Clean Architectures in Python

Chapter 1 - A day in the life of a clean system

Must be my lucky day.

Terminator 2, 1991

In this chapter I will introduce the reader to a (very simple) system designed with a clean architecture. The purpose of this introductory chapter is to familiarise with main concepts like separation of concerns and inversion of control, which are paramount in system design. While I describe how data flows in the system, I will purposefully omit details, so that we can focus on the global idea and not worry too much about the implementation. This example will be then explored in all its glorious details in the following chapters, so there will be time to discuss specific choices. For now, try to get the big picture.

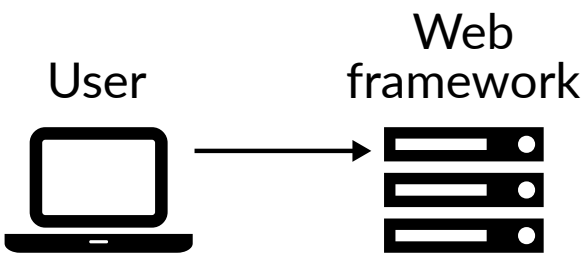
The data flow ¶

In the rest of the book, we will design together part of a simple web application that provides a room renting system. So, let's consider that our "Rent-o-Matic" application[1] is running at https://www.rentomatic.com, and that a user wants to see the available rooms. They open the browser and type the address, then clicking on menus and buttons they reach the page with the list of all the rooms that our company rents.

Let's assume that this URL is `/rooms?status=available`. When the user's browser accesses that URL, an HTTP request reaches our system, where there is a

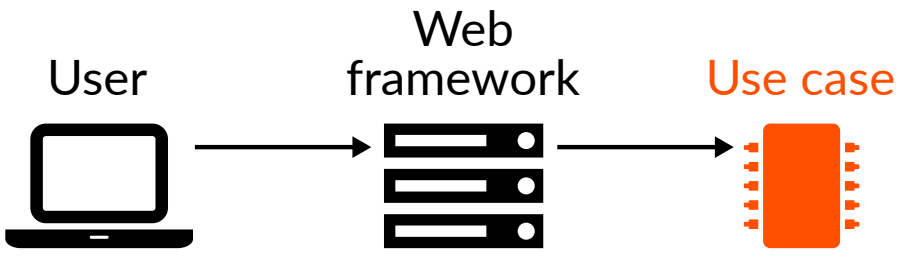
component that is waiting for HTTP connections. Let's call this component "web framework"[2].

The purpose of the web framework is to understand the HTTP request and to retrieve the data that we need to provide a response. In this simple case there are two important parts of the request, namely the endpoint itself (`/rooms`), and a single query string parameter, `status=available`. Endpoints are like commands for our system, so when a user accesses one of them, they signal to the system that a specific service has been requested, which in this case is the list of all the rooms that are available for rent.



The web framework serving HTTP

The domain in which the web framework operates is that of the HTTP protocol, so when the web framework has decoded the request it should pass the relevant information to another component that will process it. This other component is called *use case*, and it is the crucial and most important component of the whole clean system as it implements the *business logic*.



The business logic

The business logic is an important concept in system design. You are creating a system because you have some knowledge that you think might be useful to the world, or at the very least marketable. This knowledge is, at the end of the day, a way to process data, a way to extract or present data that maybe others don't have.

A search engine can find all the web pages that are related to the terms in a query, a social network shows you the posts of people you follow and sorts them according to a specific algorithm, a travel company finds the best options for your journey between two locations, and so on. All these are good examples of business logic.

Business logic

Business logic is the specific algorithm or process that you want to implement, the way you transform data to provide a service. It is the most important part of the system.

The use case implements a very specific part of the whole business logic. In this case we have a use case to search for rooms with a given value of the parameter `status`. This means that the use case has to extract all the rooms that are managed by our company and filter them to show only the ones that are available.

Why can't the web framework do it? Well, the main purpose of a good system architecture is to *separate concerns*, that is to keep different responsibilities and domains separated. The web framework is there to process the HTTP protocol, and is maintained by programmers that are concerned with that specific part of the system, and adding the business logic to it mixes two very different fields.

Separation of concerns

Different parts a system should manage different parts of the process. Whenever two separate parts of a system work on the same data or the same part of a process they are *coupled*. While coupling is unavoidable, the higher the coupling between two components the harder is to change one without affecting the other.

As we will see, separating layers allows us to maintain the system with less effort, making single parts of it more testable and easily replaceable.

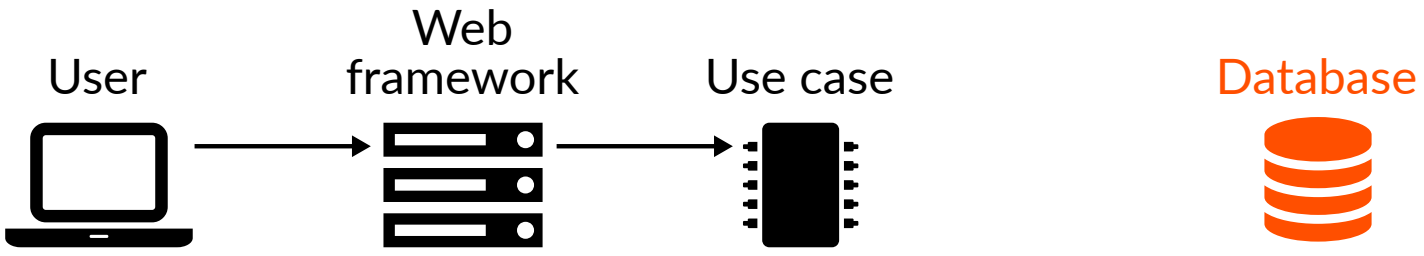
In the example that we are discussing here, the use case needs to fetch all the rooms that are in an available state, extracting them from a source of data. This is the business logic, and in this case it is very straightforward, as it will probably consist of a simple filtering on the value of an attribute. This might however not be the case. An example of a more advanced business logic might be an ordering based on a recommendation system, which might require the use case to connect with more components than just the data source.

