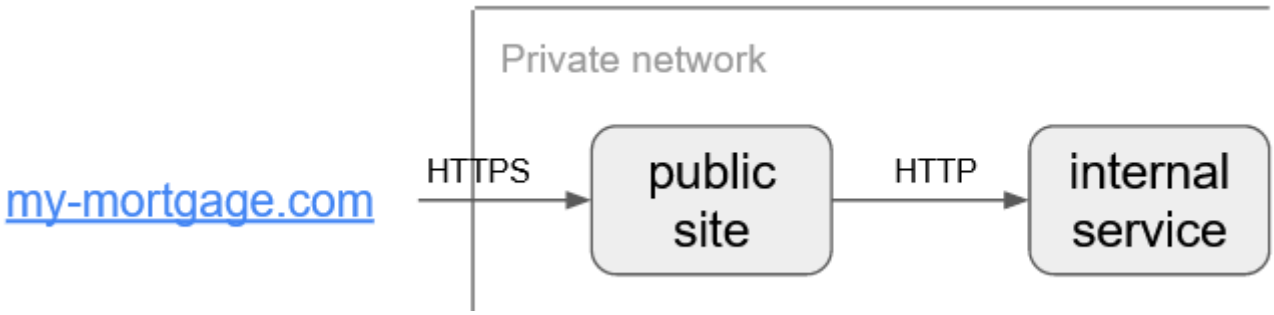
Architectural decisions choosen at this scale must consider the objectives previously enumerated (Maintainability, Testability, etc.) as well as the human factor.

Teams working on different applications are made of men and women with different work habits, rhythms and technical maturity.

It is necessary to consider these elements to protect the quality of service and simplify interactions between teams.

2.3.1. Example 1: asynchronous communication

Consider an application under heavy but irregular traffic, as a website performing mortgage simulations which is mainly used between 12 P.M. and 2 P.M.



The public website gathers simulation requests and sends them to an internal service which perform these tasks:

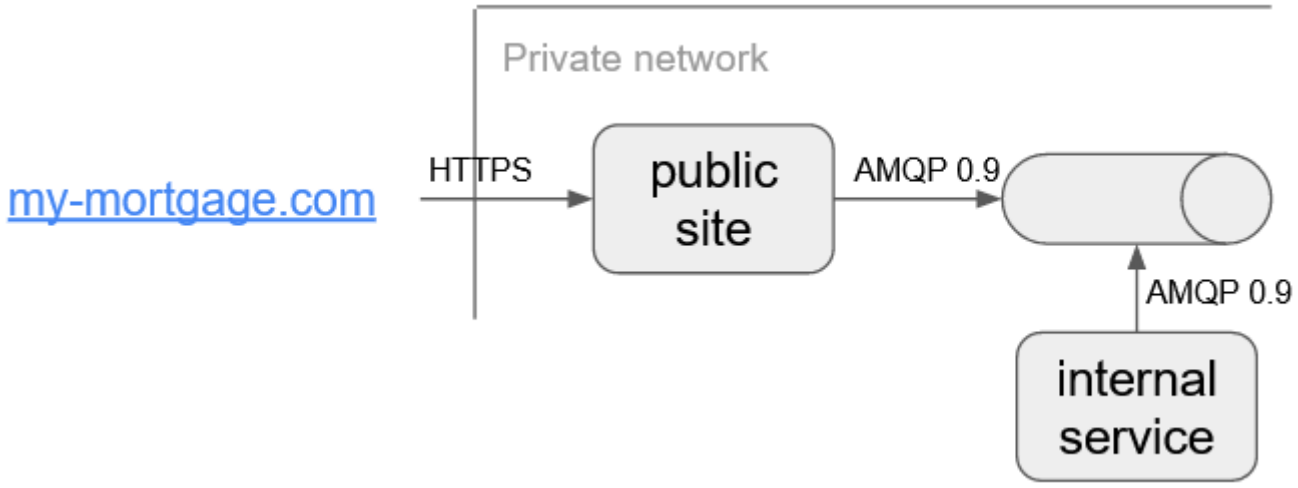
- Connect periodically to external APIs of various banks to maintain fresh data for computation
- Compute the simulations (we imagine them slow for the sake of the demonstration) based on user request information
- Send back simulation results to the public website

Even with a low traffic, a service interruption of a few minutes can damage the business, by loosing customers or visibility.

However with a synchronous communication between the public website and the internal service, an update of the latter will *de facto* lead to the public website service interruption or data loss.

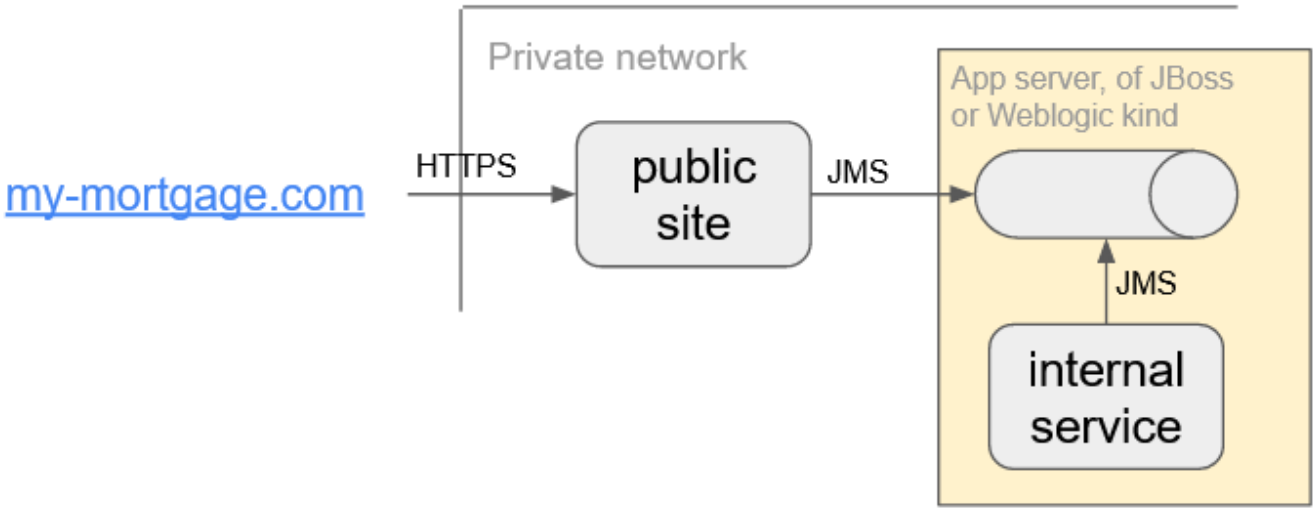
In this case, it is relevant to switch the communication model to an asynchronous one, through a message broker which will buffer user requests.

Thus, no service interruption will be *visible* and no data will be be lost.



2.3.2. Example 2: spread the load

Following on from the same mortgage application, consider that the message broker is coupled to the internal service for legacy reasons.



Sending a lot of events all at once (water hammer) to such a system could reduce its performance, or even crash it.

In this case, it could be simpler to have the publishing application to spread the load (*throttling*), to avoid the pressure coupling between the two applications.

Setting up this kind of mechanism requires to define a maximum speed (in messages/sec for example).
It can be built using different solutions, such as:

- Periodically read the X oldest messages in a database (be careful however, batch systems are difficult to scale)
- A small dedicated application using off-memory state (database, to be able to simply scale if needed)

- Using some brokers functionalities

3. Common architecture styles

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
>
> — attributed to Melvin Conway, Conway's Law

Everything has a cost, development itself is often the highest one.

Architecture choices should tend toward matching development cost and expected gain, hence it is strategic for a company.

Behind the mercantile facet, it is more pleasant to add new features on top of a good code base, rather than waiting for another team to deploy a new version of some API our team needs, or even be buried under constant bug-bashing.

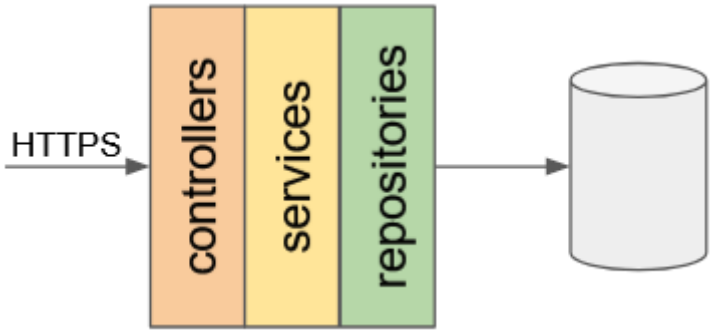
Architecture decisions are relevant in the context of development iterations, in order to focus energy on added value at the time being.

3.1. N-tier

N-tier architecture divides an application into technical layers.

For a simple application (HTTP API on one side, some database on the other for example), we often find **3** tiers:

- The Controller layer, responsible for
  - HTTP transport
  - Securing the access (Authentication and Authorization)
  - Serializing and deserialize data into *anemic* objects (**D**ata **T**ransfer **O**bject)
  - Performing surface validation, particularly on received data structure
- The Service layer, responsible for
  - Supplying high-level business features to *controllers*, so no business logic leak into this other layer
  - Ensuring data consistency, in regard to *business rules*
  - (Handle transactions , if the database is transactional)
- The Persistence layer (aka *repositories*, or **D**ata **A**ccess **O**bjects), responsible for
  - Communicating with the database
  - Serializing and deserialize data into *anemic* objects (**D**ata **T**ransfer **O**bject or entities)
  - Translate database technical errors to be handled by a business rule in the *service* layer



This architecture style is often applied as-is to the entire code base, and brings confusion when it comes to understanding the different business domains.

It would be preferable to introduce a division *matching* business domains

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── controllers
                ├── services
                └── repositories
```

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── blog
                ├── subscriptions
                └── twitter
```

That way, understanding of the key domains of an application is simpler, and it avoids to mix by mistake components between domains.

Furthermore, code can still be divided in technical layers inside each domain.

```
my-app
└── src
    └── main
        └── java
            └── com.my_company.my_app
                ├── blog
                │   ├── controllers
                │   ├── services
                │   └── repositories
                ├── subscriptions
                │   ├── controllers
                │   ├── services
                │   └── repositories
                └── twitter
```

3.2. Hexagonal

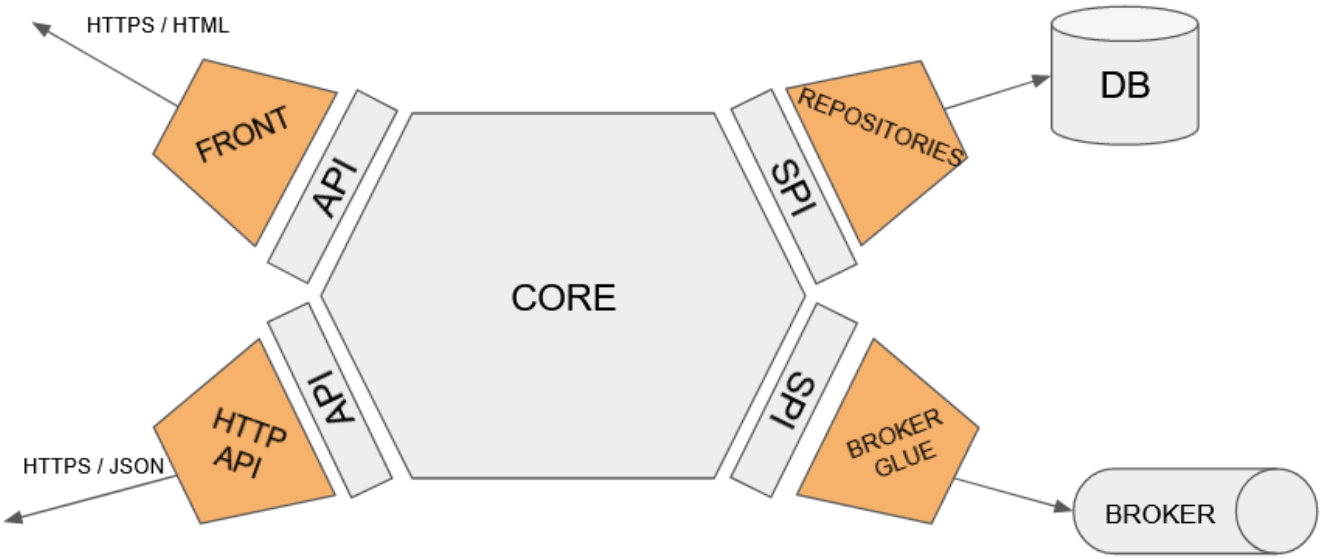
Hexagonal architecture aim for dividing *business code* from *technical code*.

