Introduction to full stack web development with Go and Vue.js

Enrico Bassetti, Emanuele Panizzi

Contents

Preface

Who is this book for?

The ideal reader is a student enrolled in a Computer Science university program: someone with a basic knowledge of at least one programming language and willing to learn how to build a full-stack web application.

This book is also suitable for developers who want to develop real-world web applications professionally using state-of-the-art technologies.

Requirements

To fully understand and follow the examples in this book, you should have some basic knowledge of how to code in at least one programming language and use your operating system's command line for launching tools and managing files.

We suggest you test the examples in this book in a Debian GNU/Linux real or virtual machine. We tested these examples on GNU/Linux; they might work in other operating systems, provided that tools are installed and configured correctly (however, some commands might be different, and you need to translate them). If unsure, you can download the Debian ISO freely from https://www.debian.org and VirtualBox from https://www.virtualbox.org/, and then use VirtualBox to create a Debian virtual machine.

If you want to deepen your knowledge of the GNU/Linux command line, we suggest "*The Linux Command Line*" book from *William E. Shotts, Jr.* here: https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/download.

Typographic conventions

Examples that involve running programs on the command line will use the dollar sign $ to denote the shell prompt as a standard user. Since the dollar sign indicates your shell prompt, you should not type it in.

Lines without the dollar sign represent the output for the command. For example, in this block of text:

```
$ git --version
git version 2.30.2
```

The text `git --version` is the command you need to run, and `git version 2.30.2` represents the command's output.

Note that, in UNIX philosophy, commands that run successfully should have no output (if not explicitly requested using a flag). Many of the commands that we will use have this behavior.

Note for e-book readers

Some e-book readers have a small screen, and some images/tables may overflow out of the screen. To scroll laterally and see the rest of the image or table, consult the manual of your e-book reader.

About the Authors

Emanuele Panizzi is an Associate Professor in Computer Science at Sapienza University of Rome, Italy. He leads a research team focusing on human-computer interaction, app design, gamification, and context-aware mobile interaction. In the two areas of smart parking and earthquake detection, his current study uses AI to recognize users' behaviour and context. Designing mobile user interfaces with implicit interaction and crowdsensing applications is the experimental component of this study. Panizzi supervised several large software projects for Sapienza University and for companies and start-tups. He teaches HCI and software architecture. He has served as a consultant for major national and international corporations.

Enrico Bassetti is a post-doctoral researcher at Sapienza University of Rome, Italy. His research focused on network security, fully distributed sensing systems, and Internet-of-Things devices. In 2019, he earned his Master's in Cybersecurity from Sapienza University and defended his Ph.D. in 2023. In addition to his research experience, Bassetti has ten years of company experience as a system architect consultant, DevOps advocate, system/network administrator, and trainer on security and Linux.

If you have any feedback about any aspect of this book, email us at panizzi@di.uniroma1.it and bassetti@di.uniroma1.it.

1 Introduction

1.1 Types of Applications in Web Development

In the world of web development, applications are the heart and soul of the digital landscape. These computer programs enable users to accomplish a wide array of tasks, from managing their finances to playing games or simply staying connected with friends and family. Applications come in various shapes and sizes, tailored to suit different platforms and user needs.

1.1.1 Desktop and Mobile Applications

Desktop applications, also known as native desktop applications, are designed to run directly on a desktop computer's operating system. To use a desktop app, users typically need to install it on their computer. These applications can vary widely in their complexity, from lightweight utilities to sophisticated software suites.

Mobile applications are tailored for smartphones, tablets, or even wearable devices like smartwatches. Like desktop apps, mobile apps also need to be installed on the user's device. They can be further categorized into two types:

- *Native Apps*: Native apps are explicitly developed for a specific mobile operating system, such as Swift for iOS, Java or Kotlin for Android, or Objective-C for older iOS versions. This approach provides developers with the maximum control and access to device-specific features and capabilities.
- *Hybrid Apps*: Hybrid apps, on the other hand, are created using web technologies like HTML, CSS, and JavaScript and then cross-compiled to run on multiple operating systems. Tools like Apache Cordova or frameworks like Ionic allow developers to build hybrid apps that can be deployed on both iOS and Android platforms.

1.1.2 Web Applications

Web applications, often referred to as web apps, are distinct from their desktop and mobile counterparts. They run within a web browser, which means users don't need to install them on their devices.

Instead, web apps can be accessed simply by navigating to a specific URL. These applications are ubiquitous on the internet, offering a wide range of services, from email clients to project management tools.

A specific subset of web apps is known as *Single Page Applications* or SPAs. SPAs interact with users by dynamically updating the current web page with new data from the web server, without requiring a full page reload. This approach results in a smoother, more seamless user experience, as only the necessary content is updated, making the application feel more like a traditional desktop or mobile app.

Progressive Web Apps, or PWAs, are a modern evolution of web applications. They offer the best of both worlds, combining the accessibility of web apps with the capabilities of native apps. Users can install PWAs by adding them to their device's home screen, providing easy access like a native app. Additionally, PWAs can work offline or in low-quality network conditions, making them highly resilient and user-friendly.

1.1.3 Web APIs

Web API stands for Web Application Programming Interface. It is a set of rules and protocols that allows one software application (desktop, mobile or web application, or even servers) to request and exchange data or functionality with a server over the Internet. Web APIs are the building blocks that enable different web services, servers, and clients to communicate effectively.

At their core, web APIs define a collection of endpoints or URLs that developers can use to send requests and receive responses in a structured format, typically in JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). These endpoints are like doors into a web service, each offering a specific function or data resource.

1.2 Full stack web development

Full stack web development refers to the process of creating web applications that encompass both the front-end and back-end aspects. A full stack developer is proficient in working with all the technologies and layers involved in building a web application, including the client-side, server-side, and the underlying database.

The typical full stack application can be described by these layers:

- the "**Front-End**" (also known as "client-side"): is what users directly interact with through their web browsers; it includes the visual elements, layout, and interactive features that users see

and engage with. Front-end technologies commonly include HTML, CSS, and JavaScript, along with various front-end libraries and frameworks like Vue.js, React, or Angular.

- the "**Back-End**" (also known as "server-side"): is responsible for handling business logic, data processing, and communication with databases and external services. It manages requests from the client-side, processes them, and sends back the necessary data or responses. Back-end development involves working with programming languages like PHP, Go, Python, Node.js, Ruby, or Java, as well as frameworks such as Express, Django, or Spring.
- the **database layer** of a full stack application stores and manages the application's data. It could be a relational database like MySQL or PostgreSQL, a NoSQL database like MongoDB or Redis, flat files, or any combination of those.

In this book, we will go through these layers, and we will present some of these technologies, namely OpenAPI for API definition, Go for backend development, Vue.js for frontend development. We won't cover the database part,as there are multiple valid books on the topic.

Additionally, we will see how a full stack application can be deployed, and we will present the "containers" as deployment strategy.

1.3 Structure of a web application

There are several different approaches when dealing with web application. Regardless of programming languages and frameworks that are involved, you may have *server-side* or *client-side* rendered application.

In *server-side* rendering, the frontend code (JavaScript, HTML, CSS) is rendered on the server side. The frontend browser receives web pages already formatted and it draw them on the screen. Using this approach, the HTML code (and its generator) are embedded in the backend project, and there is no need for serialization and de-serialization of data. Responses may also contains partial web pages: their content replaces a portion of an already-shown web page. Requests may be synchronous or asynchronous (i.e., in background).

In *client-side* rendering, the backend and the frontend are two different applications. Data and message exchanged between the two applications are serialized and encoded in HTTP messages. The frontend is in change of preparing the HTML and CSS based on data and messages exchanged with the backend. Differently from the *server-side* rendering, the backend does not produce HTML: data are serialized to the frontend, which is expected to fully manipulate them. In this type of rendering, JavaScript is heavly used to request data from the backend and build UI elements. Requests are almost always asynchronous.

2 Version Control with Git

To introduce you to *version control*, let us tell you a story (based on `notFinal.doc` PHDcomics). Let's say that you are at the end of your final year at the university, and you are writing your final thesis. You write the document, and when you think that it's ready, you name it `final.doc`, and you send it to your supervisor/professor for revision and approval.

The professor reads your thesis, adds comments, and marks paragraphs of the text that you should rewrite. And they send back the document, named `final.comments.doc`. You eagerly start modifying your thesis, and you send it as `final_rev2.doc`. And then your professor sends back a `final_rev2.comments2.doc`. Now you send back `final_rev2.comments2.corrections1.doc`, and they send back… you get the point.

At the end of this back and forth, you end up with many files with inconsistent naming at best, hoping that the last created file is also the latest and greatest version.

This problem is named *versioning*, and it is common, especially when multiple people work on the same project, or the project lasts for years.

2.1 Version control systems

Version control systems (VCS) start from a set of needs every team or solo developer face after a few years. Projects grow in complexity and size; several changes are made by different collaborators/developers, and tracking and evaluating those changes is necessary. Sometimes, projects may have multiple versions being developed or tested concurrently (consider, for example, trying various optimizations to the same algorithm). And when the project is finally months/years old, there should be a way to track down when a specific feature or bug was introduced in the code.

Version Control Systems are now essential tools for modern software development and collaborative projects (even in low-code or no-code projects). They provide:

- Consistent labeling of revisions: multiple versions of a file are labeled using a standardized way (enforced by the tool);
- Tracking of changes: each difference between two revisions is tracked, and it can be recalled in the future (for example, if you need to explore an older version of the code);

- Metadata (date, authors, etc.) to revisions, to help, e.g., tracking down when and who made a particular change;
- Consistent **point-in-time revisions**: each revision is saved explicitly, and it may contain multiple files;
- Branches: revisions may diverge, and the same project may have multiple parallel "versions", clearly labeled;
- Merges: different branches may be "merged" by merging their changes;
- Synchronization between multiple users and computers.

Various applications provide these features, such as CVS, Subversion, Mercurial. We will look at Git, a modern solution that quickly became industry standard a few years ago. Note that most (if not all) concepts are shared between different version control systems.

2.2 Git

Git, a distributed version control system, was created in 2005 by Linus Torvalds to manage the source code of the Linux Kernel. It utilizes directed acyclic graphs (DAG) and data structures similar to Merkle trees. Thanks to its distributed nature and robust capabilities, Git has become the industry standard for version control. It enables developers to collaborate seamlessly, manage complex projects, and maintain a comprehensive historical record of code modifications.

We will introduce Git incrementally in the following sections.

2.2.1 Repository

A *repository* is a set of *commits*, *branches*, and *tags*, usually for the same project. We will define all these terms in the following sections; for now, think of the repository as a "bucket" which contains all Git-related items for a given project.

A project (for example, a mobile app) may be divided into multiple repositories for organizational reasons (for example, because some parts have a different life cycle). Nevertheless, for the sake of simplicity, in this book we assume that a repository contains a project and that a project is in one repository.

2.2.2 Working copy

The *working copy*, or *working directory*, is a local copy of the project. It is the set of files tracked by the version control system, and it's usually the directory that contains your project. It includes subdirectories as well. Each time you add, modify or delete a file, you need to tell Git about that.

Git manages the files in the working copy. When switching between revisions, Git changes all files in the working directory to align the content to the one they had at the time of that revision.

Example: let's say we have a file named `travels.txt` in our working copy. Suppose that the content during the first revision was:

```
2022 Rome
```

Now, suppose that we have another travel to add into the file, and we change the content by creating the second revision with the following content:

```
2022 Rome
2023 Paris
```

If we switch back to revision 1 (using the appropriate command), `git` will *automatically* change the content of the file to:

```
2022 Rome
```

We will see this example again in the next section.

2.2.3 Commit

A *commit* is the snapshot of the working copy in a given moment. It can include some or all files in the project. When you prepare your working copy for a commit, you decide which files should be in a commit and which don't.

In our examples, we will represent a Git commit with uppercase letters, like A.

Usually, each commit has one parent. Sometimes, when there is a *merge* between *branches*, a commit may have multiple parents (we will see in a moment what *merge* and *branches* are). In some exceptional cases, like the first commit, a commit may have no parent. In that case, the commit is called "orphan".

In Git, a commit is uniquely identified in a Git repository by an alphanumeric string. The string is the SHA1 hash of the commit content described below. **Every field described below is part of the hash**. For example, `20ea1e7d13c1b544fe67c4a8dc3943bb1ab33e6f` is a commit identifier.

> SHA1 is a hash function. A hash function is a function that maps data of arbitrary size to fixed-size (or reduced-size) values. Due to the pigeonhole principle, this function may create collisions (two input values can return the same output). Values returned by a hash function are called hash values or digests.
>
> One fundamental property of hash functions is the "collision-resistance": a hash function is collision-resistant if it is hard to find two inputs that result in the same hash value.

Sometimes you may find shorter version of the commit identifier. The identifier can be shortened to any size by truncating the string after a given number of characters. However, to avoid mistakes due to potential ambiguities, do not use short identifiers when modifying a repository: always use the full ID.

A commit **requires** a meaningful *message*. The commit *message* comprises a mandatory summary of 72 characters at most and a detailed explanatory text after a blank line. The latter is optional; if present, it should wrap at 72 characters.

Example from the commit ID `20ea1e7` in the Linux kernel tree:

```
file: always lock position for FMODE_ATOMIC_POS

The pidfd_getfd() system call allows a caller with ptrace_may_access()
abilities on another process to steal a file descriptor from this
process. This system call is used by debuggers, container runtimes,
system call supervisors, networking proxies etc. So while it is a
special interest system call it is used in common tools.

That ability ends up breaking our long-time optimization in fdget_pos(),
which "knew" that if we had exclusive access to the file descriptor
nobody else could access it, and we didn't need the lock for the file
position.
```

The commit also contains the name and the e-mail of the *committer*, which is the person that created the commit in the first place, and the commit *date*. This information is useful when you look for the code by specifying a time window.

The *author's* name, e-mail, and date are also present. Typically, these correspond to the *committer* information. In some cases, they may differ, e.g., when the code's author is not the same person that created the commit in the repository.

Commits also contain a copy (sort of) of the working directory at that moment in time. The field is named *tree*. If you are interested in Git internals, you can look at the Git Book (links are at the end of the chapter).

Finally, a commit contains the identifier of the parent or parents commits, if any.

> **Recap:** a git commit contains
>
> - **Commit message**
> - **Committer** and commit date
> - **Author** and author date
> - **Tree** (hash of all files in the commit, sort of)
> - **Parent** commits
>
> It's identified by an alphanumeric ID, the SHA1 of the above fields.

2.2.4 Staging area

We said before that a Git commit may contain only a subset of modified files. To achieve this result, Git has the "staging area" concept. The *staging area* is the set of all changes selected to be in the next commit.

Before issuing the command for creating a commit, you should add all added, modified, or deleted files to the staging area.

> While counterintuitive, deleted files should also be added in the staging area, as the file deletion is a *change* that must be tracked.

Note that a file may have multiple changes (e.g., changes in different lines): Git allows you to put only some parts in the staging area, leaving the others outside the commit. For simplicity, in our examples, we will always put the whole file in the staging area.

2.2.5 Branch

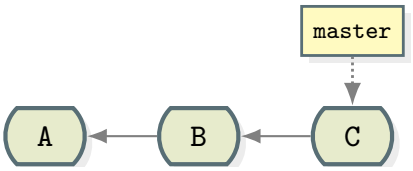
A *branch* is a line of development, an ordered set of commits linked together in a *Direct Acyclic Graph (DAG)* by a parent-child relationship. It has a name and starts with a commit. **The branch always points to the latest commit for the branch**, and its history starts from the first orphan in its history

(which usually is the very first commit of the repository). In other words, the history of a branch contains all parent commits until you reach an "orphan". Part of the branch history is usually shared with other branches.

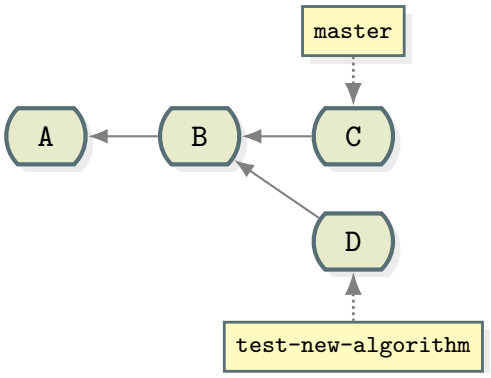
When you start a new Git repository, it's empty (no branches, no commits). As soon as you begin creating commits, you make the first branch. Git automatically assigns a name to this branch (by default, `master`). Each subsequent commit will be a child of the last commit in that branch.

> Actually, you can create a commit with no branch associated. However, it's not desirable, as it is difficult to retrieve, and the Git *garbage collection may delete it*.
>
> The *garbage collection* (GC) is a periodic operation that Git performs to remove the old stuff and optimize the repository.

Example: a simple repository may have the first commit A, then a second commit B (a child of A), and then the third commit C (a child of B). The arrow represents the parent-child relationship, and it always points to the parent commit. The `master` branch name is pointing at the latest commit of the branch:



Example 2: two branches, `master` and `test-new-algorithm`. They share commits A and B; however, they *diverged* after that. C changes are in the `master` branch but not in the `test-new-algorithm` branch. At the same time, changes in the D commit are present in the `test-new-algorithm`, but not in the `master` branch. Changes in A and B are present in both branches, as these commits are in common.



Branching (i.e., the action of creating one or more *branches*) is a powerful feature, as it allows developers to create a parallel line of development. For example, testing a new algorithm while other developers are still working on the code is facilitated by branching, as the new algorithm can have its

own dedicated history.

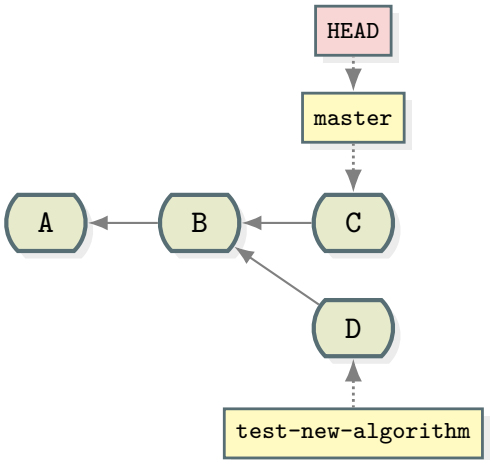
At the end of the parallel development, a branch may be abandoned (e.g., if the experiment failed), or it can be merged into other branches (more on this later on).

Note: the main line of development (`master` in the example above) is a branch itself - nothing special!

2.2.6 The HEAD

In Git, the special name HEAD is the pointer to the current location in the Git history. The "current location" determines the state of the working copy. In other words, files in the working copy are aligned to their content at the commit pointed by HEAD.

HEAD may point to a commit, a branch, or a tag. Git allows us to move HEAD with a dedicated primitive, named *checkout*, which updates our working copy to reflect the change of HEAD.



When HEAD is NOT pointing to a branch (for example, when we moved directly to a commit or a tag), we are in a special case named `detached` HEAD. In this state, we cannot create new commits. We should move HEAD to a branch to create new commits.

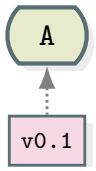
> Note that, as stated before, commits can be created outside branches (even in `detached` HEAD); however, the risk of data loss is concrete. To be safe, **never** create a commit outside branches.

2.2.7 Tags

Sometimes you want to label a specific commit so that you can quickly return to that commit in the future. This is common when dealing with documents or software *versions*: you release a new ver-

sion of your software by building it from a specific commit, and you tag that commit with the version number.

For example, the commit A corresponds to the code of the released application at version `v0.1` because that version was built when HEAD was pointing to A. In the future, if and when we need to return to the code at version `v0.1`, we can use the tag to find the correct commit.



You may remove tags, although it's discouraged as a tag is considered "immutable", and doing so may break other systems if the repository is shared.

2.2.8 Merge

When changes in a branch are ready to be moved into another branch (which can be the main line of development or any other branch), we can *merge* them.

The *merge* action will result in a *merge* of changes between the two branches. Usually, when you merge one branch into another, the latter contains both changes, while the former is left unchanged.

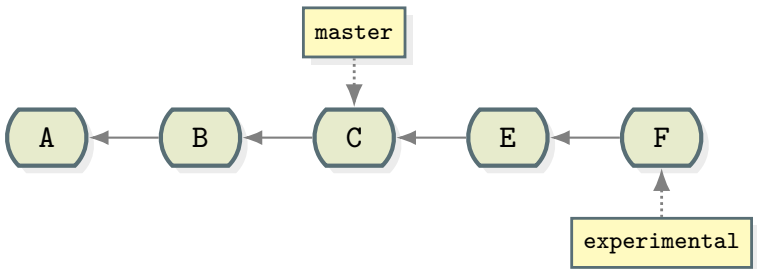
There are different ways of merging branches, named "strategies": *fast-forward*, *non-fast-forward* (or "*merge commit*"), and *rebase*. Depending on the situation, you can choose any legal strategy to do.

In the following sections, we will see these strategies. In the examples, we will merge the `experimental` branch into `master` (i.e., adding all changes from `experimental` to `master`).

2.2.8.1 Fast-forward strategy

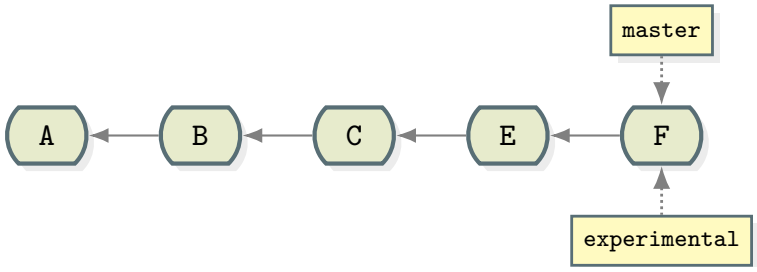
The *fast-forward* strategy is the most straightforward. It can be applied only when the new branch `experimental` is a direct continuation of the branch where we are merging (`master`, in this example). The merge action will simply move the `master` branch pointer on the same commit of the `experimental` branch.

Example: let's suppose that we have this situation:



In this case, `master` contains A, B, and C, while `experimental` has A, B, C, D, E, and F.

To merge `experimental` into `master`, we move the `master` label from C to F. By doing so, we are telling Git that the next time we want to move our HEAD to `master`, we want to see the state of the repository at F (which means F plus all the previous changes).

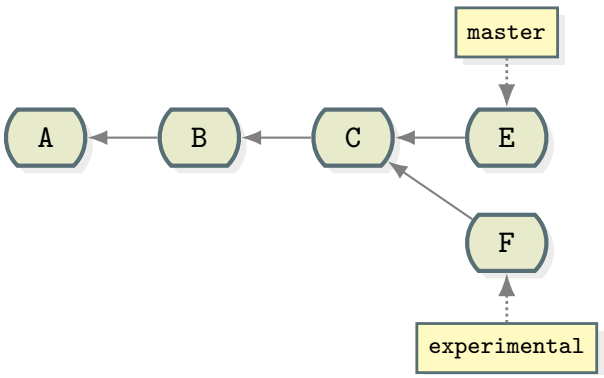


The merge is done, and the `experimental` branch may be removed, as now `master` is the same as `experimental`.

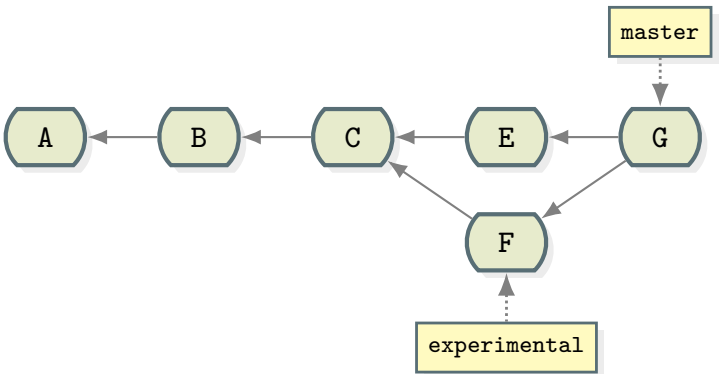
2.2.8.2 Non-fast-forward strategy

A more versatile strategy is the *non-fast-forward*. It can be applied even if the history is non-linear. This strategy requires the creation of a new commit in the `master`, which contains the changes from the `experimental` branch. To do so, **the new commit will have two parents**: the last commit of `master`, and the last commit of `experimental`. Then, `master` is moved to point to this newly created commit.

Example: suppose that you have the repository below, and that you want to merge `experimental` into `master`:



We need to create a new commit, G, which has two parents: E and F. Then, `master` is moved to G.



The difficulty of this strategy is that, differently from the *fast-forward* strategy, we may have *conflicts* here. We will see how to deal with conflicts later in this chapter.

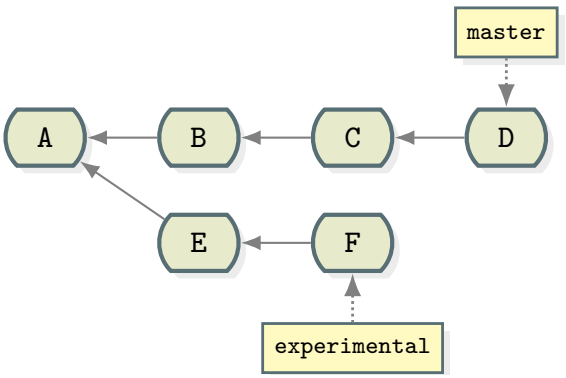
2.2.8.3 Rebase strategy

The idea behind the *rebase* strategy is to modify the history *before* merging so that the new history will be linear. Then, we can apply the *fast-forward* strategy.

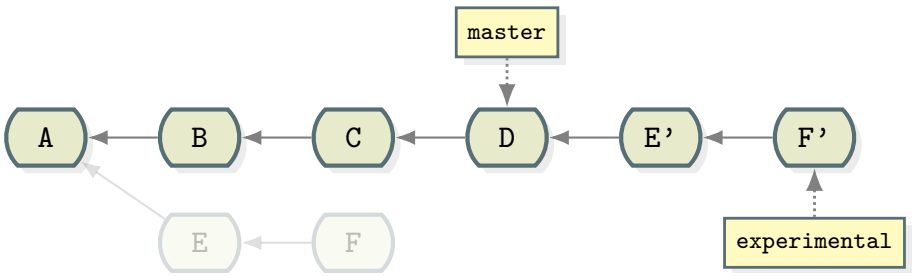
To modify the history, we should create a new copy of all commits in the `experimental` branch that are not in common with the `master` branch. The first copied commit that is not in common should have the latest `master` commit as parent. All other commits will follow as usual.

Think of the *rebase* strategy as if you create a new branch starting from the latest commit of `master`, and then you re-create each commit that `experimental` has by applying the same changes.

Example: suppose that you have the repository below, and that you want to merge `experimental` into `master` using the *rebase* strategy:



The first step is to create `E'` ad `F'` commits. The `E'` commit, differently from E, will have D as parent. `F'` will have `E'` as parent. Note that E is different from `E'` and F is different from `F'` (see the commit definition for the reason of this). Once these two commits have been created, `experimental` may be moved to `F'`. At this point, we are ready for a *fast-forward* merge (`master` can be moved to `F'` easily).



Note that, in this case, E and F won't be attached to any branch after the merge (*dangling commits*). This means that they will be removed.

> Dangling commits are commits that are not attached to a branch or a tag. Theoretically, these commits won't exist as they are removed during the rebase. In practice, they remain in the repository until the next *garbage collection*.

This strategy also may expose *conflicts*. In the next section, we will see how to handle these conflicts.

2.2.9 Merge conflicts

When a *non-fast-forward* or *rebase* merge occurs, there might be a situation where the same file has been changed in both branches (`master` and `experimental` in the examples above). In this case, Git tries to merge these changes by applying them from both branches. This works if the changes are "distant" in the file (i.e., in different lines, with some untouched lines between them).

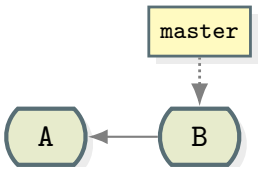
If, on the other hand, the changes are colliding along the same lines, Git cannot provide a solution. So, we have a *merge conflict*.

> A conflict is also present when the file is changed in one branch and deleted in the other.

Let's see a visual example of this problem. Suppose that we have the following file `example.go` (the programming language is not important at the moment):

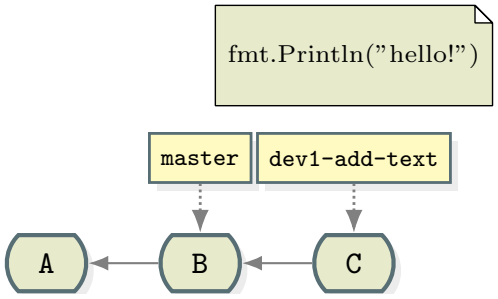
```
package main

import "fmt"

func main() {
  fmt.Println("")
}
```

The current status of the history is the following:

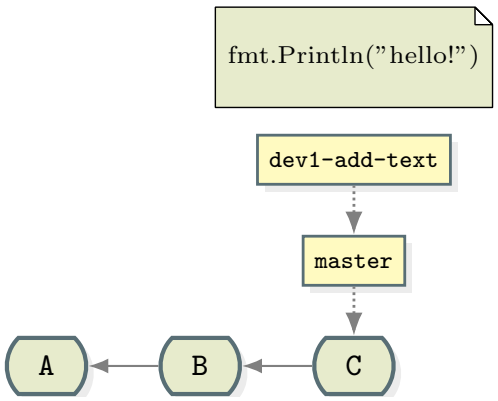


Suppose that two developers change the message in parallel to two different texts.

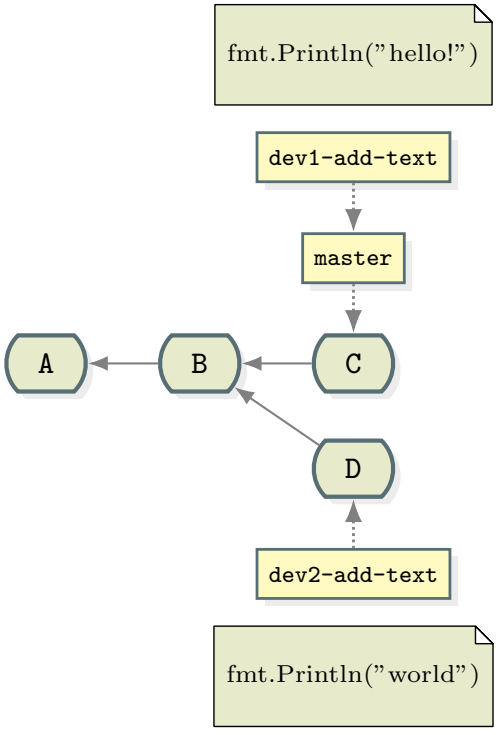
The first developer, `dev1`, creates a new commit in a new branch `dev1-add-text`, replacing the `fmt.Println("")` with `fmt.Println("hello!")`:



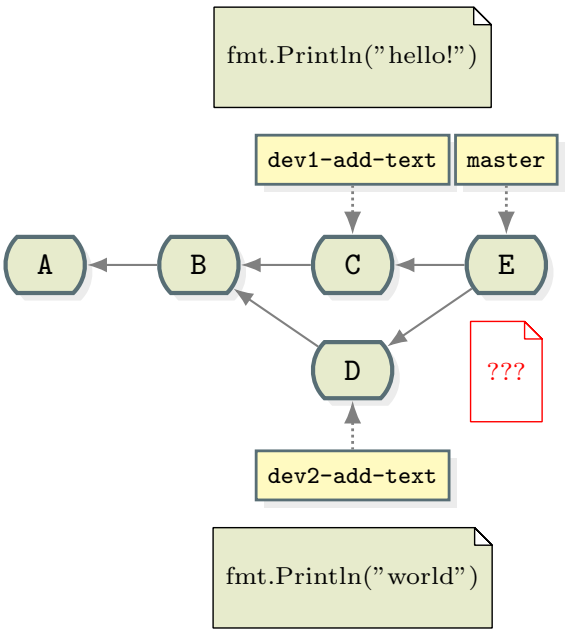
We can merge `dev1-add-text` to `master` using a simple *fast-forward* merge. The result will be:



Suppose that, while dev was working at his commit, or while we were merging his branch, dev2 created a new commit in a new branch `dev2-add-text`, replacing the `fmt.Println("")` with `fmt.Println("world")`:



Now we have a conflict: the same file has been changed in two different branches simultaneously. When we try to do a *non-fast-forward* merge from `dev2-add-text` to `master`, Git won't be able to resolve the conflict automatically. We need to do something.



When a conflict occurs, you will notice that Git will halt the merging procedure, and it will present you a description of the current status, informing you about what is in conflict.

At that moment, if we open the `example.go` file, we will see something like this:

```
package main

import "fmt"

func main() {
<<<<<< HEAD
  fmt.Println("hello!")
=======
  fmt.Println("world")
>>>>>> dev2-add-text
}
```

The file has been modified: Git added **both** lines inside three markers. In the first part, we have the line at the HEAD (which is C from the example above, as we just fast-forward-merged `dev1-add-text` into `master`). In the second part, we have the line as it is in the `dev2-add-text` branch.

There are three ways to resolve these conflicts:

1. By keeping the already-merged file from dev1: in this case, we can tell git to accept "ours" version;
2. By using the new file from dev2 by telling Git to accept "theirs";

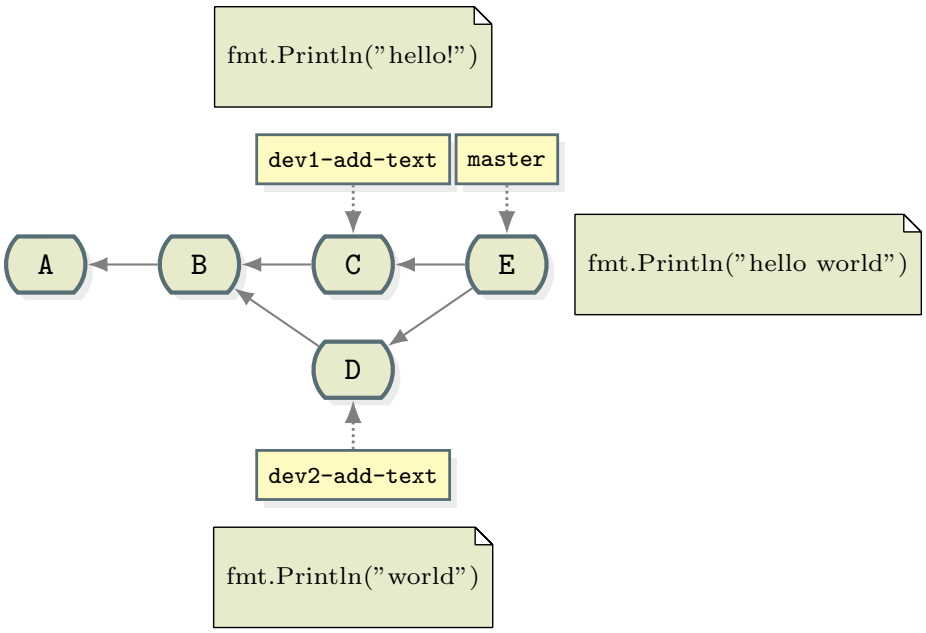
3. By editing the file manually.