So, the information that the use case wants to process is stored somewhere. Let's call this component *storage system*. Many of you probably already pictured a database in your mind, maybe a relational one, but that is just one of the possible data sources. The abstraction represented by the storage system is: anything that the use case can access and that can provide data is a source. It might be a file, a database (either relational or not), a network endpoint, or a remote sensor.

Abstraction

When designing a system, it is paramount to think in terms of abstractions, or building blocks. A component has a role in the system, regardless of the specific implementation of that component. The higher the level of the abstraction, the less detailed are the components. Clearly, high-level abstractions don't consider practical problems, which is why the abstract design has to be then implemented using specific solutions or technologies.

For simplicity's sake, let's use a relational database like Postgres in this example, as it is likely to be familiar to the majority of readers, but keep in mind the more generic case.



The storage

How does the use case connect with the storage system? Clearly, if we hard code into the use case the calls to a specific system (e.g. using SQL) the two components will be *strongly coupled*, which is something we try to avoid in system design. Coupled components are not independent, they are tightly connected, and changes occurring in one of the two force changes in the second one (and vice versa). This also means that testing components is more difficult, as one component cannot live without the other, and when the second component is a complex system like a database this can severely slow down development.

For example, let's assume the use case called directly a specific Python library to access PostgreSQL such as psycopg. This would couple the use case with that specific source, and a change of database would result in a change of its code. This is far from being ideal, as the use case contains the business logic, which has not changed moving from one database system to the other. Parts of the system that do not contain the business logic should be treated like implementation details.

Implementation detail

A specific solution or technology is called a *detail* when it is not central to the design as a whole. The word doesn't refer to the inherent complexity of the subject, which might be greater than that of more central parts.

A relational database is hundred of times richer and more complex than an HTTP endpoint, and this in turn is more complex than ordering a list of objects, but the core of the application is the use case, not the way we store data or the way we provide access to that. Usually, implementation details are mostly connected with performances or usability, while the core parts implement the pure business logic.

How can we avoid strong coupling? A simple solution is called *inversion of control*, and I will briefly sketch it here, and show a proper implementation in a later section of the book, when we will implement this very example.

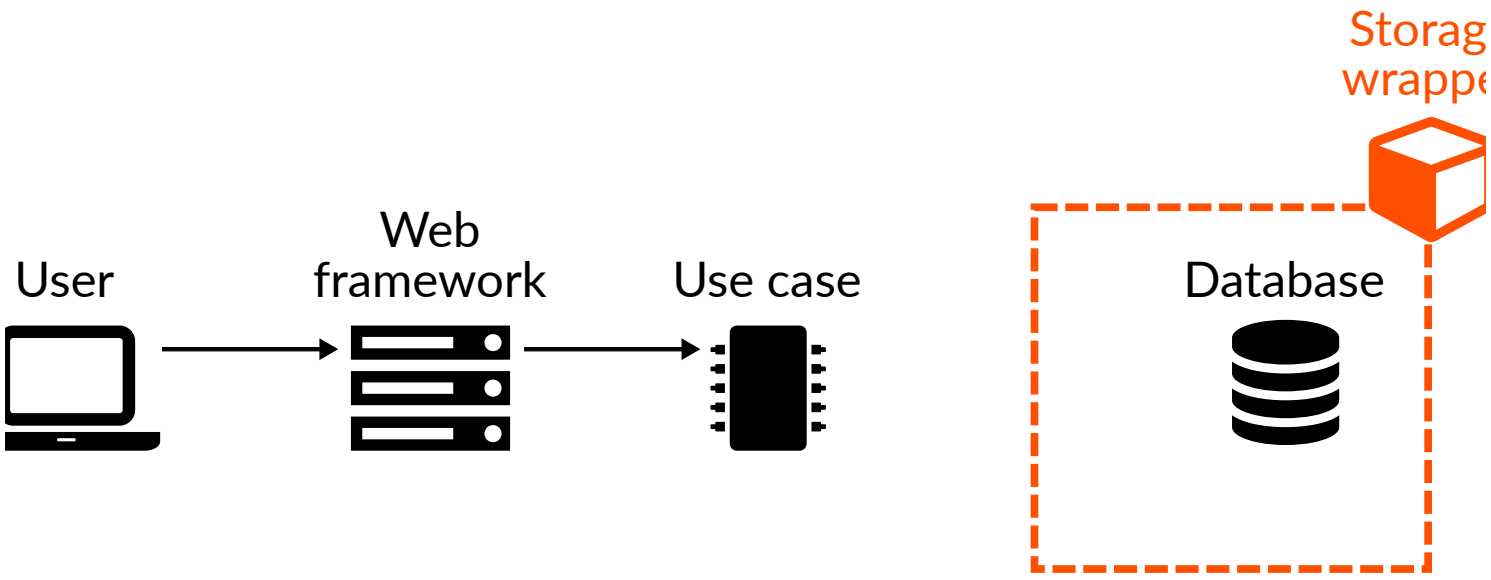
Inversion of control happens in two phases. First, the called object (the database in this case) is wrapped with a standard interface. This is a set of functionalities shared by every implementation of the target, and each interface translates the functionalities to calls to the specific language[3] of the wrapped implementation.