The division isolates the application *core*, from the *adapters* which link *business code* to the outside world, *machine-to-machine* communication (HTTPS, AMQP, etc.), or *human-to-machine* (GUI, email, SMS, etc.)

The objectives are as follow:

- Maintain a low coupling between technical stakes and business rules
- Simplify tests on the *business code*, since there is no framework or library involved

- Allow to replace one *adapter*, without modifying the rest of the application



Some vocabulary:

- **API** (**A**pplication **P**rogramming **I**nterface) an interface which is implémented by the business code, and called by an adapter
- **SPI** (**S**ervice **P**rovider **I**nterface) an interface which is called by the business code, and implemented by an adapter.

Development starts by writing the business code of a feature.
This code must not use any dependency to a framework or library.
To this end, this code can be isolated in a dedicated module so that the compilation can enforce this no-dependency rule.

Afterwards, other modules can be added for adapters, these modules having a dependency to the core one.

3.3. Monolith vs micro-services

> I'll keep saying this... if people can't build monoliths properly, microservices won't help.
>
> — Simon Brown, @simonbrown

Building a monolith is creating one only application to handle all features.

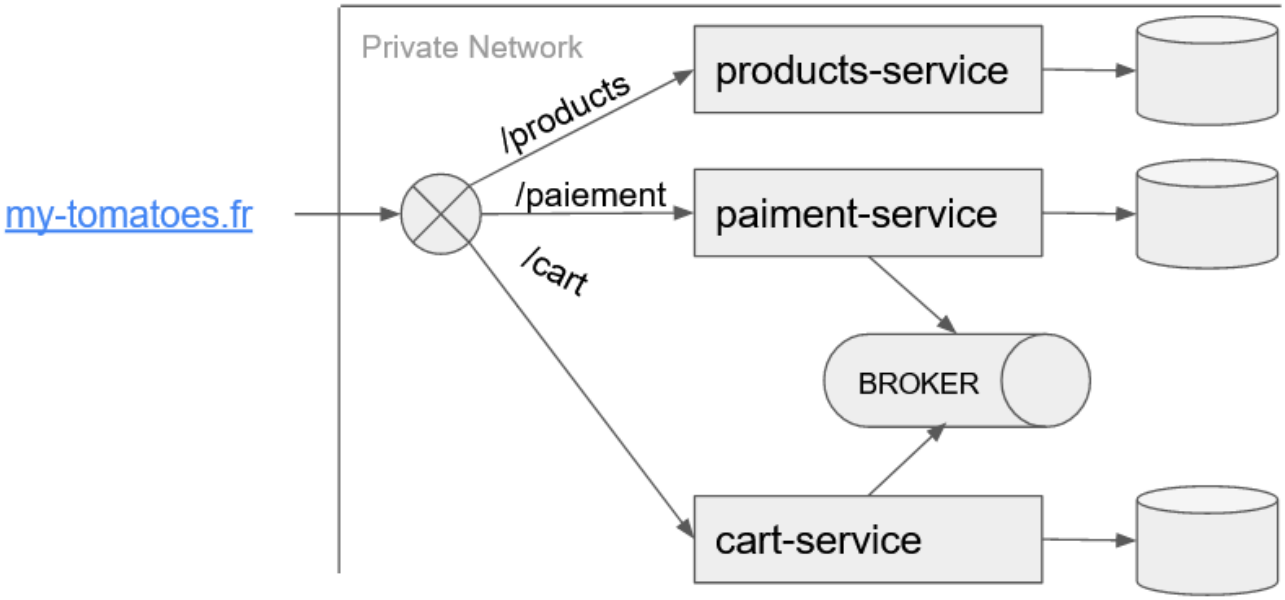
In a monolith, modularize the code to isolate parts of the code that must not interact with each other is vital.

If not followed, the risk is to see an exponential growth of the complexity emerging at each feature addition.

On the other hand, the micro-service architecture style forces a kind of division, as micro-services must not depend (in terms of code) to each other.

L'approche micro-service a ceci de différent que les différents domaines sont gérés par des applications différentes.
As it is still possible to share code trough the use of shared libraries, domain isolation, as in a monolith, is essential.
The rule that expresses this concern the most is that each micro-service must have (if needed) its own database / data persistence model.

The same way, if micro-services must communicate, using asynchronous communication will avoid time-related coupling.



A micro-services architecture has downsides:

- Multiple applications must be deployed at a fast pace

  → Delivery and deployment must be robust and automated
- Issue investigation is more complex, as a single user action can be spread across multiple applications

  → Logs must be unique, informative and centralized

  → A correlation mechanism is needed (correlation-id header, via header, APM, etc.)
- The infrastructure cost is higher

And upsides:

- The code of one micro-service is smaller, simpler
- Micro-services can evolve independently of each other, and be maintained by different teams
- Micro-services can scale separately from each other

  → If the *product* domain of an e-commerce website is under more load than the *payment* one, it is possible to dispatch more resources to the one needing it

4. Scaling & relative considerations

Some time ago, the habit of building monolithic applications with in-memory state (such applications are said *stateful*) needed more powerful servers to follow an increase in usage.

What needed to be increased:

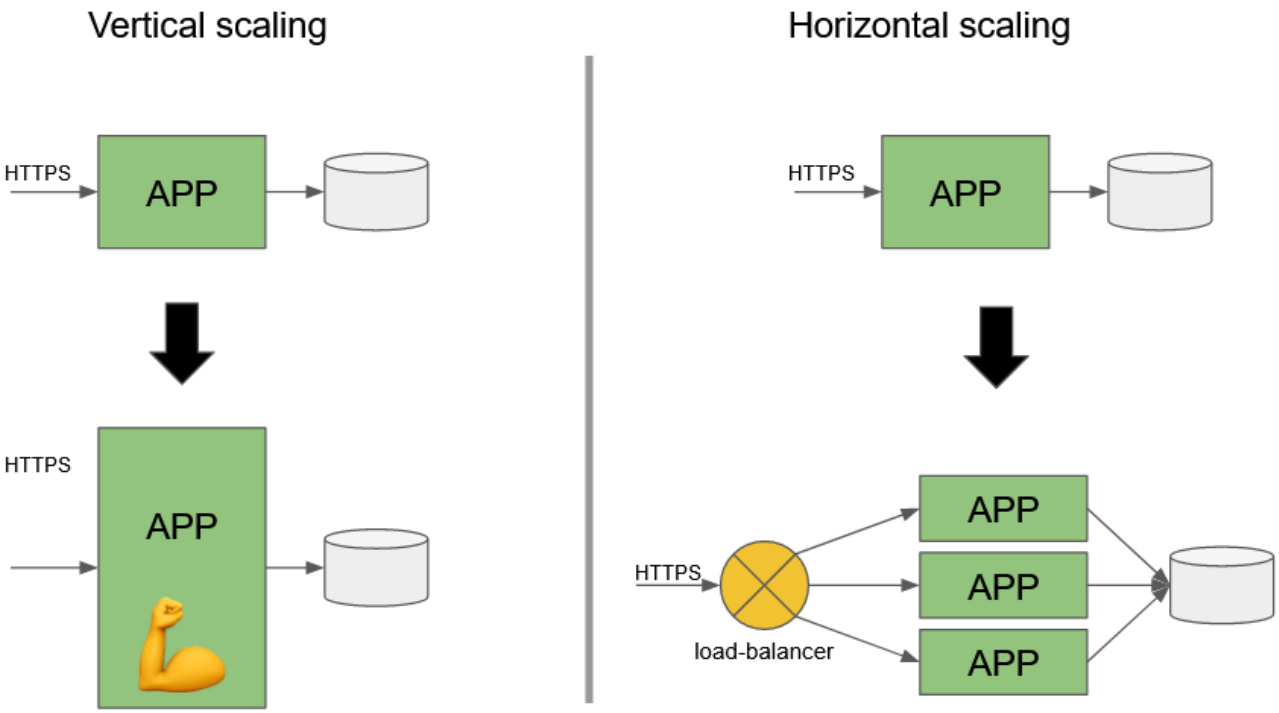
- RAM (memory)
- CPU frequency
- Number of CPU cores
- Disk size

Such upgrades fall under **vertical scaling**.
However hardware prices are not proportional to power, but rather exponential.
This scaling type rapidly faces technical limit (maximum power of a CPU at a certain time) or financial limit.

Some systems, such as traditional SQL databases (Oracle, PostgreSQL, MySql, etc.) hav no choice but to use this kind of scaling.

Conversely, applications can be structured in a way that they can be scaled differently.

In this case, **horizontal scaling**, which involves adding more containers / VMs / serveurs to increase the system capacities in a **linear** way.



Another downside of having a unique instance, is that any change needing a reboot (configuration, new version, etc.) will interrupt the service.

Whereas with several instances of the same service, rebooting one is transparent from the user perspective.

Rebooting all instances, each one at a time, is called **rolling update** and used to update an application without service interruption.

The only warning with such a system, is to pay attention to N-1 retro-compatibility, as during an update, both versions, the old one and the new one, will run behind the same load-balancer at some point.
So API of the old version must have the same contracts and behave the same when used on the new version.

4.1. Identify SPOFs

SPOFs (**S**ingle **P**oint **O**f **F**ailure), are parts of a system that cannot have more than one instance and by design lead to service interruption in case of a stop (crash, update, etc.).

It is important to have monitoring on these components in order to rapidly take action in case of an issue.

This reaction can be automated (switch to a replica, passive until then) or manual.

Knowing where SPOFs are, can also orient the system architecture to avoid flooding these components by setting up throttling or caching.

SQL databases are often SPOFs; but most of them have replication mechanisms capable of keeping several other instances up-to-date and ready to use if the main one fails.