Options 1 and 2 are implemented using a specific Git command. The third option requires opening the file and modifying it directly.

For example, let's say we want to merge the two changes manually to print `hello world!`. We can open the file with any text editor, remove markers and replace the two prints with a new one:

```
package main

import "fmt"

func main() {
  fmt.Println("hello world!")
}
```

At this point, we tell Git that we changed the file (by adding the file to the staging area), and we can continue the merge:



2.3 Remote git repositories

Earlier in this chapter, we claimed that a VCS enhances collaboration, and we described Git as a distributed version control system. However, for now, we used Git in a local fashion.

Git can retrieve and send commits, tags, and branches to remote repositories in remote hosts, like a server. A URL and a name identify these remote hosts. These are named *remotes*.

> Actually, Git is fully capable of using other PCs as "remote". For example, if there are three developers in a group, each one can set up two remotes, one for each other colleague.
>
> Git can also use *directories* as "remote", for example, in a USB disk. See the Git manual for all options.

The default remote is named `origin`. This name is used in operations where a remote is added without a name. However, Git supports different remotes and arbitrary names.

When Git downloads commits and tags from remotes, they are copied as-it-is in the local repository (no changes). Branch names, however, are prefixed with the name of the origin. For example, the `master` branch in the remote `origin` will be named `origin/master`. These labels are used to keep track of the status of remotes: using this technique, Git is able to tell whether synchronization is necessary (more on this later).

Note that you can't create a commit in a branch prefixed by a remote: their purpose is to keep track of remote branches. To create a commit in a remote branch, you must create a "local" branch linked to the remote one. Git handles this automatically, as we will see later on.

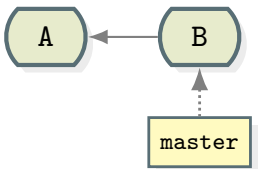
The following sections will show the basic primitives for synchronizing Git repositories across computers.

2.3.1 Clone

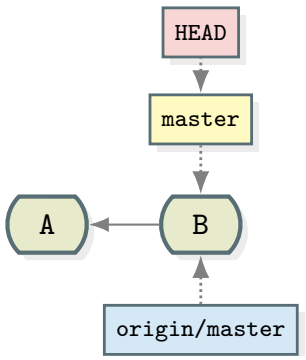
The action of downloading a remote repository for the first time is named *clone*. This action creates a full copy of the remote Git repository in our local PC and creates a working directory. The URL that is provided to the clone action will be the `origin` remote.

By default, the working directory is checked-out to the same HEAD the remote repository uses.

Example: let's say that the remote host contains this repository:



When issuing a *clone* action, Git creates a local repository by downloading all commits (A and B) and all branches (in this case, only `master`, which becomes `origin/master`). Git also creates a local branch `master`, which corresponds to `origin/master`, and sets the local HEAD to the remote HEAD.

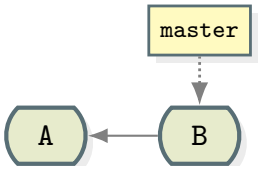


The repository is ready. We will see shortly that other primitives are available to synchronize commits between remotes and local repositories and to synchronize branches.

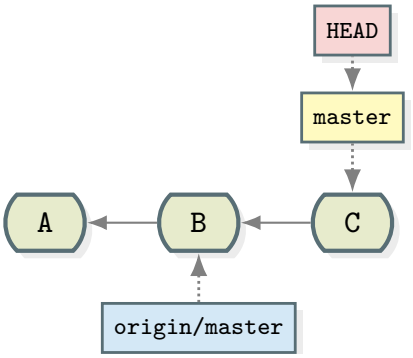
2.3.2 Push

We use the *push* action to update remote branches or tags. When using the push action, Git sends the specified branch or tag to the remote. In particular, if the local branch is tracking a remote branch, the remote branch is advanced to the same position as the local branch. If the remote branch or tag does not exist, it is created. New commits (i.e., commits for that specific branch/tag that exists locally but not in the remote host) are sent automatically.

Example: suppose we have the remote `origin` with this repository:

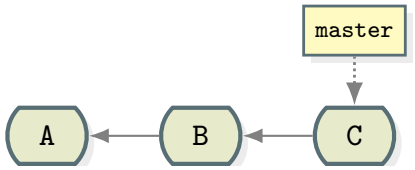


Suppose, also, that we have a local repository (created using *clone*) where we create a new commit C in the `master` branch:

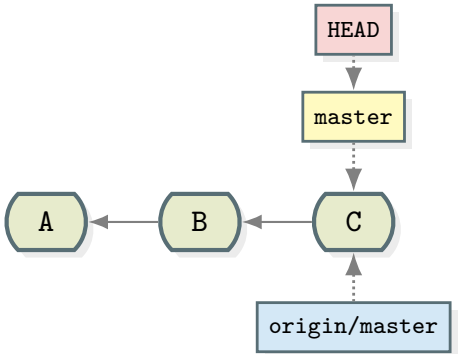


As you can see, when we create C, we update the position of `master` (local branch), but the remote branch `origin/master` still points to B. When issuing a *push* action for `master` against the `ori-`

`gin` remote, Git will send C commit and the new position for remote `master`. The remote repository is updated as follows:



Also, the local repository is updated to reflect this change, i.e., `origin/remote` is synchronized to the location of the remote `master` branch, which is now pointing to C:

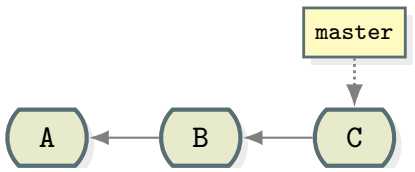


2.3.3 Fetch

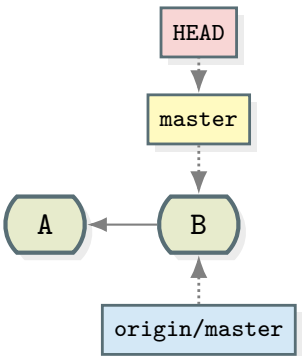
The *fetch* primitive synchronizes remote branches and tags into the local repository. Note that *fetch* updates the labels for remote branches only, meaning that **only prefixed branches are updated**. Suppose the remote is `origin` and exists a branch named `master` both locally and in `origin`. In that case, only `origin/master` in the local repository is updated. The local `master` is left untouched.

All missing commits are downloaded from the remote into the local repository.

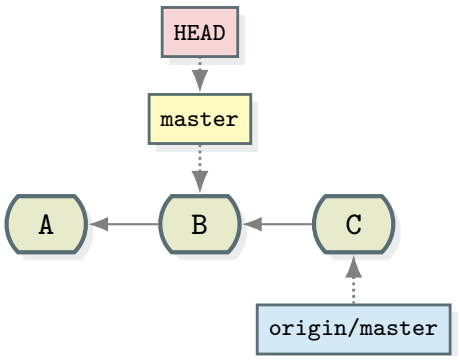
Example: suppose that we have this remote repository in `origin`:



Suppose that our local repository is older: C is missing, and `master` is still pointing to B. Note that, before executing a *fetch*, both `master` (the local one) and `origin/master` (the reference to the remote `master` on `origin`) are still on B, as Git doesn't know that the remote repository has changed:

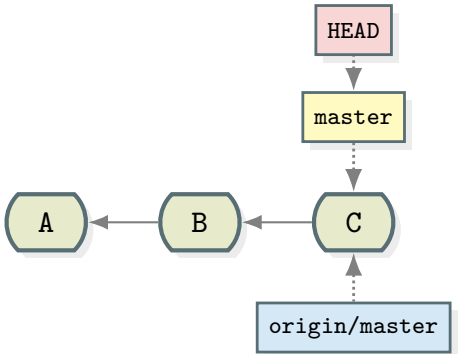


Upon executing a *fetch*, our local repository will be updated to that `origin/master` points to C (which is downloaded as part of the fetch operation):



You can inspect the C commit and the `origin/master` branches using any Git tool, as they are in your local repository. However, your local `master` will remain B unless you decide otherwise.

At this point, it should be clear what operation can be used to "move forward" your local `master` to the location of `origin/master` (C). The operation is a *fast-forward* merge of `origin/master` into `master`:



As this operation is very common, Git provides a single command: the *pull*.

> Note that you can use other strategies for merging, like non-fast-forward or rebase. This may happen when the remote and local branches diverge; for example, when you create a new com-

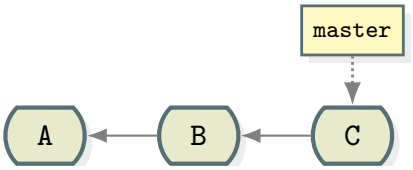
> mit locally before synchronizing with the `origin`, and the corresponding branch in the `origin` gets updated after your last synchronization. In that case, you may also experience *merge conflicts*!
>
> To avoid this problem, we strongly advise issuing a Git pull (or Git fetch + merge) regularly, or at least when you start editing your working copy.
>
> Also, we strongly advise having only one developer working on a branch. If multiple developers want to work on the same code, there are several alternative solutions, such as IDE synchronization tools like "Code With Me" from JetBrains or "Live Share" for Visual Studio Code.

2.3.4 Pull

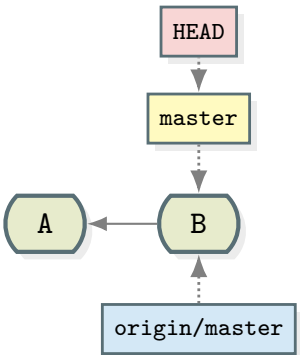
During a typical development workflow, Git *fetch* is hardly used without a subsequent *merge*. Usually, developers want to update their local branch to the remote one. So, the *pull* operation is provided as a shortcut for *fetch* and *merge* actions.

Using *pull*, Git automatically fetches the information from the remote and then executes a *merge* from the remote branch to the current branch where HEAD is.

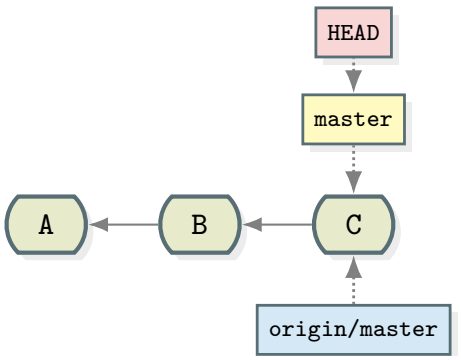
Example: suppose that we have this remote repository:



Suppose also that we have this local repository, which is an older copy of the remote one (note that C is missing):

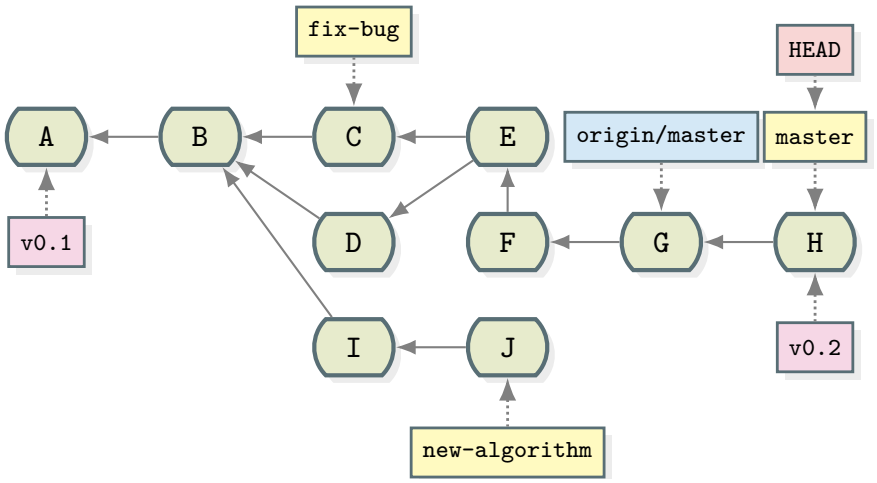


By executing the *pull* action, Git automatically fetches the remote information, downloads the C commit, updates `origin/master` to its new location, and merges `origin/master` into `master`, updating the local HEAD.



Now, the working directory is synchronized with the remote origin.

2.3.5 Full example of commit history and branches



In this example, A is the root of our repository (the first commit) and the version `v0.1`. A and B are shared by all branches. `fix-bug` and `master` share C, but `master` also contains D (which was parallel to C), E, F, G, and H (which is both the version `v0.2` and the current HEAD). The `new-algorithm` branch contains only A, B, `I` and `J` changes. The remote `origin` has an old copy of the `master` branch up to G, as we can see thanks to the label `origin/master`.

2.4 Git repository hosting

Until now, we assumed that the Git remote was some host somewhere that we could reach using a URL. Although Git can use a Linux box via SSH to store the remote repository, many developers use third-party applications to host their Git repositories, as they provide additional tooling for project management and easy collaboration. These third-party applications are usually called “forges”.

Using a "forge" has multiple benefits:

- It is always online: you and your collaborators may synchronize your repository with no time restrictions;
- integrated solution for bug and feature request tracking;
- integrated documentation as wikis;
- tooling to manage forks and pull/merge requests (we will see later these two concepts);
- and many other things, like pipelines for continuous integration and continuous delivery, web IDEs, integrations with other tools, etc.

Widely known "forges" are GitLab, GitHub, BitBucket, and Gitea. Some can be self-hosted, others are offered only as a service.

We will briefly introduce three concepts present in all "forges". These are the main reason why you want to use these platforms.

2.4.1 Bug and feature requests tracking

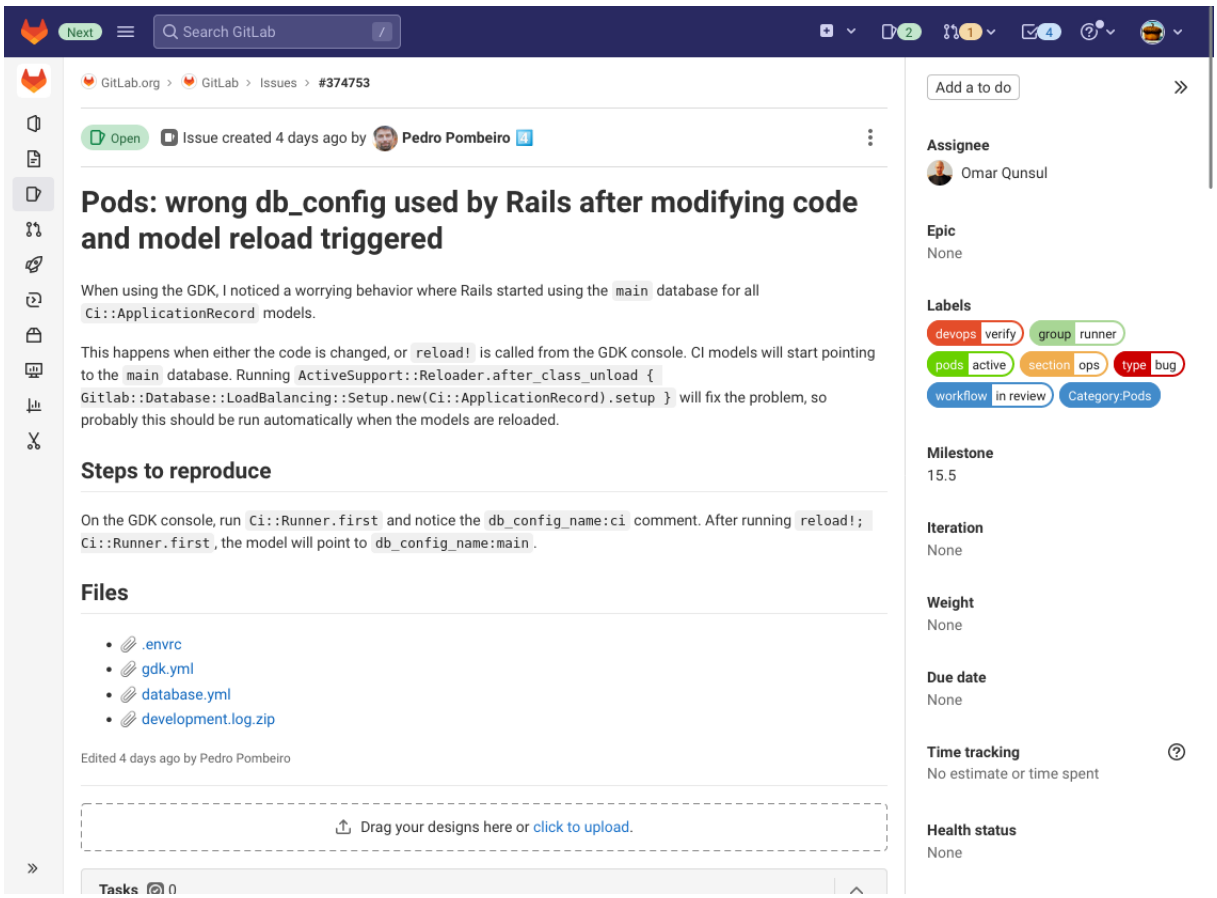
During project development, especially when collaborating with other people, you need to track and plan the work on the code. The "forges" can help you with a feature capable of tracking problems with your application (bugs, unexpected behaviors, etc.), feature requests from users or stakeholders, future enhancements for the application itself (such as improving the speed of some algorithm), or even periodic tasks to do on the code itself. All these things are named *issues*.

When you create an issue, the platform expects a summary/title of the issue: it should contain the very essence of the problem. It is strongly suggested to add a long description to provide more context, and, if it's a bug, possibly every piece of information on how to reproduce the problem. Usually, projects have rules on what is mandatory in these fields.

When the issue is created, it will assume the "open" state. When the issue is finished (meaning that the problem or feature has been resolved/implemented or that the problem or feature won't be resolved/implemented), it will assume the "closed" state. Some platforms may have more detailed statuses, like "completed", "wontfix", "invalid", etc.

An *issue* can be assigned to a developer to keep track of who is doing what. Also, usually, issues have *labels* to categorize them by importance or which part of the project they affect. Issues can have a deadline and a *milestone* (i.e., a future version of the app, even an internal one).

Example: this is an open issue in the GitLab project.



2.4.2 Forks

Sometimes you don't have the right to send commits or branches to a repository, but they will accept your code using pull or merge requests (we will see them soon). In some other situations, you want to create your own version of the code to apply some modifications that are not accepted by the original authors (effectively creating a new project based on the original one). In these cases, you can create a "fork".

> Note about software licenses: before creating a fork from another project, you must check their software license. **You may be restricted by their license or even forbidden to fork, change or redistribute the code** (yes, even if the code is already public!).
>
> Consult the license and ask the original authors to understand your rights.

A "fork" happens when a developer creates a copy of the source code from an existing project (usually by copying the entire Git repository) and starts independent development, creating distinct and separate software.

In Git, creating a fork is easy, as the working directory contains a copy of the complete repository and their commits can be *pushed* somewhere else. However, Git forges help you by providing an easier way to create a fork by using a button on the web page. When you click on that button on the page of an existing project, the Git repository will be copied into a new project under your forge account.

In terms of features, a forked project is identical to the existing one: you will be provided a URL where you can clone, push, and pull the repository.

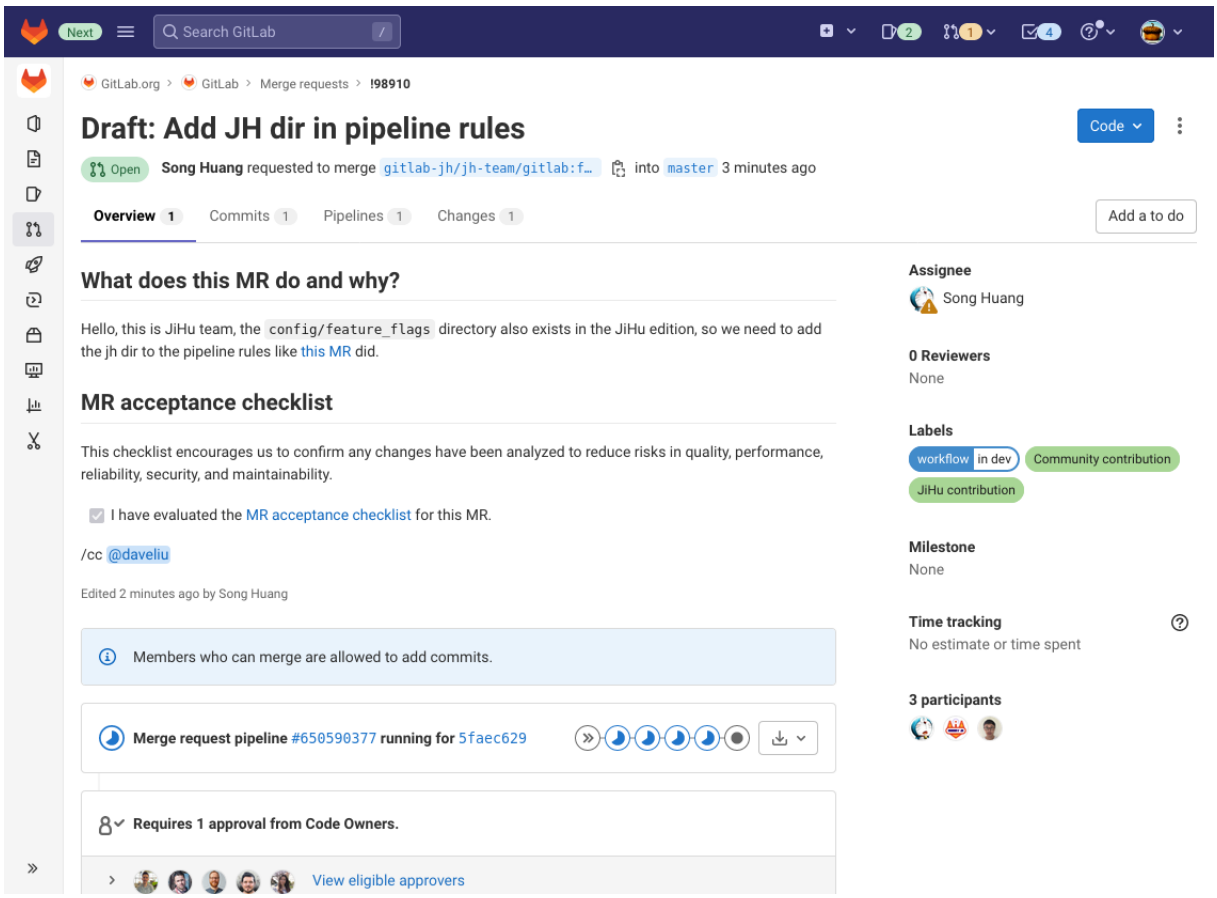
2.4.3 Pull/Merge requests

Whether you have a fork or a simple branch to merge into a main line of development, you may be in a situation where you have no right to perform the merge. In that case, forges allow you to create a "pull request" (or "merge request" in some platforms).

In essence, a "pull/merge request" is a request (from you) to merge your code in another branch, usually the principal one. Pull/merge requests have the same attributes as the issues: summary/title, description, labels, assignee, etc. They also show your changes using diffs, where you and other developers can "review" the code by adding comments, asking questions, or suggesting changes.

When a pull/merge request is complete, the web platform can automatically merge your branch into the target branch using a configured strategy. A pull/merge request may also be closed but not merged.

Example: this is an open merge request in the GitLab project.



2.5 Practical Git

2.5.1 Before starting: configure your local Git instance

In this book, the initial branch is named `master`. This name is still the default in Git, but it might change in the future. To avoid mistakes and problems, we explicitly set the initial branch name to `master`, so Git will always use this name as the initial branch name.

```
$ git config --global init.defaultBranch master
```

Git requires a name and an email for the committer. These can be anything - no need to be the same as your GitHub/GitLab account. Set these data using:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "some@email.address"
```

Also, in our examples we will use a compact version of the log command to avoid wasting space in this document. This command will create an alias `lg` for that log command:

```
$ git config --global alias.lg "log --graph --abbrev-commit --decorate
 ↪  --date=relative --format=format:'%C(bold blue)%h%C(reset) - %C(bold
 ↪  green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)-
 ↪  %an%C(reset)%C(bold yellow)%d%C(reset)' --all"
```

> If you copy and paste this command, make sure that:
>
> - the command is in one line
> - colors have space between "bold"/"dim" and the actual color
> - no space is between the color (or "reset") and any parenthesis

2.5.2 Before starting: remote Git repositories

In some examples we will use a remote Git repository. You need an internet connection and an account on GitLab.

After that, you need to fork the example repository at https://gitlab.com/sapienzaapps/greetings because you don't have permission to modify the original project. Use the "Fork" option (upper-right corner) and put the new repository in the namespace corresponding to your username. This action creates a copy that you own, where you can modify the repository as you wish.

2.5.3 Creating a (local) repository

To create commits, we must first create a new repository in an empty directory. Assuming that you're on your home directory, create a directory and switch to it:

```
$ mkdir -p wasa/test-init
$ cd wasa/test-init
$ git init
Initialized empty Git repository in /tmp/test/.git/
```

Git just created its directory for storing git-related files. There is no need for you to change any file in there. Leave the directory as it is.

The repository is now created, but it's empty. To see the current status, you can run:

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Even if Git is telling us that the `master` branch exists, the repository history is empty:

```
$ git log
fatal: your current branch 'master' does not have any commits yet
```

That is expected, as we don't have any commit yet.

2.5.4 Our first commit

To create a commit, we need *something* to commit. Let's create a file with "Hello world" as content:

```
$ echo "Hello world" > english.txt
```

Now you have the file in the working directory; however, it's not in the staging area (and Git is not tracking it). If you issue `git status`:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        english.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Git says that there is a file that is not tracked, and it's correctly suggesting we add the file to the staging area. Let's do it:

```
$ git add english.txt
```

Now, if you look at the status, Git says:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   english.txt
```

For now, we are OK with this change, so we commit:

```
$ git commit -m "Add a simple text file"
[master (root-commit) c82109a] Add a simple text file
 1 file changed, 1 insertion(+)
 create mode 100644 english.txt
```

Congratulations! You created your first commit in your repository. Its ID is `c82109a` (actually, this is the "short version"), and it's a "root-commit" or "orphan".

> What if we don't specify a commit message (or we omit `-m` flag)? As Git needs a commit message, it will open your default editor (specified in the `EDITOR` environment variable) and put some default text in it. You are required to edit the content and confirm (by saving the file) or to abort (by not saving the file).

If you look at the repository status now, you will see that status and log output have changed:

```
$ git status
On branch master
nothing to commit, working tree clean

$ git log
commit c82109a96bfe1e4bd8257f195a3200d173bffa5e (HEAD -> master)
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:07:42 2022 +0200

    Add a simple text file
```

From this output, you can see that:

- the commit ID is `c82109a96bfe1e4bd8257f195a3200d173bffa5e` (but you can address it using the shorter version `c82109a`);
- the author's name is `Enrico204`, and their email address is `enrico204@gmail.com`;
- the commit was created at `Mon Oct 3 12:07:42 2022 +0200`;
- the commit is in the branch `master`, the current HEAD.

2.5.5 Switch between commits

To switch between commits, we need multiple commits! Let's create two more in the `master` branch:

```
$ echo "Ciao mondo" > italian.txt
$ git add italian.txt
$ git commit -m "Add Italian translation"
[master 4c87b32] Add Italian translation
 1 file changed, 1 insertion(+)
 create mode 100644 italian.txt

$ echo "Hallo Welt" > german.txt
$ git add german.txt
$ git commit -m "Add German translation"
[master b4ea645] Add German translation
 1 file changed, 1 insertion(+)
 create mode 100644 german.txt
```

Now, let's check the status and the log:

```
$ git status
On branch master
nothing to commit, working tree clean

$ git log
commit b4ea645832930fcaaa0cdf8769d9762486060bfc (HEAD -> master)
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:18:24 2022 +0200

    Add German translation
```

```
commit 4c87b320853f0d697c57c29ddc39b1c277457f70
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:16:16 2022 +0200

    Add Italian translation

commit c82109a96bfe1e4bd8257f195a3200d173bffa5e
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:07:42 2022 +0200

    Add a simple text file
```

Now we have three commits. The latest one in the `master` branch is also the HEAD.

Let's suppose we want to go back to "*Add Italian translation*". We can use "checkout" command for switching between commits (replace `4c87b32` with your commit ID for that commit):

```
$ git checkout 4c87b32
Note: switching to '4c87b32'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 4c87b32 Add Italian translation
```

As we switched to a specific commit, we are now in a "detached HEAD" state. The message suggests creating a branch if we want to retain commits (`git switch` is a new version of `git checkout` - for now, we will stick to `git checkout`).

Also, Git `status` command is again telling us that we are in "detached HEAD" state:

```
$ git status
HEAD detached at 4c87b32
nothing to commit, working tree clean
```

Now, if you look at the log, you can see that it finishes in the current HEAD, which is not a branch:

```
$ git log
commit 4c87b320853f0d697c57c29ddc39b1c277457f70 (HEAD)
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:16:16 2022 +0200

    Add Italian translation

commit c82109a96bfe1e4bd8257f195a3200d173bffa5e
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:07:42 2022 +0200

    Add a simple text file
```

To see `master` again, we need to be explicit and run:

```
$ git log master
commit b4ea645832930fcaaa0cdf8769d9762486060bfc (master)
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:18:24 2022 +0200

    Add German translation

commit 4c87b320853f0d697c57c29ddc39b1c277457f70 (HEAD)
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:16:16 2022 +0200

    Add Italian translation

commit c82109a96bfe1e4bd8257f195a3200d173bffa5e
Author: Enrico204 <enrico204@gmail.com>
Date:   Mon Oct 3 12:07:42 2022 +0200

    Add a simple text file
```

As you can see, the HEAD is not in a branch - `master` is at `b4ea645`, while HEAD is at `4c87b32`. If we look at the directory, we will see that the `german.txt` created in `b4ea645` is missing:

```
$ ls -1
italian.txt
english.txt
```

Or, in other words, it doesn't exist *yet*.

Note that you can look at the difference between commits using Diff. For example, if we want to diff the first (`c82109a`) and the second (`4c87b32`) commit in the repo:

```
$ git diff c82109a 4c87b32
diff --git a/italian.txt b/italian.txt
new file mode 100644
index 0000000..abb27dc
--- /dev/null
+++ b/italian.txt
@@ -0,0 +1 @@
+Ciao mondo
```

This output means that the file `italian.txt` has been added. How to read this output:

- the commit ID order is important: first `c82109a` then `4c87b32` means that our previous situation is `c82109a` and our future is `4c87b32`. In this context, an "added" line means that the line was not present in `c82109a`, but it's present in `4c87b32`;
- `diff --git a/italian.txt b/italian.txt` is the corresponding UNIX `diff` utility command line for the output - it's saying that the output is relative to the `italian.txt` file;
- `new file mode 100644` indicates that the file has a new set of permissions;
- `index 0000000..abb27dc` indicates the two indexes for the file: the previous one (`0000000`, which is a special case for not existing file) and the future one (`abb27dc`);
- `--- /dev/null` indicates that lines removed (those starting with a dash) are for the `/dev/null` file (again, special case for not existing file);
- `+++ b/italian.txt` indicates that lines starting with a plus sign are inside the file `italian.txt` (usually, this name is the same as above unless the file is renamed in the same commit);
- `@@ -0,0 +1 @@` is the "context" for the diff (line 0 in the previous and future file, one line was added);
- and then the file: lines starting with a dash are those deleted by the future commit, while those with plus are added.

2.5.6 Switch between branches

Now that we are in a "detached HEAD", let's switch back to `master`. We can use the same Git checkout command for this:

```
$ git checkout master
Previous HEAD position was 4c87b32 Add Italian translation
Switched to branch 'master'
```

Git tells us that we switched from `4c87b32` (commit) to `master`. Let's see the log to confirm that (we will use the alias `lg`):

```
$ git lg
* b4ea645 - (4 hours ago) Add German translation - Enrico204 (HEAD -> master)
* 4c87b32 - (4 hours ago) Add Italian translation - Enrico204
* c82109a - (4 hours ago) Add a simple text file - Enrico204
```

What if we want to switch to another branch? We can use the checkout action again - however, we need another branch! Let's create a branch from the current HEAD:

```
$ git branch experiment
```

See what happened in the log:

```
$ git lg
* b4ea645 - (4 hours ago) Add German translation - Enrico204 (HEAD ->
 ↪  master, experiment)
* 4c87b32 - (4 hours ago) Add Italian translation - Enrico204
* c82109a - (4 hours ago) Add a simple text file - Enrico204
```

We have two branches now: `master` and `experiment`. HEAD is currently pointing to `master`, meaning that every commit will be added to the `master` branch.

Let's switch to `experiment`:

```
$ git checkout experiment
Switched to branch 'experiment'
```

As we can see in the log, HEAD is pointing to the `experiment` branch now:

```
$ git lg
* b4ea645 - (4 hours ago) Add German translation - Enrico204 (HEAD ->
 ↪  experiment, master)
* 4c87b32 - (4 hours ago) Add Italian translation - Enrico204
* c82109a - (4 hours ago) Add a simple text file - Enrico204
```

However, these two branches are on the same commit, meaning they share all commits since the first one. Let's add a commit to the `experiment` branch:

```
$ echo "Hola Mundo" > spanish.txt
$ git add spanish.txt
$ git commit -m "Add Spanish translation"
[experiment 4f8a791] Add Spanish translation
 1 file changed, 1 insertion(+)
 create mode 100644 spanish.txt
```

Now, if you look at the log:

```
$ git lg
* 4f36f44 - (5 seconds ago) Add Spanish translation - Enrico204 (HEAD ->
 ↪  experiment)
* b4ea645 - (4 hours ago) Add German translation - Enrico204 (master)
* 4c87b32 - (4 hours ago) Add Italian translation - Enrico204
* c82109a - (4 hours ago) Add a simple text file - Enrico204
```

As we expected, `experiment` is one commit "ahead". We can switch back to `master` again using checkout:

```
$ git checkout master
Switched to branch 'master'
```

Note that `git log` is showing you the history starting from HEAD by default, while our alias is showing all history - so if you run `git log` now you'll see that the Spanish translation commit is "missing"; it's not: you can see it if you specify the branch (`experiment`) or the `--all` flag.