Inversion of control

A technique used to avoid strong coupling between components of a system, that involves wrapping them so that they expose a certain interface. A component expecting that interface can then connect to them without knowing the details of the specific implementation, and thus being strongly coupled to the interface instead of the specific implementation.

A real world example of this is that of power plugs: electric appliances are designed to be connected not with specific power plugs, but to any power plug that is build according to the specification (size, number of poles, etc). When you buy a TV in the UK, you expect it to come with a UK plug (BS 1363). If it doesn't, you need an *adapter* that allows you to plug electronic devices into sockets of a foreign nation. In this case, we need to connect the use case (TV) to a database (power system) that have not been designed to match a common interface.

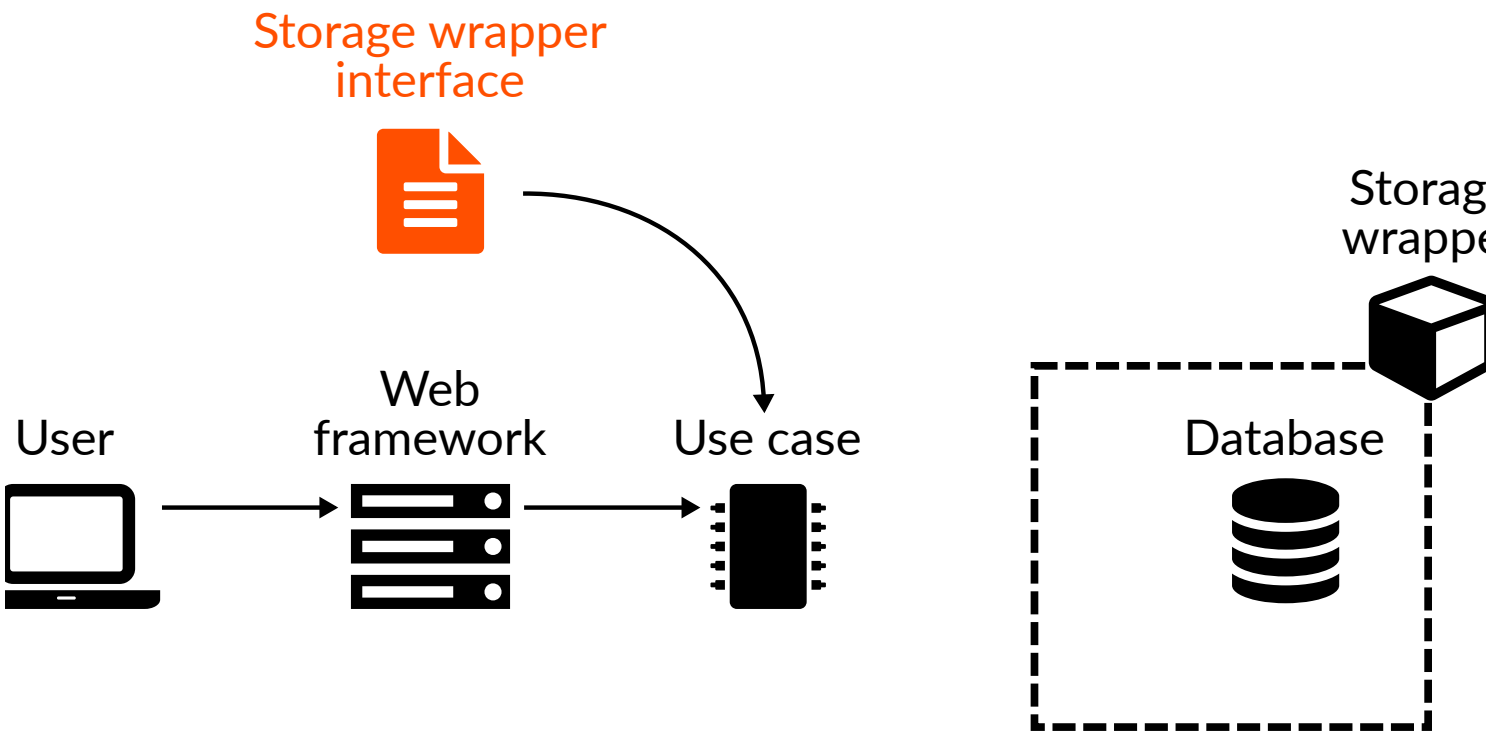
In the example we are discussing, the use case needs to extract all rooms with a given status, so the database wrapper needs to provide a single entry point that we might call `list_rooms_with_status`.