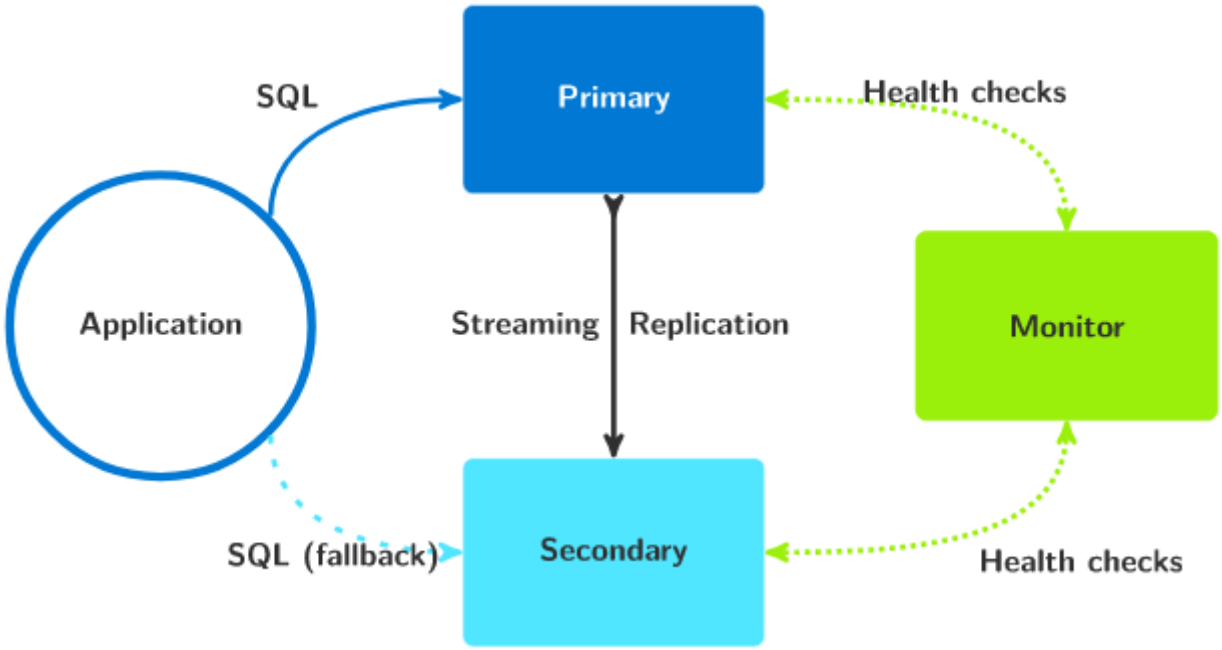


Figure 2. PostgreSQL replica switch mechanism (`pg_auto_failover`)

4.2. Identify points of contention

When a system is under heavy load, points of contention may appear.

We are talking about parts of the system with slower processing which defines the maximum speed

of the overall system.

NOTE	Like with a stormwater run-off system, if one of the pipe has a low throughput, the whole system appears slow, risking congestion

A point of contention can induce cascading reactions or hide several other contentions.

To identify and improve them if necessary, a precise supervision of each components of the system is required.
Supervision consists, among other things, of a probe set which collect data periodically so they can be graphed.
These graphs reflect the system evolution over time and allow to make correlations between exogenous events and endogenous consequences.

4.3. Distribute the load

4.3.1. For synchronous communications

To distribute the load on multiple instances of a same service, we use load-balancers, whether they are hardware (F5, Altéon, etc.) or software (HAProxy, Nginx, Traefik, etc.).

Multiple configurations are available, to weight target nodes or establish *sticky session*.

Software load-balancers are simpler to configure and can even be controlled through an API.
This last capability allow to automate operations such as the previously mentioned **rolling update**.

4.3.2. For asynchronous communications

When using asynchronous communication protocols, its is best to rely on a dedicated message broker which acts as a buffer between applications.

Several tools and mechanisms are available to make a broker ***H**igh **A**vailability* and capable of ingesting large quantities of data.

Such a broker can ingest occasional pressure surges and damper them for listening applications.

When communicating asynchronously, using pull flow allow a listening application to consume events at its own pace.

Load distribution is available *by design* by adding more instances of listening applications.

Most brokers offer different delivery strategies:

- **at most once**: in this case, events can be loss
  - Use case: notifications, data of little importance
- **at least once**: in this case, events can be delivered more than once, listening applications must be able to behave in an idempotent way
  - Use case: every other cases, where loosing data is not an option (customer order, bank

transaction, etc.)

4.4. Architecture evolution example

Consider a *typical* application such as:

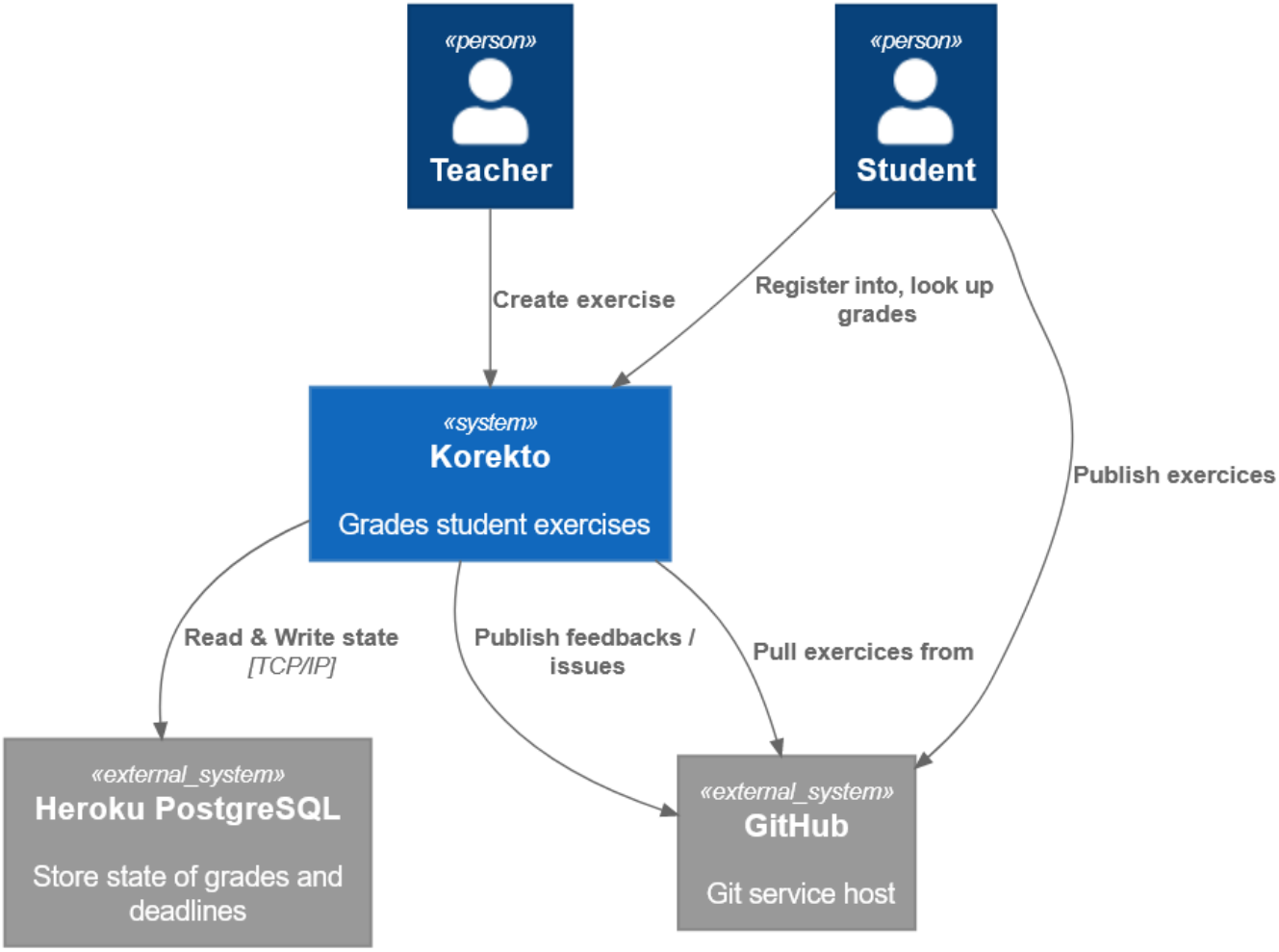


Figure 3. Context diagram

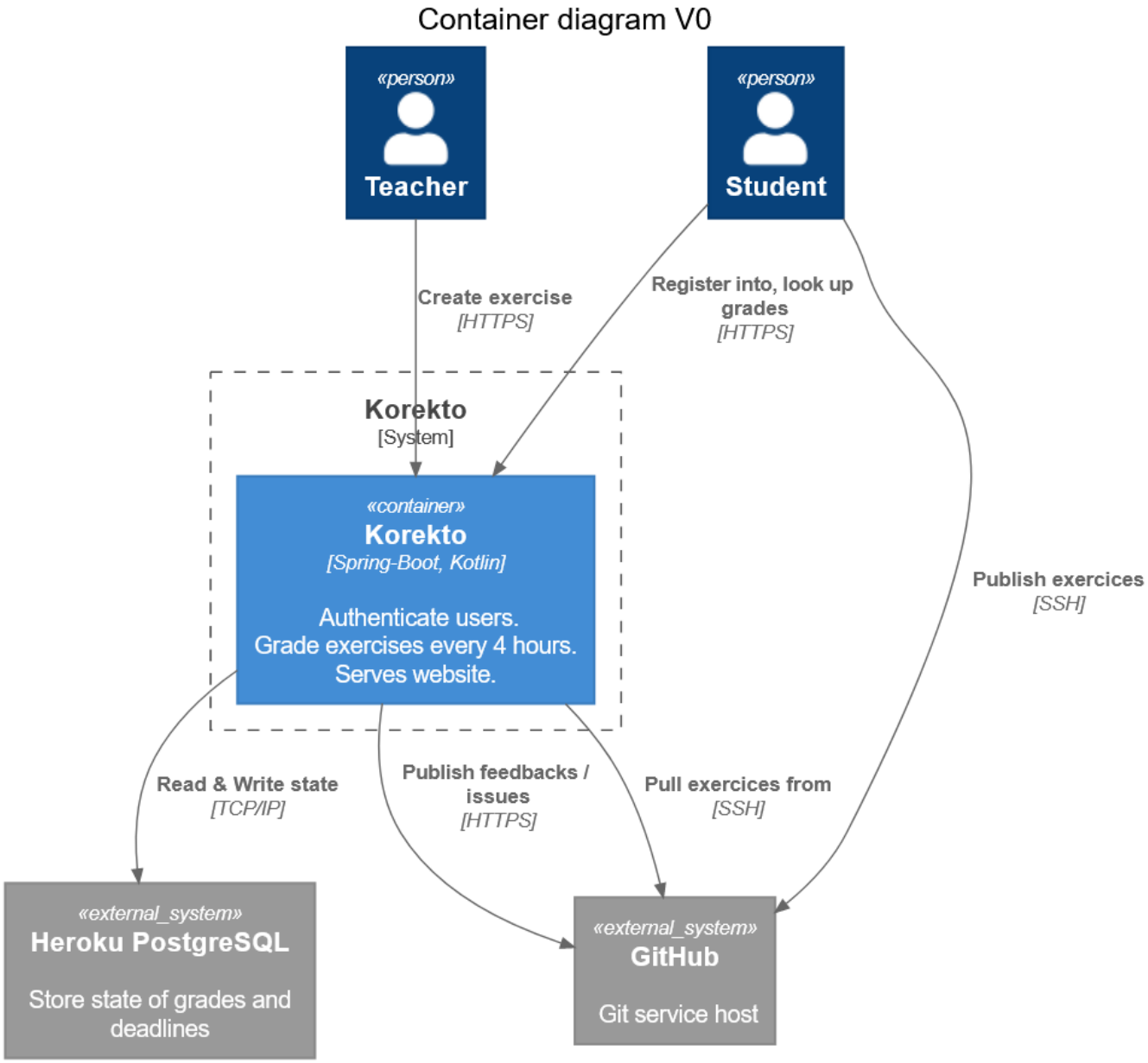


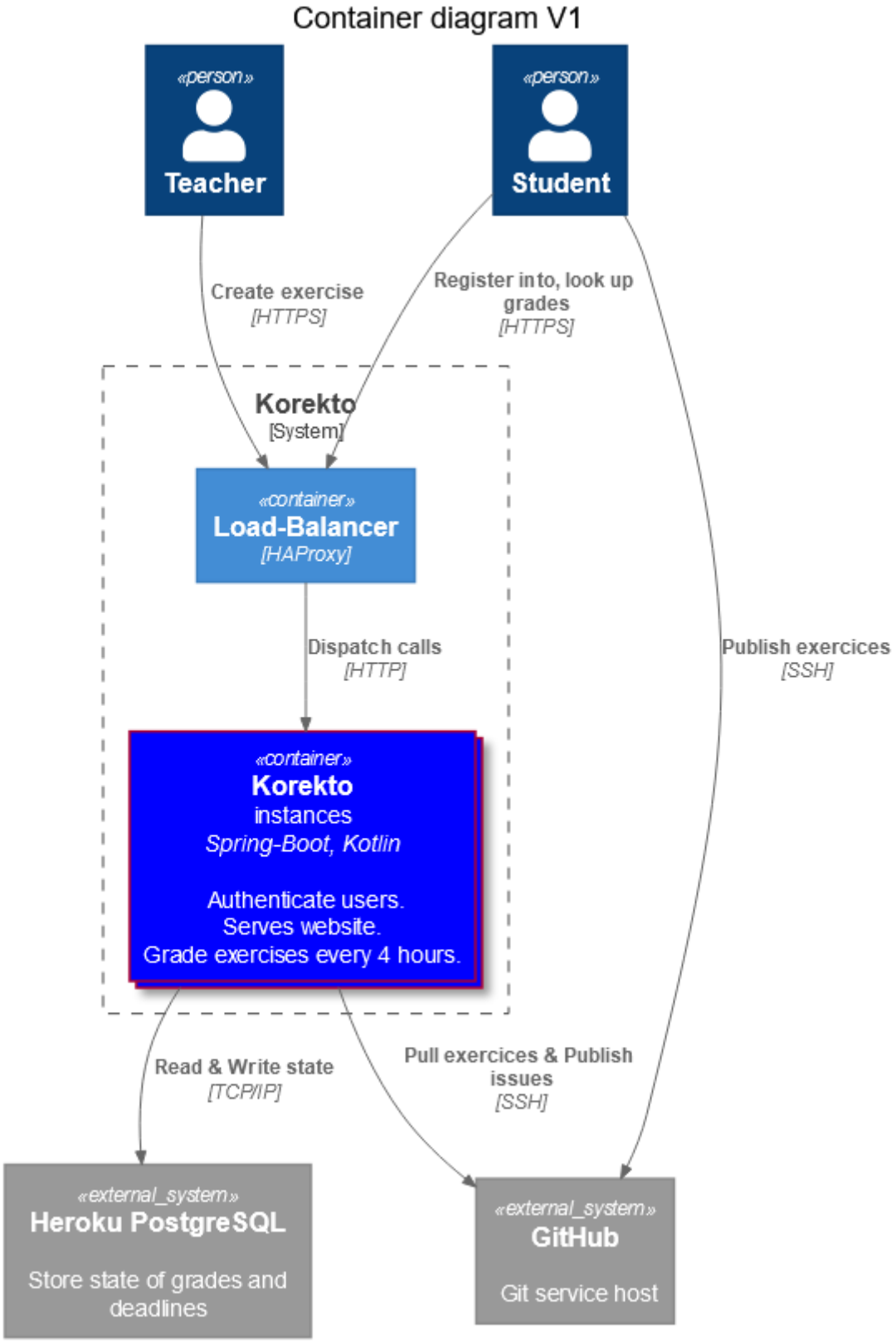
Figure 4. Container diagram

4.4.1. Introduction of a load-balancer

Horizontal scaling, is the ability to increase a system capacity by adding more instances (or nodes). But from the users perspective, there is only one service (https://korekto.io for example).
To map from one service to multiple instances, we use a load-balancer.

Warning, the part that executes a task every 4 hours must only be active on one instance, under penalty of doing the same job multiple times, which can lead to:

- Pointless CPU load
- Incoherent data



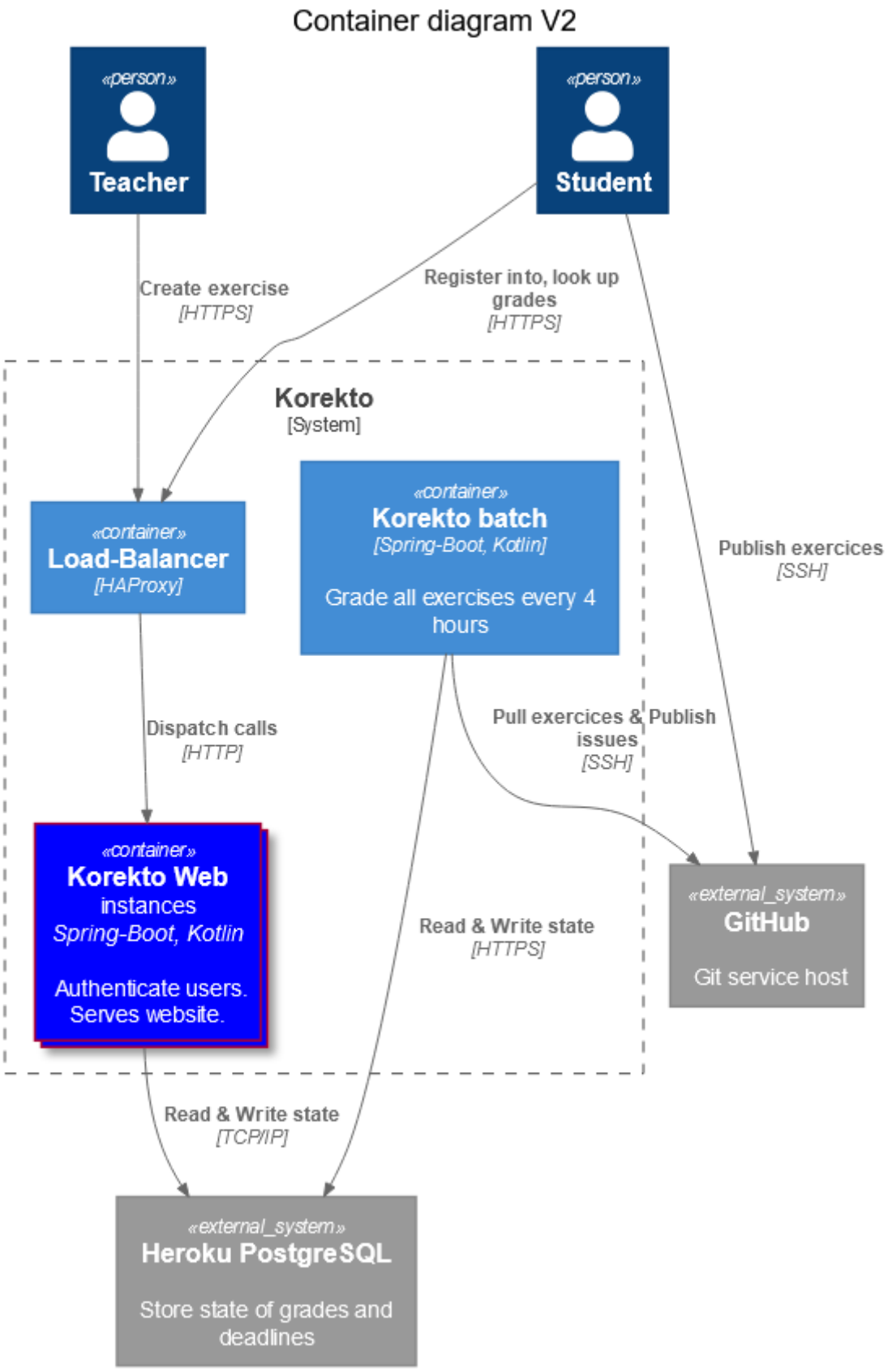
4.4.2. Split the application

Some features are used less often and uses less CPU.
It can be interresting to split our monolithic application into smaller ones, to be able to update them independently and scale them differently.
This separation must respect one golden rule to be durable:

IMPORTANT	Applications must be independent technically AND logically (from a business

perspective).

This means that if one application is to evolve, it can do so without impact to others.



4.4.3. Deleting SPOFs

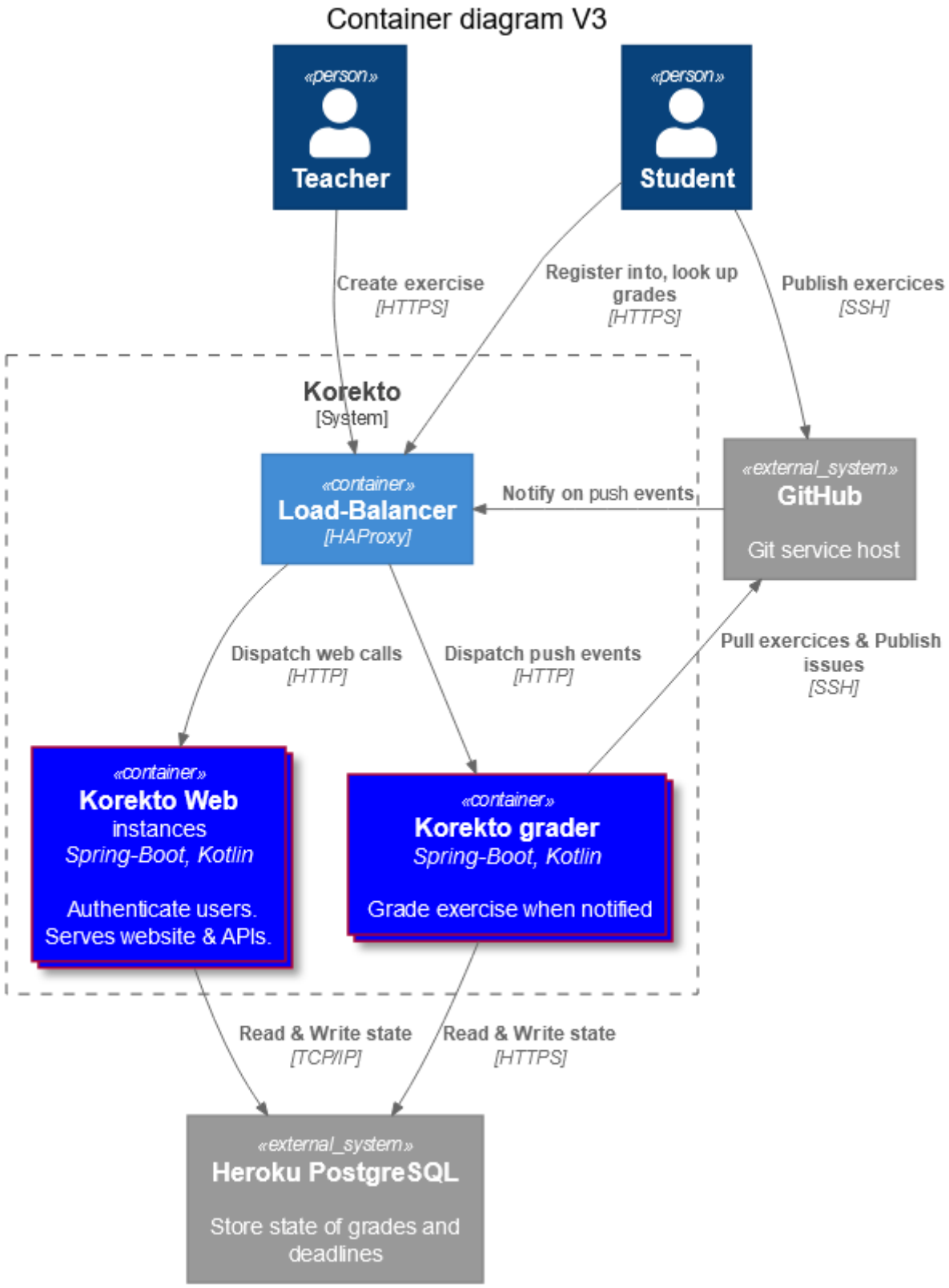
Single **P**oint **O**f **F**ailure are components of the system that cannot be replicated.
This causes 2 issues:

- If the component stops working, it causes a service disruption
- If the traffic increases, the only possibility is vertical scaling

SPOF nature is often due to a state stored in memory or a low latency storage need.
This is the case of the **[batch]** component.