2.5.7 Merge the experiment! (fast-forward)

Now we want to merge the `experiment` branch back to the `master`, using fast-forward strategy. To do that, be sure to have HEAD pointing to `master`:

```
$ git status
On branch master
nothing to commit, working tree clean
```

(if not, use `git checkout` to switch back to `master`)

Now, we can use `merge` action:

```
$ git merge --ff experiment
Updating b4ea645..4f36f44
Fast-forward
 spanish.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 spanish.txt
```

We explicitly used `--ff` flag to force Git to use fast-forward strategy (because it might not be the default one). What just happened? Let's see the log now:

```
$ git lg
* 4f36f44 - (13 minutes ago) Add Spanish translation - Enrico204 (HEAD ->
 ↪  master, experiment)
* b4ea645 - (4 hours ago) Add German translation - Enrico204
* 4c87b32 - (4 hours ago) Add Italian translation - Enrico204
* c82109a - (4 hours ago) Add a simple text file - Enrico204
```

The `master` branch was moved (along with HEAD) from `b4ea645` to `4f36f44`, and now both branches have the same set of commits (they're identical).

Now we can get rid of the `experiment` branch:

```
$ git branch -d experiment
Deleted branch experiment (was 4f36f44).
```

Note: if you try to delete a branch that is not merged with any other branch, Git will warn you that you'll lose those commits that are present only in that branch. Be aware that if you go ahead and delete that branch, commits that were only in that branch will be removed too.

2.5.8 Non fast-forward merge

We will create two parallel branches and try to merge one of them using non-fast-forward into the main branch. First, let's start with the first branch:

```
$ git branch dev1
$ git checkout dev1
Switched to branch 'dev1'
$ rm spanish.txt
$ git add spanish.txt
$ git commit -m "Removed spanish translation"
[dev1 317b9ea] Removed spanish translation
 1 file changed, 1 deletion(-)
 delete mode 100644 spanish.txt
```

Now we switch back to master, and we create another branch with one commit:

```
$ git checkout master
Switched to branch 'master'
$ git branch dev2
$ git checkout dev2
Switched to branch 'dev2'
$ rm german.txt
$ git add german.txt
$ git commit -m "Removed german translation"
[master c2ef725] Removed german translation
 1 file changed, 1 deletion(-)
 delete mode 100644 german.txt
```

If everything is right, you'll see something like that:

```
$ git lg
* c2ef725 - (3 hours ago) Removed german translation - Enrico204 (HEAD ->
 ↪  dev2)
| * 317b9ea - (3 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
* 4f36f44 - (3 hours ago) Add Spanish translation - Enrico204 (master)
* b4ea645 - (7 hours ago) Add German translation - Enrico204
* 4c87b32 - (7 hours ago) Add Italian translation - Enrico204
* c82109a - (7 hours ago) Add a simple text file - Enrico204
```

As you can see in the log, the two branches, `dev1` and `dev2`, have diverged. Let's suppose that `dev1` is the first to be merged into `master` - a fast-forward is sufficient (`master` and `dev1` have not diverged):

```
$ git checkout master
Switched to branch 'master'
$ git merge --ff dev1
Updating 4f36f44..317b9ea
Fast-forward
 spanish.txt | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 spanish.txt
```

Git is telling us that the merge resulted in the file `spanish.txt` deletion, as expected. Now, the situation is the following:

```
* c2ef725 - (3 hours ago) Removed german translation - Enrico204 (dev2)
| * 317b9ea - (3 hours ago) Removed spanish translation - Enrico204 (HEAD ->
 ↪  master, dev1)
|/
* 4f36f44 - (3 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (7 hours ago) Add German translation - Enrico204
* 4c87b32 - (7 hours ago) Add Italian translation - Enrico204
* c82109a - (7 hours ago) Add a simple text file - Enrico204
```

Let's proceed with the `dev2` merge into `master`, using non fast-forward merge:

```
$ git merge --no-ff dev2 -m "Merge branch 'dev2'"
Removing german.txt
Merge made by the 'recursive' strategy.
 german.txt | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 german.txt
```

As we need to create a new commit to merge the two branches, we need to specify a message with `-m` flag. Now the repository history is:

```
*   0e46de7 - (2 minutes ago) Merge branch 'dev2' - Enrico204 (HEAD ->
 ↪  master)
|\
| * c2ef725 - (3 hours ago) Removed german translation - Enrico204 (dev2)
* | 317b9ea - (3 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
```

```
* 4f36f44 - (3 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (7 hours ago) Add German translation - Enrico204
* 4c87b32 - (7 hours ago) Add Italian translation - Enrico204
* c82109a - (7 hours ago) Add a simple text file - Enrico204
```

As you can see, `0e46de7` is a new commit that is the result of merging `dev2` into `master`. Also, note that `dev1` and `dev2` branches are still there.

2.5.9 Conflicts

From time to time, code in different branches may be in conflict. Let's create two branches with a conflict:

```
$ git branch dev3
$ git checkout dev3
Switched to branch 'dev3'
$ echo "Hello" > english.txt
$ git add english.txt
$ git commit -m "Changed English translation"
[dev3 ca68455] Changed English translation
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master
Switched to branch 'master'
$ git branch dev4
$ git checkout dev4
Switched to branch 'dev4'
$ echo "world" > english.txt
$ git add english.txt
$ git commit -m "Updated English translation"
[dev4 74a6e77] Updated English translation
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now the situation should be like this:

```
$ git lg
* 74a6e77 - (81 seconds ago) Updated English translation - Enrico204 (HEAD
 ↪  -> dev4)
| * ca68455 - (2 minutes ago) Changed English translation - Enrico204 (dev3)
```

```
|/
*   0e46de7 - (48 minutes ago) Merge branch 'dev2' - Enrico204 (master)
|\
| * c2ef725 - (4 hours ago) Removed german translation - Enrico204 (dev2)
* | 317b9ea - (4 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
* 4f36f44 - (4 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (8 hours ago) Add German translation - Enrico204
* 4c87b32 - (8 hours ago) Add Italian translation - Enrico204
* c82109a - (8 hours ago) Add a simple text file - Enrico204
```

Let's fast-forward merge `dev3` into `master`, and then a non fast-forward merge for `dev4` into `master`:

```
$ git checkout master
Switched to branch 'master'

$ git merge --ff dev3
Updating 0e46de7..ca68455
Fast-forward
 english.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git merge --no-ff dev4 -m "Merge branch 'dev4'"
Auto-merging english.txt
CONFLICT (content): Merge conflict in english.txt
Automatic merge failed; fix conflicts and then commit the result.
```

OOPS! Something is wrong with our merge. Git detected something that it couldn't solve by itself, so it's asking us to do the proper modifications and continue. The status is something in between:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   english.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Let's see what the file contains:

```
$ cat english.txt
<<<<<< HEAD
Hello
=======
world
>>>>>> dev4
```

As you can see here, the file contains "Hello" in HEAD (which comes from `dev3` fast-forward merge), while we are trying to merge "world" from `dev4` branch. However, when the `dev4` branch started, the content was different, so Git can't just overwrite "Hello" with "world", otherwise it will destroy the work done in `dev3`.

Open the file with the command `nano english.txt`, remove `<<<<<< HEAD`, `=======` and `>>>>>> dev4`, and be sure that contains this:

```
Hello world
```

Save with `ctrl-o` (nano will ask for the file name, just confirm pressing `Enter` key) and return to the terminal with `ctrl-x`. Now add the file to the staging area (so Git knows that the file has been fixed) and continue the merge:

```
$ git add my_first_file.txt
$ git merge --continue -m "Merge branch 'dev4'"
[master 1b1fe98] Merge branch 'dev4'
```

Done! We merged `dev4` into `master`, resolving the conflict by hand.

2.5.10 Tagging

To tag the current commit, just use the tag action:

```
$ git tag v1
```

Now let's look at the log:

```
$ git lg
*   1b1fe98 - (46 minutes ago) Merge branch 'dev4' - Enrico204 (HEAD ->
 ↪  master, tag: v1)
|\
| * 74a6e77 - (58 minutes ago) Updated English translation - Enrico204 (dev4)
* | ca68455 - (58 minutes ago) Changed English translation - Enrico204 (dev3)
|/
*   0e46de7 - (2 hours ago) Merge branch 'dev2' - Enrico204
|\
| * c2ef725 - (5 hours ago) Removed german translation - Enrico204 (dev2)
* | 317b9ea - (5 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
* 4f36f44 - (5 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (9 hours ago) Add German translation - Enrico204
* 4c87b32 - (9 hours ago) Add Italian translation - Enrico204
* c82109a - (9 hours ago) Add a simple text file - Enrico204
```

As you can see, the `1b1fe98` commit is tagged `v1`. If we go ahead with `master`, the `v1` tag will remain there.

Tags are not supposed to be removed or changed. However, if you made a mistake, you can use the `-d` flag to remove a tag.

Let's try to go ahead with a commit on `master`, then go back to the tag. To create some changes, we will overwrite the `italian.txt` file with new content:

```
$ echo "Salve mondo" > italian.txt
$ git add italian.txt
$ git commit -m "Changed italian text"
[master 0dea299] Changed italian text
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout v1
Note: switching to 'v1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
```

```
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 1b1fe98 Merge branch 'dev4'
```

Git is (again) in `detached` HEAD state. Why? Because we changed HEAD to point to a tag, not a branch (see in "Switch between commits" what this means).

Let's look at the content of `italian.txt` file:

```
$ cat italian.txt
Ciao mondo
```

As we expected, the file contains "Ciao mondo", and not "Salve mondo" (which is the new text) because `v1` commit is on the previous commit:

```
$ git lg
* 0dea299 - (45 minutes ago) Changed italian text - Enrico204 (master)
*   1b1fe98 - (2 hours ago) Merge branch 'dev4' - Enrico204 (HEAD, tag: v1)
|\
| * 74a6e77 - (2 hours ago) Updated English translation - Enrico204 (dev4)
* | ca68455 - (2 hours ago) Changed English translation - Enrico204 (dev3)
|/
*   0e46de7 - (3 hours ago) Merge branch 'dev2' - Enrico204
|\
| * c2ef725 - (5 hours ago) Removed german translation - Enrico204 (dev2)
* | 317b9ea - (5 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
* 4f36f44 - (6 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (10 hours ago) Add German translation - Enrico204
```

```
* 4c87b32 - (10 hours ago) Add Italian translation - Enrico204
* c82109a - (10 hours ago) Add a simple text file - Enrico204
```

If you forgot to put a tag, you can always go back in time and put a tag:

```
$ git checkout c82109a
Previous HEAD position was 1b1fe98 Merge branch 'dev4'
HEAD is now at c82109a Add a simple text file
$ git tag v0
$ git lg
* 0dea299 - (46 minutes ago) Changed italian text - Enrico204 (master)
*   1b1fe98 - (2 hours ago) Merge branch 'dev4' - Enrico204 (tag: v1)
|\
| * 74a6e77 - (2 hours ago) Updated English translation - Enrico204 (dev4)
* | ca68455 - (2 hours ago) Changed English translation - Enrico204 (dev3)
|/
*   0e46de7 - (3 hours ago) Merge branch 'dev2' - Enrico204
|\
| * c2ef725 - (5 hours ago) Removed german translation - Enrico204 (dev2)
* | 317b9ea - (5 hours ago) Removed spanish translation - Enrico204 (dev1)
|/
* 4f36f44 - (6 hours ago) Add Spanish translation - Enrico204
* b4ea645 - (10 hours ago) Add German translation - Enrico204
* 4c87b32 - (10 hours ago) Add Italian translation - Enrico204
* c82109a - (10 hours ago) Add a simple text file - Enrico204 (HEAD, tag: v0)
```

2.5.11 Clone a remote repository

The first step to working on the remote repository is creating your local working copy. To do that, you can use `git clone` command (replace `Enrico204` with your GitLab username):

```
$ git clone https://gitlab.com/Enrico204/greetings.git
Cloning into 'greetings'...
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 14 (delta 4), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (14/14), done.
Resolving deltas: 100% (4/4), done.
```

Now Git created a directory (named after the repo name) with the working copy. Enter it and look for files:

```
$ ls -1
english.txt
german.txt
italian.txt
spanish.txt
```

What is the log showing?

```
$ git lg
*   f208b06 - (10 minutes ago) Merge branch 'dev2' - Enrico204 (HEAD ->
 ↪  master, tag: v1, origin/master, origin/HEAD)
|\
| * c68746d - (10 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (11 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
|/
* c6884ff - (12 minutes ago) Add Italian translation - Enrico204
* 055d743 - (13 minutes ago) Add English translation - Enrico204 (tag: v0)
```

Now you can manage your repository as shown in previous sections. However, those commands will work locally: to synchronize your working copy with the remote part, you need to learn how to push and pull branches and tags.

2.5.12 Send commits/branches/tags to the remote

Try to create a new commit in `master` branch:

```
$ echo "Salve mundi" > latin.txt
$ git add latin.txt
$ git commit -m "Add Latin translation"
[master 6bebfae] Add Latin translation
 1 file changed, 1 insertion(+)
 create mode 100644 latin.txt
```

Let's look at the log:

```
$ git lg
* 6bebfae - (43 seconds ago) Add Latin translation - Enrico204 (HEAD ->
 ↪  master)
*   f208b06 - (16 minutes ago) Merge branch 'dev2' - Enrico204 (tag: v1,
 ↪  origin/master, origin/HEAD)
|\
| * c68746d - (17 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (17 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
|/
* c6884ff - (18 minutes ago) Add Italian translation - Enrico204
* 055d743 - (20 minutes ago) Add English translation - Enrico204 (tag: v0)
```

As you can see in the log, your local `master` has a new commit, `6bebfae`, which is not present in the remote repository (`origin/master`). To send the commit, use the push action:

```
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 6 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 281 bytes | 281.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To gitlab.com:Enrico204/greetings.git
   f208b06..6bebfae  master -> master
```

Git is saying that the remote repository just updated its `master` reference from `f208b06` to `6bebfae`, meaning that the `origin/master` is now aligned with our local `master`:

```
$ git lg
* 6bebfae - (3 minutes ago) Add Latin translation - Enrico204 (HEAD ->
 ↪  master, origin/master, origin/HEAD)
*   f208b06 - (19 minutes ago) Merge branch 'dev2' - Enrico204 (tag: v1)
|\
| * c68746d - (19 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (20 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
|/
```

```
* c6884ff - (21 minutes ago) Add Italian translation - Enrico204
* 055d743 - (22 minutes ago) Add English translation - Enrico204 (tag: v0)
```

Push action also works for tags:

```
$ git tag v2
$ git push origin v2
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To gitlab.com:Enrico204/greetings.git
 * [new tag]         v2 -> v2
```

2.5.13 Fetch commits/branches/tags from remote

Fetch action works by getting info about the status of the remote repository. However, if you try to fetch now in your repo, you'll notice that nothing happened as no one (not even you) modified the remote repository in the meantime:

```
$ git fetch
```

To create a commit in the remote repository (but not in the local repository), you have different ways:

- Find a partner/colleague and make they push a new commit in your repo; or
- Use another PC to create the commit; or
- Clone the repository in another directory and push from there; or
- Use the web editor (to simulate another PC)

For simplicity, use the web editor on the project page (on GitLab) and edit a file. Add/remove/change the content and commit directly to `master` on the web editor (if asked).

Now, if you run fetch command:

```
$ git fetch
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), 200 bytes | 100.00 KiB/s, done.
From gitlab.com:Enrico204/greetings
   6bebfae..cc45437  master     -> origin/master
```

Git informs us that the `origin/master` has been updated with a new commit. Let's see the log:

```
$ git lg
* cc45437 - (45 seconds ago) Delete spanish.txt - Enrico (origin/master,
 ↪  origin/HEAD)
* 6bebfae - (37 minutes ago) Add Latin translation - Enrico204 (HEAD ->
 ↪  master, tag: v2)
*   f208b06 - (53 minutes ago) Merge branch 'dev2' - Enrico204 (tag: v1)
|\
| * c68746d - (53 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (54 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
|/
* c6884ff - (55 minutes ago) Add Italian translation - Enrico204
* 055d743 - (56 minutes ago) Add English translation - Enrico204 (tag: v0)
```

As you can see, the reference `origin/master` has an additional commit. Now, if we want to update our local `master` to match the remote one, we can use any merge strategy (in this case, a fast-forward is advisable):

```
$ git merge --ff origin/master
Updating 6bebfae..cc45437
Fast-forward
 spanish.txt | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 spanish.txt
```

Just like any other merge, Git is telling us that it has updated the working copy to match what we requested. The log should be something like:

```
$ git lg
* cc45437 - (5 minutes ago) Delete spanish.txt - Enrico (HEAD -> master,
 ↪  origin/master, origin/HEAD)
* 6bebfae - (42 minutes ago) Add Latin translation - Enrico204 (tag: v2)
*   f208b06 - (57 minutes ago) Merge branch 'dev2' - Enrico204 (tag: v1)
|\
| * c68746d - (58 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (58 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
```

```
|/
* c6884ff - (59 minutes ago) Add Italian translation - Enrico204
* 055d743 - (61 minutes ago) Add English translation - Enrico204 (tag: v0)
```

You can use the pull action to simplify this operation (fetch + merge). For example, if we create a new commit again remotely, we can run pull to synchronize:

```
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), 198 bytes | 198.00 KiB/s, done.
From gitlab.com:Enrico204/greetings
   cc45437..29fac79  master     -> origin/master
Updating cc45437..29fac79
Fast-forward
 latin.txt | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 latin.txt
```

As you can see, the output is the sum of the fetch and fast-forward merge action. The log will show that we are aligned to the remote:

```
$ git lg
* 29fac79 - (62 seconds ago) Delete latin.txt - Enrico (HEAD -> master,
 ↪  origin/master, origin/HEAD)
* cc45437 - (6 minutes ago) Delete spanish.txt - Enrico
* 6bebfae - (42 minutes ago) Add Latin translation - Enrico204 (tag: v2)
*   f208b06 - (58 minutes ago) Merge branch 'dev2' - Enrico204 (tag: v1)
|\
| * c68746d - (58 minutes ago) Add Spanish translation - Enrico204
 ↪  (origin/dev2)
* | 4968d26 - (59 minutes ago) Add German translation - Enrico204
 ↪  (origin/dev1)
|/
* c6884ff - (60 minutes ago) Add Italian translation - Enrico204
* 055d743 - (61 minutes ago) Add English translation - Enrico204 (tag: v0)
```

2.5.14 Forges and projects

Before switching to see how issues and pull/merge requests works in practice, we need to remove the fork relationship in GitLab. To remove, open your project, go to "Settings" in the lateral menu, then General (just under "Settings" in the menu). In the main part of the page, expand "Advanced" (the last section) and go to the "Remove fork relationship" section. Click on the red button to remove the fork relationship.

2.5.15 Create an issue

You can use issues to track bugs or tasks for your repository. For example, let's say we want to follow the addition of a French translation.

First, we create an issue in the project: open the project page on GitLab, click on "Plan" on the lateral menu, and then "Issues". Now click on "New Issue" (center of the page, or right-upper corner). Fill the issue fields:

- Title: *French translation missing*
- Type: *Issue*
- Description: *The French translation is missing. Add a French translation for the "Hello world" greeting in a dedicated file*

Click on "Create issue". Now the browser is redirected to the (newly created) issue page. Using the lateral menu, you can assign the issue to other project members (even yourself), put a label on it, and set a milestone or a due date. At the center of the page, you can add a comment: usually, you want to discuss with others a possible solution and keep track of what is required/done (for example, if you are working on it, and you discover something that may complicate the matter, it's better to leave a note for future reference).

For now, assign the issue to yourself. We'll work on it in the next paragraph.

2.5.16 Pull/Merge request

Now you want to contribute to the project by resolving an issue. To do that, you can create a pull/merge request.

In GitLab, you can create a merge request directly from an issue: open the "French translation missing" issue you just made, click on the "Create merge request" button and fill all fields you think are appropriate. You may want to assign the merge request to yourself. Then click on "Create merge request".

Your browser will show the merge request you just created. At the same time, GitLab automatically created a new branch named `1-french-translation-missing`, and it's prepared to merge that branch on `master` branch.

Let's create a file: switch to your shell and run a fetch to download the new branch GitLab just made:

```
$ git fetch
From gitlab.com:Enrico204/greetings
 * [new branch]      1-french-translation-missing ->
  ↪  origin/1-french-translation-missing
```

Now, switch to the new branch:

```
$ git checkout 1-french-translation-missing
Branch '1-french-translation-missing' set up to track remote branch
 ↪  '1-french-translation-missing' from 'origin'.
Switched to a new branch '1-french-translation-missing'
```

Git tells us that we switched to the new branch and that it is coming from the remote `origin`.

Create a new file, and then commit:

```
$ git add french.txt
$ git commit -m "Add French translation"
[1-french-translation-missing f537070] Add French translation
 1 file changed, 1 insertion(+)
 create mode 100644 french.txt
```

And then push the commit to the remote repository:

```
$ git push origin 1-french-translation-missing
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 6 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 357 bytes | 357.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: View merge request for 1-french-translation-missing:
remote:   https://gitlab.com/Enrico204/greetings/-/merge_requests/1
remote:
```

```
To gitlab.com:Enrico204/greetings.git
   29fac79..f537070  1-french-translation-missing ->
    ↪  1-french-translation-missing
```

Now, if you open the merge request on your browser (or refresh the page if it was already open), you will see that the view has changed:

- "Commits" tab contains the commit you just did
- "Changes" tab contains a summary of all modifications that all commits contain
- A prominent message appeared regarding the fact that the merge is blocked as it is still a draft

You can send as many commits as you like to the merge request: the "Commits" view will summarize them, and the "Changes" view will show all changes together.

When the merge request is ready, click on "Mark as ready" in the "Overview" tab. The view will change again, and now a "Merge" button has appeared. Click on it to merge your branch.

If everything goes right, you can see the effect of the merge by fetching the new status and looking at the log:

```
$ git fetch
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 277 bytes | 277.00 KiB/s, done.
From gitlab.com:Enrico204/greetings
   29fac79..2665e6c  master     -> origin/master

$ git lg
*   2665e6c - (59 seconds ago) Merge branch '1-french-translation-missing'
 ↪  into 'master' - Enrico (origin/master, origin/HEAD)
|\
| * f537070 - (7 minutes ago) Add French translation - Enrico204 (HEAD ->
 ↪  1-french-translation-missing, origin/1-french-translation-missing)
|/
* 29fac79 - (3 hours ago) Delete latin.txt - Enrico (master)
* cc45437 - (3 hours ago) Delete spanish.txt - Enrico
* 6bebfae - (3 hours ago) Add Latin translation - Enrico204 (tag: v2)
*   f208b06 - (4 hours ago) Merge branch 'dev2' - Enrico204 (tag: v1)
|\
| * c68746d - (4 hours ago) Add Spanish translation - Enrico204 (origin/dev2)
* | 4968d26 - (4 hours ago) Add German translation - Enrico204 (origin/dev1)
```

```
|/
* c6884ff - (4 hours ago) Add Italian translation - Enrico204
* 055d743 - (4 hours ago) Add English translation - Enrico204 (tag: v0)
```

GitLab, by default, uses non fast-forward strategy even when a fast-forward is possible. You can change this setting if you like.

2.6 Further readings

Git theory is widely explained on the web. The Git Book (from the Git authors) is a complete description of Git. Other resources are:

- https://www.atlassian.com/git/tutorials
- https://threesaplings.co/articles/explaining-basic-concepts-git-and-github/
- https://github.com/RomuloOliveira/commit-messages-guide
- https://nitaym.github.io/ourstheirs/
- https://en.wikipedia.org/wiki/Version_control
- https://www.atlassian.com/git/tutorials/what-is-version-control
- https://about.gitlab.com/topics/version-control/

3 Web protocols

3.1 Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is a string of characters identifying a resource on the internet. URIs are *uniform*, in the sense that they are defined in the same way even if resource types are different or the semantics to access those resources is different.

The RFC 3986[1] describes a "resource" as anything that can be identified by a URI, such as electronic documents, images, and web services, but also non-electronic items like books, humans, and abstract concepts like mathematical equations.

URIs have two specializations known as *Uniform Resource Locator* (URL) and *Uniform Resource Name* (URN).

A *Uniform Resource Locator* (URL) is a specific type of URI that additionally provides the means to locate a resource on the internet and specify how to access it.

A *Uniform Resource Name* (URN), instead, is a type of URI that provides a globally unique name to a resource. A URN name is expected to remain globally unique and persistent even when the resource is no longer available. By convention, URNs usually start with `urn:`.

The formal definition of a URI from RFC 3986 is:

```
URI = scheme ":" ["//" authority] path ["?" query] ["#" fragment]
authority = [userinfo "@"] host [":" port]
```

The `scheme` and the `path` are always required for a URI (but they can be empty). Other components are optional.

Example: the following strings are all URIs (but only the first three are URLs, and only the last one is a URN):

- `tel:+39-06-49911`
- `mailto:bassetti@di.uniroma1.it`

[1] RFC 3986 "Uniform Resource Identifier (URI): Generic Syntax" - https://datatracker.ietf.org/doc/html/rfc3986

- `https://datatracker.ietf.org/doc/html/rfc3986`
- `urn:oasis:names:specification:docbook:dtd:xml:4.1.2`

3.2 Hyper-text Transfer Protocol (HTTP)

The Hypertext Transfer Protocol, commonly known as HTTP, is a fundamental protocol that describes the communication between web browsers and servers on the Internet. It serves as the backbone of the World Wide Web, facilitating the transfer of data, such as web pages, images, videos, and other resources, across the vast digital network.

HTTP was first introduced in 1991 by Tim Berners-Lee, the inventor of the World Wide Web, and has undergone various iterations and enhancements over the years. Almost all browsers, servers, and applications support two active standards: HTTP/1.1 and HTTP/2; a newer standard, HTTP/3, was released in 2022 and is gaining traction.

HTTP communication occurs over the Transmission Control Protocol (TCP) until version 2 and over User Datagram Protocol (UDP) in version 3. The standard port for HTTP (1.1 and 2) is 80, while the secure version named HTTPS (HTTP over TLS, Transport-Layer Security) uses port 443.

> UDP has no concept of connections. However, HTTP/3 integrates a basic connection control mechanism over UDP. For the sake of simplicity, we will use the term "connection" to indicate both TCP and HTTP/3 connections.

HTTP operates on a client-server model, where a client ("**User Agent**"), typically a web browser or application, sends requests to a server ("**Origin Server**"), and the server responds with the requested data or performs the requested action. It primarily relies on **stateless interactions**, meaning each request from a client to a server is independent and lacks any knowledge of previous communications. As a result, every request must contain all the necessary information required for the server to process it effectively.

> A User Agent is any software that is capable of issuing HTTP requests: not only a web browser but also a mobile app, web spider (indexing services for search engines), household appliances (like a fridge asking for the current weather), etc.
>
> An Origin Server is any software that can originate authoritative responses for a given resource: web server, web cameras, Wi-Fi router web console, video-on-demand platforms, etc.
>
> An application may assume both roles: it may need to ask different Origin servers to respond to a given resource asked by its clients. For example, an Origin server that provides the weather to a mobile app ("User Agent") may need to ask another Origin server for the exact coordinates of

> a given city.

The HTTP protocol defines several request methods, including GET, POST, PUT, DELETE, HEAD, and more, each serving a specific purpose in web interactions. Additionally, it specifies a range of response status codes that inform clients about the outcome of their requests, such as 200 OK, 404 Not Found, and 500 Internal Server Error. We will see them later in detail.

3.2.1 Request/response

In HTTP, data is exchanged through a sequence of request-response messages sent in the connection's session layer. A connection may host multiple (different) exchanges (even concurrently in newer protocol versions). Within a connection, a typical HTTP exchange is:

1. The *User Agent* sends its request to the *Origin server*.
2. The *Origin server* elaborates the request and sends back an HTTP response message;

These messages may contain an attached payload named "body" or "content".

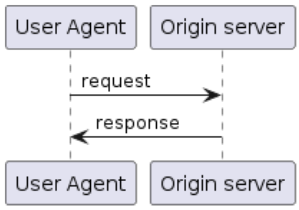


Figure 3.1: A simple HTTP request and response between a User Agent and the Origin server.

In HTTP/1.1, messages are encoded in ASCII and are human-readable. Starting from HTTP/2, these messages are encoded in binary frames for performance reasons. Nevertheless, the human-readable representation is still used everywhere regardless of the protocol version, from debug tools to software configuration to development libraries.

3.2.2 Anatomy of an HTTP request

An HTTP request contains three parts:

- The request line, which contains the HTTP method, the resource path, and the protocol version;
- the request headers;
- an (optional) message body after a blank line.

The request line contains the URI for the resource. It can be an absolute path or a URL. Absolute paths are URI, so they may have a *query string* after a `?`, which is additional information for the server. The query string is a series of key-value pairs divided by `=`, and joined together by `&`. For example, the path `/hello.txt?key1=value&key2=value2` contains a query string.

Request headers are key-value pairs. They are expressed as `key: value`. Although no headers is mandatory by the standard, in practice some of them are required, and some are useful to exploit HTTP completely. Some examples:

- `host: www.example.com` indicates that the resource path is in the context of `www.example.com` website (helpful if the Origin server handles more than one website);
- `accept-language` indicates preferred languages for the body;
- `authorization` provides information for authentication and authorization of the *User Agent*;
- `content-type` indicates the type of the request body;
- `content-length` indicates the length of the request body (not specified if there is no payload/body).

Case sensitivity is different in a request:

- The method name and the protocol version in the request must be *uppercase*;
- The resource path and its query string are case-sensitive (`/hello.txt` is different from `/HELLO.TXT`);
- header names are case-insensitive (`user-agent` is the same as `User-Agent`). From HTTP/2, they must be lowercase during the transmission.

> In some examples, you may see Camel Case HTTP headers, like `Content-Type`. It's not an error: the HTTP/2 standard mandates that headers are lowercase *during the transmission* only. They can be configured or shown otherwise by tools and libraries.

Example: this is a `GET` request for the resource `/hello.txt` in `www.example.com` using the `HTTP/1.1` protocol. The software acting as *User Agent* is `curl` version `7.64.1`, and it accepts a resource in English or Italian. There is no payload/body attached.

```
GET /hello.txt HTTP/1.1
user-agent: curl/7.64.1
host: www.example.com
accept-language: en, it
```

Example: this is a `PUT` request for the resource `/hello.txt` in `www.example.com` using the `HTTP/2.0` protocol. The request contains a body of length 5 bytes. The MIME type for the content is `text/plain`.

```
PUT /hello.txt HTTP/2.0
host: www.example.com
content-length: 5
content-type: text/plain

world
```

3.2.3 Anatomy of an HTTP response

An HTTP response contains three parts:

- The status line, which contains the protocol version, the status code, and the status message;
- the response headers;
- an (optional) message body after a blank line.

The status code indicates the status of the request issued by the *User Agent*, and they are represented by a number from 100 to 599 (although some status codes are not defined). Precisely, status codes are divided into five classes:

- `1xx` informational responses: the request has been received, but no status is available yet;
- `2xx` successful responses: the request has been received and successfully accepted or processed;
- `3xx` redirection responses: to complete the request, the *User Agent* must take further actions;
- `4xx` client error responses: the request was rejected/aborted because of an error by the client;
- `5xx` server error responses: the request was rejected/aborted because of an error by the server;

The `1xx` status codes are the only ones that are not "final": a new response with a different status code is always expected after `1xx` responses.

The status message is usually the human-readable description of the status code, and it has been removed in HTTP/2 and HTTP/3.

Response headers, as request headers, are key-value pairs. They are expressed as `key: value`. Depending on the response, some headers may be required. For example, `date` is almost always required. Some examples of response headers are:

- `date` indicates the date and time at which the message originated;
- `server` contains the name and version of the Origin server;
- `content-type` indicates the type of the response body;
- `content-length` indicates the length of the response body.

Case sensitivity is different in a response:

- The protocol version in the response must be *uppercase*;
- header names are case-insensitive (`server` is the same as `Server`), and from HTTP/2, they must be lowercase.

Example: this is a successful response using the `HTTP/1.1` protocol. The software that is acting as *Origin* is `Apache`, and the message originated `Mon, 27 Jul 2009 12:28:53 GMT`, although the body was last modified a few days earlier (`Wed, 22 Jul 2009 19:15:56 GMT`). The length of the `text/plain` content is `51` bytes. Other useful headers are present.

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain

Hello World! My content includes a trailing CRLF.
```

3.2.4 Proxy Servers

The HTTP protocol supports intermediate hosts that forward messages. These hosts can scan, alter or reject requests and responses. They are called "proxies".

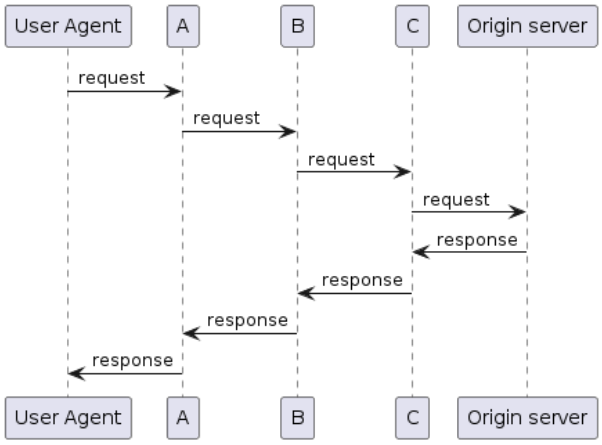


Figure 3.2: HTTP requests may go through by intermediate devices.

There are two different kinds of proxy servers:

- Forward proxy;
- Reverse proxy.

A *forward proxy* receive all requests from a network (company, university campus, etc.), and forward them to each different *Origin server*. Typically, they are used for local caching, policy enforcement (i.e., denying/allowing access to some websites), and auditing (keeping track of who visited what).

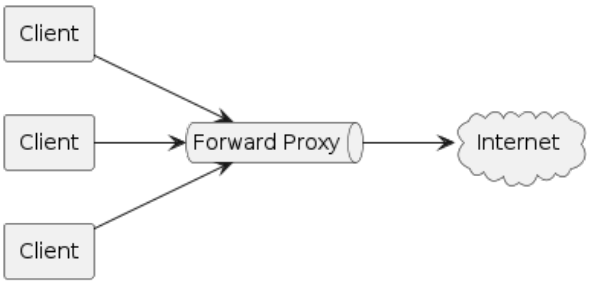


Figure 3.3: The architecture of a network with a forward proxy.