The storage interface

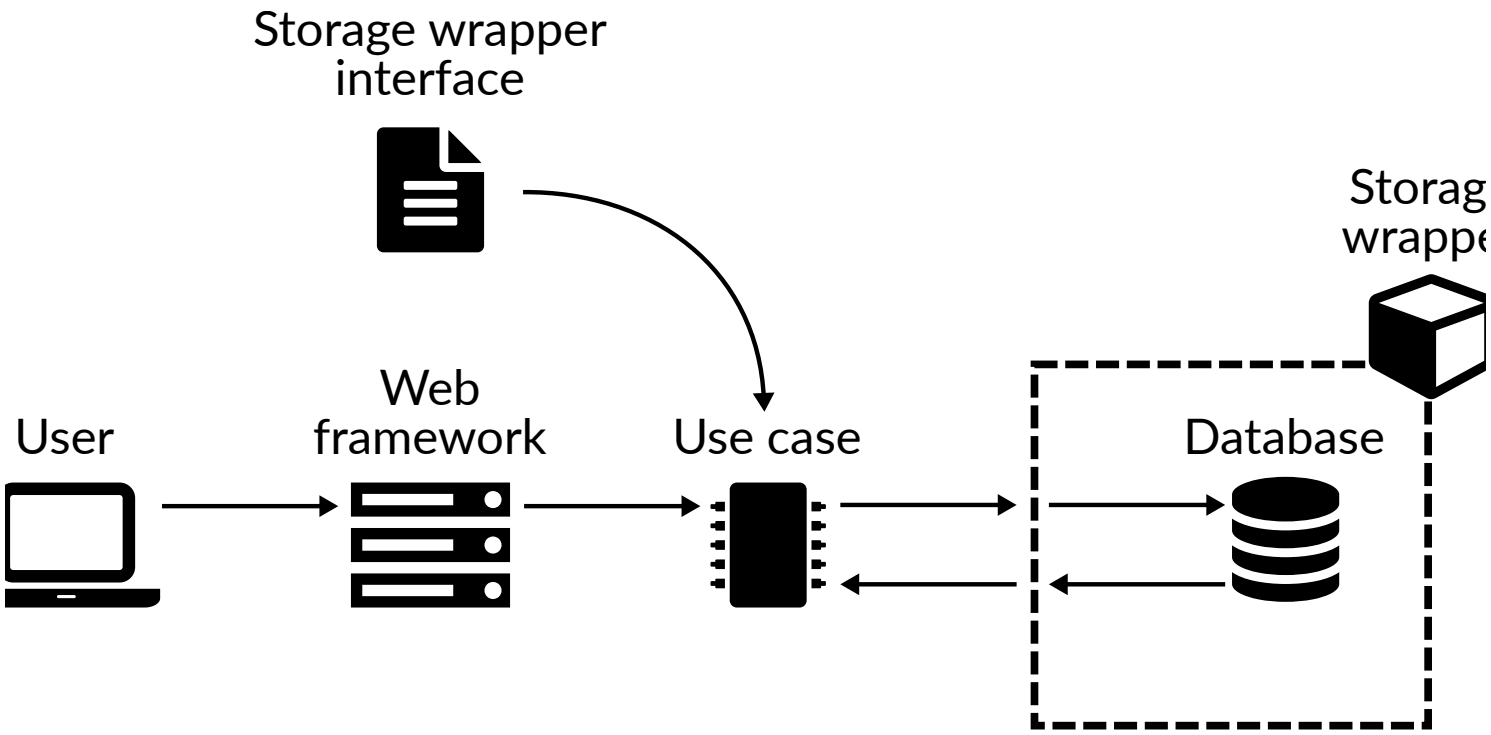
In the second phase of inversion of control the caller (the use case) is modified to avoid hard coding the call to the specific implementation, as this would again cou-

ple the two. The use case accepts an incoming object as a parameter of its constructor, and receives a concrete instance of the adapter at creation time. The specific technique used to implement this depends greatly on the programming language we use. Python doesn't have an explicit syntax for interfaces, so we will just assume the object we pass implements the required methods.



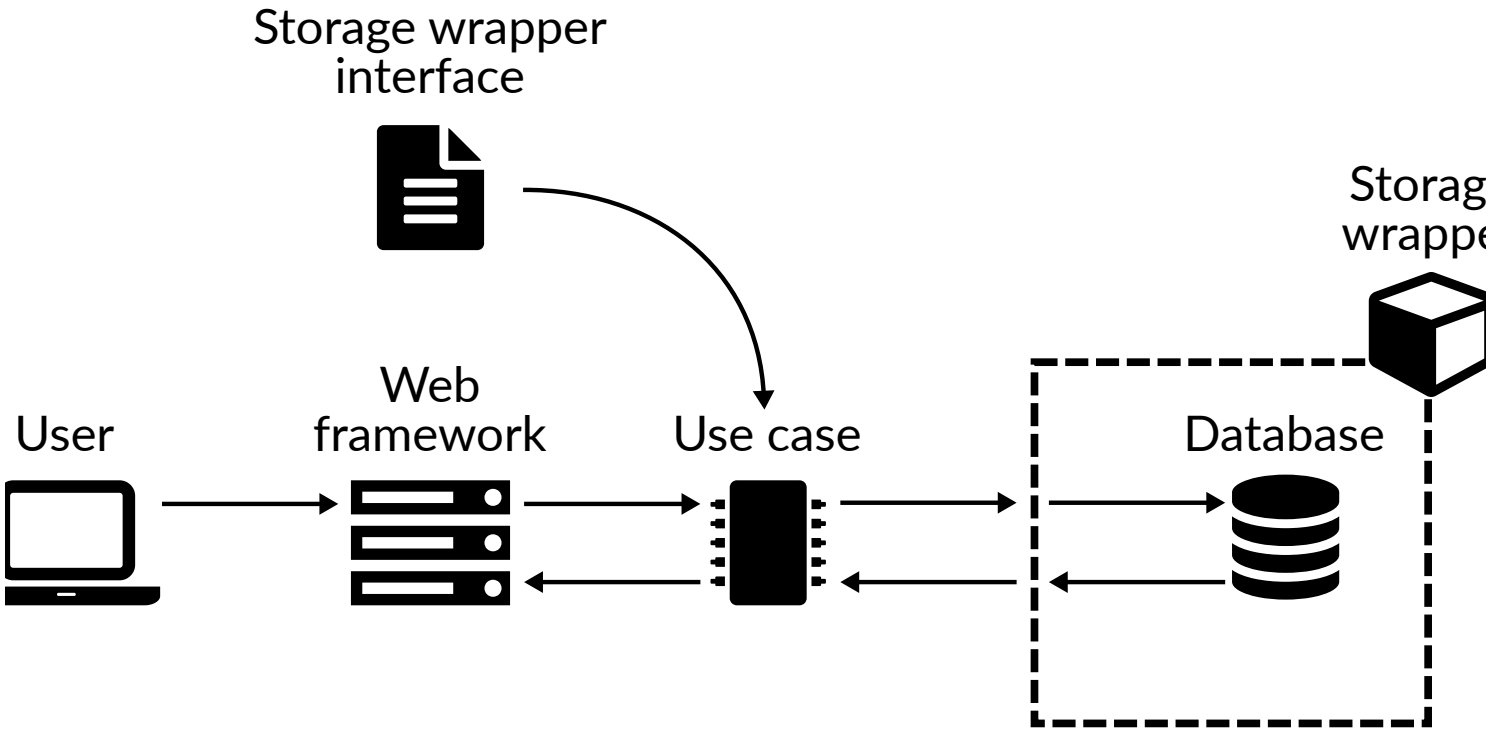
Inversion of control on the storage interface