If multiple instances were to execute simultaneously, how could they synchronize to distribute the work and restart jobs of failing instances ?

In our case, the simplest move is to use GitHub webhooks to be notified in case of a change, rather than periodically scan for change.



4.4.4. Do not exchange data through storage

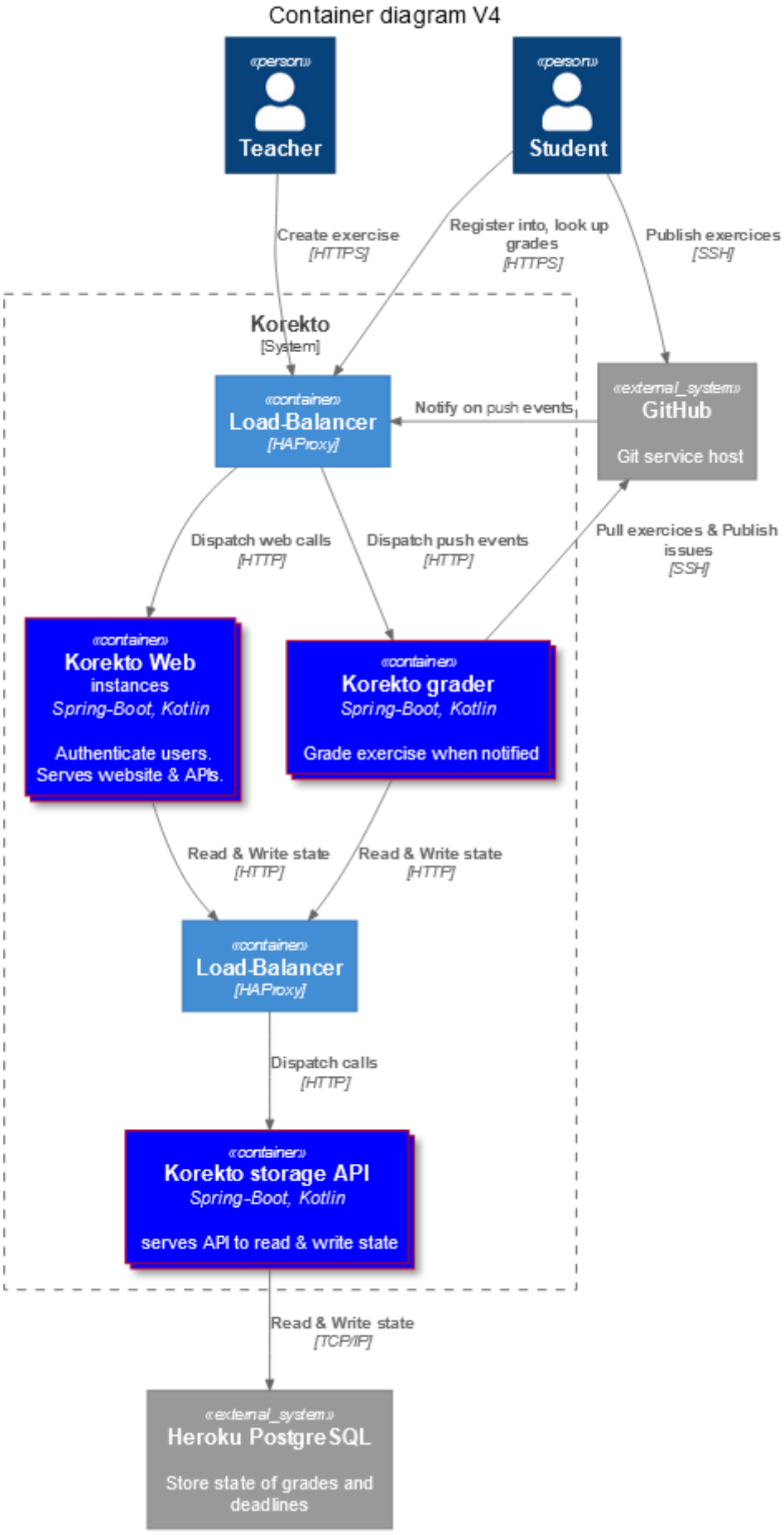
During the split, a mistake was made.
Indeed, the **[web]** container and the **[grader]** one exchange data through the database.

This is well known anti-pattern, because:

- How to change the database model without impact for both containers?
- How to change the way data is stored without impact for both containers?

These two containers are coupled.

The objective now, is to introduce a new container which will abstract the storage.



Therefore, when

- Some of the data must be stored in a different way (in a relational database for example)

  → the change occurs in one only container, the **[storage API]** one

- The exercise model evolves to include more data

  → **[storage API]** supplies a new API (GET / POST / PUT / DELETE) /api/admin/specification/exercise/${name}/v2

  → **[web]** may use this API so teachers can better define properties of an exercise

  → **[grader]** may use this API to improve grading

  → **[storage API]** may delete the old API (/api/admin/specification/exercise/${name}/v1) when no container use it anymore

These kind of changes can be applied with no service disruption

5. Architectural documentation

Documentation of an IT project was historically very verbose (Technical specifications, Functional specifications, **T**echnical **A**rchitecture **D**ocument, etc.)

Yet, an IT project evolves fast, faster than these documents.
This documentation become rapidly outdated, and leads to confusion or bugs as some pieces of information are simply false?

However, documenting a project is necessary to pass on knowledge among changing stakeholders.

Hence, the objective is to write the minimum required to understand the context, and to hook this documentation up to code, as much as possible.

Thiw kind of documentation which has its roots in code is called living documentation, as it evolves naturally as the project does.

5.1. README

The README file, written in Markdown or AsciiDoc (like this course) is found at the root of a project / repository.
It is the gateway for an unknown project.

It answers the following questions:

- What is the use of this project ?
- How to use it ? (build tool, etc.)
- How to contact other stakeholders

Very popular in open-source projects, the README file is also important in a corporate context, since multiple projects coexists in the same information system and multiple teams must work together.

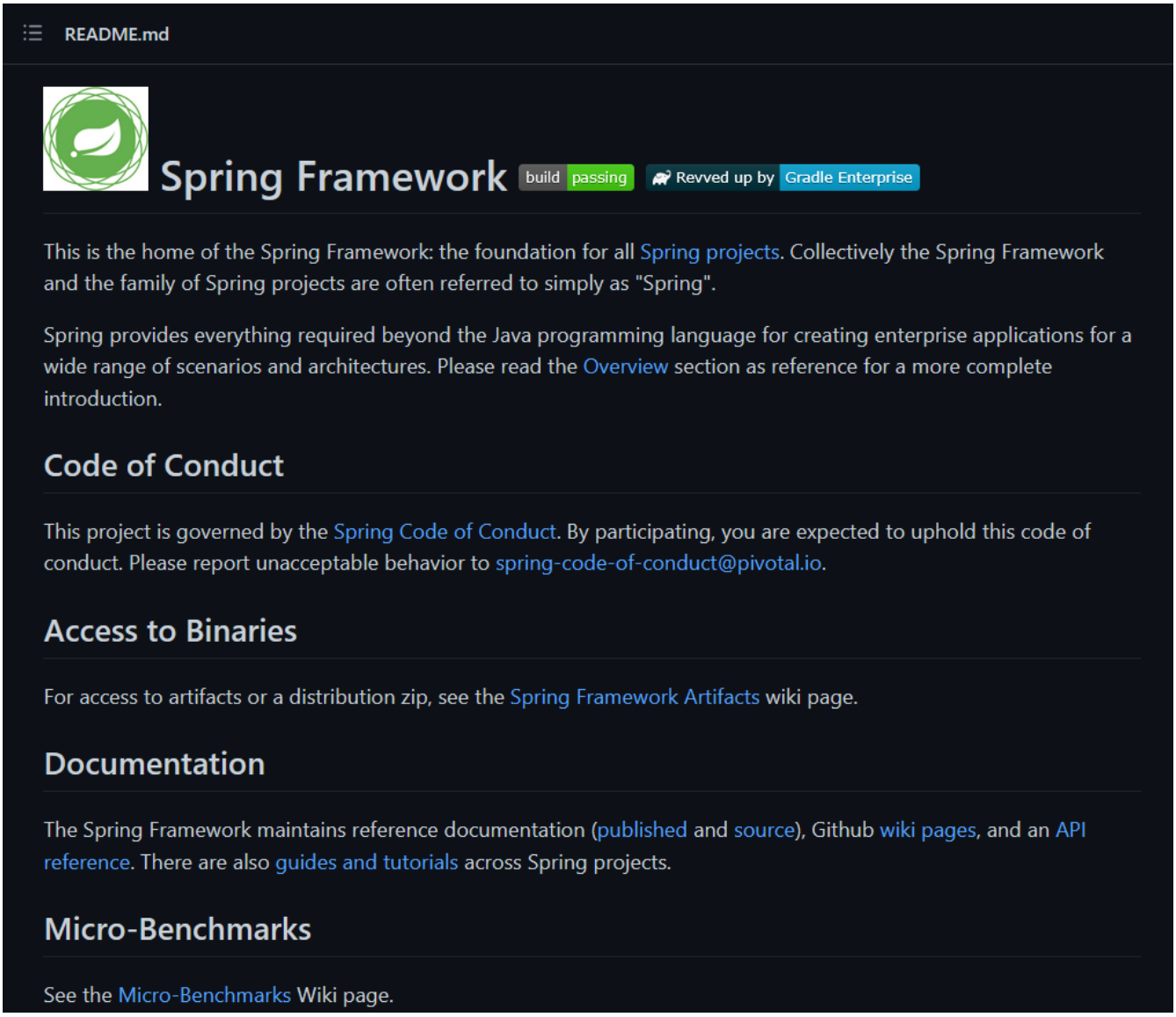


Figure 5. README file of the Spring project

5.2. C4

C4 is an approach for documenting the architecture, without mixing contexts.

The first thing needed, is a common vocabulary (as in DDD) with which all stakeholders ***understand*** each other.

This is a decisive step, which avoids lack of understanding afterward.