A *reverse proxy* receives all requests from the Internet (thus acting as an *Origin server*), and then forward all requests to the actual *Origin server*. The goal of this proxy is to take away the load from the *Origin server*. For example, a *reverse proxy* may be in charge of blocking malicious requests, which is computationally intensive, or it may be used to cache some responses.

Figure 3.4: The architecture of a network employing a forward proxy.

> The term "cache" or "caching" refers to the ability of applications to store the response for an HTTP request and provide that response without involving the *Origin server*.
>
> Caching significantly speeds up web browsing; however, only some HTTP responses can be cached. Proxies and browsers decide whether they should cache the response by looking at both the request and the response. The criteria include headers, response status codes, the request path, and payloads.

Example: an intermediate proxy server replies with a cached version of the response:

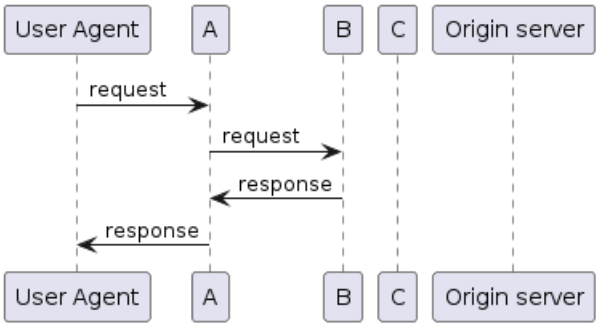


Figure 3.5: HTTP responses may be cached by intermediaries.

As you can see, the *Origin server* doesn't know about this message exchange: its resource usage for this request is zero.

3.2.5 Request methods

Each request has a method. The method indicates which action the *User Agent* wants to perform on the server and mandates how to build requests and responses. Methods are defined in RFC 9110[2].

They have these properties:

- **Safe**: no side effect on the resource, i.e., it is "read-only". `GET`, `HEAD`, `OPTIONS`, and `TRACE` are safe;
- **Idempotent**: multiple identical requests have the same effect as one of such requests. All safe methods are also idempotent. `PUT` and `DELETE` are also idempotent;
- **Cacheable**: allow a cache to store and use a response. `GET`, `HEAD` (and `POST` under some conditions) are cacheable.

Some widely used methods for web applications are:

- `GET`: request a representation of a resource;
  - It is **safe**: no changes to the resource
  - It is **cacheable**: responses can be cached by an intermediary and reused; conditions for caching the responses (e.g., expiration date, etc.) can be specified.
  - It is **idempotent**: due to the safeness property.
- `POST`: create or modify a subordinate of the resource indicated in the URI. The URI identifies the resource that will handle the request.

[2]RFC 9110 "HTTP Semantics" - https://datatracker.ietf.org/doc/html/rfc9110

  - It is **cacheable**: response can be cacheable if specified.
  - It is **not safe**: it changes the server state;
  - It is **not idempotent**: `POST` is used when actions are not idempotent, like submitting a new order in e-commerce.

- `PUT`: create or replace a resource as specified in the request;
  - It is **idempotent**: any successive identical PUT request does not modify the resource;
  - **Neither safe nor cacheable**.

- `DELETE`: request that the origin server removes the target resource;
  - It is **idempotent**: deleting an already-deleted resource does not produce any new effect;
  - **Neither safe nor cacheable**.

Lesser used methods include `HEAD`, `CONNECT`, `OPTIONS`, `TRACE`, and `PATCH`.

Method	Request has body	Response has body	Safe	Idempotent	Cacheable
`GET`	Not really *	Yes	Yes	Yes	Yes
`HEAD`	Not really *	No	Yes	Yes	Yes
`POST`	Yes	Yes	No	No	Yes
`PUT`	Yes	Yes	No	Yes	No
`DELETE`	Not really *	Yes	No	Yes	No
`CONNECT`	No †	Yes	No	No	No
`OPTIONS`	Not really *	Yes	Yes	Yes	No
`TRACE`	No	Yes	Yes	Yes	No
`PATCH`	Yes	Yes	No	No	No

* The standard does not define semantics for the request body using this method. It means that, although it is not explicitly forbidden, the result of sending a request body may result in unpredictable responses, except for specific conditions. You should avoid sending a request body with this method.

† The `CONNECT` method does not support a request body due to the semantic difference of the method itself. Data in the request body are up to the HTTP protocol itself for the purpose of the tunnel.

3.2.6 Common status codes

- 2xx success
  - **200 Ok**: the request has been fulfilled, and no further details are available;
  - **201 Created**: the request has been fulfilled, resulting in the creation of a new resource;
  - **202 Accepted**: the request has been accepted, and it's being processed;
  - **204 No Content**: the server successfully processed the request, but no content is available;
- 3xx redirection
  - **301 Moved Permanently**: this and all future requests should be directed to the given URI;
  - **302 Found**: (previously "Moved Temporarily") indicates that the content is available in another URL;
- 4xx client errors
  - **400 Bad Request**: the client sent a request that the server cannot parse or understand;
  - **401 Unauthorized**: authentication is required to access the resource, and no authentication information has been provided;
  - **403 Forbidden**: the request is valid and was understood by the server, but the action is prohibited (note that, although this code is commonly used in a context where the authorization is not sufficient, it is not the only cause for this error);
  - **404 Not Found**: resource could not be found, but it may become available in the future;
  - **405 Method Not Allowed**: the request method is not supported; e.g., a PUT request on a read-only resource.
- 5xx server errors
  - **500 Internal Server Error**: an internal error in the server itself prevents a response;
  - **501 Not Implemented**: a valid request was submitted, but the server cannot respond because of an incomplete implementation;
  - **502 Bad Gateway**: the proxy server received an invalid response from the upstream server;
  - **503 Service Unavailable**: the server is overloaded, or some of its internal processes are temporarily not available;
  - **504 Gateway Timeout**: the proxy server did not receive a response from the upstream server within a specified timeout.

3.2.7 HTTP examples

We can use the cURL command line utility to see HTTP in action. cURL conveniently divides the output using three markers at the beginning of the line: an asterisk * denotes connection information (mainly TLS certificate information), the greater-than symbol > is used to indicate the request, and the less-than symbol < is used for the response.

In the examples below, **we removed the lines starting with** * (connection information) from the cURL output **to focus on HTTP messages**.

If your system does not have cURL, you can download the source code or a pre-compiled binary from the website https://curl.se/ or install it using the package manager of your operating system.

Example: we can use cURL to request the web page `https://httpbin.org/user-agent` (which is a public API that returns your user agent) using `HTTP/1.1`:

```
$ curl -v --http1.1 https://httpbin.org/user-agent
> GET /user-agent HTTP/1.1
> Host: httpbin.org
> User-Agent: curl/7.74.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Thu, 27 Jul 2023 20:48:41 GMT
< Content-Type: application/json
< Content-Length: 34
< Connection: keep-alive
< Server: gunicorn/19.9.0
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Credentials: true
<
{
  "user-agent": "curl/7.74.0"
}
```

A lot of information here! Let's analyze this output.

- the request contains the following:
  - method: `GET` (the cURL default if we don't specify otherwise)
  - path: `/user-agent`
  - protocol: `HTTP/1.1`
  - the host we are trying to contact: `httpbin.org`

  - our *User Agent*: `curl/7.74.0`
  - `Accept: */*` indicates that we (cURL) accept every type of content;

- the response is a `200` with OK as the description, and:
  - the date when this response was created: `Thu, 27 Jul 2023 20:48:41 GMT`;
  - the response type, `application/json` identifies a JSON and its length of 34 bytes;
  - `Connection: keep-alive` indicates that the server supports keeping this TCP connection opened for further HTTP requests;
  - the *Origin server* is using `gunicorn/19.9.0`
  - `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials` are headers for CORS (we will see CORS later in this book).

The body (content) of the response is a JSON document:

```
{
  "user-agent": "curl/7.74.0"
}
```

We will see JSON in the following chapter.

Example: let's see an HTTP/2 message exchange instead:

```
$ curl -v --http2 https://httpbin.org/user-agent
> GET /user-agent HTTP/2
> Host: httpbin.org
> user-agent: curl/7.74.0
> accept: */*
>
< HTTP/2 200
< date: Thu, 27 Jul 2023 21:04:26 GMT
< content-type: application/json
< content-length: 34
< server: gunicorn/19.9.0
< access-control-allow-origin: *
< access-control-allow-credentials: true
<
{
  "user-agent": "curl/7.74.0"
}
```

As you can see, request and response have mostly the same structure. As expected, the response has no human-readable status line, and all header names are lowercase.

Example: what if we want to modify the method? Let's say we want to DELETE the resource at that URL:

```
$ curl -v --http2 -X DELETE https://httpbin.org/user-agent
> DELETE /user-agent HTTP/2
> Host: httpbin.org
> user-agent: curl/7.74.0
> accept: */*
>
< HTTP/2 405
< date: Thu, 27 Jul 2023 21:20:24 GMT
< content-type: text/html
< content-length: 178
< server: gunicorn/19.9.0
< allow: GET, OPTIONS, HEAD
< access-control-allow-origin: *
< access-control-allow-credentials: true
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
```

Oops! The server returns `405 Method Not Allowed`. That is because if we look at the https://httpbin.org documentation, the `/user-agent` API does not support DELETE. We need to use a path that supports this method:

```
$ curl -v --http2 -X DELETE https://httpbin.org/delete
> DELETE /delete HTTP/2
> Host: httpbin.org
> user-agent: curl/7.74.0
> accept: */*
>
< HTTP/2 200
< date: Thu, 27 Jul 2023 21:30:06 GMT
< content-type: application/json
< content-length: 214
< server: gunicorn/19.9.0
< access-control-allow-origin: *
< access-control-allow-credentials: true
<
```

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.74.0"
  },
  "json": null,
  "url": "https://httpbin.org/delete"
}
```


The DELETE is successful! The server is also sending us content (in this case, an example JSON).

3.3 Cross-Origin Resource Sharing (CORS)

It is common for web applications to need resources from different domains. This can be as simple as requesting data from a third-party API or embedding content from another website. However, due to security concerns, browsers enforce a strict "same-origin" policy by default, which restricts web pages from making requests to domains other than their own.

To manage this restriction and enable controlled cross-origin communication, the Cross-Origin Resource Sharing (CORS) mechanism comes into play.

> Note that CORS applies only to web browsers! Mobile applications and native applications are not affected by CORS.

3.3.1 Understanding the Same-Origin Policy

Before diving into CORS, let's first understand the concept of the same-origin policy (SOP). The same-origin policy is a security feature implemented by web browsers to prevent web pages from making unauthorized requests to different domains. Under this policy, web pages can only make requests to resources (e.g., JavaScript, images, stylesheets, or APIs) located on the same domain from which the web page originated. This policy aims to protect users from potentially malicious actions by ensuring that resources are only loaded from trusted sources.

This is where CORS comes into play, allowing web applications to relax the same-origin policy selectively.

3.3.2 Introducing Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) is a security feature that enables controlled access to resources on different domains. It defines a set of HTTP headers that both the web server and the web browser use to determine if a cross-origin request should be allowed.

CORS works by adding HTTP headers to responses from the server and evaluating these headers on the client side before allowing or blocking requests. The key CORS headers include:

- `Access-Control-Allow-Origin`: specifies which origins (domains) are permitted to access the resource. It can be a specific domain or a comma-separated list of domains (the standard indicates also the asterisk as a wildcard, but we strongly discourage its use as it affects other CORS flags as well). For example, if you want to allow access from `http://localhost:3000` as the origin, you can set the header `Access-Control-Allow-Origin: http://localhost:3000`
- `Access-Control-Allow-Methods`: lists the HTTP methods (e.g., GET, POST, PUT, etc.) that are permitted when accessing the resource. It helps define what actions can be performed on the resource from a different domain.
- `Access-Control-Allow-Headers`: specifies the HTTP headers that the client can include in the actual request when making a cross-origin request.
- `Access-Control-Expose-Headers`: lists the headers that the client can access after the response is received.
- `Access-Control-Allow-Credentials`: indicates whether the browser should include credentials (e.g., cookies or HTTP authentication) when making the request.
- `Access-Control-Max-Age`: specifies how long the results of a pre-flight request (an initial request sent before the actual request) can be cached.
- `Origin`: sent by the client, indicates the origin (domain) of the requesting page.

CORS defines two different types of requests: simple requests, and pre-flighted requests. A request qualifies for one of these two categories based on its characteristics, and a different policy applies.

3.3.3 Simple requests

Simple requests are the most common type of CORS requests and are used for straightforward cross-origin requests. A request is considered a simple request if it meets all of the following conditions:

- the HTTP method used in the request is one of the following: GET, HEAD, or POST;
- the only allowed custom headers in the request are those that the browser considers "simple headers", such as `Content-Type`, `Content-Language`, and `Accept`.

For simple requests, the browser follows these steps when handling CORS:

1. the browser adds an `Origin` header to the request, indicating the origin (domain) of the requesting web page;
2. the browser sends the request to the server;
3. the server checks the `Origin` header and decides whether to allow or deny the request based on its CORS policy. If the server allows the request, it responds with the appropriate CORS headers, including `Access-Control-Allow-Origin`, to specify which domains are permitted to access the resource;
4. The browser enforces the same-origin policy by checking the `Access-Control-Allow-Origin` header. If the header allows the origin of the requesting page, the browser allows the web page to access the response data. Otherwise, it blocks the request.

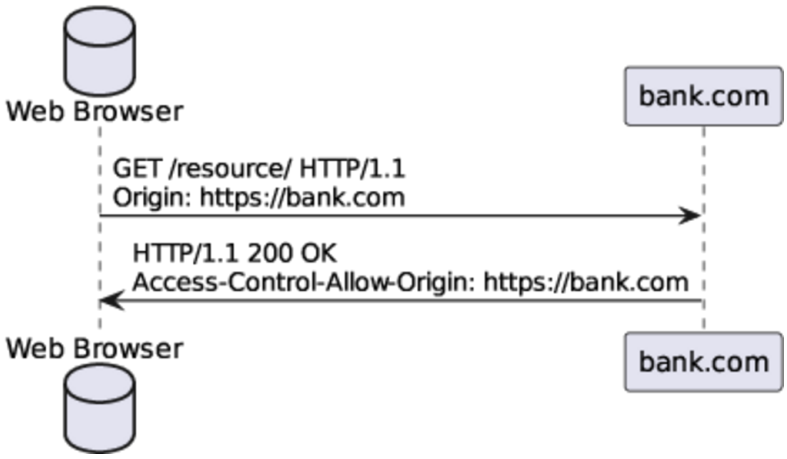


Figure 3.6: Simple CORS request.

3.3.4 Pre-flighted Requests

Pre-flighted requests, on the other hand, are used for more complex cross-origin requests that do not meet the criteria of a simple request.

For pre-flighted requests, the browser follows these steps when handling CORS:

1. Before making the actual request, the browser sends a preflight `OPTIONS` request (a separate HTTP request with the HTTP method `OPTIONS`) to the server. This preflight request includes an `Origin` header and additional headers such as `Access-Control-Request-Method` and `Access-Control-Request-Headers` to inform the server about the actual request.
2. The server processes the preflight request and determines whether the actual request should be allowed. It checks the `Origin`, `Access-Control-Request-Method`, and `Access-`

`Control-Request-Headers` headers and responds with appropriate CORS headers, including `Access-Control-Allow-Origin`.

3. If the server allows the preflight request, the browser proceeds to make the actual request. Otherwise, it blocks the request.
4. After receiving the actual request, the server handles it and includes the relevant CORS headers in the response.
5. The browser enforces the same-origin policy by checking the `Access-Control-Allow-Origin` header. If the header allows the origin of the requesting page, the browser allows the web page to access the response data. Otherwise, it blocks the request.

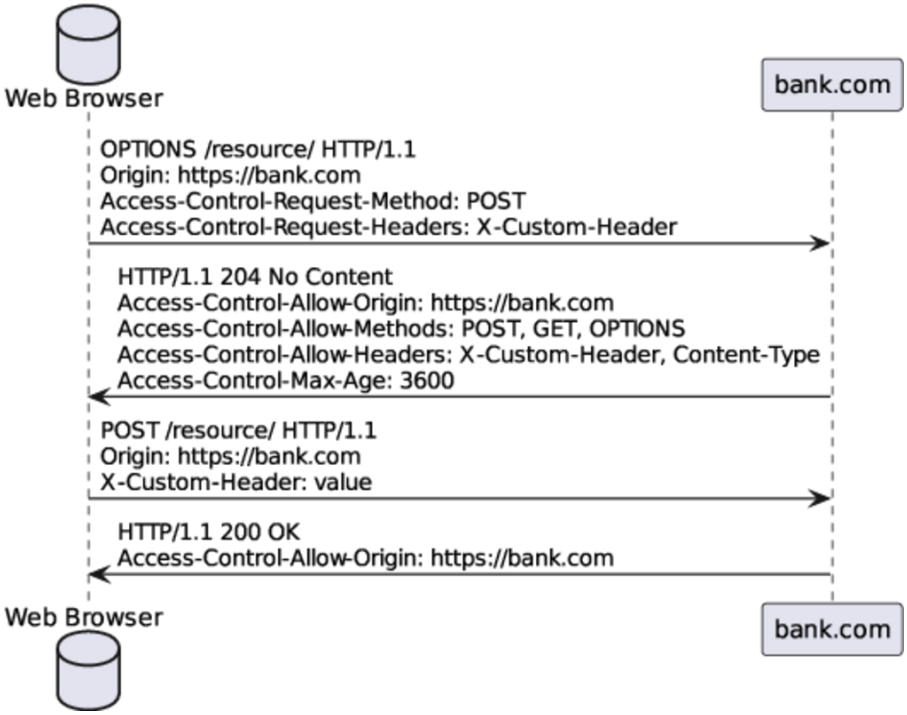


Figure 3.7: Pre-flighted CORS request

3.4 JavaScript Object Notation (JSON)

JSON stands for *JavaScript Object Notation*. It is a lightweight text-based format for serializing and exchanging data. Thanks to this property, it serves as a language-independent, human-readable, and easy-to-parse way of representing structured data. It is used almost everywhere in RESTful APIs as format to represent resources. It is used often also to store, retrieve and exchange data via serialization/deserialization.

Although it was born in the JavaScript language (hence the name *JavaScript Object Notation*), parsers and generators are available for virtually every programming language.

JSON data is organized in a key-value format, similar to the way objects are represented in various programming languages. JSON allows the representation of various data types, including scalars (strings, numbers, and booleans), arrays, associative arrays (using dictionaries or maps), and nested data structures. Values are `null`-able, meaning that all fields may assume the special value `null`.

JSON consists of two primary data structures:

- JSON Object: an **unordered** collection of key-value pairs, enclosed within curly braces `{}`. Each key is a string that uniquely identifies a value within the object, and the key-value pairs are separated by colons `:`. The keys and values can be of data types: strings (enclosed by quotes `"`), numbers, booleans, arrays, objects, `null`.
- JSON Array: an **ordered** list of values, enclosed within square brackets `[]`. Values can be of data types: strings (enclosed by quotes `"`), numbers, booleans, arrays, objects, `null`. Differently from arrays in many programming languages, items within the same JSON array may have different types.

Example: a simple JSON that describes an object that contains one item, named `user`, which is itself an object. The inner object contains three fields: `firstName` and `lastName` of type strings, and `age` of type number:

```
{
    "user": {
        "firstName": "John",
        "lastName": "Smith",
        "age": 27
    }
}
```

Example: a simple JSON that describes an object that contains one item, named `wasaWeekdays`, which is an array with two strings:

```
{
    "wasaWeekdays": ["tuesday", "thursday"]
}
```

Example: a more complex JSON example. Note that spaces and newlines outside strings quotes are ignored by parsers.

```
{
    "anObject": {
        "aNumber": 42,
        "aString": "This is a string",
        "aBoolean": true,
        "nothing": null,
        "anArray": [
            1,
            {"name": "value", "anotherName": 12},
            "something else"
        ]
    }
}
```

3.5 YAML Ain't Markup Language

YAML, which stands for *YAML Ain't Markup Language* is a human-readable data serialization format. It is often used for configuration files and data representation in software applications. YAML is designed to be user-friendly and easy to read and write.

YAML is characterized by the following key features:

- *Human-Readable*: YAML uses indentation and whitespace to structure data, making it easy for humans to understand and modify without the need for complex tags or symbols;
- *Whitespace-Sensitive*: the indentation in YAML is crucial for defining the hierarchy and nesting of data elements. This means that spaces and tabs are significant and must be used consistently;
- Data Structure: YAML allows the representation of various data types, including scalars (e.g., strings, numbers, and booleans), arrays (using lists), associative arrays (using dictionaries or maps), and nested data structures.
- Comments: YAML supports comments, allowing users to add notes or explanations within the data without affecting the data processing.

A YAML document is defined in this way:

- Maps (objects) can be defined:
  - using one level of indentation; or
  - using curly braces `{}`;
- lists (arrays) can be defined:

  - using the hyphen symbol `-`; or
  - using square brackets `[]`;

- strings are not required to have delimiters, except when they may be ambiguous: in that case they must be enclosed in single `'` or double `"` quotes;
- comments are prefixed by the hash symbol #
- a single YAML file may have multiple YAML documents, separated by `---` or `...`
- a multi-line string can be defined by specifying a "block indicator", which has different meaning depending on the symbols:
  - `>` means that newlines are removed (effectively transform the newline in a single line when parsed) except when the line is empty;
  - `|` means that newlines are preserved;
  - by default, only one newline is preserved at the end of the string: a hyphen `-` removes it, a `+` preserve all newlines at the end of the string;
- YAML supports references: block can be defined using `*` and a name, and they can be referenced by using `&` and the block name.

Note that, per YAML definition above, JSON files are valid YAML. So, YAML is a *superset* of JSON.

Example: a simple YAML document that describes an object that contains one item, named `user`, which is itself an object. The inner object contains three fields: `firstName` and `lastName` of type strings, and `age` of type number:

```yaml
user:
  firstName: John
  lastName: Smith
  age: 27
```

Example: a more complex JSON example. Note that indentation is used to define objects and lists boundaries:

```yaml
anObject:
  aNumber: 42
  aString: This is a string
  aBoolean: true
  nothing: null
  anArray:
  - 1
  - anotherObject:
      someName: some value
```

```
      someOtherName: 1234
  - something else
```

Example: a YAML using JSON syntax for lists and nested objects:

```
anotherArray: [1,2,3]
anotherObject: {"city": "Rome", "country": "Italy"}
aStringWithColon: "COVID-19: procedure di accesso"
aNumericString: "0649911"
aLongString: |
  this string spans
  more lines
```

3.6 REpresentational State Transfer (REST)

3.6.1 REpresentational State Transfer

REST stands for *Representational State Transfer*, an architectural style for designing loosely coupled networked applications and web services. It was introduced by Roy Fielding in his Ph.D. dissertation in 2000. REST is commonly used to develop Web APIs (Application Programming Interfaces) over HTTP.

REST has software engineering principles and interaction constraints like other architectural styles. A *RESTful* service implements all REST principles and satisfies all constraints.

The REST style does not define the details of the implementation. Instead, it is a design style focused on components and their roles in a described system with constraints.

3.6.2 REST building blocks

The **resource** is the fundamental abstraction of data in REST. A resource is a set of elements or values that can vary over time. Anything in REST can be a resource: a document, an image, a collection of resources, etc. Multiple resources can be mapped to the same value.

The current or intended state of the resource, at any particular time, is known as the **resource representation**. It consists of data and the metadata describing the data. The data format of a representation is known as a *media type* (or MIME type): it is a two-part identifier for file formats and format contents transmitted on the Internet. It defines how a representation is to be processed.

> The *current* state of the resource is usually used in responses, as it is the resource's state on the server. The *intended* state of the resource is usually sent in requests that change the state of the resources on the server.

REST components (such as clients and servers) can exchange data and perform actions on resources by using a representation of the resource in their requests and responses.

3.6.3 Constraints and principles

Let's analyze the architectural constraints by discussing the general principles of REST.

3.6.3.1 Stateless

The *Stateless* constraint requires that each interaction between a client and a server is stateless, meaning that each request by the client must contain all the necessary information to understand the request. The server must not retain any state (or context). Optionally, the state/context may be stored on the client.

This constraint indirectly creates multiple features in a REST system:

- visibility: as interactions are self-described, there is no need for a component to look beyond a single request. In other words, any decision, modification, or elaboration can be done by looking only at the current request and not in some hidden context;
- reliability: as there is no state, and interactions are self-described, recovering from partial failures is trivial;
- scalability: due to the stateless nature of interactions, there is less the need for coordination and data sharing among components and their resource consumption.

3.6.3.2 Cache

The *cache* constraint requires that all response data be marked implicitly or explicitly "cacheable". If a response is cacheable, the client can use that response for an identical request in the future.

The cache constraint reduces the number of interactions, improving the throughput, scalability, and performance, as responses are immediately available with no load on other components in the REST system.

3.6.3.3 Separation of concerns

The separation of concerns applied in REST defines the architecture model as a client-server. In this model, the two components have different roles. The rationale behind this constraint on the architecture is to decouple data storage from the user interface, improving portability and scalability.

This new constraint allows the client and the server to evolve independently, an essential requirement in applications running on the Internet.

3.6.3.4 Uniform interface

The *Uniform interface* is the core of REST. It defines that components have a *uniform* interface, meaning that every interaction uses the same basic structure (same protocol, same message structure, etc.). This strong constraint helps simplify the system's overall design at the cost of efficiency.

> By implementing all services behind a uniform interface, we are limiting the customization that would improve some of these services.

This generalization also helps to decouple the implementation from the definition of the service, allowing the former to evolve and change independently.

The constraints REST defines to correctly implement a *Uniform interface* are four: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state (i.e., the client needs only the initial URL).

3.6.3.5 Layered System

The idea behind this principle is that layers in the transmissions (i.e., intermediate hosts) can isolate, modify, replay, and encapsulate REST messages. This allows the architecture to have intermediate agents that, by separating "layers" in the architecture, lower the overall complexity and decouple services in different layers. Components in different layers cannot see past the immediate previous/next layers.

> For example, intermediate hosts can provide caching to lower the load on the service provider, enforce policies, or encapsulate legacy services in REST.

Each component in an intermediate layer in REST can effectively modify the content of the interaction flowing through, as messages are self-descriptive, and the semantics is visible to every component.

By definition, intermediate hosts may reduce the overall system throughput. The presence and the number of layers in a REST design should be balanced between the performance gain by offloading to intermediate layers and removing them to create more visibility between the client and the server.

3.6.3.6 Code-on-demand

The *Code-on-demand* principle of REST extends client functionality by downloading and executing code. Although this reduces the client's complexity and allows flexibility, it reduces also the visibility of the system itself. For this reason, this principle is optional, and it is often skipped in implementations.

3.7 Further readings

- HTTP
  - https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
  - https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
  - https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
  - https://www.rfc-editor.org/rfc/rfc7231
  - https://www.rfc-editor.org/rfc/rfc9110.html
  - https://http.cat
  - https://http3-explained.haxx.se/
- JSON
  - https://www.json.org/json-en.html
  - https://en.wikipedia.org/wiki/JSON
  - https://jsonlint.com
  - https://en.wikipedia.org/wiki/YAML
  - https://yaml.org
  - https://onlineyamltools.com/validate-yaml
  - https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html
  - https://en.m.wikipedia.org/wiki/Recursive_acronym
- REST
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
  - https://restfulapi.net
  - https://datatracker.ietf.org/doc/html/rfc3986
  - https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

4 Designing and documenting APIs

Now that we have seen which web technologies we will use, it's time to focus on *designing* and *documenting* our REST API. Designing a good API from the beginning helps you decouple the backend and the frontend; this has several benefits when organizing the work and developing the components of your systems.

During the design phase, we will document our REST API using an industry-standard document format named "OpenAPI". Documenting an API is an essential step toward implementation and maintenance. A well-documented API helps frontend and backend developers to reduce the amount of misunderstanding and friction they may encounter during their work.

When starting a new project, the right time to write the API documentation is during the planning phase before starting the coding. Writing what your backend should return and what your frontend should send helps you to check whether the planning phase successfully considered the boundary between client and server.

We will use OpenAPI version 3.0.0, the most common version, and YAML as the language to write the documentation file.

To simplify our journey, we learn how to document them as we go so that we can discuss them using unambiguous writing. In this chapter, "dictionary", "object", and "map" are used as synonymous.

4.1 Introduction to APIs

An *Application Programming Interface* (API) is a set of rules and protocols that allows different entities to communicate and interact with each other. It serves as an intermediary that enables seamless integration and cooperation between disparate systems, enabling them to effectively exchange data, functionalities, and services.

The interface is considered a *contract* between two entities, which we will refer to as "*provider*" and "*consumer*". The *provider* is the entity that provides a service accessible by *consumers* using the specification in the contract (e.g., a REST server). The *consumer* is the entity that accesses and uses the service from the *provider* by issuing commands and queries defined in the contract (e.g., a REST client).

The need for an API arises because we want to *abstract* the complexity of a service, hiding the *implementation*. In API contracts, only input and output data are described in detail: *how* a particular function works is not. **A good API is abstract enough to provide a usable interface while allowing developers to change the underlying implementation without breaking the contract.**

You can define APIs over any communication media: network protocols, shared memory, named pipes, inter-process communication in general, or even function calls internal to a single application. For example, the operating system exposes a set of API named "system calls"; in Java, the "interface" type is a contract between classes; the timekeeping software in your PC synchronizes your local clock by using an API over a network protocol.

APIs can be either *private*, or *public*. A *private* API is meant to be used internally in a software, company, or a private group of entities. A *public* API is meant to be shared with others, freely or behind authorization. Private APIs may be tailored to specific usage, while public APIs should be defined as general as possible to cover unexpected use cases.

> Smartphone and web applications usually use private APIs while exchanging data over the Internet. In fact, just because an app uses the Internet to access an API, it doesn't mean that the API is freely usable!
>
> Companies usually have dedicated public APIs different from private APIs to allow third-party services/companies/developers to exchange data with them.
>
> For example, Deutsche Bahn (a German train company) has private APIs for official apps and a set of public APIs for developers at https://developers.deutschebahn.com/

Sometimes it is possible to update the *contract* without breaking it by adding a new action, resource or collection in a way that is not disrupting who is still using other actions, resources or collection. For those who are not using the "*updated contract*", nothing will change.

Some other times you may be forced to *break the contract*, i.e., to change some definition in the API due to some change in the underlying implementation that won't fit in your existing contract. We refer to this as "**breaking changes**". For example, the management of a train company may decide to use letters for train platforms instead of numbers: existing implementation can't represent train platforms anymore.

There are different ways to deal with these changes. Some solutions are presented in the "API versioning" section.

4.2 OpenAPI Specification Overview

There are different ways of documenting an API. In some contexts, a Word document is more than enough. Linux "system calls" are documented in the Linux kernel Git repository. Private APIs in small libraries may be documented directly in the code.

Web developers have different needs. The large number of public APIs creates a problem of *scalability*: each (small) API is documented using a separate tool, and some critical information is missing, outdated or unclear. Out of this mess, different proposals appeared on the web, aiming to standardize the documentation to other developers, especially for public APIs.

OpenAPI Specification (OAS), formerly known as *Swagger Specification*, is a vendor-neutral open standard for defining and documenting Web APIs defined over HTTP. It quickly became the industry standard for describing APIs, especially RESTful services.

The specification is *machine-readable*, meaning that it is easy to read and process using the software. An OAS document can be defined using YAML or JSON. The machine-readable property is exploited by various tools to produce human-readable documentation, automatic validation of APIs, code generation for both *provider* and *consumer* (including mocking servers), and other features.

> Even if OAS supports JSON, we strongly suggest YAML as it's less verbose and more human-friendly. Since YAML is a superset of JSON, OAS tools support both with no differences.

An OpenAPI document is composed of multiple sections:

1. The OpenAPI document version;
2. the metadata describing the service that is documented (e.g., name of the service, version of the document, etc.);
3. the list of servers (URLs) where the service is found;
4. a dictionary that describes all API endpoints/URIs, the method used for accessing them, the parameters, the request and response body, and more;
5. the definition of common schemas (such as resources) in the document;
6. the information about security in this service (authentication/authorization);
7. tags, external documents, and other information;

4.2.1 Preamble

Every OpenAPI file starts with the definition of the OpenAPI version. Then, you may want to provide some information about the API, like a title and a description. Also, API "version" is essential.