Now the use case is connected with the adapter and knows the interface, and it can call the entry point `list_rooms_with_status` passing the status `available`. The adapter knows the details of the storage system, so it converts the method call and the parameter in a specific call (or set of calls) that extract the requested data, and then converts them in the format expected by the use case. For example, it might return a Python list of dictionaries that represent rooms.



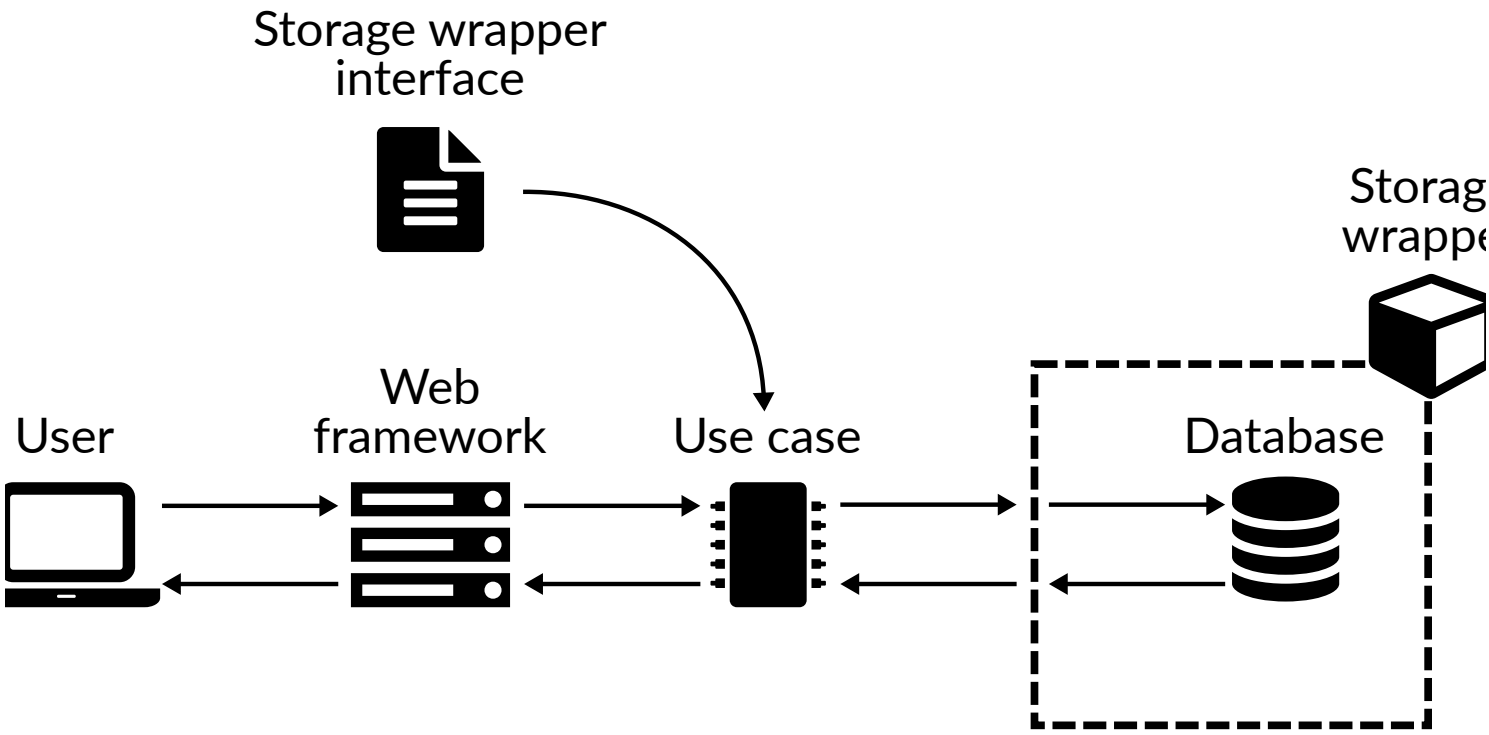
The business logic extracts data from the storage

At this point, the use case has to apply the rest of the business logic, if needed, and return the result to the web framework.