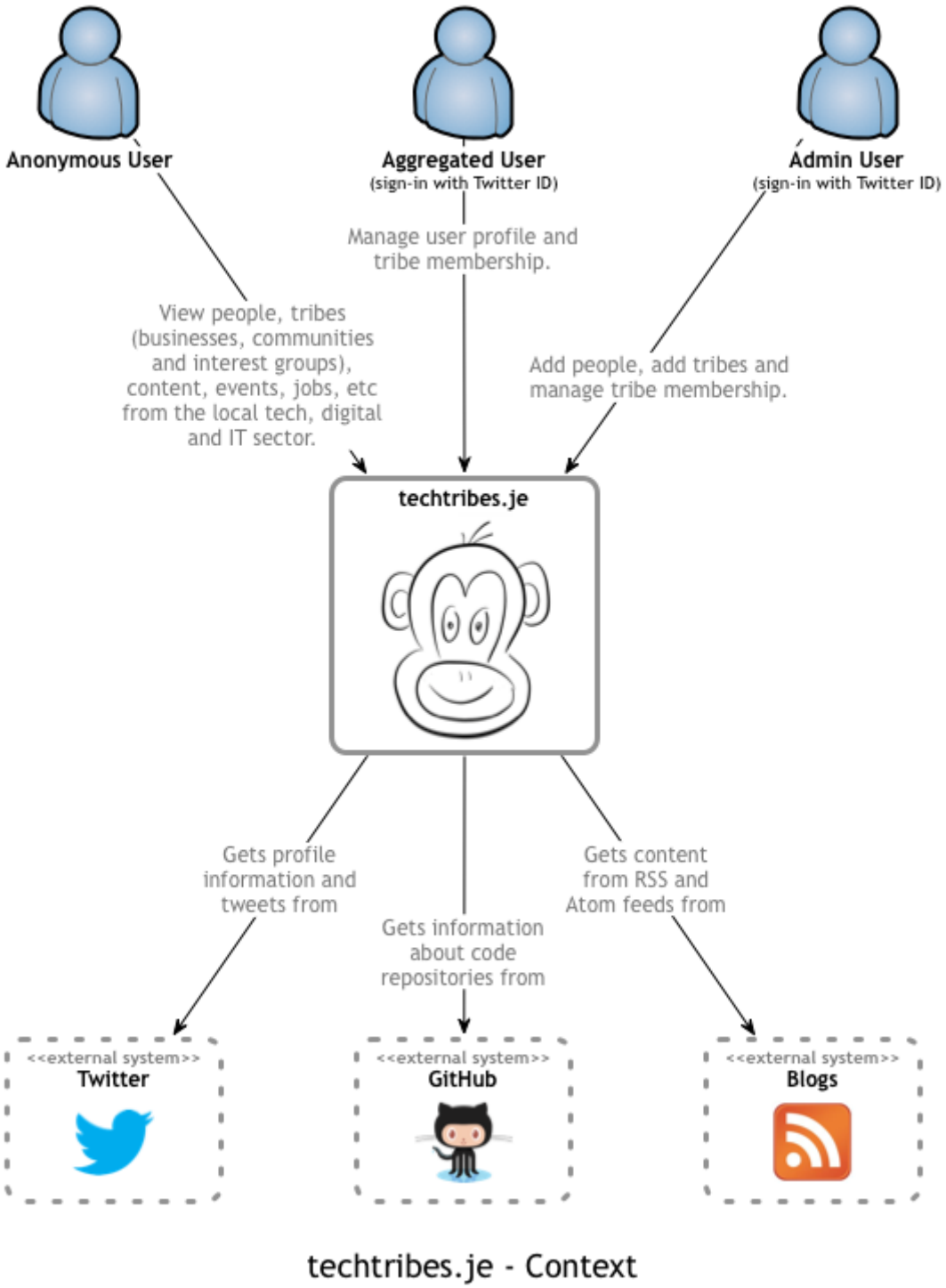
This vocabulary is used to draw the diagrams representing the project.

5.2.1. Context

The ***context* diagram** allows to understand key features of the system without displaying how the system is organized.

The system is represented by a unique shape (black box) which interacts with user personas or other systems.

The details (technologies, protocols, etc.) are irrelevant in this diagram, which should be understood by non-technical stakeholders.



techtribes.je - Context

5.2.2. Container

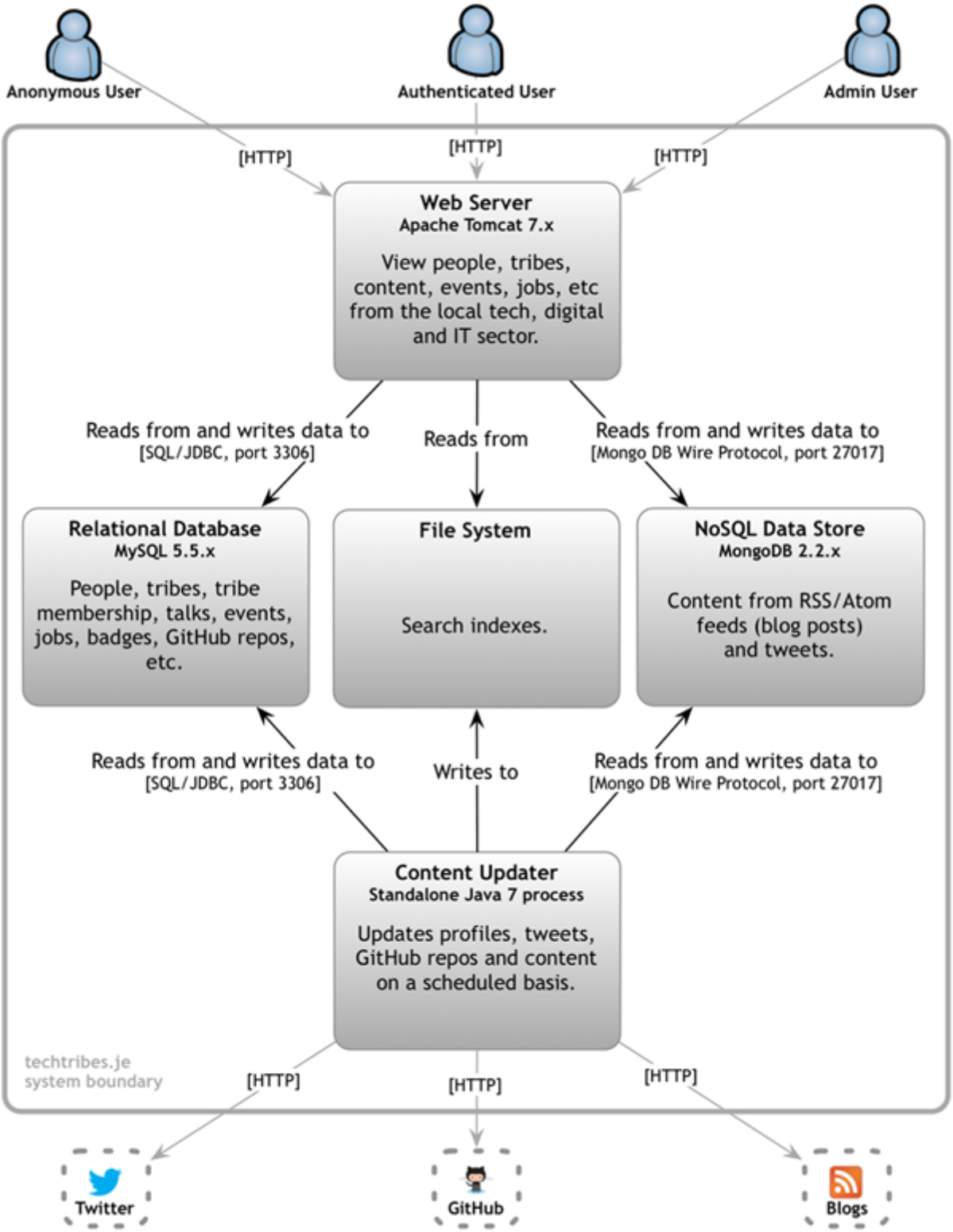
Once features are laid down, we can zoom in one notch with the ***container* diagram**

By *container*, we mean all system hosting code or data (application, database, broker, etc.).

This diagram reflects the "big-picture" architecture of the system and allow to see how responsibilities are distributed among the different software pieces.

Can be represented:

- **Security** : networks, encoded traffic, traceability, etc.
- **Scalability** : load-balancers, (SPOF), etc.
- **Reliability** : backups, replicas, etc
- **Supervision** : probes, metric storage, IHM, etc.



techtribes.je - Containers

5.2.3. Component

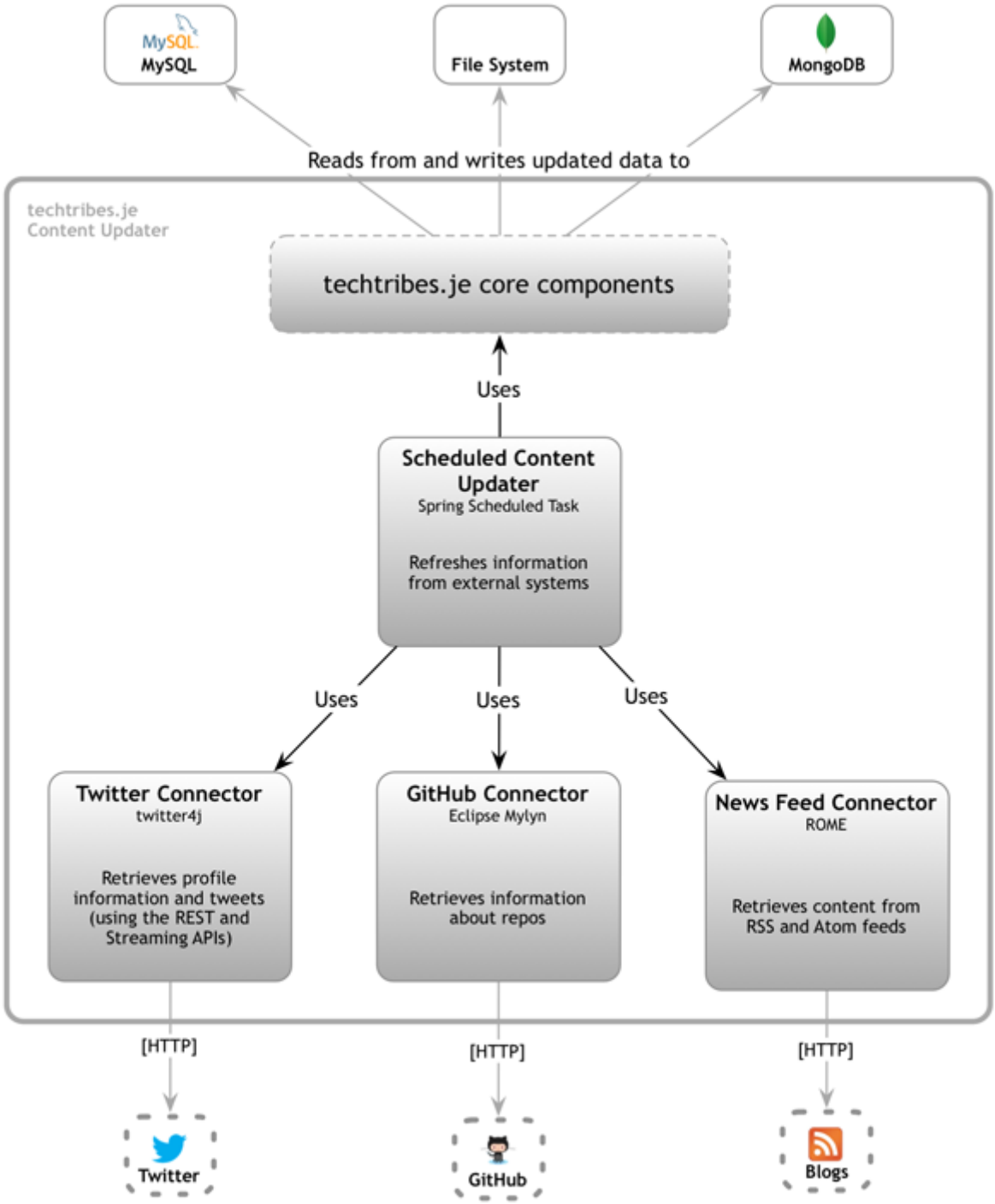
Zooming in one more notch on each container is possible with the ***component* diagram**.