```yaml
openapi: 3.0.0
info:
  description: Fontanelle app backend REST API
  title: Fontanelle API
  version: 1.0.0
```

Optionally you can add "servers" (also known as "base URLs"): they can be used by OpenAPI tools (like the Swagger Editor[1]) to help you test the API. If you don't have any servers yet, you may omit the "servers" section entirely. However, when you have a server that implements these APIs, you can write something like this:

```yaml
servers:
  - url: http://localhost:3000
```

> `localhost` is a special hostname that points your local PC to itself — also known as "loopback address". It's often written via its IP representation: `127.0.0.1` or `::1`. The URL `http://localhost:3000` indicates that the server is present on the same PC (not in a remote host) at port `3000` via HTTP.

Now that the preamble is complete, we will see how to populate the rest of the document while designing our API.

4.3 Designing REST API

The first step in designing an API is determining which concepts we will represent as REST *resources* and *collections*. Generally speaking, for now we can assume that every (abstract or concrete) object we need to manipulate can be represented with a *REST resource*, and every set, list, or collection of those objects can be defined with a *REST collection*.

To capture the essence of these guidelines, let's assume that we have a project in our hands, "Fountains" (we will use this project during the rest of the book).

4.3.1 Fountains

In Rome, drinking water is available thanks to little public fountains, also known as *nasoni* ("large noses" due to their nose-like shape). They were installed in late 1800 by the local municipalities in the capital city and nearby areas. More than 2500 are still working.

[1] https://editor.swagger.io/

Figure 4.1: A typical *nasone* in Rome.

The "Fountains" project includes a mobile app and a website that allow citizens and tourists to explore nearby drinking fountains (see their location and status). Also, they will be able to change the state or location of any fountains, add missing ones, and remove those that are no longer present.

A fountain's status is "good" when it is in working condition and "faulty" if broken.

Frontends will communicate with a central server via REST and JSON. No authentication is needed, nor is user identification.

4.3.2 Identifying resources

From the description above, we can distinguish one candidate to be a resource: the "fountain". In fact, it matches two main criteria, by definition: first, it is a concept that is so important to be represented on its own, and second, the frontend performs actions over a "fountain": create a new fountain, delete an old one, update an existing one.

Now we can focus on the content: what *is* a fountain (in our system), by the way? The answer is again in the project specifications: a fountain is a *location* (coordinates) and a *state*. We describe the location as latitude and longitude, expressed using the decimal degree format, and the state as a string with only two possible values, "good" or "faulty". We also add an integer unique identifier to distinguish multiple fountains.

In OpenAPI, we describe *Fountains* under the *schema* key, which is inside *components*:

```
components:
  schemas:
    Fountain:
```

```
    title: Fountain
    description: "This object represent a single Fountain (\"nasone\")."
    type: object
    properties:
      id:
        description: Unique fountain identifier.
        type: integer
        example: 1
        readOnly: true
      state:
        description: "Describe the state of the fountain. If it's
↪   \"good\", the fountain is in working condition; if it's \"faulty\", the
↪   fountain is broken."
        type: string
        enum: ["good", "faulty"]
        example: good
      latitude:
        description: Location latitude in decimal degrees format.
        type: number
        format: float
        example: 12.34
        minimum: -90
        maximum: 90
      longitude:
        description: Location longitude in decimal degrees format.
        type: number
        format: float
        example: 56.78
        minimum: -180
        maximum: 180
```

The `type` of the `Fountain` schema is `object`, meaning that it's a dictionary. You can describe its properties by indicating the property's name as a key under the `properties` dictionary, and then describe of the format of the content as value of that key.

Let's take, for example, the `id` property:

```
      id:
        description: Unique fountain identifier.
        type: integer
        example: 1
        readOnly: true
```

We have a `description` field, to help the developer understand the meaning of this property. Then, we have the `type` key: in this case, the value is an integer.

> Some types may have additional fields that specify subtypes or restrictions. Examples:
>
> - `type: number` often requires a `format` key to specify the kind of number (see `latitude` and `longitude`)
> - `type: string` may have a `enum`, which indicates the only values that can be accepted in that field.
> - `type: string` also may use `format` for restricting the subtype; for example, `format: date-time` is used for RFC3339 dates (which are stored using `string`)

Good API documentation should also provide examples: the value in the `example` field should be something developers can expect to be in that property when reading/writing the resource.

The definition of our *Fountain* resource object also contains a special constraint for the `id`. Thanks to `readOnly: true`, we can describe that the unique identifier is something that cannot be changed (directly) using the REST API.

> The reason for having `readOnly` is that you don't want the frontend to provide the value for these properties when sending an object (in other words, you don't want to allow them to modify this property).
>
> A typical example is a server-provided identifier, such as a server-side counter. In that case, you want the developer to receive the property in objects and use the value in request parameters, but you don't want them to include it in the object when it's sent to the server.
>
> Server-provided identifier are common because they solve different problems, such as collisions (multiple instances of the frontend try to create an object with the same identifier).

4.3.3 Identifying collections

By definition, our project has multiple *Fountain* resources. Also, in the specifications, there is a reference to an action ("retrieve nearby fountains") that works on numerous *Fountains*. So, we also have a *collection of Fountains*.

> Why not a *collection of **nearby** fountains*? In fact, we may do that, and it would be correct too. However, given that "nearby" is some sort of "filter" for a more general collection, we decided to have a general collection and filter it later.
>
> See later for more details about what would be the difference when using a stricter approach.

4.3.4 Assign URIs

Now that we identified all our resources and collections, we must complete them by defining their *URIs*. We will specify the URI as absolute paths in the web server, as everyone typically does (although this is a convention, not a rule).

We can define, for example, the URI for our collection as `/fountains/`, while the *Fountain* is identified by the URI `/fountains/{id}` (where `{id}` represent a placeholder for the identifier we defined for the resource).

> If we define the collection as *collection of **nearby** fountains* (see note above), it would be best to use another path, such as `/nearby_fountains/`, as it would be logically unsound to have `/fountains/` as collection for nearby fountains (because `/fountains/{id}` identifies *any* fountain, not only the one near you).
>
> Defining `/fountains/{id}` as resource URI and `/nearby_fountains/` as collection URI for nearby fountains is correct. Note that this does *not* imply that there is a third collection (`/fountains/`), although it might exist if we decided to have both collections.
>
> Also note that resource URIs are not required to share part of their URI with their collections. Having one collection in `/nearby_fountains/` and resources on `/fountains/{id}`, and NO collection on `/fountains/`, is perfectly valid.

4.3.5 Identifying actions

By looking at the description of the project, we need to create at least the following actions:

1. Create a new fountain;
2. Report a new state (faulty/good) or the new location of an existing fountain;
3. Delete an existing fountain;
4. Get the list of (nearby) fountains with their location and state.

Actions 2 and 3 are on an existing resource, while actions 1 and 4 are on the collection.

> The reason for 1 to be over a collection is simple: given that we decided above that the *Fountain* URI contains its ID and that the ID is *read-only* in our API, we cannot construct the URI of the new resource before getting a new ID from the server.
>
> This means that, during the resource creation, we must send the resource to the server **minus** its ID (as we don't have it yet), and the server will create the resource for us, giving us back the resource with its ID.

> The logical place where we can do this action is on the collection of all *Fountains*.

4.3.5.1 API: Create a new fountain

The first step to define our action is choose the URI: as we stated above, the URI for creating a new *Fountain* will be the collection URI, `/fountains/`.

The second step is choose the HTTP method. As we need to create a new *Fountain*, we need an HTTP method that can modify the server state - this excludes methods for retrieving data, like `GET` and `HEAD`. We don't need to delete the resource, so we can obviously exclude `DELETE`. From the remaining methods (`POST` and `PUT`), we use `POST`.

> Why not PUT? PUT is idempotent. However, for a request to be idempotent, the server needs to have some value for identifying identical/duplicated requests, such as an ID, otherwise there is no way to distinguish whether a second request is identical to the previous one.
>
> Given that the server will provide the ID *after* the resource creation, we don't have it (yet). So we are forced to use `POST`.

Now we can focus on the request's body. We already described the *Fountain* structure in OpenAPI, so we add a reference to that definition; we still need to choose its *representation*. For example, we can say that the server accepts a JSON: to do that, we can specify a content using the MIME type `application/json` under the `requestBody` key, and we can create a reference to the "Fountain" definition.

> Note that the ID in the *Fountain* scheme is marked `readOnly`: this means that it won't be present in any request body. If the client erroneously specifies the ID property, the server must ignore it (or reply with HTTP status code `400 Bad Request`).

The last part is the definition of possible responses that the client should expect. Here we enumerate all response codes as keys and describe the meaning and the response body (if any) as value. In case of success, we can use HTTP status `201 Created`, returning the *Fountain* JSON object to the client (this time, the object will contain the ID).

```yaml
paths:
  /fountains/:
    post:
      operationId: createFountain
      summary: Create a new fountain
      description: |-
        Create a new fountain using the properties in the request body.
```

```
        The server will create a new unique ID, the client can find it
        in the response.
      requestBody:
        content:
          application/json:
            schema: { $ref: "#/components/schemas/Fountain" }
      responses:
        "201":
          description: Fountain created successfully
          content:
            application/json:
              schema: { $ref: "#/components/schemas/Fountain" }
```

4.3.5.2 API: Update fountain attributes

To update a fountain, we need to define an action for changing fountain attributes. One method is to replace the resource content with new content using HTTP method PUT, another is replacing single attributes using HTTP PATCH. We will see how to define an action with PUT; defining the action using PATCH is almost identical. In either case, the path of the resource is `/fountains/{id}` as we defined above.

> Why PUT instead of POST? Like before, we need to ask ourselves what property we need from HTTP. In this case, idempotency is good: updating a resource multiple times with the same content should be the same as updating it once (as we send the same content over and over, we should not expect changes).

We have one parameter in the URL, which is `{id}`. We need to specify the schema for that parameter, and we can define it in the top-level `components` key, so we can reuse it in other requests.

As in the "create new fountain" API, we have a body that contains a "Fountain" in JSON format.

Apart from error replies, a successful request can use `204 No Content` if we don't need to send anything back to the client (e.g., all `readOnly` fields are immutable even in the server), or we can use `200 Ok` with a complete "Fountain" object if needed.

```
paths:
  /fountains/{id}:
    parameters:
      - $ref: "#/components/parameters/FountainID"
    put:
```

```
      operationId: updateFountain
      summary: Update fountain properties
      description: |-
        Replaces all fountains properties with the Fountain specified
        in the body of the request.
      requestBody:
        content:
          application/json:
            schema: { $ref: "#/components/schemas/Fountain" }
      responses:
        "200":
          description: Fountain updated successfully
          content:
            application/json:
              schema: { $ref: "#/components/schemas/Fountain" }
        "404": { description: Fountain not found }

# ...
components:
  parameters:
    FountainID:
      schema:
        type: integer
        example: 1
        readOnly: true
      name: id
      in: path
      required: true
      description: Fountain ID
```

4.3.5.3 API: Delete a fountain

To delete a fountain, we must use the `DELETE` method against the path of the resource. In this case, the path will be `/fountains/{id}`.

We don't need to specify any parameter or body in the request. A successful request can have `204 No Content` as a response.

> If you need, you can return something in the request body (for example, the "Fountain" object that the client just deleted).

```
    delete:
      operationId: deleteFountain
      summary: Remove a fountain
      description: Remove a fountain from the system.
      responses:
        "204": { description: Fountain deleted successfully }
        "404": { description: Fountain not found }
```

4.3.5.4 List (nearby) fountains

To list fountains in the collection, we can use the GET HTTP method on the collection `/fountains/`. The GET HTTP method is perfect for our collection: it specifies the action of *getting* something, it has no side effect, and it is idempotent.

However, the project specifications indicate that we should retrieve *nearby* fountains. How can we accomplish that?

If you remember the previous chapter, a URL (and so a URI) can have a *query* component that we can use to provide additional information to the server. We can exploit this query string to pass some "filters". For example, we can decide that `/fountains/?latitude=1&longitude=2&range=3` is the way we ask the server to filter for fountains at `(1, 2)` within a range of `3`.

> Why don't we put these parameters in the URL path (before ?)? These parameters are not defining a resource or the collection itself, so we must not put them in the URL path.

We also define that the server indicates a successful response with `200 Ok`, returning a list of "Fountain" resources.

```
paths:
  /fountains/:
    get:
      operationId: listFountains
      summary: Get the list of available fountains
      description: |-
        Return the list of all fountains in the system.
        If the client specifies a set of coordinates for a point,
        the list will be sorted by distance from that point
        and filtered for a range.
        Optionally, the client can specify a custom range.
      parameters:
        - name: latitude
```

```
          in: query
          required: false
          description: Latitude for sorting/finding fountains.
          schema: { $ref: "#/components/schemas/Latitude" }
        - name: longitude
          in: query
          required: false
          description: Longitude for sorting/finding fountains.
          schema: { $ref: "#/components/schemas/Longitude" }
        - name: range
          in: query
          required: false
          description: |-
            Range for the location filter. Default: 10.
            If coordinates are not specified, this parameter is ignored.
          schema:
            type: number
            format: float
            example: 12
            minimum: 1
            maximum: 200
      responses:
        "200":
          description: List of fountains
          content:
            application/json:
              schema:
                type: array
                items: { $ref: "#/components/schemas/Fountain" }
# ...
components:
  schemas:
    Latitude:
      description: Location latitude in decimal degrees format.
      type: number
      format: float
      example: 12.34
      minimum: -90
      maximum: 90
    Longitude:
      description: Location longitude in decimal degrees format.
      type: number
```

```
      format: float
      example: 56.78
      minimum: -180
      maximum: 180
```

4.3.6 Resource representation

As you may remember from the REST chapter, we introduced the concept of "representation" of a resource. In our definitions above, we decided to use JSON as representation for the resource. The resulting JSON object for a fountain is something like this:

```
{
  "id": 1,
  "state": "good",
  "latitude": 12.34,
  "longitude": 56.78
}
```

However, if we want to use XML instead, we can use the `application/xml` MIME type under the `content` key in the definition of actions, and the result will be something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Fountain>
  <id>1</id>
  <status>good</status>
  <latitude>12.34</latitude>
  <longitude>56.78</longitude>
</Fountain>
```

You can specify that an API supports multiple MIME type by adding multiple keys under `content`. The client specifies the desired MIME type using the `Accept` HTTP header, and the server should follow it, if documented in the OpenAPI (if not, the server should reply with `406 Not Acceptable`).

4.4 API versioning

Whether you are designing private or public API, it is a good habit to have a solution to deal with changes in the contract. The standardized way for indicating changes is *API versioning*.

You can do *API versioning* in different ways. Still, they all end up with a similar mechanism: the client should send something to the server that (directly or indirectly) indicates a version of the API it requests.

One widely used mechanism is putting the API version in the URI as a common prefix for all URIs. In this way, the client indicates that it is doing an action on a specific resource version (or collection). The version number is incremented at each revision of the API document. For example, the API design may require an HTTP GET on `/v1/fountains/` to retrieve the list of all fountains in version 1.

Another typical solution is to include the version in an HTTP header. For example, the same GET on `/fountains/` may be like:

```
GET /fountains/ HTTP/1.1
user-agent: curl/7.64.1
host: www.example.com
acmecorp-api-version: 1
```

Note that we used a custom HTTP header, as there is no standard header specific to API versions.

> To build the custom header, we prefixed it with `acmecorp`. We do this to avoid clashing the name with a future standardization of the `api-version` header. For further information, see the RFC 6648 [a].

[a] RFC 6648 - Deprecating the "X-" Prefix and Similar Constructs in Application Protocols, https://datatracker.ietf.org/doc/html/rfc6648

4.5 Best practices

Here we have collected some suggestions on REST API over HTTP. We suggest you to use them to avoid pitfalls and bugs that may slow down your development.

Also, remember that in HTTP and REST, the paths `/users/`, `/Users/`, and `/users` are different. Make sure always to use the exact URL that you define for the resource.

4.5.1 Use nouns instead of verbs in resource URIs

You should use nouns in URIs, as URIs represents resources, and resources are objects. Never use or include verbs in the resource URI: the action is expressed using the HTTP method! Different actions can be executed on the same URI. For example:

```
GET http://example.com/managed-devices/{id}

PUT http://example.com/managed-devices/{id}

DELETE http://example.com/managed-devices/{id}
```

Also, it is common to use singular nouns for a single resource (e.g., `http://example.com/users/admin`) and plural nouns for a collection of resources (e.g., `http://example.com/users/`).

4.5.2 Use logical nesting / hierarchy in URIs

We suggest using the *forward slash* (/) to express hierarchy, and the *trailing slash* when addressing a collection of resources. This organization helps when the API grows in size.

Example:

```
http://example.com/users/{userid}/devices/{deviceid}
```

4.5.3 Return the correct HTTP status code

Many errors can be mapped directly to an HTTP status code. For example, missing required parameters or bad value for a parameter can be mapped to `400 Bad Request`.

If you want to provide additional information for the developer or the user, you can define a common body for errors (for example, a JSON containing various fields with details about the error).

4.5.4 Use query to filter and paginate

As we saw earlier in the GET operation for the fountains collection, you should use the `query` component of a URI to filter and paginate the result. Never use the body or the URI for filtering: the collection is still one, you are just filtering it!

Example:

```
http://example.com/managed-devices/?region=USA
```

4.5.5 Do not define body in GET and DELETE requests

You might be tempted to send a body inside a `GET` or `DELETE` request. While technically you can have a body inside a `DELETE` request (per HTTP standard), **this definition is forbidden in REST** as it creates risks due to the undefined behavior in the HTTP definition[2].

4.5.6 Use big binary streams/files directly in JSON

It is not advisable to embed binary data inside JSON or other formats for performance reasons. If the API is returning or accepting only a single piece of information (the binary file, e.g., the photo), then it's better to define MIME types directly (like `image/png` and `image/jpeg`) and put the file in the body.

Otherwise, if you need to return or accept multiple information, one strategy is:

- In requests, you can use one of these two strategies:
  - you can use `multipart/form-data` and put the image and the structured data in one request; or
  - you can send the image in a dedicated API (e.g., `/images/`), and then insert the image URL in the JSON in the request with the structured data.
- In responses, you can reply with `application/json` with structured info, and a handle for the image; e.g.,
  - a full URL where the image can be found; or
  - some identifier the client should use in another API (e.g., `/images/{identifier}`) to get the binary file directly.

4.5.7 Do not use the body for filters

If you want to provide a filter for a collection, implement them in the **query** string part of the URL. Do not use the body of the HTTP request or the path for filtering.

Example:

```
/fountains/:
  get:
    # ...
```

[2]For more information, see this GitHub issue on the OpenAPI specifications repository: https://github.com/OAI/OpenAPI-Specification/issues/1801

```
    summary: Get the list of available fountains
    parameters:
      - name: latitude
        in: query
        required: false
        description: Latitude for sorting/finding fountains.
        schema: { $ref: "#/components/schemas/Latitude" }
    # ...
```

4.5.8 Do not swap collection name and item

Let's give an example: in a Twitter-like application, Alice wants to mute Bob (meaning that *Alice doesn't want to see Bob's posts*).

You might be tempted to define the action URL as `/users/{userid}/mute`. The URL is wrong, as there is no point on creating a `mute` resource for Bob: Bob is not muted for the entire platform, just for Alice. By using this description, it seems as if `mute` attribute is assigned to Bob no matter what. In fact, by reading the URL, we expect that the `/users/{userid}/mute` resource to exist even for different authenticated users (e.g., *John*).

The right way of implementing this is by considering "people that I muted" as a collection: `/users/{userid}/muted/{mutedid}`, where `userid` is me and `mutedid` is the muted user.

By using this URL structure, we have that:

- A user can be muted only once, respecting the fact that "mute" action is naturally idempotent;
- the unmuting action can use `DELETE` on the same path: no need for another endpoint;
- clearly, two different users have two different `muted` sub-collections.

Another way of implementing this is by collecting the user profile under a special collection (e.g., `/me/`) which is well-defined. By reading the definition, no one expects that `/me/muted/` is shared among authenticated users, while one expects that everything under `/users/` is (even if not accessible by everyone).

In other words, if `/users/` represents the collection of all users on the platform, you can't have a sub-resource (like `muted`) that is "per-authenticated-user" (Alice in the example above). However, while it's not elegant, you can have another collection (e.g., `/me/`) that contains "per-authenticated-user" resources.

4.5.9 Do not use multipart/form-data in responses

While `multipart/form-data` can be helpful when sending different content at once (upload a file while sending its metadata), it's not expected as MIME type for a reply on the server side.

If you need to send multiple contents, use the strategy described in "Use big binary streams/files directly in JSON".

4.5.10 Use PUT instead of POST when idempotency is desired

Do not use POST when the idempotency is desired. For example, use PUT when:

- Creating new resources, where the creation should be idempotent (see "Do not PUT and DELETE on a collection" below); or
- Updating an existing resource.

For example, POSTing a change of username is not correct, as the username change is naturally an idempotent action. Use PUT instead.

4.6 Further readings

- https://www.openapis.org
- https://oai.github.io/Documentation/

5 Backend Programming with Go

Welcome to the world of *Go* programming! This chapter will introduce you to the fundamentals of the Go programming language. Go is an open-source, statically typed, compiled language developed by Robert Griesemer, Rob Pike, and Ken Thompson in 2007. Since its inception, Go has gained popularity rapidly due to its simplicity, efficiency, and concurrency support, making it an excellent choice for building high-performance and scalable applications.

We will assume that you have Go installed on your machine. If not, you can install Go by following the instructions on this page: https://go.dev/doc/install.

5.1 Introduction to Go Language

Go was designed as an easy-to-learn, readable, and productive programming language while still offering powerful features. It automatically manages memory allocation and de-allocation, offering memory safety[1] and garbage collection[2]. Go also provides built-in support for concurrency through goroutines and channels.

Although influenced by C syntax, Go syntax is simpler. The Go Authors deliberately omitted some syntax constructors that make the code too complex. As a result, a Go source code tends to be more verbose when compared with C or other languages developed in the same period (such as Rust).

As we will see later, Go offers a comprehensive standard library and a dependency management system. The official Go compiler can also build static and dynamic executables and easily cross-compile Go for different platforms and architectures.

[1] A programming language is "memory safe" when it protects from several issues related to memory management and access. For example, "pointer arithmetic" (a way to access memory) may lead to out-of-bounds errors; memory-safe languages do not support "pointer arithmetic".

[2] A "garbage collector" is a piece of software that scans for unused memory (such as out-of-scope variables) and reclaims/frees the memory. It may release the memory to the operating system or keep some ready for subsequent allocations. The procedure of collecting unused memory is called "garbage collection".

5.1.1 Go packages

In Go, files within the same directory are part of the same package. Packages must have a name, and it should be declared as the first line of the file. Multiple packages in the same directory are not allowed. The package name can differ from the directory name, although we suggest using the same if possible.

```
package mypackage
//...
```

A special package is `main`; in this package, the Go compiler expects to find a function named `main()`. That function will be the *entry-point* of your program, i.e., the first function to be executed.

Within the same package, you can use any function or global variable regardless of the file where it is implemented. That is, if you have two files, `first.go` and `second.go`, you can use functions, global variables, types, structures, or interfaces between them as if all the code was in a single file.

However, when you refer to code in other packages, you need to *import* them. For example, to import the `fmt` package, you can write `import "fmt"` just after the package line. Then, you can use any **public** functions, global variables, types, structures, or interfaces in the `fmt` package by prefixing the function's name with the package name, such as `fmt.Println()`.

How do we define *public* functions, global variables, types, structures, or interfaces? Go uses a simple strategy: if the **first character** of the name is uppercase, then it is public. For example, `Println()` is public, while `printNewline()` is not. You can only use private functions, global variables, types, structures, or interfaces inside the same package.

5.1.2 Hello world!

When you start a new programming language, the first program you usually build is a simple command-line executable that prints "Hello world!" on the screen. In Go, this program is straightforward: first, we declare the `main` package; then, we import `fmt` to use its function to print messages on the screen; finally, we implement the `main()` function using `fmt.Println` to print the message:

```
package main

import "fmt"

func main() {
```

```
  fmt.Println("Hello world!")
}
```

Copy the text above in a new file named `hello-world.go` and place it in a directory called `exercises` inside your home directory. Now, open a terminal emulator and run these commands to enter that directory and run the program:

```
$ cd exercises
$ go run hello-world.go
Hello world!
```

What just happened? `go run` command builds the source file into an executable, saves it in a temporary directory, and runs it. The output you see (`Hello world!`) is the output of your Go source.

Try to change the text and issue `go run` again!

> Note: `fmt.Println("message")`, as the name suggests, sends the "message" string to the *standard-output* and adds a new line at the end. If you want a different behavior (e.g., no newline, etc.), the `fmt` package contains other functions.

What if you want an *actual* binary executable? `go run` is creating an executable using `go build` (before executing it), so we can just use that command instead:

```
$ go build -o hello-world hello-world.go
```

The `-o` flag specifies the executable's name (and optionally the path) - `hello-world` in this case. The `hello-world.go` filename is specified as the first argument. Flags are written before arguments, like in UNIX commands.

The command above created an executable in the current directory. You can execute it with:

```
$ ./hello-world
Hello world!
```

Congratulations! You wrote and successfully executed your first Go computer program!

5.1.3 Go type system

Go is a statically typed programming language; "statically typed" means that types of values are known and verified during the compilation phase by analyzing the source code. Go provides a set of built-in types; you can extend them to define others.

Built-in types include:

- `bool` contains a Boolean value;
- `string` represents strings, i.e., a list of bytes;
- `int` (signed integer), and `uint` (unsigned integer): their size depends on the platform (32-bit on a 32-bit platform, etc.);
- `int8`, `int16`, `int32`, `int64` and their unsigned variant `uint8`, `uint16`, `uint32`, and `uint64` are integers with explicit sizes;
- `uintptr` is the type for a memory pointer;
- `byte`, an 8-bit type, for representing a single byte;
- `rune`, a 32-bit type, for representing a Unicode code point;
- `float32` and `float64` to represent floating point numbers with single and double precisions;
- `complex64` and `complex128` represent complex numbers with a size of 64 and 128 bits.

All types in Go have a default value, which is the variable's value when declared without an explicit value. The default value is zero for all integer types, floating point, complex number, bytes, and runes; empty string for string type; and `false` for `bool` type.

Go syntax rules allow the use of inference to establish a type; in other words, there is no need to declare a value type explicitly if it can be inferred. Inference for literals follows these rules:

- A positive or negative integer number is given `int` type;
- A positive or negative integer number with a lowercase `i` at the end is converted to complex numbers (the real part will be set to zero);
- `true` and `false` are `bool`;
- Numbers separated with a comma are interpreted as `float64`;
- Double quote and backtick characters delimit a string;
- Single quote delimits a `rune`.

For example:

```
// This is a comment and it is ignored by Go.

// The next line declares a variable of type int
// (remember, the default value is zero).
var capacity int
```

```
// The next line assigns a value to the new variable.
capacity = 5

// The next line declared a variable of type int.
// The type is inferred from the literal value.
var length = 10

// The following lines declare two strings, a rune and a boolean.
// Types are inferred from literal values.
var message1 = "Normal string literal"
var message2 = `Raw string literal`
var letter = 'a'
var isOk = true

// You can override the inference to force a specific type:
var counter int64 = 5
```

Type conversion (or type casting) is possible by prefixing the value with the destination type and wrapping it using parenthesis, such as `int64(0)`. Unlike many languages, Go does not implicitly convert types: **all conversions must be explicit**. For example:

```
// Declare a variable as int64 and set to 5.
var counter = int64(5)

// Declare a uint variable.
var length uint = 10

// Use type conversion to align types for subtraction.
counter = counter + int64(length)
```

> From the last two examples, you see that both `var counter = int64(5)` and `var counter int64 = 5` declare a `counter` variable of type `int64` and value 5. Even if the result is the same, they are semantically different:
>
> - `var counter = int64(5)` uses type conversion to convert 5 (`int` literal) to `int64`, and then inference rules set `counter` to be an `int64` from the assignment;
> - `var counter int64 = 5` forces the inference to consider 5 as `int64` instead of `int` (no type conversion).

Go allow multiple declarations of variables in the same line when they are of the same type. Also, in

addition to the `var` keyword, Go allows declaring variables with `:=` in place of the assignment symbol `=` **the first time** the variable is encountered (in this case, the type is always inferred). For example:

```
// Declare two counters of type int.
var counter1, counter2 int

// Declare one variable by assigning a value.
// Type is inferred.
participants := 100

// Declare `result` and `err` variables here.
// Their type is inferred from the function signature.
result, err := function()
```

The same native types can be used in constants. The syntax for declaring constants is similar to the syntax for declaring variables explicitly, with `const` keyword in place of `var`. For example:

```
const Pi = 3.14
```

5.1.3.1 Pointers

Pointers are variables that store the memory address of other values. They allow you to access and manipulate data stored in memory indirectly. The reason for having such a type is that sometimes you need to access and modify the original value of a variable. We will see this concept later when talking about functions.

To declare and use a pointer, Go borrows the asterisk and ampersand symbols from C:

```
// Declare an integer variable.
var original int = 42

// Declare a pointer to an integer value.
var pointer *int

// Assign the address of `original` variable
// to the `pointer` variable.
pointer = &original

// The output of these prints will be 42, as
// both are referring to the same variable.
```

```
fmt.Println(original)
fmt.Println(*pointer)

// Now we change the value to zero: both lines
// will print zero.
original = 0
fmt.Println(original)
fmt.Println(*pointer)

// Note that if you print the raw value of a
// pointer, you will receive the memory
// address of the `original`, not its value.
fmt.Println(pointer)
```

Using the address in a pointer variable to access the value using the asterisk symbol is named "dereferencing a pointer".

The default value for pointers is `nil`.

Some native types, such as slices, maps, and channels, are implemented internally using pointers, so they can also assume the `nil` value.

5.1.3.2 Arrays and slices

Go natively supports two data structures for an ordered list of elements: *arrays* and *slices*. Both types can store elements of the same type. There is only one important difference: arrays have their size defined at creation time and cannot be extended, while slices can be extended.

Arrays are defined by using square brackets to hold the array size, followed by the type of elements. The size of the array will be the one declared when the array is created, and no further changes are allowed. For example, an array of four integers is declared using `[4]int`. You can specify elements during the array creation using curly brackets; if you don't, the type's default value will be used instead.

```
// Declare an array of four integers.
// Note that the length of this array is 4.
// Elements have zero value.
var grades [4]int

// Declare a variable that contains the
// size of the array (in this case, four).
```

```
var homeworks = len(grades)

// Print the 3rd item on screen.
// The printed value will be zero, as the
// third item has been declared but not
// populated.
fmt.Println(grades[2])

// Declare an array of four integers, and
// initialize it with four integers.
var grades = [4]int{18, 20, 30, 25}

// Print the 3rd item on screen.
// The printed value will be 30.
fmt.Println(grades[2])
```

Slices are defined similarly, but they are more complex. First, a slice has two properties: *length* and *capacity*. The capacity of a slice is the number of elements a slice can hold before resizing; the length is the number of elements currently in the slice. To add a new element in the slice, we use `append()`.

To grasp the difference between *capacity* and *length*, let's see how slices are implemented. A slice is an abstraction over an array (in other words, it *contains* an array). But arrays cannot be resized: how do we append new elements in a slice if an array backs it? When adding a new element, we might *replace* the internal array with a new one large enough to contain all the current items plus the new item.

However, this process is slow, and appending elements in a slice is a very common action. In order to optimize the append, the Go runtime allocates a larger array using some heuristic: a slice of four items (plus one that is going to be added) might become an array of eight items (but only five are really "used"). Here is the difference: the *capacity* is the size of the underlying array, while the *length* is the number of items actually used in the array.

> `append()` accepts a slice as the first argument and returns the same or new slice. A common bug is to write the result of `append()` in a different slice variable. **You should always overwrite the same variable**, i.e., `slice = append(slice, elementToAdd)`, because `append()` may need to increase the slice size, allocating new memory.

You can control the initial capacity of a slice using `make()`. The function accepts three arguments: the type of the slice, the initial length (values will assume the default value), and the capacity.

Example of arrays and slices declaration:

```
// Declare a slice of integers.
// The variable has a nil value until
// some item is appended.
var grades []int
// len(grades) == 0
// cap(grades) == 0

grades = append(grades, 30)
// len(grades) == 1
// cap(grades) >= 1 (depends on the Go runtime)

// Declare a slice of four integers,
// and initialize it with four values.
var grades = []int{18, 20, 30, 25}
// len(grades) == 4
// cap(grades) >= 4 (depends on the Go runtime)

// You can control the capacity
// using make().
// Allocate a slice with zero elements
// and a capacity of four.
var grades = make([]int, 0, 4)
// len(grades) == 0
// cap(grades) == 4

grades = append(grades, 30)
// len(grades) == 1
// cap(grades) >= 4 (depends on the Go runtime)
```

Note that slices may assume a `nil` value (for example, when declared using `var grades []int`). `nil` slices are similar to empty slices: their length and capacity are zero, and `for` loops will gently execute zero iterations. But there are some differences: the most notable one is that an empty slice is not equal to `nil`, while a `nil` slice is equal to `nil`.

```
var gradesNil []int
// gradesNil == nil is true

var gradesEmpty = make([]int, 0)
// gradesEmpty == nil is false

// len(gradesNil) == len(gradesEmpty) == 0
```

```
// cap(gradesNil) == cap(gradesEmpty) == 0
```

5.1.3.3 Maps

Another built-in type in Go is the `map`. Map variables can store values indexed by a key. The key must be a type that is comparable[^gocomparable], while values can be of any type. In some programming languages, this structure is called a "dictionary".

Unlike slices, maps must be allocated using the `make()` function. If not, the variable contains a `nil` value.

Example: a map that contains integers indexed by strings:

```
// The variable is declared but not allocated
// (it contains nil).
var dictionary map[string]int

// assign a value here will result in a crash!

// The variable is declared and allocated.
var dictionary = make(map[string]int)

dictionary["John"] = 42
```

Maps can also be initialized using literals in place of `make()`:

```
var temperatures = map[string]int{
  "Rome":   27,
  "London": 18,
  "Madrid": 21,
}
```

You can use the `delete()` function to delete a key in a map. It accepts the maps as the first parameter and the key you want to delete as the second. For example:

```
delete(temperatures, "Rome")
```

To read a value from a dictionary, you can use the square bracket (e.g., `elem := temperatures["London"]`). However, if the key does not exist in the map, Go will crash. To check whether

a key exists, you can use an extended form that returns an additional boolean which indicates if the value exists.

Example:

```
// Copy the value for the "London" key inside
// the `elem` variable.
// Note that if "London" does not exist in
// the dictionary, the program will crash.
elem := temperatures["London"]

// Copy the value for the "London" key inside
// the `elem` variable, if any.
// Set the boolean `ok` variable to `true`
// if the value exists, `false` otherwise.
elem, ok := temperatures["London"]
```

5.1.3.4 Custom types and structs

In addition to built-in types, you can define your own custom types. They can be public or private, although public variables should use public types, and public functions should return public types; otherwise, the caller might be unable to use the returned value.

To define a new type, we use the `type` keyword, followed by the type definition. We can define the custom type from built-in types or other custom types. The initialization and assignment of variables depend on the definition: in custom types defined using a built-in type only, you can assign/read the value using the type conversion. In complex custom types, you may require a different interaction.

Example: define the Age type as an unsigned integer, and define a new variable with a value of type Age.

```
type Age uint

var myvar Age
myvar = Age(25)
```

Built-in types that require `make()` or an inline assignment requires that the custom type follows the same rules.

Example: define the Phonebook type as a map between strings (names) and strings (phone numbers).

```
type Phonebook map[string]string

var pb = make(Phonebook)
pb["John Doe"] = "0039562354234"
fmt.Println(pb["John Doe"])
```

Sometimes we need a complex type: in the example above, we defined Phonebook as a map between names and phone numbers. What if we want to store more structured information? We can use the `struct`.

A `struct` is a collection of variables. Variables in a struct are named "fields" and can be of different types. You can define structs using the `struct` keyword and a code block that declares its fields. The definition of a new struct creates a type, so we need to use the `type` keyword.