The business logic returns processed data to the web framework

The web framework converts the data received from the use case into an HTTP response. In this case, as we are considering an endpoint that is supposed to be reached explicitly by the user of the website, the web framework will return an HTML page in the body of the response, but if this was an internal endpoint, for example called by some asynchronous JavaScript code in the front-end, the body of the response would probably just be a JSON structure.



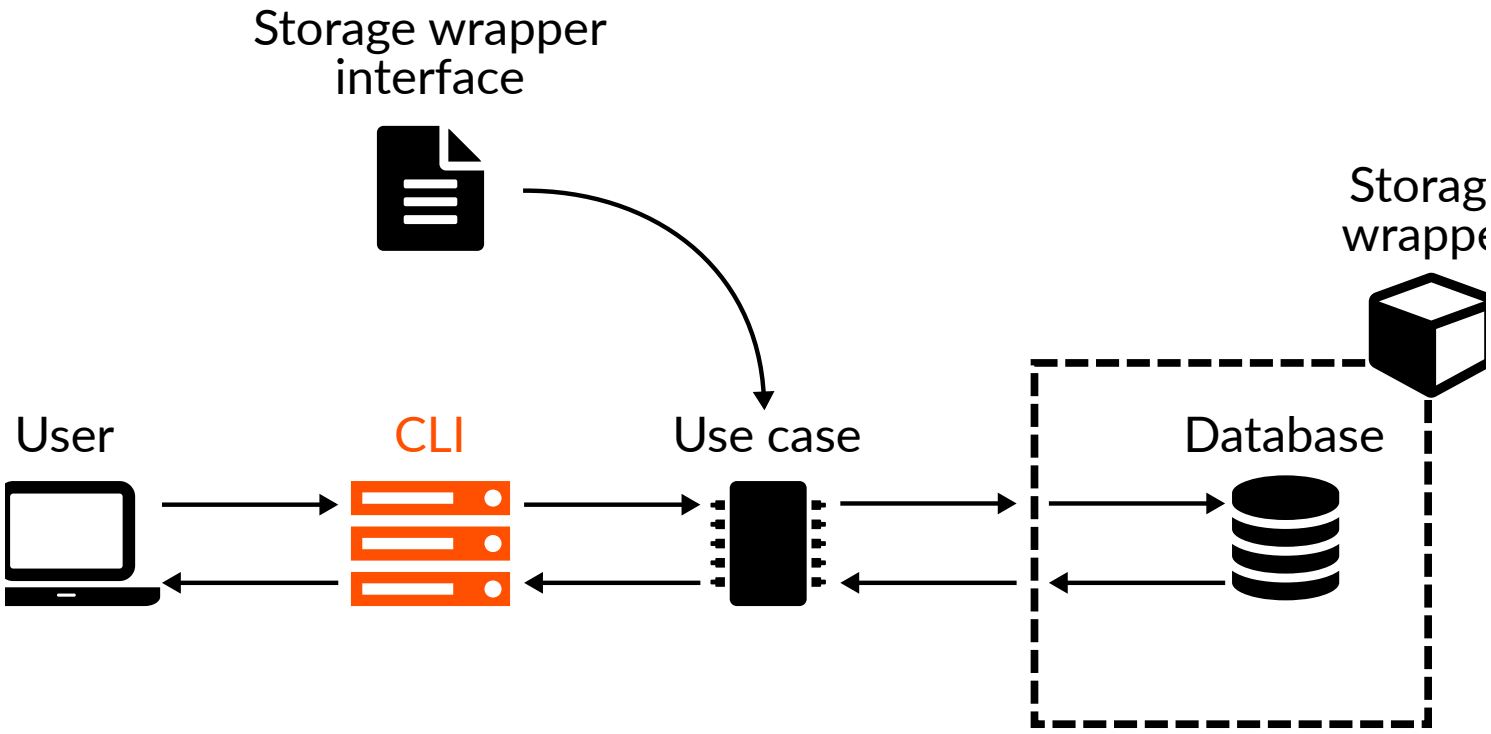
The web framework returns the data in an HTTP response