This diagram details the composition of one container, with possible representation of:

- Intersecting features
- Business service
- Workflow

A component as a logical identity, one responsability, and interacts with other components.

Here, implementation details and technical choices are relevant since these diagrams can be the most technical of a project.

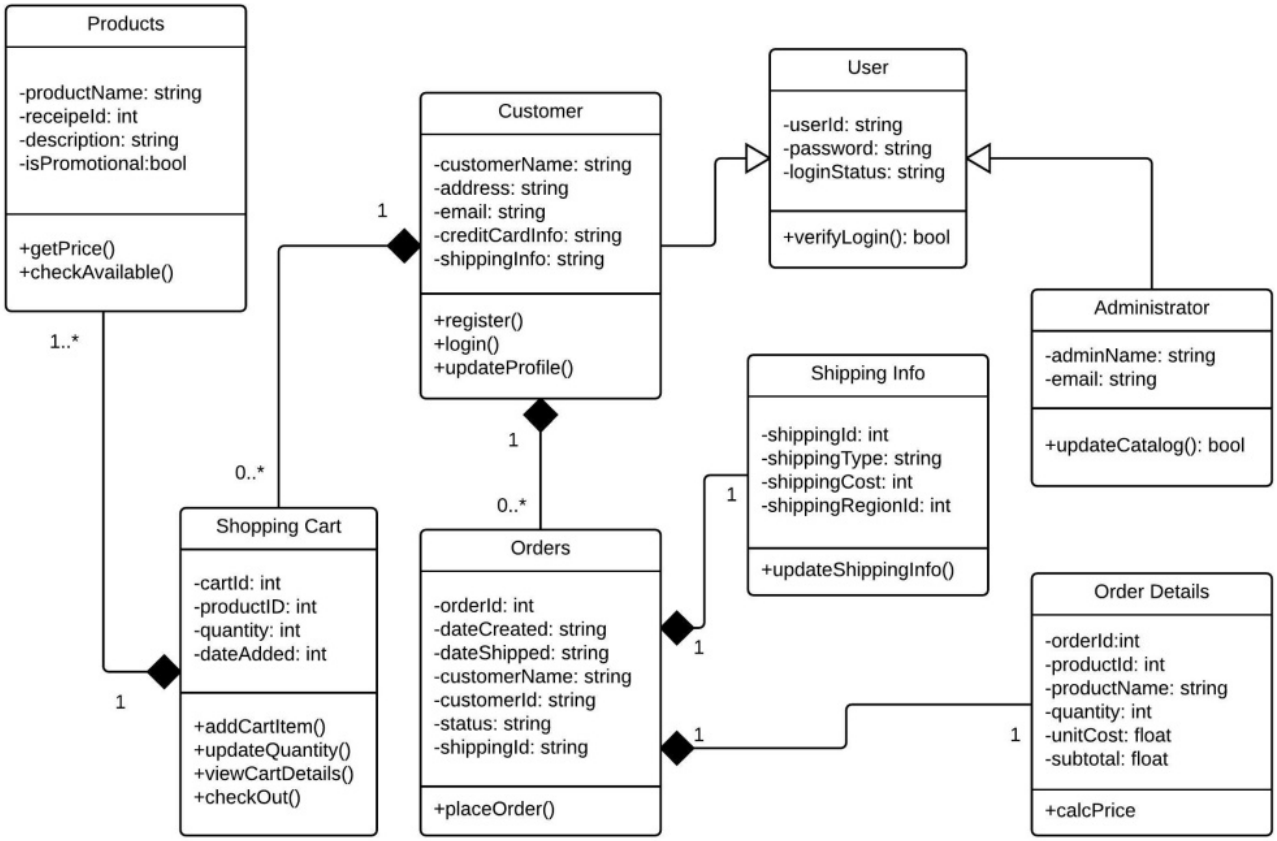


5.2.4. Class

If necessary, we can go down more using the classic **UML *class* diagram**.

Rarely useful if needs are clear and responsibilities well divided.

However to represent a complex behavior, this diagram can shine a light on a complex interaction between several objects.



5.3. ADR (Architectural Decision Records)

ADR are a log with each entry regarding a structuring choice made in the project architecture.

It is written by the software engineers and stored (most of the time) next to the code, in the same repository.

Reading this log allows to understand why and how an application was built this way.

Each entry is composed of:

- The date
- Involved stakeholders
- The context: what lead to this choice (security, organisational, performance issue, etc.)
- Different solutions considered
- The decision
- Expected consequences
- Optionally a state, which shows that the question was mentioned, analyzed, but that nothing came out of it

5.4. API documentation

APIs can be documented in two ways:

- **Contract First** : the API specification is written (OpenAPI, RAML, WSDL, etc.) first, then the matching code is written or generated.
  If the code is generated, there always is a manual step to wire it to the business code.
- **Code First** : the code is written first.
  This code can be enhanced with metadata of the API (description, explanations for each error codes, etc.)
  The documentation is then generated based on the code, particularly with the emergence of **OpenAPI** (formerly Swagger) and various tools in its ecosystem.
  Such a documentation evolves as code does.

The **code first** approach is the most durable, as it avoids a drift between documentation and code to grow over time.

In any case, modifying an API must be taken carefully, as it may lead to breaking changes for end users (and this can have business consequences).

Some changes have no consequences:

- Add a new field in a response
- Remove an optional field from a request

→ *What do the most can also do the least*.

Other changes, have consequences:

- Delete a field from a response
- Add a mandatory field in a request

And will lead to communication errors and needs to be operated differently.

In the case of a change with breaking changes is needed, it must be introduced with a new version of the API, while maintaining the old version.

This way, customers have the time to operate the necessary change of code on their side to use the new version.
When all customers hav moved on to the new version, the previous one can be deleted.

This approach introduces a ***low coupling*** between systems allowing them to evolve at different paces.

5.5. BDD and ATDD

Behavior **D**riven **D**evelopment and **A**cceptance **T**est **D**riven **D**evelopment are two similar technics focused on describing the features in a way all stakeholders can understand them.
The result is often written in natural language (English, French, etc.) and uses unequivocal vocabulary.

This vocabulary is used to form sentences in different tests, which are associated to code.

These tests are ***executable*** and *understood by all.*

As with the TDD, it is more a work management system than a testing technique.

The way these tests are constructed encourage composition and gives stakeholders a better independence and understanding.

BDD formalize how tests are written, with constraints similar to those of a *unit test*.
Sentences must start with:

- **Given** : initial conditions, zero, one, or more → **[0..*]**
- **When** : triggering event, there must be one and only one → **[1]**
- **Then** : assertions, there must be at least one → **[1..*]**

Listing 7. Feature description using the **Gherkin** *format*

```
Feature: Is it Friday yet?
Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"
```

Listing 8. Sentences associations to code

```
public class Stepdefs {
    private String today;
    private String actualAnswer;

    @Given("today is Sunday") ①
    public void today_is_Sunday() {
        today = "Sunday";
    }

    @When("I ask whether it's Friday yet")
    public void i_ask_whether_it_s_Friday_yet() {
        actualAnswer = IsItFriday.isItFriday(today);
    }

    @Then("I should be told {string}")
    public void i_should_be_told(String expectedAnswer) {
        assertEquals(expectedAnswer, actualAnswer);
    }
}
```

① Annotation supplied by the **Cucumber** test framework

6. Technics & best practices

6.1. KISS, YAGNI & DRY

Some popular principles among developers communities:

- **K**eep **I**t **S**tupid **S**imple : The simpler, the better.
  However do something in a simple way can be a difficult exercise.
- **Y**ou **A**in't **G**onna **N**eed **I**t : Only add code useful at the time of its writing.
  Predictions on how a piece of software will evolve are often wrong and code structure that was planned may be an obstacle for real evolutions.
- **D**on't **R**epeat **Y**ourself : Duplication is frowned upon by developers, and bring several issues :
  - It's more work, as the same code needs to be written, tested and maintained several times
  - In case of a bug, multiple fixes must be done, and some may be forgotten

This same simplicity can be found in two of the twelve principles of the **Agile Manifesto**:

- Our highest priority is to satisfy the customer through early and continuous delivery of **valuable** software
- Simplicity [the art of maximizing the amount of work not done] is essential.

7. Log management

Listing 9. Example of useless log

```
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.74
##################
2017-02-04 22:33:12 [thread-2] com.github.some.project.DatabaseService l.75 START
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.93 STOP in
17ms
2017-02-04 22:33:12 [thread-2] com.github.some.project.DatabaseService l.94
##################
2017-02-04 22:33:12 [thread-1] com.github.some.project.DatabaseService l.124
java.sql.SQLIntegrityConstraintViolationException
at org.h2.message.DbException.getJdbcSQLException(DbException.java:345)
at org.h2.message.DbException.get(DbException.java:179)
at org.h2.message.DbException.get(DbException.java:155)
at org.h2.command.CommandContainer.update(CommandContainer.java:98)
at org.h2.command.Command.executeUpdate(Command.java:258)
at org.h2.jdbc.JdbcPreparedStatement.execute(JdbcPreparedStatement.java:201)
... 50 more
```

Listing 10. A little better

```
2017-02-04 22:33:12.332 INFO  5jhgd45ui74h c.g.s.p.UserService.create [Joshua] [Bloch]
Save successful
2017-02-04 22:33:12.758 INFO  6jyg45hgduyg c.g.s.p.UserService.create [Doug] [Lea]
Save successful
2017-02-04 22:33:12.964 ERROR hg457gehe4rt c.g.s.p.UserService.create [James]
[Gosling] Save KO: already exists
```

Log *files* contain all information needed to diagnose issues.
These information can also bu used to build metrics for supervising the system.
Logs are often the sole source of the event chronology and are especially important in high traffic applications where finding the issue of a particular user can seem like looking for a needle in a haystack.

7.1. Log libraries

On the app side, there are many library to help write log events.

Just in the Java ecosystem, we can find:

- JUL (**J**ava **U**til **L**ogging) minimalist, supplied by the JDK
- Apache JULI (**J**ava **U**til **L**ogging **I**nterface)
- Apache Commons Logging
- Apache Log4J

- JBoss Logging
- Logback
- etc.

We find in these various tools common concepts:

- Low coupling between the component that collects log events (`Logger`) and the one which write them outside the system (`Appender`)
- Configuration outside the code (file, system properties, etc.)
- Severity levels:
  - **DEBUG** : use during the development phase, invisible once the application is deployed
  - **INFO** : information about an event in the system (state change, or reaction to an outside trigger)
  - **WARN** : an error happened, the system did not behave as expected, but no manual intervention is immediately needed
  - **ERROR** : an error happened, a human intervention is required to fix the system

NOTE	A business error is not necessarily a technical error. For example: a user entering the wrong password is a normal business case, handled in the code, and the matching event (if recorded) is of **INFO** level.

7.1.1. SLF4J architecture

The issue with having this many tools for writing log is that the ecosystem is mixed.

- Tomcat uses **JUL** through the **JULI** abstraction
- Spring uses **Apache Commons Logging**.
- **HBase** official client uses **Log4j**.
- Etc.

One application,using different frameworks and libraries, should then configure each of these *logging* libraries in a consistent way (same event format, files, retained severity, etc.).

It would be not only annoying, but also error prone (race condition on the same file, configuration oversight, etc.).

However, **SLF4J** was designed to unify all these tools.