Example: define the Phonebook with more fields per person.

```
type PhonebookEntry struct {
  HomeAddress string
  HomePhone   string
  MobilePhone string
}
type Phonebook map[string]PhonebookEntry

var bp = Phonebook{}
bp["John Doe"] = PhonebookEntry{
  HomeAddress: "Brooklyn Avenue, 15",
  HomePhone:   "0039562354234",
  // Unspecified fields are set to the default value.
}

// We can update the value later.
bp["John Doe"].MobilePhone = "0039562354234"

// And we can print a field.
fmt.Println(pb["John Doe"].HomeAddress)
```

Go supports composition in structs, allowing multiple structures to be merged together:

```
package main

import "fmt"
```

```
func main() {
  type First struct {
    A string
  }
  type Second struct {
    B string
  }
  type Third struct {
    // You can have an arbitrary number
    // of types here.
    First
    Second

    C string
  }

  // When assigning a value, we need
  // to specify each type individually,
  // as if they were fields in the
  // struct.
  var myvar = Third{
    First: First{A: "A"},
    C:     "C",
  }

  // We can access fields from any
  // type (in this case, `A` is from
  // `First`) directly, or explicitly.
  fmt.Println(myvar.A)
  fmt.Println(myvar.First.A)
}
```

5.1.3.5 Notes on strings

Strings in Go are declared using the `string` type. Differently from other languages (such as C), `string` is a *native* Go type. Example:

```
var text string
text = "Hello"
text = text + `World`
```

String literals are delimited either by double quotes " or backtick `. A double-quoted string literal admits escape sequences (such as `\n` for new line, `\t` for tab character, etc.); a backtick quoted string does not.

They are **immutable**: you cannot change a single character of a string (or append other characters) without allocating a new string. Even if this behavior is not explicit by the developer (meaning that you can still write something like `a = "b" + "c"` to concatenate two strings), the effect is that **Go allocates a new variable each time**. To avoid this problem when building a string from multiple strings, use `strings.Builder`.

A common misconception is that Go strings are always UTF-8: they are not. Strings are lists of bytes, regardless of their encoding. In other words, a string in Go can hold whatever set of bytes you like: not only different encodings (like UTF-16 or ISO-8859-1) but also arbitrary bytes! **Go does not enforce a particular encoding on string variables**.

> Text *encoding* is a technique to represent characters on a computer system. Computers work with bits; they don't have the concept of "character" or "number". The *encoding* indicates which combination of bits represents a specific character.
>
> For example, ASCII is an old (but still used) encoding scheme. In ASCII, the a character is represented by the value 97, encoded in binary as `01100001`.
>
> The most common encoding scheme on the web nowadays is UTF-8, which is part of a family named "Unicode". Go supports all Unicode character sets using the native `rune` type.

There are two exceptions to the rule above:

- The source code of your Go application must be encoded in UTF-8;
  - so, string literals with **no** escape sequences are UTF-8;
- When looping over a string using a `for range` loop, the value for each iteration is a `rune` representing a Unicode code point, not a sigle byte;
  - however, looping over a string using `for` and an index accesses its bytes directly.

Remember: a `rune` is larger than a `byte` (because Unicode code points are up to 32-bit long), and a string is a list of `bytes`! Due to this difference, **the number of characters in a string may be less than the string size in memory**. For example, `utf8.RuneCountInString("süß")` is 3 (we have three Unicode characters, `s`, ü, and ß), while `len("süß")` (the length in bytes) is 5, as both ü and ß are represented using two bytes each, while `s` uses only one byte.

> Flaws in handling UTF-8 may result in all kinds of bugs. Do not underestimate the difference between the length and the number of characters in a string!

5.1.4 Flow control

Just like other programming languages, Go has flow control mechanisms, such as `if`, `for`, and `switch`. They works similarly to other languages, with some "syntax sugar" for the `if` and `switch`.

Example: We can use `if` to check if the current second is even:

```
package main

import (
  "fmt"
  "time"
)

func main() {
  var p = time.Now().Second()
  if p % 2 == 0 {
    fmt.Println(p, "is even")
  } else {
    fmt.Println(p, "is odd")
  }
}
```

`if` supports inlining one statement. If used, variables declared in that statements will have the scope of the `if` blocks (i.e., you can only use the variable inside `if` blocks).

Example: This is the same example as above rewritten to use inline statement and the `:=` shortcut for assignment and variable declaration:

```
package main

import (
  "fmt"
  "time"
)

func main() {
```

```
  if p := time.Now().Second(); p % 2 == 0 {
    fmt.Println(p, "is even")
  } else {
    fmt.Println(p, "is odd")
  }
}
```

The only iterator available in Go is `for`. It has three main syntaxes, two of which may be familiar to you from other languages:

- `for boolean-condition {}`: loops until the boolean-condition evaluates to `false`;
- `for initialization-statement; boolean-condition; update-statement {}`: executes `initialization-statement` once, then loops until `boolean-condition` evaluates to false, executing `update-statement` at the end of each loop (before the next iteration);
- `for a, b := range c {}`: loops over the slice, array, map or string named c;
  - in slices, array and strings, a will be the index and b the value;
  - in maps, a will be the key and b the value;
  - a shorter form is `for a := range c {}`, where a will be the index.

Example: What if we want to generate an even random number?

```
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func main() {
  var found = false
  for !found {
    var p = rand.Int()
    if p % 2 == 0 {
      fmt.Println(p, "is even")
      found = true
    } else {
      fmt.Println(p, "is odd")
    }
```

```
  }
}
```

We use the `math/rand` Go package to pick a random number.

> `math/rand` is not "cryptographically secure". It means that you must use it for non-sensitive operations. If you need a random number for cryptography or sensitive operations, see the `crypto/rand`.

> Note: before Go 1.20, the *pseudo-random number generator* inside `math/rand` was not seeded automatically, and you were supposed to use the `rand.Seed()` function to initialize it before use. Since Go 1.20, there is no need to use `rand.Seed()` anymore, as the PRNG is automatically seeded.

The `switch` statement in Go is similar to the `switch` statement in other languages, with some exception. In many programming languages, once a case is matched, the code is executed until a `break` is found, or until the end of the `switch` statement. If a `break` is not present, all the code in subsequent cases from the case that matched are executed. In Go, the execution terminates when the next case starts (or until the end of the `switch` statement).

Example: What if we want to print a different message depending on the random number?

```
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  var floor = rand.Intn(5)
  switch floor {
  case 0:
    fmt.Println("Ground floor")
    // Note that, differently from other languages,
    // the `break` keyword is not necessary, as the
    // program won't continue the execution of other
    // cases automatically.
  case 1:
    fmt.Println("First floor")
```

```
  case 2:
    fmt.Println("Second floor")
  default:
    fmt.Println("The building is too high for me")
  }
}
```

If we want to continue the execution for subsequent cases, we can use the `fallthrough` keyword.

Another peculiarity of Go `switch` statement is the ability to use conditions on the `case` itself, potentially matching different variables, or building more complex boolean statements.

Example: A switch statement that uses conditions in `case` matches, and uses `fallthrough` to continue the execution:

```
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  var floor = rand.Intn(5)
  switch {
  case floor == 0:
    fmt.Println("Ground floor")
  case floor > 1:
    fmt.Println("Upper floors")
    fallthrough // continue in the next case
  case floor > 3:
    fmt.Println("The building is too high for me")
  }
}
```

5.1.5 Functions

Functions are first-class citizens in Go. You can define functions using the `func` keyword, followed by the function signature. The signature comprises the function name, parameters (enclosed in parenthesis), and return value or return values.

Example: a simple function that prints `Hello world!`:

```
func printHelloWorld() {
  fmt.Println("Hello world!")
}
```

Parameters in the function signature are defined using the parameter name, followed by the type. They are passed by value, meaning that the value is copied into the new variable defined in the function parameters.

```
func printMessage(message string) {
  fmt.Println(message)
  // The next statement does nothing,
  // as the variable has been created
  // at the beginning of the function,
  // and it is destroyed at the end of it.
  message = ""
}

func main() {
  var message = "Hello world!"
  printMessage(message)
  // Here `message` contains `Hello world!`
}
```

Note that if you pass a pointer or a structure that is a pointer (e.g., maps), you are copying the *pointer variable* in the function. The original value won't be copied. So, the called function can modify the variable's value in the caller. For example:

```
func printMessage(message *string) {
  fmt.Println(*message)
  // The next statement sets the variable
  // on the caller to an empty value,
  // as the variable is a pointer.
  *message = ""
}

func main() {
  var message = "Hello world!"
  printMessage(&message)
```

```
  // Here `message` is empty (due to
  // the last statement in the function).
}
```

Functions may return one or more values using the `return` statement. There are two ways to define return values: un-named and named returns. The former is more straightforward: the function signature establishes a list of return values, and Go expects values in the same order at each `return`:

```
func next(value int) int {
  return value+1
}

func integerDivision(dividend int, divisor int) (int, int) {
  remainder := dividend % divisor
  quotient := (dividend - remainder) / divisor
  return quotient, remainder
}

func main() {
  var i = 5
  i = next(i)

  q, r := integerDivision(5, 2)
  fmt.Println("Quotient:", q)
  fmt.Println("Remainder:", r)
}
```

Another way to handle the return values is "named returns". In this case, we define a name for each return value; Go creates a variable for each return value, and all these variables are automatically returned when a "naked `return`" is encountered.

> A "naked `return`" is a return with no return values. It is used in named returns because Go already knows which variables must be returned to the caller, and in functions that return no value when we want to terminate the function call before the end.

```
func integerDivision(dividend int, divisor int) (quotient int, remainder
↪  int) {
  remainder = dividend % divisor
  quotient = (dividend - remainder) / divisor
  return
```

```
}

func main() {
  q, r := integerDivision(5, 2)
  fmt.Println("Quotient:", q)
  fmt.Println("Remainder:", r)
}
```

If we are not interested in some values of a function that returns multiple values, we can use the underscore to indicate that we intend to ignore that value:

```
func main() {
  q, _ := integerDivision(5, 2)
  fmt.Println(q)
}
```

Functions can be assigned to variables; their signature (without the name) is a type. For example:

```
// Defines a new type. Variables of this type
// must contains a function with the specified
// signature (one string parameter, one bool
// return parameter)
type printFunction func(string) bool

func main() {
  var printFn printFunction

  // Assign an "anonymous function" to the
  // variable.
  printFn = func(message string) bool {
    fmt.Println(message)
    return true
  }

  // Call the function.
  var b = printFn("Hello world!")
  fmt.Println(b)
}
```

Function types are identical to other types. For example, you can have a function type as a parameter to a function:

```
type printFunction func(string)

func printHelloWorld(printFn printFunction) {
  printFn("Hello world")
}

func main() {
  var printFn printFunction = func(message string) {
    fmt.Println(message)
  }

  printHelloWorld(printFn)
}
```

Finally, a special kind of function is named "*method*". A *method* in Go is a function associated with a custom type or a struct. In the definition, their signature includes a *receiver*, the variable over which they operate. For example:

```
type PhonebookItem struct {
  HomeAddress string
}

// HasHomeAddress returns true if the item
// has a HomeAddress, false otherwise.
func (item PhonebookItem) HasHomeAddress() bool {
  return item.HomeAddress != ""
}
```

The function signature is still `HasHomeAddress() bool`, but the receiver `item` of type `PhonebookItem` is added in the implementation. The variable can be used just like other variables. If defined as a pointer, their values can be changed too:

```
type Age uint

func (age *Age) Increment() {
  *age++
}
```

5.1.6 Interfaces

Interfaces are a fundamental concept that enables abstraction in Go. An interface defines a set of method signatures but doesn't provide the implementations for those methods. Instead, it serves as a contract that specifies what behavior a concrete type must implement. This allows different types to satisfy the same interface as long as they provide the required methods.

> Interfaces are building blocks for API within the same Go application!

Go interfaces are defined using the interface keyword, followed by a block of method declarations. For example:

```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

In this example, the Shape interface specifies that any type that wants to satisfy this interface must have two methods: `Area()` and `Perimeter()`, which return floating-point values.

A type automatically satisfies an interface if it implements all the methods declared in the interface. There is no need to declare that a type implements an interface explicitly. This approach contrasts with other languages where interfaces must be explicitly stated in the class or struct definition. Using this technique, we can define interfaces for already present types (even in external packages or libraries) with no changes to their code.

Example: we can define a custom type, `Circle`, which implements the Shape interface. We only need to define the two functions indicated in the interface:

```
package main

import (
  "fmt"
  "math"
)

type Shape interface {
  Area() float64
  Perimeter() float64
}
```

```
type Circle struct {
  Radius float64
}

func (c Circle) Area() float64 {
  return math.Pi * c.Radius * c.Radius
}

func (c Circle) Perimeter() float64 {
  return 2 * math.Pi * c.Radius
}

func main() {
  var s Shape

  s = Circle{Radius: 5}

  fmt.Println("Area:", s.Area())
  fmt.Println("Perimeter:", s.Perimeter())
}
```

> A best practice for Go interfaces is to have small interfaces. Interfaces can use the composition in the same way as `struct` does.

As interfaces are types, they can also be used in function signatures, allowing the developer to abstract and provide reusable functions and methods. A best practice is to use the smallest interface we need.

Example: a function that needs to write "Hello world!" in the standard output may accept `io.Writer`, as the function needs only the `Write()` method, defined in the `io.Writer`.

```
package main

import (
  "fmt"
  "io"
  "os"
)

// printHelloWorld accepts anything that
// implements `io.Writer`, such as a file,
```

```go
// the standard output, a network stream.
func printHelloWorld(out io.Writer) {
  _, _ = fmt.Fprintln(out, "Hello world!")
}

func main() {
  // `os.Stdout` has the `Write()` method
  // with the same signature as the method
  // in the `io.Writer` interface. So, we
  // can use `os.Stdout` here.
  printHelloWorld(os.Stdout)
}
```

> The Go standard library source code is a good place where you can read a Go code that respects best practices for interfaces, composition, and others: https://cs.opensource.google/go/go/+/refs/tags/go1.21.0:src/io/io.go;l=99

5.1.7 Defer

The defer keyword is used to schedule a function call to be executed **just before the function returns**. It's a mechanism for ensuring that cleanup or finalization tasks are performed regardless of how the function exits, whether due to a successful return, a crash, or an explicit return statement.

The syntax for using defer is simple: you place the defer keyword before a function **call**, and that function call will be postponed until the surrounding function completes. For example:

```go
func main() {
    defer fmt.Println("Deferred call")
    fmt.Println("Regular call")
}
// The output will be:
// Regular call
// Deferred call
```

Even though the deferred function call appears before the `Regular call` print, it's executed after that line because it runs when the function block ends. Deferring function is common when dealing with files or external connections: in both cases, you need to clean up some resources at the end of the function.

```
func main() {
  // Open a file, and close it at the end
  // of the current function.
  fp, _ := os.Open("filename.txt")
  defer fp.Close()

  fileContent := fp.Read()
  // ...
}
```

Multiple `defer` keywords are possible, and functions are called in reverse order: deferred function calls are placed on a stack, and the stack is executed in reverse order when the surrounding function returns.

> You can defer an anonymous function call as well:
>
> ```
> defer func() {
>   // ...
> }()
> ```
>
> Note the parentheses at the end of the code block. You need those parentheses because you are deferring a function *call*.

5.1.8 Errors handling

In Go, errors are treated as values and are represented by the `error` type. The `error` type is an interface with a single method, `Error() string`, which returns a string representation of the error message. They can be created using `errors.New()`, by defining a custom type that implements the `error` interface.

```
// Define an error using "My custom error" as
// error message.
var err error = errors.New("My custom error")

// Define a custom type for the error.
// This is useful when dealing with complex
// errors, and you want to give the caller
// more context.
type MyCustomError struct {
  lineno uint
```

```
  msg string
}

func (e MyCustomError) Error() string {
  return fmt.Sprintf("Error line %d: %s", e.lineno, e.msg)
}
```

> Although the `error` interface begins with a lowercase `e`, it is available everywhere because it is a built-in type.

When functions encounter errors, they can return error values to indicate that something unexpected or erroneous has occurred. It's common practice for functions to return an error as the *last* return value, following the actual result(s).

```
func divide(a, b float64) (float64, error) {
  if b == 0 {
    return 0, errors.New("division by zero")
    // Or they can use a custom type, like:
    // return 0, MyCustomError{...}
  }
  return a / b, nil
}
```

In turn, **it is expected that every error is checked by the caller immediately after it is returned**, before reading the actual results of the function. In fact, the other values returned from the function are usually invalid if an error is present unless specified otherwise.

```
v, err := divide(10, 2)
if err != nil {
  fmt.Println("Error!", err)
  return
}
fmt.Println("Result:", v)
```

> To extract some context from custom error types, see `errors.As` and `errors.Is` functions.

5.1.9 Commenting the code

Code comments in Go are supported via C-style comments (`/*` for opening a multi-line comment block and `*/` for closing it) and C++-style single-line comments (prefixed by `//`).

Usually, Go source is commented using C++-style single-line comments everywhere except before the package statement.

Comments that appear just before top-level declarations (e.g., global variables, functions, types, etc.), without any empty lines, are considered code documentation. They have a particular format, depicted in the Go documentation on "Doc Comments".

Writing comments in the code is a crucial task to lower the burden of maintaining software for an extended period. Comments help newcomers to become productive and experienced engineers to reduce their cognitive load.

The Go standard library contains several examples of comments; we suggest looking at it. Another complete guide on how to write comments is the blog post "Writing system software: code comments." from `<antirez>`.

5.2 Concurrency

Go has been designed natively for *concurrency*. It supports a lightweight version of threads, named "Go routines", and some primitives to communicate between Go routines. The `main()` function itself is executed in a Go routine.

> Concurrency is not parallelism! Although these two concepts are interconnected, they are not the same. *Concurrency* is about designing a composition of independent processes, while *parallelism* is the simultaneous execution of processes.
>
> For example, a program may have a concurrent design that allows parallel execution for some Go routines. Whether these Go routines are executed in parallel or interleaved in a single CPU and executed in series depends on the runtime environment.
>
> Rob Pike explained this difference clearly in his talk at Heroku's Wasa conference: https://go.dev/blog/waza-talk

5.2.1 Go routines

Let's introduce the Go routines with an example. Suppose that you have two statements: a function call and an assignment. If the assignment must be executed after the function call (in other words, if there is a dependency between them), then you would write something like:

```
func1()
x := y + 1
```

In the example, the assignment is executed *after* `func1()` finishes.

However, if the two statements are **independent**, you can create a concurrent execution by prefixing the function call with the `go` statement. The `go` statement spawns a new Go routine, launching the function:

```
go func1()
x := y + 1
```

Unlike the previous example, Go immediately executes the assignment once the Go routine is launched. It does not wait for `func1()` to complete.

> Remember: as stated before, the code above might execute both the function and the assignment simultaneously (*parallelism*) or not. In any case, the two statements are *concurrent*.

5.2.2 Channels

To coordinate and talk between Go routines, Go offers a feature named "channel". A "channel" is a typed FIFO queue where Go routines can write and read data of a specific type. A Go routine that writes data in a channel is also named "producer", while the Go routine that reads data from a channel is called "consumer". Multiple Go routines can read and write data in channels concurrently.

There are two ways of declaring a channel:

- Buffered channels:
  - The producer can write data until the buffer is full, then it is blocked until a consumer reads at least one value (to free the buffer);
  - The consumer can read data until the buffer is empty, and then it is blocked until a producer writes at least one value.
- Unbuffered channels:
  - The producer is always blocked until a consumer reads a value;
  - The consumer is always blocked until a producer writes a value.

> A "blocking operation" is an operation that stops the execution until something happens. For example, a write in a full buffered channel is a blocking operation as the current Go routine is stopped until the operation can be completed.
>
> Unbuffered channels have only blocking operations.

Example: A buffered channel with a buffer size of 5 is used to exchange an integer between a producer Go routine and the main Go routine (acting as consumer):

```
var valueChannel = make(chan int, 5)
go func() {
  valueChannel <- 4
}()

recvdValue := <-valueChannel
fmt.Println(recvdValue)
```

5.2.3 Select

You can use the ' select ' statement when you have multiple channels and want to receive from all of them. The `select` waits until at least one channel is ready, and the code block associated with that channel is executed (note that if multiple channels are ready, only one gets handled). For example:

```
select {
  case v := <-ch1:
    fmt.Println(v)
  case p := <-ch2:
    fmt.Println(p)
}
```

The `select` is a blocking operation unless you specify a `default` keyword. If present, the `default` code block gets executed if no channel is ready:

```
select {
  case v := <-ch1:
    fmt.Println(v)
  case p := <-ch2:
    fmt.Println(p)
  default:
    fmt.Println("No channel is ready yet")
}
```

The `default` code block makes the `select` a non-blocking operation.

5.3 Sharing and using external code using Go Modules

Go Modules provides a standardized way to manage dependencies and versioning within your projects. We will see how to use Go Modules to import and use external libraries in your Go projects.

5.3.1 Initializing a Go Module

You need to initialize a module to start using Go Modules in your project. Open a terminal window and navigate to a new empty directory. This directory will be the root directory of your project. Use the following command to initialize a new Go module:

```
$ go mod init your_module_name
```

Replace `your_module_name` with the actual name of your module. The module name should be unique. The best practice is to use the schema `domain/path` and name the package using the address of your Git repository for the module. For example, `github.com/gofrs/uuid`.

The command above creates a `go.mod` file in your project's root directory. This file serves as the manifest for your module and tracks its dependencies.

5.3.2 Adding Dependencies

To import external libraries and dependencies into your project, you should use `go get` command to download the dependency and then use `import`.

Let's say you want to import the popular `github.com/gin-gonic/gin` package into your project. You can do this by executing this `go get`:

```
$ go get github.com/gin-gonic/gin@v1.7.2
```

Optionally, you can use `latest` instead of `v1.7.2` to download the latest version of the library.

To use `gin` in our Go program, we add an `import` statement in the code:

```
import (
    "github.com/gin-gonic/gin"
)
```

5.3.3 Sharing your code as a module

To share your project as a Go module, you should save it in a Git repository and then use the same procedure we saw above for including an external module.

All packages except `main` and those inside a directory named `internal` are available to projects that include your Go module. You can use `internal` for packages that you don't want to share when publishing a Go module: source code is still shared, but other projects won't be able to use it as a package directly from your project (they can still fork your project and modify the structure).

5.4 Building web services with Go

Go standard library contains a fully fledged HTTP server that supports TLS and HTTP/1.1 and 2. You can build entire web services using the `net/http` package only! However, should you need to customize its behavior, there are different packages implementing components thanks to `net/http` Go interfaces, such as `http.Handler`.

5.4.1 A dead simple web service

Suppose we want to develop a web service that returns a number, and if the number is even or odd. We can exploit the `math/rand` package to generate a random number, and the modulo operation `%` to check the parity of the number.

```
package main

import (
  "fmt"
  "time"
  "math/rand"
  "net/http"
)

func evenRandomNumber(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "text/plain")
  var p = rand.Int()
  if p % 2 == 0 {
    fmt.Fprintf(w, "%d is even", p)
  } else {
    fmt.Fprintf(w, "%d is odd", p)
```

```
  }
}

func main() {
  http.HandleFunc("/", evenRandomNumber)

  fmt.Println("Starting web server at http://localhost:8090")
  http.ListenAndServe(":8090", nil)
}
```

In `evenRandomNumber` the `w` variable is a `ResponseWriter`: everything you write in this variable (using `.Write()` or other `ResponseWriter` functions) is sent to the client, **in the response body**. If you want to set something in the header, you need to use the `w.Header().Set()` function **before** any write in `w`.

> In HTTP, headers are sent before the body. When the header section finishes, HTTP mandates that only the body is allowed.
>
> Go does not send any header to clients until the first `w.Write()` call (which writes the body); after that call, you can set no additional headers (nor can you change existing ones) using `w.Headers()` as they have already been sent.

> The header `Content-Type` indicates the MIME type of the response body. A dead simple text type (no formatting, etc.) is `text/plain`.

Note that `http.ResponseWriter` is an interface that includes `io.Writer` (meaning that every `http.ResponseWriter` is also a valid `io.Writer`); this means that you can use `w` in any function that accepts a generic `io.Writer`, like `fmt.Fprintf()`.

> `http.ResponseWriter` can be used when `io.Writer` is requested by the function, but not the other way around!
>
> For example, `os.Stdout` is also a `io.Writer`; however, it does not contain `.Header()` (for obvious reasons: the standard output is not "speaking" HTTP). So, while `os.Stdout` and `http.ResponseWriter` are both `io.Writer`, they are not the same.

Run this program and open http://localhost:8090.

If you want to use the terminal, you can issue HTTP requests using `curl` utility:

```
$ curl "http://localhost:8090/"
```

5.4.2 I salute you! (or: Query string parameters)

If we want to receive some parameters from the user, one option is reading query string parameters:

```
package main

import (
  "fmt"
  "net/http"
)

func hi(w http.ResponseWriter, r *http.Request) {
  name := r.URL.Query().Get("name")

  w.Header().Set("Content-Type", "text/plain")
  fmt.Fprintf(w, "Hi %s!", name)
}

func main() {
  http.HandleFunc("/", hi)

  fmt.Println("Starting web server at http://localhost:8090")
  http.ListenAndServe(":8090", nil)
}
```

The query string is part of the URL, so it's stored in the `http.Request` field URL: `r.URL`. The type `url.URL` provides us a function to access the parsed query string, named `.Query()`. The returned value of `.Query()` is a `url.Values`, which contains the `.Get()` function for retrieving query string parameters from their key name (in this case, "`name`").

> Query string parameters are formatted as key-value pairs, separated by &. For example: `key1=value1&key2=value2`.
>
> The `url.Values` type handles the parsing for us (which is good!).

Run this program and open http://localhost:8090?name=John+Doe. Try to change the name in the URL and see what happens!

If you want to use the terminal, you can issue HTTP requests using `curl` utility:

```
$ curl "http://localhost:8090/?name=John+Doe"
```

5.4.3 POSTing data

What if we want to receive data using POST? Let's receive a plain text string:

```
package main

import (
  "fmt"
  "io"
  "net/http"
)

func hi(w http.ResponseWriter, r *http.Request) {
  body, _ := io.ReadAll(r.Body)

  // We assume that the body contains the raw string.
  // In a real server, you want to check this before proceeding.
  name := string(body)

  w.Header().Set("Content-Type", "text/plain")
  fmt.Fprintf(w, "Hi %s!", name)
}

func main() {
  http.HandleFunc("/", hi)

  fmt.Println("Starting web server at http://localhost:8090")
  http.ListenAndServe(":8090", nil)
}
```

The POST string is sent in the body. We read the body as `[]byte` slice and cast it to a string. Then, we write a string back to the client.

> The underscore in the 2nd return value for `io.ReadAll()` means we want to ignore that value.

> For the sake of simplicity, we assume that the POST here is a raw string, which is not always the case (you should check the `Content-Type` request header!). Also, structured data (like `application/json` or `application/x-www-form-urlencoded`) needs an intermediate parsing step first.

Browsers can send POST requests, but they need some client-side code (JavaScript or plain HTML). So, to test a POST request, we can use the `curl` utility:

```
$ curl -d 'John Doe' -H 'Content-Type: text/plain' http://localhost:8090
```

> cURL supports different options (`man curl` explain them). We use: * `-d` for sending the body, which accepts one argument (the body itself) * `-H` for sending a header; in this case, the `Content-Type`

5.5 Example: fountains

Let's implement the list of fountains REST endpoint that we defined in the previous chapter about OpenAPI. The server is very simple, it is a production-grade server (although it is very limited).

```
package main

import (
  "encoding/json"
  "fmt"
  "net/http"
  "strconv"
)

type Fountain struct {
  ID        int     `json:"id"`
  State     string  `json:"state"`
  Latitude  float64 `json:"latitude"`
  Longitude float64 `json:"longitude"`
}

func getFountainsFromDataStorage(lat, lng, filterRange float64) []Fountain {
  // Retrieve the list of fountains from somewhere
}
```

```
func listFountains(w http.ResponseWriter, r *http.Request) {
  // First, we check the HTTP method, as the default Go router
  // does not check for it.
  if r.Method != http.MethodGet {
    w.WriteHeader(http.StatusMethodNotAllowed)
    return
  }

  // Now, we should check for the query string.
  // First, we check if the required values are set.
  if !r.URL.Query().Has("latitude") {
    w.WriteHeader(http.StatusBadRequest)
    return
  }
  if !r.URL.Query().Has("longitude") {
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  var lat, lng, filterRange float64
  var err error

  lat, err = strconv.ParseFloat(r.URL.Query().Get("latitude"), 64)
  if err != nil {
    // The latitude is not a valid float.
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  lng, err = strconv.ParseFloat(r.URL.Query().Get("longitude"), 64)
  if err != nil {
    // The longitude is not a valid float.
    w.WriteHeader(http.StatusBadRequest)
    return
  }

  // Check the optional parameter "range"
  if r.URL.Query().Has("range") {
    filterRange, err = strconv.ParseFloat(r.URL.Query().Get("range"), 64)
    if err != nil {
      // The value is not a valid float, reject the request.
      w.WriteHeader(http.StatusBadRequest)
```

```
      return
    } else if filterRange < 1 || filterRange > 200 {
      // The value is out of range, reject the request.
      w.WriteHeader(http.StatusBadRequest)
      return
    }
  } else {
    filterRange = 10
  }

  // Retrieve the list of fountains
  fountains := getFountainsFromDataStorage(lat, lng, filterRange)

  // Send the list to the user.
  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(fountains)
}

func main() {
  http.HandleFunc("/fountains/", listFountains)

  fmt.Println("Starting web server at http://localhost:8090")
  http.ListenAndServe(":8090", nil)
}
```

5.6 Exercises

We have included some exercises that you can use to start with Go.

5.6.1 Help the train company to arrive on time!

Write a program that reads a JSON structure from a file representing a train track and writes an updated JSON file with calculated values.

You should calculate the following:

- The total distance
- The total travel time in minutes
- The arrival time

Input file example:

```
{
  "line": "FL7",
  "stations": [
    {
      "name": "Latina",
      "distance": 0,
      "ETA-from-previous-station": 0,
      "platform": "1T",
    },
    {
      "name": "Cisterna di Latina",
      "distance": 18,
      "ETA-from-previous-station": 7,
      "platform": "3",
    },
    {
      "name": "Campoleone",
      "distance": 26,
      "ETA-from-previous-station": 10,
      "platform": "4",
    },
    {
      "name": "Pomezia - S. Palomba",
      "distance": 19,
      "ETA-from-previous-station": 7,
      "platform": "2",
    },
    {
      "name": "Torricola",
      "distance": 21,
      "ETA-from-previous-station": 8,
      "platform": "2",
    },
    {
      "name": "Roma Termini",
      "distance": 32,
      "ETA-from-previous-station": 12,
      "platform": "16",
    },
  ],
```

```
  "departingTime": "2022-11-02T09:00:00Z"
}
```

The `distance` is expressed in km, and the `ETA-from-previous-station` is the number of minutes to reach the station from the previous one.

> Hint: use `time.Time` for `departingTime`, so that Go will parse the timestamp for you!

An example of the output is:

```
{
  "totalDistance": 116,
  "travelTime": 44,
  "arrivalTime": "2022-11-02T09:44:00Z"
}
```

> Hint: use `struct` for reading/writing JSON. See `encoding/json` for details.

5.6.2 Calculate your GPA

Write a program to calculate the Grade point average (GPA) of a given set of scores. The GPA for Computer Science courses in Sapienza is calculated as follows:

- The average algorithm is the "Weighted arithmetic mean". For each grade to sum, the weight is the ratio between the CFU/ECTS for that grade and the total number of CFU/ECTS.
- The program should ask the user for grades. The number of grades is not defined. You can assume that no one will take more than 40 exams, but you should not require the user to input 40 values if the user did fewer exams.
- Grades in Italy are a natural number within the range `[0, 30]`, but you should ignore grades from zero to seventeen as they don't count in the GPA.
- CFU/ECTS for each grade are a natural number in the range `[1, 40]`.
- A score over 30 is referred to as "lode" or "*cum laude*" (Latin). For the GPA, scores over 30 are counted as 30.
- You can use some values for special meaning – e.g., zero score means "done inserting scores"

Examples (grade/CFU):

- Alice: 30/6 30/9 25/6 22/12 26/6 28/6 = 26.4
- Bob: 18/6 20/6 30/12 30/9 24/6 18/6 29/12 28/9 = 25.91

> Hint: see `bufio.NewReader()` and its `ReadString`. You need to pass them the reference to the standard input: in Go, operating system-related functions/variables are inside the `os` package.

> Hint: `strconv` package might be useful.

5.6.3 Word count

Write a program that opens and read a file (the name can be hard-coded inside the source code) and counts all occurrences for every different word. The output should be the list of words and the number of times the word is present in the text (one per line).

To test with a long text, you can download The Project Gutenberg eBook of Alice's Adventures in Wonderland, by Lewis Carroll (link).

5.6.4 Web-train

In this exercise, you need to create a web server that implements the exercise "Help the train company to arrive in time!" via HTTP. JSON should be received via an HTTP POST request (in the body), and the response JSON should be sent in the body of the HTTP response.

5.6.5 Web-GPA

This exercise is identical to the "Calculate your GPA", except that input data must be sent as JSON via an HTTP POST request to the server, and the response should be provided as plain text.

5.6.6 Web-count

This exercise is identical to the "Word count", except that input data must be sent as plain text via an HTTP POST request to the server, and the response should be provided as plain text.

5.6.7 Further readings

Go tour (https://go.dev/tour/list) is an excellent place to start. For a complete view of Go features and functions, look at the official Go documentation(https://go.dev/doc/) or its source code(https://github.com/golang/go/tree/master/src).

An example of a structure for RESTful backends is available at https://github.com/sapienzaapps/fantastic-coffee-decaffeinated.

6 Web Front-end Programming with Vue.js

6.1 HTML

HyperText Markup Language (HTML) is a standard markup language used to describe web pages. It was first released in 1991, and we now use version 5.2.

HTML is a description language and not a procedural language. It means that an HTML file describes what a page is composed of, and then some other program (i.e., a browser) needs to parse this description and execute the proper actions to render it on the screen. The HTML description, in particular, does not enter into rendering details. However, it addresses the essential structures of the pages, leaving to other languages (e.g. CSS) or to the browser defaults to take the most appropriate rendering decisions.

For example, the .html file can describe several elements composing the page, such as a title, a block of text with its content, or an image. Each element is known as a tag, described by its name and content. Most tags require an opening and a closing, as in the code below:

```
<tagname>
  content
</tagname>
```

The angular braces delimit tag names, and the backslash indicates the closing tag. Tags can be nested but cannot be interleaved.

HTML pages are described inside the tag named `<html></html>` and is composed of two parts: the `<head></head>` and the `<body></body>`. The code below represents a file with a title and two paragraphs.