Advantages of a layered architecture ¶

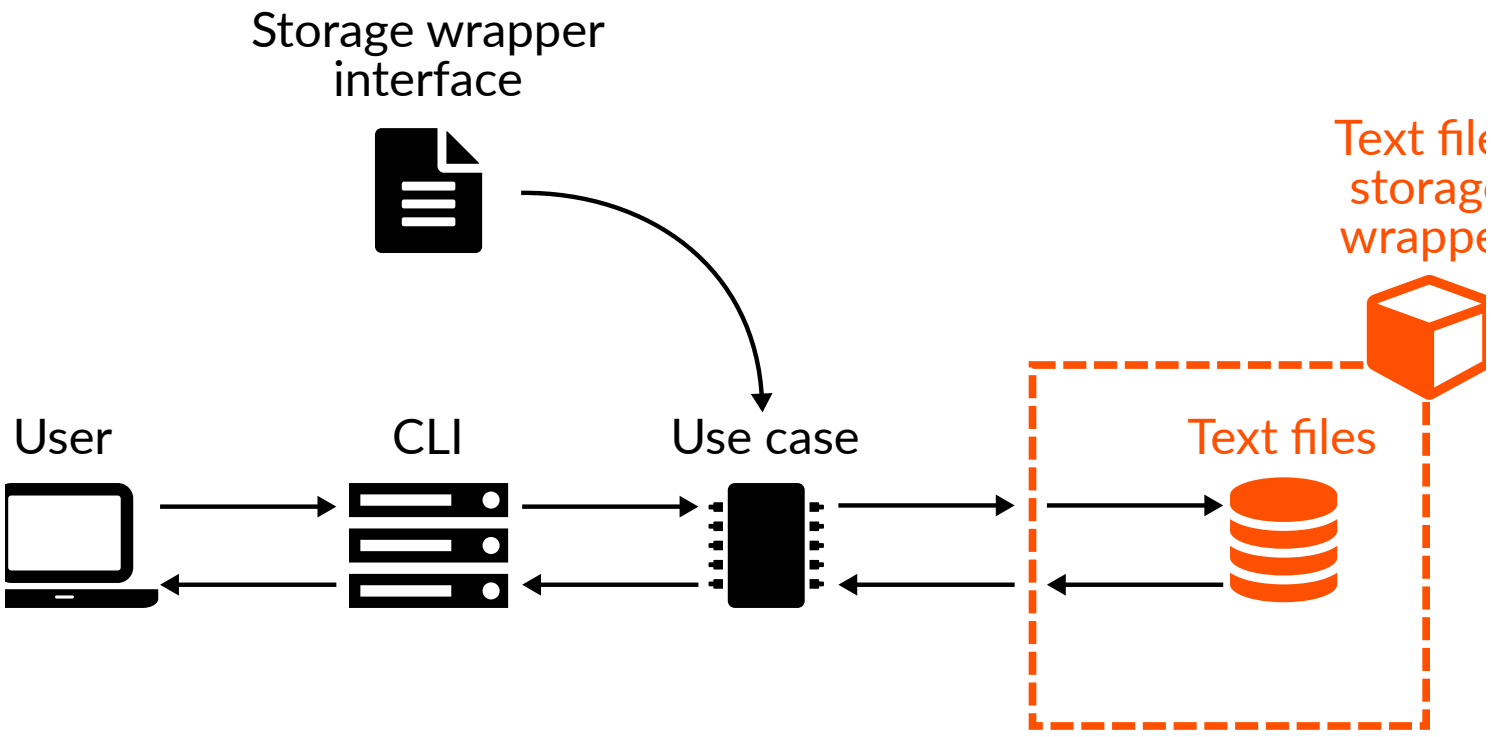
As you can see, the stages of this process are clearly separated, and there is a great deal of data transformation between them. Using common data formats is one of the way we achieve independence, or loose coupling, between components of a computer system.

To better understand what loose coupling means for a programmer, let's consider the last picture. In the previous paragraphs I gave an example of a system that uses a web framework for the user interface and a relational database for the data source, but what would change if the front-end part was a command-line interface?

And what would change if, instead of a relational database, there was another type of data source, for example a set of text files?