SLF4J is made of:

- An ***abstract API*** forming a unique facade for various but common logging features
- ***Adapters*** between this API and existing log libraries
- ***Bridges***, libraries having the same binary compatibility (same qualified class names, same method signatures) as existing log libraries, but redirecting calls to the ***abstract API***

[slf4j bridges] | *slf4j_bridges.png*

Using **SLF4J** means redirecting all logs to this ***abstract API*** and send them one unique implementation.

Configuration is then done once.

7.1.2. MDC (Mapped Diagnostic Context)

The **MDC** is a tool supplied by many log libraries which allows to transport information in a same thread.

The goal is to enhance event information with common data as they are available, without having to pass them as parameters in each methods.

Consider this code:

Listing 11. File MyController.java

```
class MyController {

    private final MyService service;

    public User newUser(User user, @Header("correlationId") String correlationId) {
        return service.newUser(user, correlationId);
    }
}
```

Listing 12. File MyService.java

```
class MyService {

    private final Logger logger = LoggerFactory.getLogger(MyService.class);
    private final MyRepository repository;

    public User newUser(User user, String correlationId) {
        if(isValid(user, correlationId)) {
            return repository.save(user, correlationId);
        } else {
            throw new InvalidUserException();
        }
    }

    private boolean isValid(User user, String correlationId) {
        if(user.age > 110) {
            logger.info("[" + correlationId + "] Invalid User: too old");
            return false;
        } else if(user.age < 1) {
            logger.info("[" + correlationId + "] Invalid User: too young");
            return false;
        }
```

```
        return true;
    }
}
```

The `correlationId` variable is systematically passed because it is required to log it each time, allowing later reconciliation of multiple events matching the same user trigger.

It is a good fit for the **MDC** use:

Listing 13. File MyController.java

```
class MyController {

    private final MyService service;

    public User newUser(User user, @Header("correlationId") String correlationId) {
        MDC.put("correlationId", correlationId);
        return service.newUser(user);
    }
}
```

Listing 14. File MyService.java

```
class MyService {

    private final Logger logger = LoggerFactory.getLogger(MyService.class);
    private final MyRepository repository;

    public User newUser(User user) {
        if(isValid(user)) {
            return repository.save(user);
        } else {
            throw new InvalidUserException();
        }
    }

    private boolean isValid(User user) {
        if(user.age > 110) {
            logger.info("Invalid User: too old");
            return false;
        } else if(user.age < 1) {
            logger.info("Invalid User: too young");
            return false;
        }
        return true;
    }
}
```

MDC variables we want to appear should then be configured in the output format.

Listing 15. File logback.xml

```
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - [%X{correlationId}]
%msg%n</pattern> ①
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

① Pattern used to write events in the console

7.2. GC logs

Causes of an issue can be technical, and related to the **G**arbarge **C**ollector.

So it is important to enable its logs, using a flag when starting the JVM:

```
-Xlog:gc=debug:file=gc.log:pid,time,uptimemillis:filecount=5,filesize=1M
```

In this example, the JVM will write GC events of minimal **debug** severity in file named **gc.log** (with a rotation on 5 files maximum of 1M each) with additional information: PID (processus ID), time and execution duration.

More options here: https://openjdk.java.net/jeps/158

Alternately, metrics collection tools like **Micrometer** can export these information to time-series ingesting systems, such as **Graphite**, **Warp10** or **Prometheus**.

7.3. Heap Dump

When the JVM stops unexpectedly, this can be due to an **O**ut **O**f **M**emory error, and the investigation can be done using a **Heap Dump**.

A **Heap Dump** is the projection in a file of the memory state of the JVM.

This flag needs to be added when starting the JVM, so that such a file is produced in case of a crash:

```
-XX:+HeapDumpOnOutOfMemoryError
```

Produced files can be analyzed with tools such as **Eclipse MAT** (free) or **JProfiler** (licensed).
=== Centralized log management

In order to simplify the process of searching some event in log files, it is important to centralized them.

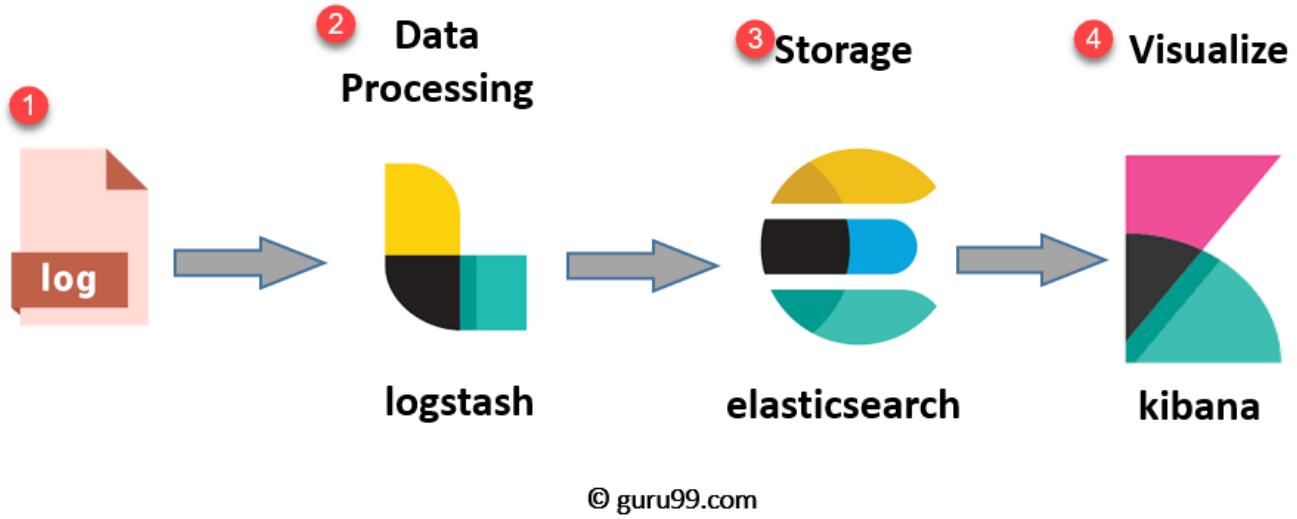
This means send them in the same place, whether it be:

- A file system, in which the `grep` command can be used
- *ElasticSearch*, *Loki* or equivalent, in which logs can be queried
- A SAAS solution whose job it is (*Datadog*, *Logz.io*, etc.)

Logs must be obvious and provide the maximum information possible.
This is why visual visual logging or *debug* events must not be sent to such systems.

7.3.1. The Elastic stack

The most popular solution of log centralization (among others) is the *Elastic* one: **ELK** (**E**lasticSearch, **L**ogstash, **K**ibana).
Each of its component can be replaced by another tool, and the chain can be more complex to handle scaling, but the structure, nevertheless, stay the same.



- **Logstash** slices log lines into structured documents (JSON) and send them to **ElasticSearch**
- **ElasticSearch** indexes them and supply an API to query these indexes
- **Kibana** is a web interface to create dashboards and search arbitrary data

8. Supervision

Collecting logs and metrics is the ground on which supervision is built.

On this, we add components to refine data into:

- Alerts, triggered when an abnormal behavior is detected
- Dashboards, to investigate an issue, find correlations and converge on the issue origin

Supervision is the tool that allows to follow the system state, understand its operation and even predict its use.

ADVICE

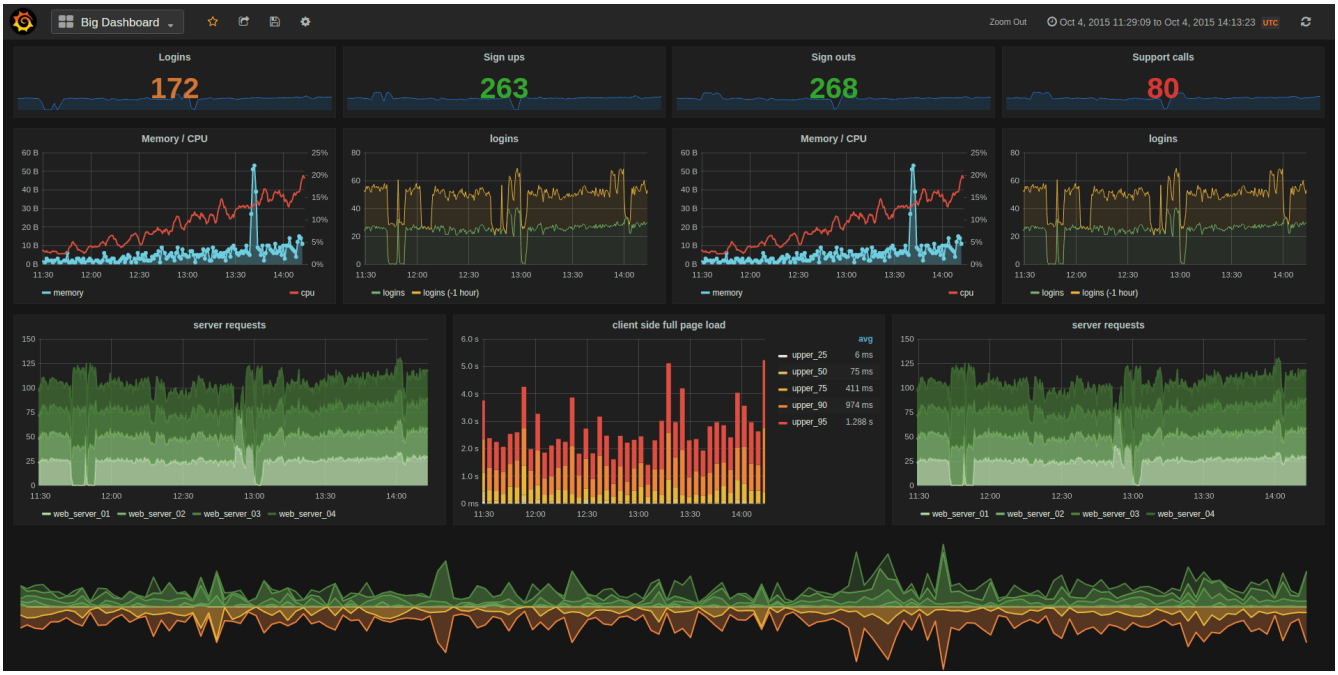
Beware to only keep relevant data in such systems, as it can grow very fast and it can become tricky to navigate in the significant amount of data.

The end product is built with the help of these three questions:

- What are the behavior of the system identified as abnormal? (unavailable service, high resource usage, etc.)
- What system behavior do we want to understand? (water hammer, external service unavailability, etc.)
- Where to put the probes that will generate necessary metrics ?

Tools for document indexation, storage of time-series or databases, all are information source that can be leveraged to build dashboards and alerts.

Today, the most popular choice among open-source tools for aggregating such data is Grafana.



Conclusion

To the contrary of what french industry tries to make us believe, being a software architect is not a job separated from being a developer.

Architectural questions must be handled by the development team, each one having its own experience and interest to a specific matter.

It is the job of software engineers to think about the consequences of their doing, as they will face them anyway, once the product is in production.

Software architecture, as for technology monitoring, is an integral part of the tools portfolio of the developer.

To go further

- Out of the tar pit, essay on complexity
  https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf
- Agile Manifesto
  https://agilemanifesto.org/
- Blogs of
  - Martin Fowler
    https://martinfowler.com/
  - Simon Brown
    https://www.codingthearchitecture.com/
  - Jessie Frazelle
    https://blog.jessfraz.com/
- Conference videos on YouTube
  - Devoxx
  - goto;
  - DevFest
  - JUGs (**J**ava **U**ser **G**roup)
  - etc.
- Tech news websites
  - InfoQ https://www.infoq.com/fr/
  - Developpez (FR) https://www.developpez.com/