```
<html>
  <head>
    <title>This is the title of my page</title>
  </head>
  <body>
```

```
    <h1>This is a heafing</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
  </body>
</html>
```

The following example shows a page composed of a title and a couple of paragraphs. Though the second paragraph is on three lines in the code, the browser will represent it according to the window size; thus, it can fit just one line or need more. It is up to the browser to position the elements on the screen.

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Page Title</title>
    </head>
  <body>
    <h1>Heading</h1>
    <p>A paragraph.</p>
    <p>A
      longer
      paragraph.
    </p>
  </body>
</html>
```

This file can be rendered as:

Figure 6.1: A very simple page

6.1.1 Main HTML elements

In the following we report a list of the main tags: the document's headings and subheadings are named `<h1>` to `<h6>`; `<div>` and `<span>` are used to create page sections; `<ul>` and `<ol>` describe unordered lists (bullet points) and ordered lists (numbered) respectively; `<li>` is used for each list element (for both list types); `<table>` creates a table and contains one or more rows (`<tr>`) which in turn contain one or more columns (`<td>`, or `<th>` for the heading row).

- `<h1></h1>..<h6></h6>` headings
- `<p></p>` paragraph
- `<br>` line break (no end tag)
- `<a></a>` link (aka hyperlink)
- `<img>` image (no end tag)
- `<div></div>` section
- `<span></span>` section
- `<iframe></>`
- `<ul></ul>` unordered list
- `<ol></ol>` ordered list
- `<li></li>` list item
- `<table></table>`
- `<tr></tr>` table row
- `<td></td>` row element
- `<th></th>` heading element

The `<a>` tag refers to the hyperlink, i.e., it describes a portion of text that points to another webpage or document. This tag contains the text to be displayed and the `href` attribute specifying the destination URL.

```
<a href="http://gamificationlab.uniroma1.it/wasa">
Our course
</a>
```

6.1.2 Attributes

Attributes like `href`:

- provide additional information about elements
- are always specified in the start tag
- are name/value pairs like name="value"

The image tag `<img>` has no content nor a closing tag. All the information is contained in the attributes:

- `src` specifies the path to the image
- `alt` specifies an alternate text
- `width`: specifies a size, which can also be relative to the container the image is in (e.g., `width="50%"`)

For example:

```
<img
src="logo.png"
alt="This website's logo"
width="250px"
>
```

The browser will read the HTML and look for the resource indicated in the `src` attribute. It can be displayed when it is served from the server (by default, the same server and path as the current page). The `alt` attribute contains some alternate text that can be shown for accessibility, while `width` will force the image to resize to the indicated width.

6.1.3 Nesting elements

Tables are composed nesting the different tags as in this simple example:

```
<table>
 <tr>
   <th>Room</th>
   <th>Places</th>
 </tr>
 <tr>
   <td>1L</td>
   <td>120</td>
 </tr>
 <tr>
   <td>2L</td>
   <td>96</td>
 </tr>
</table>
```

Each `<tr>` is a row, from the top to the bottom of the table. Each `<th>` or `<td>` is an element of that row. Elements are placed in the table columns from left to right. This example can be rendered as follows:

Room	Places
1L	120
2L	96

Figure 6.2: A simple table

In lists, `<li>` elements are nested within `<ul>` or `<ol>` elements. The following code will produce the two lists in the image below:

```
<ul>
<li> Alice </li>
<li> Bob </li>
</ul>

<ol>
<li> Alice </li>
<li> Bob </li>
</ol>
```

- Alice
- Bob

1. Alice
2. Bob

Figure 6.3: Unordered and ordered lists

Different elements can be nested, as in this example:

```
<table>
<tr><td><a
href="https://www.uniroma1.it"
>
<img
src="logo-sapienza.png"
alt="Sapienza's logo"
width="150px"
>
</a></td></tr>
</table>
```

Here, the Sapienza University logo is inside the table element and linked to the University website's URL, so the user can click on the logo image when the browser shows it.

HTML has many different elements and attributes, which we do not describe in this introduction. The reader can easily retrieve them in the documentation links at this section's end.

6.2 CSS

Cascading Style Sheets (CSS) represent the current way to format the layout of a web page. Cascading Style Sheets describe the page's appearance, defining the properties and style of all the HTML elements contained in the page.

HTML documents can include CSS in 3 ways:

- Inline - by using the style attribute inside HTML elements
- Internal - by using a `<style>` element in the `<head>` section
- External - by using a `<link>` element to link to an external CSS file

6.2.1 Inline CSS: Apply a unique style to a single HTML element

We can apply a style to an instance of HTML element describing it in the `style` attribute. This style will not impact other HTML elements, as shown in the code below:

```
<html>
  <body>
    <p style="color:red;">This is red</p>
    <p>This is not red</p>
```

```
  </body>
</html>
```

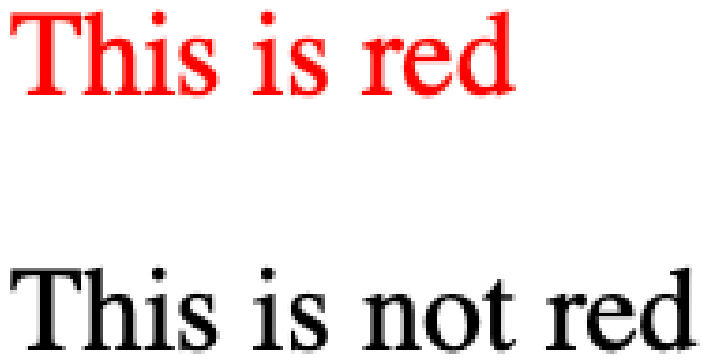


Figure 6.4: Inline CSS definition

6.2.2 Internal CSS: Define a style for a single HTML page

In the page's <head> section, we define a <style> element in which we put all the style descriptions. The rule to specify each style is very simple, e.g.:

```
h1 { color:blue; font-size:12px;}
```

It comprises a selector (h1 in this example), followed by one or more declarations separated by semi-colons, all included in braces. Each declaration is in the form property:value. The style applies to all the elements of the page that correspond to the selector. For example, all the h1 headers on the following page are red, and the background of all the paragraphs is gray:

```
<html>
  <head>
    <style>
      h1 {
        color:red;
```

```
      }
      p {
        background-color:gray;
      }
    </style>
  </head>
  <body>
    <h1>This is red</h1>
    <p>This is on gray background</p>
    <h1>This is red</h1>
  </body>
</html>
```

Figure 6.5: Internal CSS definition

6.2.3 External CSS: Define the style for many HTML pages

When the style becomes complicated, and we do not want a very long HTML <head>, we use external CSS. In this case, the browser can import a file containing the style definitions. External CSS is also helpful for applying the same style to different HTML pages that import the same file. For example, we can create a new file `style.css` containing some styles:

```
body {
  background: white;
  color: gray;
}
h1 {
  color:red;
}
```

And then we include a <link> tag in the <head> section of our page, e.g. :

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>This is red</h1>
    <p>This is gray</p>
  </body>
</html>
```

The browser will request the file from the server, thus getting all the style definitions.

Figure 6.6: External CSS definition

6.2.4 Overridding

When an element matches more rules that may conflict, there is an order that determines which will be applied.

First, an inline style overrides any other styles, e.g.:

```
<html>
  <head>
    <style>
      h1 {
        color:yellow;
      }
```

```
    </style>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 style="color:blue;">This is now blue!</h1>
    <p>This is gray</p>
  </body>
</html>
```

Figure 6.7: Inlyne style has precedence

Then, the type of selectors (see below) and the number of elements are considered. Finally, the latter definition has precedence, and external CSS styles are considered to come before any internal CSS. Thus, internal CSS has higher precedence.

For example, in the code below, the internal style overrides the corresponding external one:

```
<html>
  <head>
    <style>
      h1 {
        color:yellow;
      }
    </style>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1>This is now yellow!</h1>
    <p>This is gray</p>
  </body>
</html>
```

Figure 6.8: Last / internal declaration has precedence.

Note that a style is composed of different properties, which may be defined in separate rules or inherited by a parent element. Thus, an element can get part of its style from a CSS rule and part from another rule. For example, the h1 element below mixes the blue color defined inline and the yellow background defined internally. The inline color overrides the external red color. The yellow internal background overrides the white background of the parent element (body), which would be inherited otherwise.

```
<html>
  <head>
    <style>
      h1 {
        background-color:yellow;
      }
    </style>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 style="color:blue;">This is now blue on yellow background!</h1>
    <p>This is gray</p>
  </body>
</html>
```

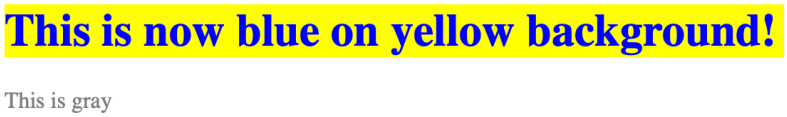


Figure 6.9: Adding style properties.

6.2.5 Selectors

Selectors, i.e., the first part of the CSS rule, are used to determine which styles apply to each element in the page. If an element matches a selector, the style is applied. For example, the `<p>Just centered</p>` element in the code below will match the selector `p` in the rule `p {text-align:center;}`. The element name matches the selector in this case. The paragraph will be centered horizontally on the page, as shown in the image below. `p` is a *simple selector*.

CSS has three types of simple selectors: element names, id selectors, and class selectors.

Id selectors refer to identifiers used in HTML elements' `id` attribute. For example, in the code below, we have a significant element that we want to identify as *hot*. To any *hot* element, we apply the style described by the rule `#hot {color:red;}`. This is why the second paragraph `<p id="hot">Centered red</p>` is rendered in red color, besides being centered. It matches both selectors: `p` and `#hot`.

Class selectors refer to the `class` HTML attribute. In our example, we define a huge font style (size 300% more than the default font) with the rule `.xxl {font-size:300%;}`, and any element containing the class attribute valued "xxl" will match.

```
<head>
  <style>
    body {background-color:white;}
    p {text-align:center;}
    #hot {color:red;}
    .xxl {font-size:300%;}
  </style>
</head>
<body>
    <p>Just centered</p>
    <p id="hot">Centered red</p>
    <p class="xxl">Centered large</p>
</body>
```

We can combine simple selectors in the same rule. There are four combinator types:

- descendant selector (space)
- child selector (>)
- adjacent sibling selector (+)
- general sibling selector (~)

The code below shows a descendant selector.

```
<head>
  <style>
    div p {color:red;}
  </style>
</head>
<body>
  <div>
    <p>Child</p>
    <table><tr><td>
      <p>Descendant, not child</p>
    </td></tr></table>
  </div>
  <p>Not in a div, adjacent sibling</p>
  <p>Not in a div, general sibling</p>
</body>
```

This descendant selector is composed of two simple selectors separated by a space and will match any element p contained in a div, either directly (a child) or not (a descendant). In our case, both the paragraphs `<p>Child</p>` and `<p>Descendant, not child</p>` will be red, as they are in a `div`. On the other hand, the last two paragraphs will not be red as they are outside the div.

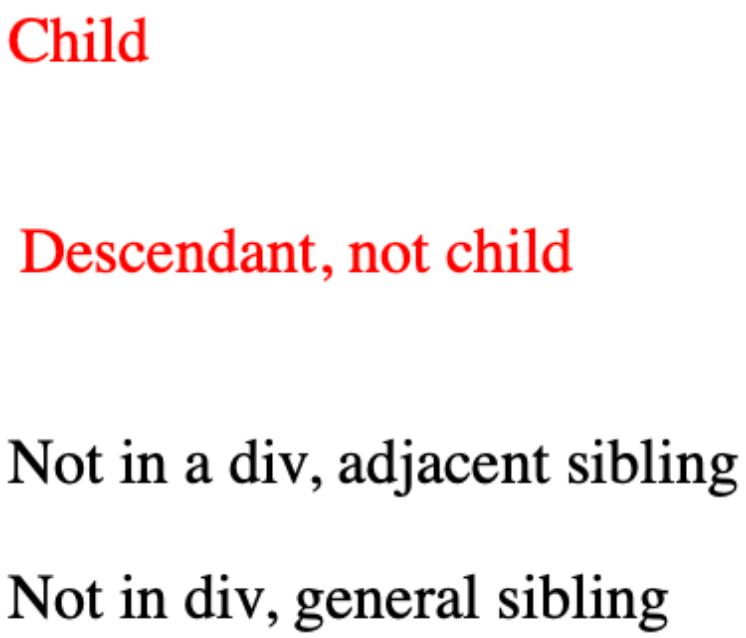


Figure 6.10: Results of descendant selector in our example

If we want to apply the style only to direct children, we can use the child selector, combining the two selectors by the symbol `>`. Thus, if we modify the rule as `div > p {color:red;}` we obtain the result shown in the picture below:

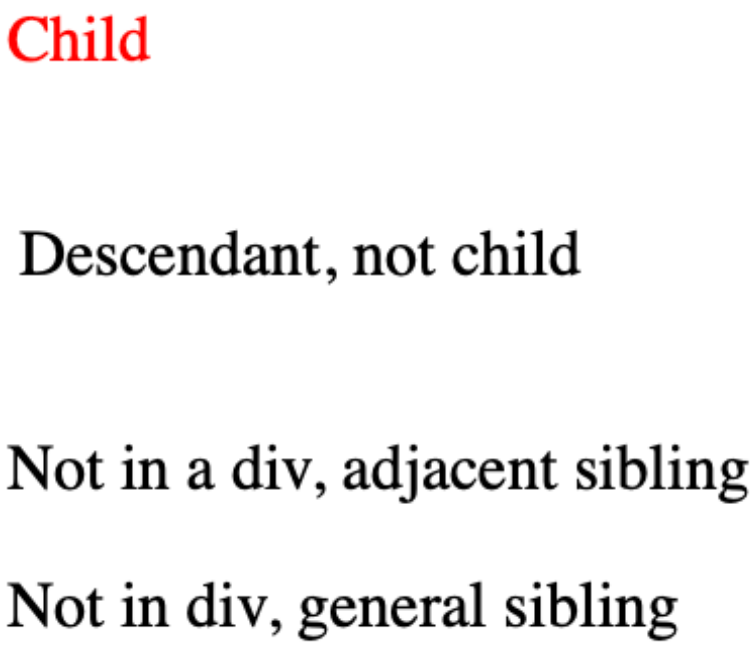


Figure 6.11: Results of child selector

The other two combinators refer to sibling elements, i.e., elements with a common parent in the DOM. The adjacent sibling selector matches only the first following sibling: if we modify the rule in our example with `div + p {color:red;}`, the paragraph `<p>Not in a div, adjacent sibling</p>` will become red, as shown in the figure:

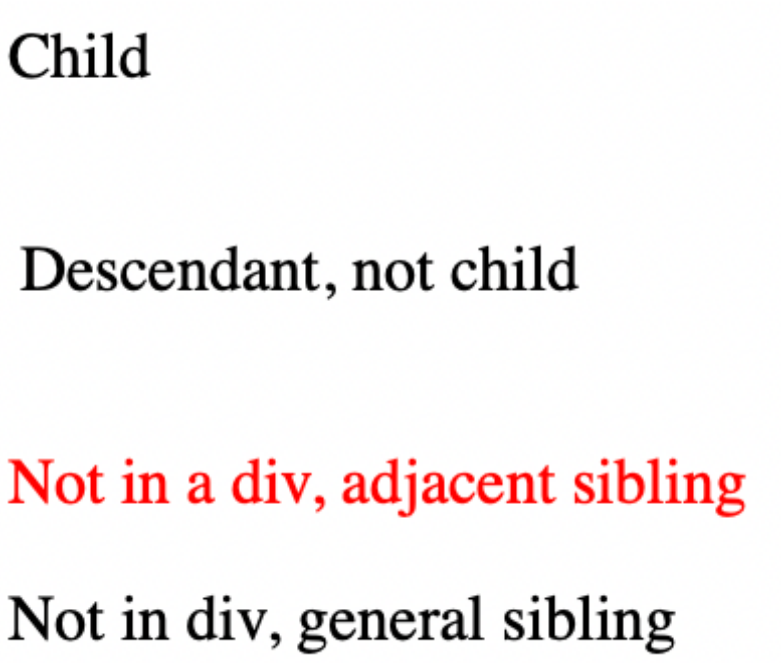


Figure 6.12: Results of adjacent sibling selector

On the other hand, if we use the last combinator type, the general sibling selector, both the paragraphs after the div will be styled in red, as this combinator matches all the subsequent siblings. To do this, we modify the rule as `div ~ p {color:red;}`, to obtain:

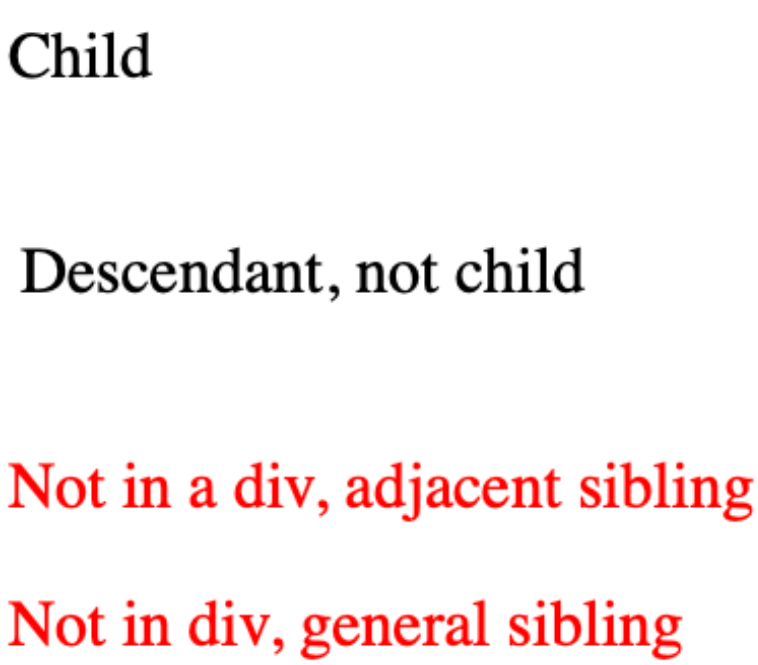


Figure 6.13: Results of general sibling selector

We can group more selectors in a rule using commas: the following will apply the style to both paragraphs inside a div and paragraphs directly following a div:

```
div p, div + p {color:red;}
```

Finally, we can use the universal selector * to match any element.

6.3 JavaScript

Javascript (JS) is a just-in-time compiled programming language broadly used on the web. We are interested in using it on the client side to perform UI operations and make requests to the backend APIs. JS has many characteristics similar to other programming languages, like object orientation, first-class functions, and dynamic typing. It also can access and modify the DOM, thus allowing the modification of the content and the presentation of the web page according to user interaction and API responses. Current browsers have a JS engine to execute the JS code contained in web pages on the user's device.

6.3.1 Types and variables

The most important Javascript primitive types are `string`, `number` (double-precision 64-bit format), `boolean`, `null`, and `undefined`. Javascript is dynamically typed, and the variables' types do not need to be specified in the declaration but are defined based on the assigned values. A variable's type can change during the program execution.

JS is weakly typed, and type conversions can occur automatically in expressions, for example, when operands of different types are used, e.g., in:

```
'123'+456
```

the right operand is a number and is automatically converted to a string, resulting in:

```
123456
```

It is possible to convert any value to a `string`, a `number`, or a `boolean`. Implicit (automatic) conversion can enhance the code readability but also cause behavior that can be unexpected for the programmer, leading to errors.

Variables can be declared with the `var` or the `let` keywords, which differ in the scope that the variable will have:

```
var a;        // function-scoped
var b = 1;
let c;        // block-scoped
let d = 3/2;  // it's 1.5
```

The value and type of a variable can change, e.g.:

```
b += 5;       // reassigned, now it's 6
d = "Hello";  // reassigned, changed type
d += " world";// now "Hello world"
d = d + 1;    // now "Hello world1"
const k = 30; // block-scoped, cannot be reassigned
x = "foo";    // deprecated assignment (global scope)
```

6.3.2 Objects

Objects are collections of properties. Property keys are strings while property values can be of any JS type. For example:

```
var course = {
  name: "WASA",
  room: "4"
}
```

We can access property values using the dot notation (console.log() prints the argument on the console):

```
console.log(course.room) // 4
```

Objects are mutable, we can modify their content:

```
course.year = 3; // now: {name: "WASA", room: "4", year: 3}
```

Object orientation in Javascript is prototype-based (this topic is outside the scope of this book and is not necessary for the following). Array, Function, Date, RegExp, and Error are Javascript Objects.

Array indices are strings converted from integer numbers. They start from 0. The bracket notation `[]` can access array elements. As an example, we define below an array of rooms whose elements contain room names.

```
const rooms = ["1L","2L","3L"] // 1L,2L,3L
rooms[0] = "1LL";          // 1LL,2L,3L
rooms[4] = "MeetingRoom"; // 1LL,2L,3L,,MeetingRoom
var r = rooms[1];          // 2L
r = rooms["1"];            // 2L
r = rooms["01"];           // undefined
r = rooms[4];              // MeetingRoom
r = rooms[3];              // undefined
```

Using the `for...in` loop, we can iterate through the array elements:

```
for (i in rooms) {
  console.log(i) // this prints 0 1 2 4
  console.log(rooms[i]) // this prints 1LL 2L 3L MeetingRoom
}
```

> Note: The `for...in` loop can enumerate the properties of an object, too. The bracket notation can be used instead of the dot notation to access properties. The code: `for (prop in course) { console.log(prop, ": ", course[prop]); }` prints: "`name: WASA room: 4 year: 3`"

Functions are objects, too. They are *first class* as they can be passed to and returned from other functions and assigned to variables. JavaScript supports anonymous functions and closures.

The following example declares a function that accepts one numeric argument and returns its double:

```
function double(x) {
  return 2 * x;
}
```

We can invoke the function using the parenthesis notation:

```
var y = double(5);   // 10
```

We can store the function object in a variable, to reuse it later:

```
var f = double;      // function(x) { return 2 * x }
var n = f(100);      // 200
```

So, `double()` refers to the function invocation, while `double` refers to the object function.

We can define a function inside another function so that the former is created each time the latter is called:

```
function createMultiplier(k) {
  function f(x) {
    return k * x;
  }
  return f;
```

```
}
var triple = createMultiplier(3);
var z = triple(10);  // 30
var tenTimes = createMultiplier(10);
var w = tenTimes(4); // 40
```

Another notation to define a function:

```
var quintuple = (x) => x * 5;
var u = quintuple(7);  // 35
```

6.3.3 Loops

In the following code, the variable `room` defined in the loop is global, and it survives outside the loop:

```
for (j in rooms) {
  console.log(rooms[j])
}
console.log(j)  // 4
```

We can limit the scope of that variable using the `let` keyword:

```
for (let j in rooms) {
  console.log(rooms[j]) // 1LL 2L 3L MeetingRoom
}
console.log(j)  // Error: Can't find variable: j
```

When we only want to loop through the values of the array, we can use the `for ... of` loop:

```
for (let room of rooms) {
  console.log(room) // 1LL 2L 3L MeetingRoom
}
```

The standard C-like `for` loop is also available:

```
for (let i = 0; i < rooms.length; i++) {
  console.log(rooms[i]) // 1LL 2L 3L undefined MeetingRoom
}
```

> Note: while the `for...in` and `for...of` loops skip any undefined element, using the standard `for` loop, we just enumerate indices; thus, the loop above prints `rooms[3]` as undefined.

Finally, the `forEach()` method of the Array object calls a function on every iteration. In the example below, we call the console.log() function four times:

```
rooms.forEach((item) => console.log(item)); // 1LL 2L 3L MeetingRoom
```

6.3.4 HTML and DOM

Javascript can modify the page content by accessing the Document Object Model. The `document` JS object represents the DOM. Each node in the DOM is a JS object. It is possible to add, remove, and modify DOM objects, and there are methods to retrieve and access objects. For example, we can retrieve a DOM object by using the id attribute of the HTML element that it represents:

```
<p id="hot"></p>
<script>
  console.log(document.getElementById("hot")); // <p id="hot"></p>
</script>
```

We can also get all the elements of a given class:

```
<h3 class="section">First</h3>
<h3 class="section">Second</h3>
<script>
  console.log(document.getElementsByClassName("section")[0]) // <h3
  ↪  class="section"></h3>
  console.log(document.getElementsByClassName("section")[1]) // <h3
  ↪  class="section"></h3>
</script>
```

Then, we can use the `innerHTML` property of the DOM objects to access the content of an element:

```
<h3 class="section">First</h3>
<h3 class="section">Second</h3>
<script>
  console.log(document.getElementsByClassName("section")[0].innerHTML) //
   ↪ First
  console.log(document.getElementsByClassName("section")[1].innerHTML) //
   ↪ Second
  document.getElementsByClassName("section")[0].innerHTML="First: New
   ↪ content";
</script>
```

The third line of the script above modifies the content of the first element, resulting in the page shown in the following figure:

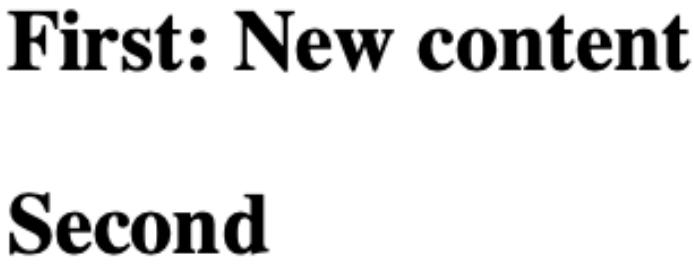


Figure 6.14: Changed element content with `innerHTML`

It is also possible to read and write HTML elements' attributes, e.g.:

```
<p id="hot">Important paragraph</p>
<script>
  document.getElementById("hot").style = "color:red";
</script>
```

Figure 6.15: The `style` attribute is set through JavaScript

Finally we can create and add a new element to the DOM:

```
<h1>WASA:</h1>
<script>
  var wasaParagraph = document.createElement("p");
  wasaParagraph.innerHTML = "Web And Software Architecture";
  document.body.appendChild(wasaParagraph);
</script>
```

The code above produces the following result:

Figure 6.16: New <p> element added to the DOM

Finally, we can remove an element from the DOM with the `document.removeChild()` method.

6.3.5 Export and Import

To achieve modularization in Javascript we can split our code in different files or *modules*. Each module can `import` variables and functions from other modules, and in turn can export (some of) the variables and functions it declares.

```
// file homework.js
export function hwAvg(grades) {
  var sum = 0;
  for (let grade of grades) { sum += grade; }
  return sum / grades.lenght;
}
```

```
<!-- file myWebApp.html -->
<p id="avg"></p>
<script type="module">
  import {hwAvg} from './homework.js';
  document.getElementById("avg").innerHTML = hwAvg([30,28,30,28])
</script>
```

6.3.6 Async-await

Operations exist that do not yield immediate results. Instead, they return a value after a certain amount of time has passed. An example is an operation involving communication with a remote server. A request is made in such cases, and a response is eventually returned. However, the time it takes to receive this response is unpredictable, depending on factors like network traffic and server load.

To manage these delayed operations, JavaScript introduces the concept of *promise*. A promise is an object representing the eventual completion or failure of an asynchronous operation. When a JavaScript function returns a promise, it indicates that its result will be available at some point in the future. The function's execution is asynchronous compared to the execution of our program, meaning it runs separately from the rest of the program, returning after the program has continued executing other tasks.

A promise can be in three distinct states:

- Pending: The initial state when the promise is in progress and has yet to be resolved or rejected.
- Fulfilled: The state when the operation completes successfully and the result is available.
- Rejected: The state when the operation fails due to an error.

We must handle the fulfilled and the rejected states to effectively use a promise. We can do this by specifying two other functions: a function that executes if the promise is successfully fulfilled (success handler) and a function that executes if an error occurs during the asynchronous operation and the result cannot be returned (error handler). We will use the `.then` method to process the result if the promise is fulfilled and the `.catch` method to handle the error if the promise is rejected.

```
var txt;
delayedOperation()
  .then((result) => {
    txt = result;
  })
  .catch((error) => {
```

```
    // Do something with the error
  });

// The following alert might execute BEFORE txt is set
alert(txt);
```

In the example above, `delayedOperation()` returns a promise. If the promise goes into the fulfilled state, the result is assigned to the `txt` variable. However, as the execution of `delayedOperation()` is asynchronous, the assignment might happen after the code execution moves to the following line `alert(txt);`. Thus, the alert can be executed before the result is available, and `txt` might not yet be assigned the value.

To simplify this challenging way of managing promises, JavaScript offers the *async/await* pattern.

The `await` keyword pauses the code execution until a promise is resolved (i.e., fulfilled or rejected).

```
...
let txt = await delayedOperation();
...
```

When the code reaches this line, it pauses until the `delayedOperation()` completes. If the promise is fulfilled, the result is stored into `txt`, otherwise an error is thrown.

The keyword `await` can only be used inside a function declared as `async`. Thus, we should include our promise inside a function that we declare as `async` , like in the example below:

```
async function buttonClicked() {
  let txt = await delayedOperation();
  alert(txt);
}
```

The assignment in the code above will actually be executed only when the promise is fulfilled: at that moment the function can countinue and the `alert()` executes, showing the value of `txt`.

The `async/await` pattern is syntactic sugar, a way to write asynchronous code using a somewhat synchronous style. Proper error handling is crucial to ensure robust applications. A `try/catch` block can be used to manage rejected state error conditions.

6.4 Introduction to Vue.js

In the realm of front-end web development, the choices for a framework are vast and diverse. One of these choices is Vue.js (created by Evan You), an open-source JavaScript framework that has garnered attention within the developer community for its pragmatic approach to building user interfaces.

Vue.js is built on standard HTML, CSS, and JavaScript. It provides a declarative and component-based programming model, and uses reactivity to update the DOM.

There are two different approaches when accessing Vue.js API: "Options API" vs. "Composition API". You can find more information about these approaches in the official documentation. We will use the "Options API" as it is simpler.

6.4.1 Vue.js applications

A Vue.js application is made of components. The first component is named the "root component", and it is the first component to be initialized. All subsequent Vue.js components are children of the root component.

The root component can be initialized like this:

```
import { createApp } from 'vue'

const app = createApp({
  /* component definition */
})
```

We will see later that, when using *Single-File Components* (SFC), we can replace the JavaScript object inside `createApp()` with the import from the SFC.

After being initialized, the application must be "mounted" by indicating which DOM element must contain the Vue.js application (in other words, the application will be confined to that element in the DOM).

```
<!-- in HTML, define a div for the Vue.js application -->
<div id="app"></div>
```

```
// in JavaScript, mount the Vue.js application inside the
// div identified by the string "app"
app.mount('#app')
```

> Note that plain JavaScript (in Vue.js) can still modify elements outside the mounting point.

This mechanism allows Vue.js to be included in existing web pages and web applications with low effort.

6.4.2 Vue.js templates

A core part of Vue.js is the template system. Vue.js templates extend the HTML syntax by adding various components, such as "mustache" syntax placeholders for variables, custom attributes for events, variable binding and conditionals, and custom elements for embedding other Vue.js components.

In templates (and in the Vue.js JavaScript code), Vue.js uses **reactivity** to notify objects when variable values change. This is done transparently, but it might cause unexpected behavior if you are not used to *reactive programming*.

To print the value of a variable in the DOM, you can use the "mustache" syntax as follows:

```
<span>Message: {{ msg }}</span>
```

The content of the `msg` variable is inserted in the DOM at that specific point. Thanks to the reactivity, the value is **automatically updated** whenever the `msg` value changes.

> Note that, for security reasons, the mustache syntax adds the value in the DOM as text – HTML markup and JavaScript code are **not** executed.

The template has no access to variables and functions outside those declared for the specific component, with some exceptions (e.g., `Math`). Consult the Vue.js code for the list of allowed globals.

> Important: in a Vue.js template there is no need for the `this` keyword to access the component variables and functions because of the limitation in global visibility. However, in JavaScript, you must use the `this` keyword.
>
> For example, accessing the `msg` variable in the template can be done using `{{ msg }}`, while in JavaScript is `this.msg`.

6.4.2.1 Add text to DOM

You can control and interact with the DOM using *directives*. Vue.js directives are custom attributes that Vue.js interprets and prepares the DOM accordingly. All Vue.js directives start with the `v-` prefix.

For example, `v-text` directive is used to provide plain text for the content of a DOM element:

```
<span v-text="variable"></span>
```

In this example, the content of the `<span>` element is filled with the content of the variable, and it is updated whenever `variable` changes. As for the mustache syntax, `v-text` does not allow markup or code. To pass raw HTML to the browser to parse, use the `v-html` syntax:

```
<span v-html="variable"></span>
```

> Warning! Do not use the `v-html` syntax on user-controlled fields. It will lead to security breaches. Use only trusted data.

6.4.2.2 Control HTML element attributes

While `v-text` and `v-html` are useful for element contents, the `v-bind` directive is used to provide values dynamically to HTML element attributes. `v-bind` requires the name of the attribute to bind to after the colon symbol `:`.

For example, the following template code updates the `id` attribute of the `<div>` element dynamically when `elementId` changes:

```
<div v-bind:id="elementId"></div>
```

`v-bind` is widely used, so Vue.js authors provide a shortcut by omitting `v-bind`. The following example is equivalent to the previous one:

```
<div :id="elementId"></div>
```

HTML elements may have boolean attributes. In this case, Vue.js acts accordingly by setting the attribute if the variable is `true`, and unsetting it if the variable is `false`. For example:

```
<button :disabled="deleteForbidden">Delete</button>
```

6.4.2.3 Show and hide DOM elements

`v-show` is a directive that is used to control the visibility of an element in the DOM. Note that an invisible element is still present in the DOM and it may still take up space in the page layout.

Example:

```
<div v-show="showWarningMessage">
    This is a warning message shown only when the
    variable `showWarningMessage` is true.
</div>
```

6.4.2.4 Add and remove DOM elements dynamically

If you want to remove completely an element from the DOM, you can use the `v-if` binding. The element is removed and destroyed when the expression is `false`, and (re-)built and added in the DOM when the expression becomes `true`.

Example:

```
<b v-if="errorPresent">
    This text is present in the DOM only when
    errorPresent is true.
</b>
```

Removed elements free their space in the page layout.

A more complex block may use `v-else` or `v-else-if` to switch between elements:

```
<div v-if="ok">
  Everything is ok!
</div>
<div v-else>
  Oops!
</div>
```

6.4.2.5 For-loop in DOM with a binding

Looping over iterable objects is possible with `v-for`. The DOM element is duplicated for each item in the loop. For example, the following template:

```
<li v-for="item in items">{{ item }}</li>
```

When executed over `['apple', 'banana', 'pear']` results in:

```
<li>apple</li>
<li>banana</li>
<li>pear</li>
```

The index of the array (or the key, if looping over a dictionary) is available as the second optional parameter:

```
<div v-for="(item, index) in items"></div>
```

6.4.2.6 Events from the DOM

The `v-on` binding is used to listen for an event regarding a DOM element. Vue.js supports listening for Native DOM events (such as `click`, `focus`, `keydown`, `scroll`, etc.) in normal HTML elements, and custom events when listening over a custom element.

Differently from other binding directives, the `v-on` directive executes a function specified as the argument when the event is fired. For example, the following code executes the function `lightsOff()` when the button is clicked:

```
<button v-on:click="lightsOff">
    Power off the lights
</button>
```

If you need to specify some function parameter, you can use the canonical form:

```
<button v-on:click="lightsOff('all')">
    Power off ALL the lights
</button>
```

Sometimes, DOM events are propagated to other handlers and elements. There are some "modifiers" to stop the propagation of events or to prevent default handlers from taking place. These are specified with a dot after the event name. For example, to stop the event propagation:

```
<button v-on:click.stop="lightsOff">
    Power off the lights and
    stop propagating.
</button>
```

This binding is widely used, so Vue.js developers added a shortcut. This is the same as above:

```
<button @click="lightsOff">
    Power off the lights
</button>
<button @click.stop="lightsOff">
    Power off the lights and
    stop propagating.
</button>
```

6.4.2.7 Additional bindings

There are additional bindings, such as `v-slot`, `v-pre`, `v-once`, and others.

The updated list is available at https://vuejs.org/api/built-in-directives.html.

6.4.2.8 Expressions inside a binding

In most cases, binding an attribute to a variable is what you need. However, sometimes, you may want to bind the value to a complex JavaScript expression. Vue.js covers that aspect too by allowing you to provide a **single** expression in all `v-` directives and in mustaches:

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div :id="`list-${id}`"></div>
```

> Note that *statements* are not *expressions*! For example, variable declaration or control flow operators are not expressions (they don't result in a value), so they cannot be used here. Functions calls are allowed (provided that they return a value).