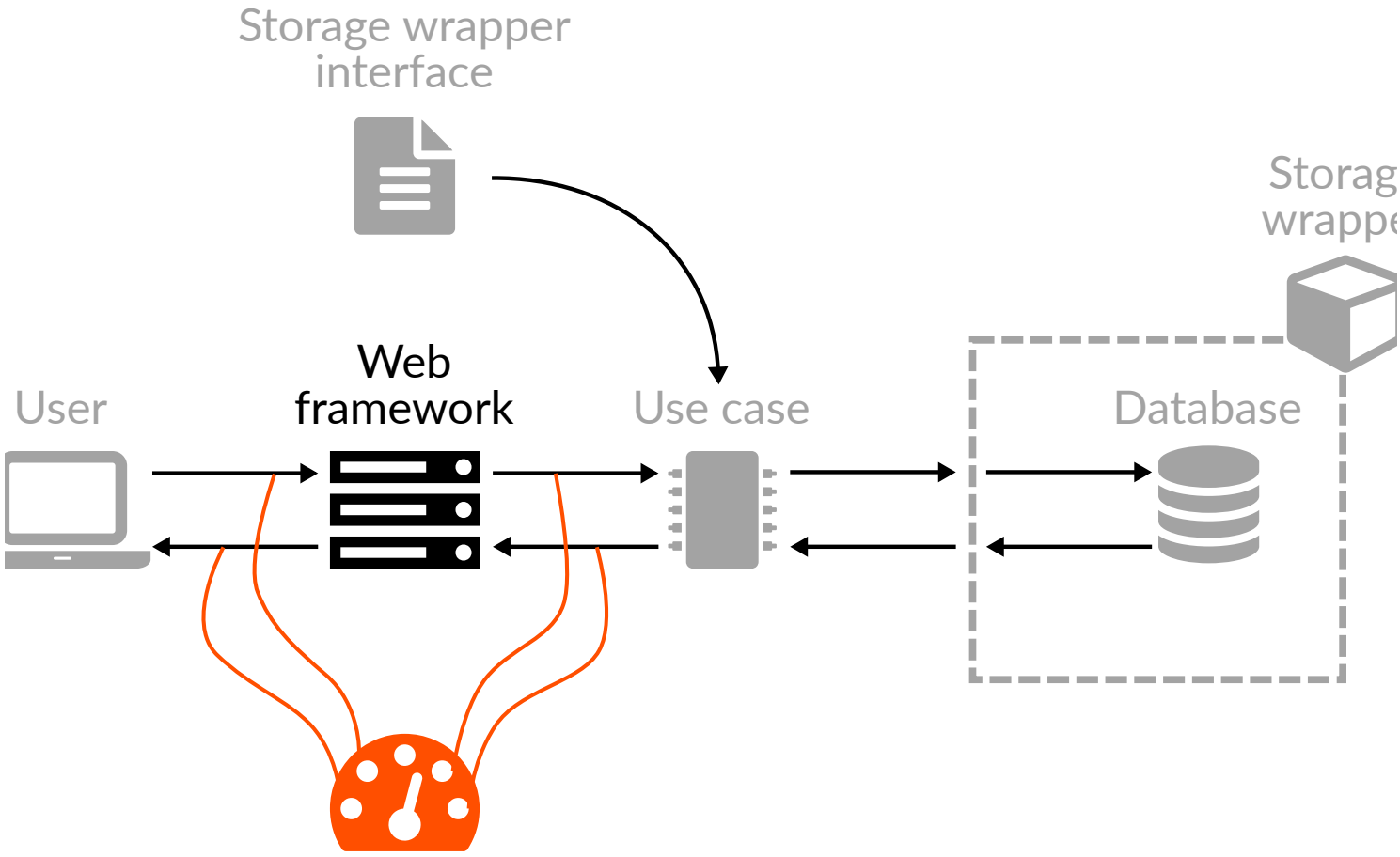
The web framework replaced by a CLI



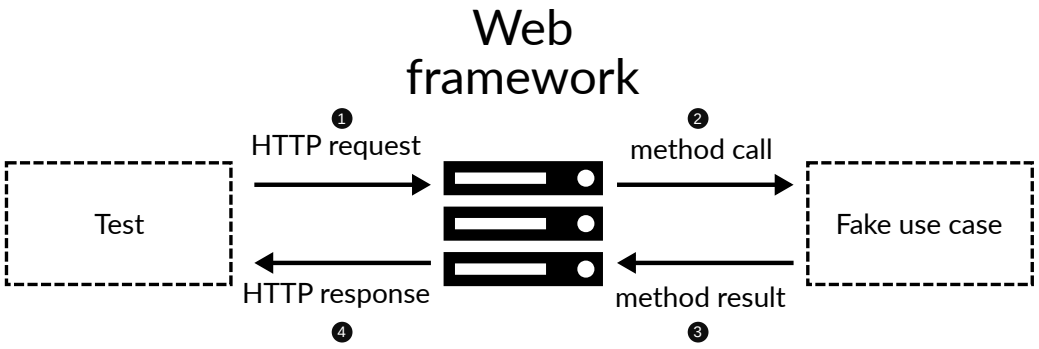
A database replaced by a more trivial file-based storage

As you can see, both changes would require the replacement of some components. After all, we need different code to manage a command line instead of a web page. But the external shape of the system doesn't change, neither does the way data flows. We created a system in which the user interface (web framework, command-line interface) and the data source (relational database, text files) are details of the implementation, and not core parts of it.

The main immediate advantage of a layered architecture, however, is testability. When you clearly separate components you clearly establish the data each of them has to receive and produce, so you can ideally disconnect a single component and test it in isolation. Let's take the Web framework component that we added and consider it for a moment forgetting the rest of the architecture. We can ideally connect a tester to its inputs and outputs as you can see in the figure



Testing the web layer in isolation



Detailed setup of the web layer testing

We know that the Web framework receives an HTTP request (1) with a specific target and a specific query string, and that it has to call (2) a method on the use case passing specific parameters. When the use case returns data (3), the Web framework has to convert that into an HTTP response (4). Since this is a test we can have a fake use case, that is an object that just mimics what the use case does without really implementing the business logic. We will then test that the Web framework calls the method (2) with the correct parameters, and that the HTTP response (4) contains the correct data in the proper format, and all this will happen without involving any other part of the system.

So, now that we had a 10,000 feet overview of the system, let's go deeper into its components and the concepts behind them. In the next chapter I will detail how the design principles called "clean architecture" help to implement and use effectively concepts like separation of concerns, abstraction, implementation, and inversion of control.

1 I was inspired by the Sludge-O-Matic™ from Day of the Tentacle

2 There are many more layers that the HTTP request has to go through before reaching the actual web framework, for example the web server, but since the purpose of those layers is mostly to increase performances, I am not going to consider them until the end of the book.

3 The word *language*, here, is meant in its broader sense. It might be a programming language, but also an API, a data format, or a protocol.

Previous
About the book

Next
Chapter 2 - Components of a clean architecture

Last update: 20/08/2021

Section 3 of Clean Architectures in Python

Previous

- Introduction
- About the book

Next

- Chapter 2 - Components of a clean architecture
- Chapter 3 - A basic example
- Chapter 4 - Add a web application
- Chapter 5 - Error management
- Chapter 6 - Integration with a real external system - Postgres
- Chapter 7 - Integration with a real external system - MongoDB
- Chapter 8 - Run a production-ready system
- Changelog
- Colophon