6.4.3 Single-File Components

Using a single big page as the template is not useful nor scalable. Thanks to the components, you can split a complex page into multiple components. Vue.js supports defining components in different files that can be re-used and included in other components. These files are named "Single-File Components" (SFC).

> Note: Vue.js also supports a "build-less" mode where templates are defined "in-line". Everything is identical, except for the fact that the template is specified as a string inside the JavaScript object for the component (instead of using Single-File components).
>
> We strongly suggest building the application and making full use of SFC files.

A Single-File component contains three parts: the JavaScript code, the template, and the optional CSS part.

A SFC component for a button-counter can be defined as:

```
<script>
// Component definition (JavaScript)
export default {
  // `data` defines a list of reactive variables.
  data() {
    return {
      count: 0
    }
  }
}
</script>

<!-- template definition -->
<template>
  <button @click="count++">Count is: {{ count }}</button>
</template>

<!-- CSS style -->
<style scoped>
button {
  font-weight: bold;
}
</style>
```

The CSS style, when present, can be "scoped": if so, the CSS applies only to elements in the component.

Note how we used the reactivity here: the `{{ count }}` is updated dynamically when the `count` variable (defined in the `data` section) is updated. We update the variable in the `@click` handler.

To use this component in other components, we can import the file and use it as a custom HTML element (suppose that the file for the SFC above is named `ButtonCounter.vue`):

```
<script>
import ButtonCounter from './ButtonCounter.vue'

export default {
  // register the component
  components: {
    ButtonCounter
  }
}
</script>

<template>
  <h1>Parent element</h1>
  <p>
    This is the child element:
    <ButtonCounter />
  </p>
</template>
```

Vue.js is aware of this "embedding", and isolates the elements: if you add multiple `<ButtonCounter />` elements, each one will have its own (internal) `count` variable – they won't interfere or communicate.

6.4.3.1 Component properties

A component can accept properties from parents. We need to declare the properties in our component:

```
<script>
export default {
  props: ['title'],
  data() {
    return {
      count: 0
    }
  }
}
</script>

<template>
```

```
  <button @click="count++">{{ title }}: {{ count }}</button>
</template>
```

The property is available as a **read-only** variable in the template – but is still reactive: if the parent changes the value, the child component is updated accordingly.

To pass a property, use it as a normal attribute:

```
<script>
import ButtonCounter from './ButtonCounter.vue'

export default {
  components: {
    ButtonCounter
  },
  data() {
    return {
      titleVariable: "dynamic title"
    }
  }
}
</script>

<template>
  <h1>Parent element</h1>
  <p>
    <ButtonCounter title="Tickets" />
    <!-- you can bind the value of a property -->
    <ButtonCounter :title="titleVariable" />
  </p>
</template>
```

Components can also emit events. For example, we can emit a `like` event by using the `$emit` function each time a specific button is clicked:

```
<script>
export default {
  emits: ['like']
}
</script>
```

```
<template>
  <!-- ... -->
  <button @click="$emit('like')">Like!</button>
</template>
```

The syntax for listening to custom events is the same as native DOM events:

```
<script>
import BlogPostEntry from './BlogPostEntry.vue'

export default {
  components: {
    BlogPostEntry
  }
}
</script>

<template>
  <h1>Parent element</h1>
  <p>
    <BlogPostEntry @like="doSomethingOnLike()" />
  </p>
</template>
```

Even the root component of a Vue.js app can be a SFC. For example, to initialize a Vue.js application using a root component from a Single-File Application, use the following snippet:

```
import {createApp} from 'vue'
import App from './App.vue'

const app = createApp(App)
app.mount('#app')
```

6.4.4 Methods in components

You can declare functions inside a Single-File Component. Functions in an SFC are named "methods". A method is a JavaScript code, and it is executed then the method is called. Inside the method, you must use `this` to access the component properties, variables, and other methods.

```
<script>
export default {
  data() {
    return {
      likes: 0
    }
  },
  methods: {
    async like() {
      this.likes++
    }
  },
}
</script>

<template>
  <button @click="like"></button>
</template>
```

6.5 Routing with Vue Router

Web applications are composed of multiple views (or pages); each view is implemented by a component. To switch between them, you can use a "router" component. Vue.js supports different routes: we will use the default one, "Vue Router".

Vue Router provides two custom HTML tags in templates: `<router-link />` to create clickable links for switching pages, and `<router-view />` which is the view where page components are placed where their page is active.

To initialize the Vue Router, we need to configure the type of navigation and the list of routes. We do so using JavaScript before creating the Vue.js application:

```
import Home from './Home.vue'
import About from './About.vue'

// Create a new router. We use the Web Hash History as
// navigation strategy, and we register two paths for
// two different components.
const router = VueRouter.createRouter({
  history: VueRouter.createWebHashHistory(),
```

```
  routes: [
    // The path is arbitrary.
    { path: '/', component: Home },
    { path: '/about', component: About },
  ],
})

// Create the application (as shown in previous sections).
const app = Vue.createApp({})

// Signal that we use the router in our application.
app.use(router)

// Mount the Vue.js app.
app.mount('#app')
```

In the template itself, you can use the new two custom elements to create navigation links and the main view.

```
<h1>Hello World!</h1>
<p>
  <!-- router-link creates a clickable link for a page -->
  <router-link to="/">Home</router-link>
  <router-link to="/about">About</router-link>
</p>

<!-- the active page component is placed here -->
<router-view></router-view>
```

If we want to control the navigation programmatically, we can use the `this.$router.push('/')` function, and specify the path of the component to open.

6.6 Interacting with Backend APIs

To interact with our REST API, we can use Axios, a JavaScript library that simplifies AJAX and allows us to send HTTP requests to a server.

We can include Axios in our Vue.js project by importing it via npm: `npm install axios`.

While Axios can be used directly, we suggest you to create an instance with some pre-defined parameters, such as the backend server URL, and save it as a global Vue object. By doing so, we don't need to duplicate the configuration around in our projects.

To create an instance, we can use the following code snippet while creating our Vue.js application:

```
import {createApp, reactive} from 'vue'
import App from './App.vue'
import axios from "axios";

const instance = axios.create({
  // This is the base URL for our API.
  baseURL: "http://localhost:3000",
  timeout: 1000 * 5
});

// Initialize the application.
const app = createApp(App)
// Set the $axios global property.
app.config.globalProperties.$axios = axios;
// Mount the application in the DOM.
app.mount('#app')
```

By doing so, we can access Axios using `this.$axios` in all components without redeclaring it. For example:

```
<script>
export default {
  methods: {
    async like() {
      let response = await this.$axios.post("/like");
      // response payload is the HTTP response.data
      // JSON payloads are already decoded!
    }
  },
}
</script>

<template>
  <button @click="like"></button>
</template>
```

6.7 Example: fountains

Let's create a basic Vue.js application that retrieves the list of fountains from a REST API server. Assuming that you have Node.js (if not, you can install it by following the instructions on their website), open a terminal and launch the following command:

```
$ npm create vue@latest
```

The command launches a wizard that installs dependencies (if any) and creates a new empty Vue.js project in a directory. The directory name is the name of the project that you type in the wizard.

```
Project name: fountains
Add TypeScript? No
Add JSX Support? No
Add Vue Router for Single Page Application development? Yes
Add Pinia for state management? No
Add Vitest for Unit testing? No
Add an End-to-End Testing Solution? No
Add ESLint for code quality? No
Add Prettier for code formatting? No

Scaffolding project in ./fountains...
Done.
```

Assuming that you typed `fountains` as the project name, you should have a `fountains` directory with the code. Switch to that directory, install the project dependencies, and launch the development server:

```
$ # Switch to the `fountains` directory
$ cd fountains
$ # Install project dependencies (only the first time
$ # or if you do a git clone)
$ npm install
$ # Run the development server
$ npm run dev
```

The last command builds the project and launches a development server, which is a server optimized for developing and debugging code. The development server is available at http://localhost:5173/ : if you open it, you will see a Vue.js page that indicates that your project has been successfully created.

Now, go back to the project, and let's take a look at the file structure together. We have some files and directories; the most important ones are:

- `index.html` is the main web page of your Single-Page Application. There is a `<div id="app"></div>` here that is the mount point of your entire Vue.js application;
- `packages.json` is the configuration file used in projects that use Node.js (such as this project). It is used to define various aspects of a project, such as its metadata, dependencies, and scripts;
- `vite.config.js` is the configuration of `vite`, the Vue.js build tool;
- `public/` is a directory that contains **static files**: Vue.js will copy these files in the `dist/` directory unmodified (no parsing, no changes will be performed on these files);
- `src/` is a directory that contains the code of your application; in particular:
  - `src/App.vue` is the Single-File component that is used as the root component for the Vue.js application;
  - `src/main.js` is the core JavaScript file of the Vue.js application. The application is created inside this file. If you use the router, Axios, and other libraries, you may want to initialize them here;
  - `src/assets/` contains "assets" (files that undergo minimal changes, such as minification), commonly used for icons and CSS;
  - `src/router/` contains the router initialization and configuration (history mode, routes, etc.);
  - `src/components/` and `src/views/` contain all SFCs (except the root `App.vue`);

Note that the structure of `src/` is mostly a formality: everything can be re-arranged, provided that paths inside all relevant files are modified accordingly.

Before editing the source code, let's add Axios to our project:

```
$ npm install axios
```

Now, let's initialize Axios and store it in a Vue.js global variable. Add this code snippet after the `createApp()` function call and before the `app.mount()` inside the `src/main.js` file:

```
import axios from "axios";
app.config.globalProperties.$axios = axios.create({
  baseURL: "http://localhost:8090",
  timeout: 1000 * 5
});
```

We are now ready to use Axios. Let's modify the `src/views/HomeView.vue` and replace it with a component like this:

```
<script>
export default {
  data: function() {
    return {
      // Declare one reactive array of fountains.
      fountains: [],
    }
  },
  methods: {
    // Declare a function that sends an HTTP GET request. Axios
    async loadFountainsList() {
      try {
        let response = await this.$axios.get("/fountains/", {
          params: {
            // These values should come from the browser
            // location/GPS or some user input.
            latitude: 1,
            longitude: 2,
          }
        });
        // Axios decodes JSON automatically
        this.fountains = response.data;
      } catch (e) {
        alert("Error: " + e);
      }
    }
  }
}
</script>

<template>
  <!-- when this button is clicked, the list is downloaded -->
  <button @click="loadFountainsList">
    Load fountains
  </button>
  <!-- separator -->
  <hr />
  <p>List of fountains:</p>
  <ul>
```

```
    <!-- loop the <li> tag for each fountain -->
    <li v-for="f in fountains">
      Lat: {{ f.latitude }} -
      Lng: {{ f.longitude }}
    </li>
  </ul>
</template>
```

6.8 Build a Vue.js application for publication

Once your application is ready to be published on the web, you need to build it for public use, which means that the Vue.js project gets compiled into plain HTML, JavaScript, and CSS. In the root directory of the Vue.js project, you can launch the `npm run build` command, which creates a `dist/` directory with the compiled code for the app.

```
$ npm run build
```

If you want to preview your application after the build, you can use the "preview" command. It launches a demo web server on your local PC:

```
$ npm run preview
```

The preview will be available at http://localhost:4173.

> Some errors in the code are somehow not shown in "development mode" (i.e., `npm run dev`). Use the "production" build mode (`npm run build`) and the preview (`npm run preview`) to check if there are errors left.

6.9 Links

- https://www.w3.org/standards/webdesign/htmlcss
- https://en.wikipedia.org/wiki/HTML
- https://www.w3schools.com/html/default.asp
- https://vuejs.org/

7 Building and Deploying Containers with Docker

Once your application is ready to be published, you need to *deploy* it on a server, so that other users will be able to use it. Deploying an application includes copying the application executable and accessory files, such as program dependencies, the `dist/` directory (created by the Vue.js "production" build), or any other external files that you might need (assets, images, etc.). All these files should be placed somewhere on this server (or even in multiple servers!), and the server must be configured to execute or serve them via HTTP.

There are three ways to deploy an application: *traditional deployment*, *virtual machine deployment* and *container deployment*. We will briefly introduce them in this section while expanding the *container deployment* in the rest of the chapter.

In a traditional deployment, processes are aware of other processes, and they can interact with them using POSIX signals, shared memory, the filesystem, or platform-specific tools. The kernel, the filesystem (thus all libraries), and the hardware are shared between processes with little constraints. Sometimes there is limited supervision (restart-on-crash, process dependencies, process limits, accounting, etc.) and it is delegated to userland tooling.

While the traditional deployment strategy is still used, it has shown many limitations. For example, sharing libraries between different applications on the same server might not be easy or desiderable. Some software might not support newer version of a given library, forcing other applications to stick to older ones (or viceversa). To overcome these limitations, virtual machines (and, recently, containers) are used instead.

In deployment via virtual machines, the *host* is the operating system instance that runs on real hardware. The *guest* is the instance running inside a virtual machine (VM), which is a software construct that emulates or simulates a real machine. Machine instructions are executed or translated by the hypervisor, a specialized software. The hypervisor is in charge of scheduling the VM execution and managing its lifecycle (start, stop, restart).

A VM provides "virtual hardware" to the guest; direct hardware access is not needed (but may be granted if needed). This provides major **security** and safety due to **isolation**: processes in different VMs are virtually separated: crashes or security incidents in a guest do not affect other guests. Should they need to interact, they must use the network or some special and explicitly configured channel in the hypervisor.

In the deployment via containers, processes are executed in a separate context within the same operating system instance. This context is created in the kernel by "virtualizing" system calls. In other words, containerized processes are running on **the same kernel** (like a traditional deployment), but every system call is executed in a specific context: a list of processes in a specific context may be different than the list of processes in another context.

Processes are still isolated: they can't interact with each other by default. Resource consumption and the span of context itself can be restricted to limit the impact of a problem within one container. However, they still share the kernel: a bug or a security incident within the kernel might affect all processes in the same server.

In general, many operating systems support containers in various shapes:

- `chroot` in most UNIXes (since 1982!)
- FreeBSD jails (2000)
- Solaris Containers/Zones (2000)
- LXC on Linux (2005)
- Docker/Podman on GNU/Linux (2013), Windows, and recently FreeBSD
- Other technologies: LXD, vServer, OpenVZ, Virtuozzo, etc.

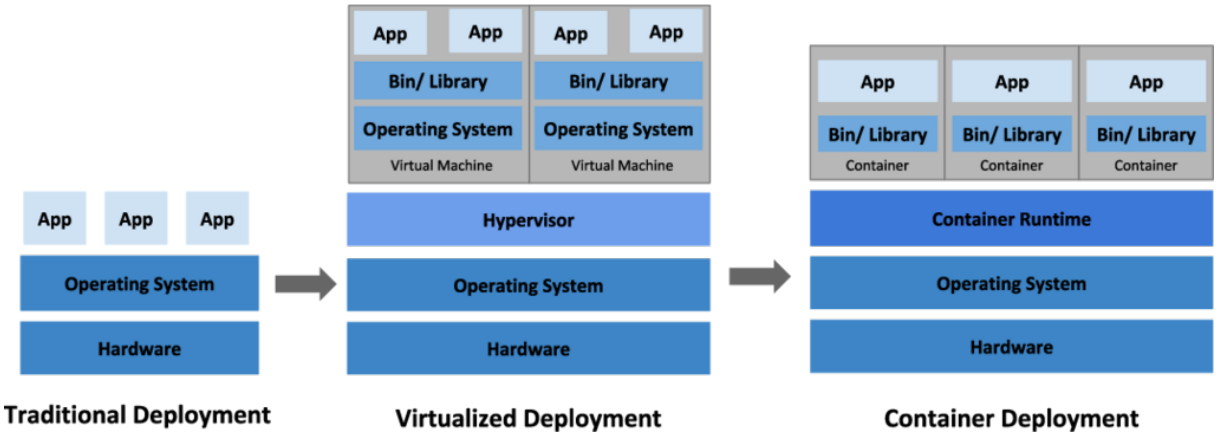


Figure 7.1: Deployment strategies - Image courtesy of The Kubernetes Authors / The Linux Foundation - CC BY 4.0

> In order to execute commands in the second part of this chapter, you should have the Docker Engine installed on your PC. Docker Desktop (which includes Docker Engine) is not required. You can install the Docker Engine for free by following the guide in the official Docker documentation at https://docs.docker.com/engine/install/.
>
> If you prefer Podman, you can replace the `docker` command with `podman` in every example

> in the second part of this chapter. To install Podman, use the installation instructions from their official website at https://podman.io/docs/installation.

7.1 Introduction to Linux Containers and Docker

In Linux, containers are built over **`cgroups`** (v2 currently). `cgroups` were designed by Google in 2006, and published in 2008. They provide four main features: limits on resources (e.g., restrict the RAM usage to a fraction of the total), prioritization (give priority to certain processes in the Linux schedule), accounting (measure the usage of resources for, e.g., billing, observability) and lifecycle control (e.g., restart-on-crash). **These features apply to all processes inside a container**.

You can control what containers can share between them using `cgroups` system calls. By default, the filesystem is not shared between containers, and processes cannot interact. There are other restrictions as well (such as the networking), controlled by *namespaces*.

7.1.1 Namespace isolation

Containers run in *namespaces*. A *namespace* may contain one or more containers; one container can be only in one namespace. Containers that run in the same namespace can share their resources with others.

There are different namespaces:

- Process ID (PID)
- Network
- UTS / hostname
- Mount
- Inter-process communication
- User
- Cgroup

Processes running in containers that runs in same namespace share these resources: suppose you have two PID namespaces, A and B, and three processes, P1, P2 in PID namespace A, and P3 in PID namespace B. When listing running processes, P1 and P2 and see each other, and they can communicate using POSIX signals. They have different PIDs (e.g., P1 has 1 and P2 has 2). Instead, P3 cannot see P1 nor P2; additionally, the P3 PID may be the same as P1 or P2 (e.g., 1) because PIDs are unique in the same namespace, not across them. The same goes for all other namespaces.

7.1.2 Docker

Docker (2013) is a set of tools for managing Linux containers, and, lately, Windows containers. They have been standardized under OCI (Open Container Initiative), and different implementations exist (e.g., `podman`).

Docker Container (instances) are started from a *container image*. A *container image* contains the "root directory" for the container. **This root directory will be ephemeral** in the container instance (data will be deleted at exit), and it will be not shared between instances. You can think of it as a sort of "template". Processes running in the container instance cannot access files outside that directory: for them, that directory is the root `/`. However, Docker can be instructed to mount an external path or volume in a specific path inside the container.

Container images are built using `Dockerfile/Containerfile`, and usually they start from a *base image* which contains some basic tools (e.g., `debian:stable` contains a basic Debian stable system; `python:3` contains a base `debian` image plus the Python 3 interpreter). The file contains instructions to build the image; each instruction creates a *layer*. Each *layer* contains the difference from the previous layers.

> The purpose of this structure is to share as much as possible common layers between images. In fact, `python:3` is built over a Debian image, allowing docker to store the Debian files only once.

Container image registries, such as the Docker Hub, allow sharing container images with your company or publicly. This is almost a requirement when running containers at scale (i.e., using multiple servers). Widely known base images, such as `python:3` and others, are frequently published on these platforms.

7.1.3 Containers in the cloud

Once a container is built, it can run on any Linux platform, regardless of the underlying system, provided that it supports cgroups and other container-related technologies. This includes not only on-premises Linux distributions, but also cloud providers such as OVH, Amazon Web Services, Microsoft Azure Cloud, ScaleWay, and others.

7.2 Running your first container

First, we will run a simple "hello world" container, `hello-world:latest`, and execute its default command (the `--rm` flag indicates that the container must be removed when stopped):

```
$ docker run --rm hello-world:latest
```

Docker `run` subcommand will download the image from Docker Hub if it is not present in the local system. Then, it runs the default command in the container (in this case, `/hello`, as you can see using `docker inspect hello-world:latest`) and prints its output in the standard output. The command, while executing, was confined in its container. To see some of these restrictions, let's spin an *interactive container*.

When you start an interactive container, your current standard input, output, and error will be linked to the container (main) process. Which means that you interact with the process which is inside the container. We can start an interactive container using the `-it` flag; we will use `opensuse/leap:latest` which provides a container with a base OpenSUSE Leap installation:

```
$ docker run -it --rm opensuse/leap:latest
```

Differently from above, the output now is a root bash prompt. You might notice that it is different from your prompt: this new shell is running inside the container. If you try to navigate the filesystem using `ls`, you will see that the root directory of this shell is different from the root directory of shells outside the container.

For example, `ls /home` shows the content of the `/home` directory. In the container, this directory is empty. In your Linux PC, this directory contains the home directories for all users.

Another test that you can do is try to enumerate all processes. To do that, you can use `ps -ef`. You will see that there are only two processes in execution: the `/bin/bash` shell, and `ps`.

Remember when we said that the container has an ephemeral root directory? Let's create a file named `keep-this-file` inside the `/home` directory (you can use `touch /home/keep-this-file` to create an empty file). Now, if you exit the bash shell using `exit`, the container will be terminated, and the file will be deleted (if you start a new container, it will have a new and different root directory).

To keep files between container executions, you can map an outside directory to a path inside the container. For example, assuming that you exited from the container bash shell (if not, use `exit` to return to your previous command prompt), let's create a directory named `shared` in the current directory, and launch a new container while mapping `shared` to `/shared` (inside the container):

```
$ mkdir shared
$ docker run -it --rm -v ./shared/:/shared --user $(id -u):$(id -g)
 ↪  opensuse/leap:latest
```

We had to add some options. First, we added the `-v` option that is used to map a directory from outside the container (in this case, `./shared/`) to a path inside the container (in this case, `/shared`). Then, we had to specify which is the user running the container with `--user` (we used a sub-shell to get them using `id`, but you can use your UID/GID if you know that, e.g., `--user 1000:1000`).

> Specifying a user is required because, by default, the OpenSUSE image starts with `root` user, while we want our user (outside the container) to be able to manage files in `./shared`. Without this option, files created inside the container in `/shared` will be owned by `root`, and you won't be able to read or write them outside the container!

Now, if you create a file at `/shared/keep-this-file` inside the container, the file is created in the `shared` directory outside the container ephemeral root, which means that you can stop the container and start another one, and the file will be there. You can also create files outside the container and use them inside it.

7.3 Creating container images

To deploy our application using containers, we must create a base image from which container instances are started. In this way, our application will be ready to go as soon as the container is created and launched.

7.3.1 A simple base image

Let's start by creating a simple container first. We won't use this container in our deployment, only to introduce the container image creation mechanisms.

Suppose that we want to add the `htop` command to a base Debian image (which does not have `htop` by default). We can create a file named `Dockerfile` with the following content:

```
# Start our image from a Debian stable
FROM debian:stable

# Update Debian package manager indexes
RUN apt-get update -yq

# Install htop
RUN apt-get install -yq htop
```

This file describes how to build the image: start from `debian:stable` image, then update the indexes of `apt` (the Debian package manager), and finally install `htop`.

To create the image, we should use the `docker build` command:

```
$ docker build -t debian-with-htop:stable .
```

We defined a new name and tag for our image (`debian-with-htop:stable`) using `-t` flag. The "dot" at the end of the command refers to the directory that is used as a "build context": some commands (such as COPY) require a "build context".

> COPY adds files inside a container image. For various reasons, files are not directly copied from the filesystem of your PC: when you build an image, you need to provide a build context with all files you need to COPY inside the image. The COPY base path will be the build context directory.

To test our image, we need to start the container in interactive mode because the default command for `debian:stable` is a shell (and we haven't changed it in our `Dockerfile`):

```
$ docker run -it --rm debian-with-htop:stable
```

If you try to launch `htop` in this container, you will see that the program is installed and runs correctly! Exit `htop` by pressing q and the container by typing `exit`.

What if we want to launch `htop` by default? We can do that using the CMD command and the full path for `htop`. Modify the content of your `Dockerfile` as follows:

```
# Start our image from a Debian stable
FROM debian:stable

# Update Debian package manager indexes
RUN apt-get update -yq

# Install htop
RUN apt-get install -yq htop

# Run HTOP by default
CMD ["/usr/bin/htop"]
```

If you build and run this container image, you will see that `htop` is executed directly. If you want to overwrite the CMD command for one container only, you can append the full path of the executable at the end of the `docker run` command.

7.3.2 Build the backend image

Now that we know how to build an image, let's build two images: one that runs our Go backend, and another serving our Vue.js frontend.

Let's start with the backend. We need an image with a Go compiler. Fortunately, the Docker Hub has a container image ready to build Go programs. We will use the version `1.19.1` (you may want to use the same Go version that you have on your PC).

Create a new file in the root directory of your Go backend project (the same directory where you have `go.mod`) named `Dockerfile` with the following content:

```
# Start from a Go base image
FROM golang:1.19.1

# Switch to the /src/ directory
# (create it if necessary)
WORKDIR /src/

# Copy Go code in the image
COPY . .

# Build the executable
RUN go build -o ./webapi ./cmd/webapi

# Inform Docker that our server uses port 3000
# Change this port with the one you used in the
# http.Listen() call in the Go project.
EXPOSE 3000

# Inform Docker that we expect a volume on this
# path (e.g., to save the database).
VOLUME /data

# Set the default program to our Go backend
CMD ["/src/webapi"]
```

Let's build and run our image:

```
$ docker build -t backend:latest .
$ docker run -it --rm -p 3000:3000 backend:latest
```

In our `docker run` command, we added `-p 3000:3000`, which is a flag that forwards a port on our PC (first number) to a port in the container (second number). In this case, we are mapping the port where the Go backend server is listening for HTTP connections (if you used another port, you should change these two numbers).

You should see the output of your Go server in your terminal. The server is reachable at http://localhost:3000 (you can use `curl` or the Swagger Editor to test it). To close the server, you can press CTRL+C.

7.3.3 Multi-stage builds

The image we just created (`backend:latest`) is working, but it is sub-optimal as it contains the executable, the source code, and the entire Go compiler. However, we don't need the Go compiler to run our application. How can we discard everything after the `go build` command?

> You might be tempted to remove files using something like `RUN rm /usr/local/...`. It doesn't work: given that a container image is a set of layers, all "removed" files are still there (even if they are not reachable anymore). You can't remove a file from an image.

We can use a technique named *multi-stage build*: we start with one (or more!) temporary images, and then we build the final one with the "artifacts" (executables, assets, etc.). Each stage starts with a `FROM` command and terminates with the next `FROM` (or the end of the file). All temporary images (i.e., all images except the last one) must have a name.

Let's modify our Go backend `Dockerfile` to use *two* stages:

```
# Create a first temporary image named "builder"
FROM golang:1.19.1 AS builder

# Copy Go code (in "builder")
WORKDIR /src/
COPY . .

# Build executables (in "builder")
RUN go build -o /app/webapi ./cmd/webapi


# Create final container
FROM debian:bookworm

# Inform Docker about which port is used
```

```
EXPOSE 3000

# Copy the executable from the "builder" image
WORKDIR /app/
COPY --from=builder /app/webapi ./

# Set the default program to our Go backend
CMD ["/app/webapi"]
```

When building this `Dockerfile`, Docker saves the last image (i.e., all commands starting from the last `FROM` command in the image to the end of the file), while all temporary images are discarded. So, in this case, we have a final image that contains only the `debian:bookworm` base image and the `webapi` executable. The Go compiler and the source code are not present because they were in a temporary image.

We can do the same for the frontend. You can create a `Dockerfile` at the root of your Vue.js project with the following content:

```
# Use latest Node LTS to build
FROM node:lts as builder

# Copy Vue.js code
WORKDIR /app
COPY . .

### Build Vue.js into plain HTML/CSS/JS
RUN npm run build-prod


### Create final container
FROM nginx:stable

### Copy the (built) app from the builder image
COPY --from=builder /app/dist /usr/share/nginx/html
```

Here we are using `node:lts` to build the image (as Vue.js requires Node), and `nginx:stable` to serve the resulting HTML/JS/CSS code (which is the Vue.js compiled application) to users. There is no need for a `CMD` at the end of the `Dockerfile`, as the Nginx image already defines the command to execute.

Let's build and run it (Nginx listens at port `80`, but we use port `8080` in our PC):

```
$ docker build -t frontend:latest .
$ docker run -it --rm -p 8080:80 frontend:latest
```

The frontend is now served at http://localhost:8080.

> Note: if you have one repository with both backend and frontend, and you want to have two different `Dockerfiles`, you can name it with different names (such as `Dockerfile.frontend` and `Dockerfile.backend`) and use the `-f` flag when building them. E.g.,
>
> ```
> $ docker build -f Dockerfile.frontend -t frontend:latest .
> ```

7.4 Deploying and managing containers

After building the container images we need for our project, it's time to deploy them to a server. The standard deployment workflow includes pushing the image on a *image registry*. This registry is a central collector for all images: developers (or specialized "build machines") push images after builds, and servers pull these images to deploy them. You can use online platforms, such as Docker Hub or Quay.io (usually protecting your image behind authentication), or private registries, such as Harbor.

In these examples, we will use the Docker Hub platform. If you don't have an account, you can subscribe to their free tier. Note that the free tier supports only **public** images (in other words, everyone will be able to pull your images).

First, we need to log in to our registry:

```
$ docker login hub.docker.com
```

Then, we need to create a new "repository" in hub.docker.com to host the new image: open the website, log in with your credentials, click on the "Create Repository" button, and use the name `backend`. Repeat the "Create Repository" to create the `frontend` repository.

Now we need to tag and push the images we created before:

```
$ docker tag backend:latest docker.io/your_username/backend:latest
$ docker push docker.io/your_username/backend:latest
$ docker tag frontend:latest docker.io/your_username/frontend:latest
$ docker push docker.io/your_username/frontend:latest
```

Replace `your_username` with your DockerHub username.

> You can build and push in one command by tagging the image directly with the full address of the repository, and using the `--push` option. E.g.,
>
> ```
> $ docker build -t docker.io/your_username/frontend:latest --push .
> ```

Once your images are in the registry, you can access the deployment server and run the container. To do that in a server where you have shell access, use the same `docker run` commands we saw before. To deploy them **in the cloud**, you should use the full path of the image (e.g., `docker.io/your_username/frontend:latest`) when creating a new container instance in the cloud provider control panel.

> Some cloud providers offer a private registry that is directly connected to their deployment panel. This simplifies the deployment as you can directly push towards your cloud provider.

7.5 Docker Compose for multi-container applications

When running multiple containers in a host (development or server machine), there are several tools to run and manage multiple containers at once. One of these tools is *Docker Compose*. It allows you to define all the services, networks, and volumes required for your application in a single, easy-to-read YAML file.

We will explore how to use Docker Compose to run the two Docker images we created before (`backend:latest` and `frontend:latest`). We will also see how to use volumes to persist data between containers and map ports to access the services.

> Note that nearly all `docker run` options have an equivalent in Docker Compose and vice-versa.

First, create a file named `docker-compose.yml` in your project directory. This file will define the configuration for your containers:

```
version: '3'

services:
  backend:
    image: docker.io/your_username/backend:latest
    ports:
      - "3000:3000"
    volumes:
      - ./shared:/shared
```

```
  frontend:
    image: docker.io/your_username/frontend:latest
    ports:
      - "8080:80"
```

Let's break down what this YAML file does:

- we specify the Docker Compose version as '3';
- we define two services: backend and frontend;
- for each service, we specify the Docker image to use;
- we map ports so that the backend container listens on port 3000, and the frontend container listens on port 8080 on the host machine;
- we create a volume in the backend service to persist data.

To start the containers defined in the `docker-compose.yml` file, run the following command (in the same directory as the `docker-compose.yml` file):

```
$ docker compose up -d
```

Docker Compose will pull the specified images (if not already available) and start the containers in the background (thanks to the `-d` option) with the configurations defined in the YAML file.

Once the containers are up and running, you can access the backend at port 3000 and the frontend at port 8080 using the IP of the machine where you launched the `docker compose up` command.

To stop the containers and remove them, run the following command:

```
$ docker compose down
```

7.6 Beyond Docker Compose: container orchestrators

While Docker Compose is a valuable tool for managing containers within a single host, it has limitations when it comes to orchestrating containers across multiple hosts in a production environment. This is where container orchestrators like Docker Swarm, Kubernetes, and others come into play. These orchestrators provide a higher level of automation and control over containerized workloads, enabling organizations to build and run resilient, scalable, and highly available applications.

Container orchestrators are beyond the scope of this book. However, if you are interested, the Docker Swarm and the Kubernetes documentation are good places to start.

Although these orchestrators may provide custom services to container instances they manage, the same image you built above (`backend:latest` and `frontend:latest`) can run **unmodified** on these orchestrators.

7.7 Links

- https://docs.docker.com/get-started/
- https://docs.podman.io/en/latest/