Functional Programming In Java

{with: Vavr.io}

Practical functional programming in Java using Vavr.io functional library

dimitris papadimitriou

FUNCTIONAL PROGRAMMING IN JAVA WITH VAVR

Practical functional programming in Java using Vavr.io functional library

Dimitris Papadimitriou

This version was published on 03/09/2020

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Feel free to contact me at:

https://www.linkedin.com/in/dimitrispapadimitriou/

https://leanpub.com/u/dimitrispapadim

https://medium.com/@dimpapadim3

https://github.com/dimitris-papadimitriou-chr

dimitrispapadim@live.com

Acknowledgments :

https://pixabay.com/photos/volcano-java-indonesia-mount-seremu-16912/

Contents

Purpose

This book is aiming to present the basics of functional programming in Java using the **Vavr.io** library. We will try to exhibit the usage of the basic Functional types: Option, Either, Future and Validation.

Resources

The online fiddle list used throughout the book:

- https://repl.it/@dimitrispapadim

github repos:

- Spring-Boot-WebAppExample
- functional-java-vavr
- Distributed-SpringBoot-CleanArchitecture

Java Functional Features

"1996 - James Gosling invents Java. Java is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Sun loudly heralds Java's novelty.

-*Brief, Incomplete and Mostly Wrong History of Programming Languages*,

1 Java Functional features

1.1 Functional Interfaces

The most prominent language feature that facilitates functional programming is the existence of First-Class functions. This means the language needs to support the ability to treat functions as you would any variable and pass them around to other functions as you see fit. Lambda functions extend this concept, allowing the creation of an anonymous function, in a compressed and easy to read syntax.

Here we are going to display in rapid succession in just a section the evolution of the Java features related to lambdas.

Functional Interfaces

Any interface with a SAM(Single Abstract Method) can be a functional interface. All functional interfaces should be decorated with an informative *@FunctionalInterface* annotation

For example, we can declare this functional interface

```
@FunctionalInterface
public interface DiscountStrategy {
    double getDiscounted(double discount, double price);
}
```

this says that anything that looks like the following :

```
double ___(double price, double discount)
```

can be assign on a variable of type `DiscountStrategy`

Method reference

Thus, if we have the following function

```
public class Discounts {
    public static double discountedPrice(double discount, double price) {
        return price - discount * price;
    }
}
```

The following is a valid declaration

```
DiscountStrategy discountStrategy = Discounts::discountedPrice;
```

The `Discounts::discountedPrice` is a method reference. This means if we try to use the `discountStrategy` variable the method `discountedPrice` inside the `Discounts` class will be used.

```
var finalPrice  = discountStrategy.getDiscounted(0.1,100);
```

Anonymous class

We can also use anonymous class to create an implementation for the interface

```
DiscountStrategy discountStrategy = new DiscountStrategy() {
     @Override
     public double getDiscounted(double discount, double price) {
        return price - discount * price;
   }
};
```

This is just an inline initialization of the functional interface `new DiscountStrategy() {…}` and inside the body we override the single method of the functional interface

```
@Override
public double getDiscounted(double discount, double price) {
        return price - discount * price;
}
```

Lambda expression

Or instead of we can use a lambda expression to inhabit the variable

```
DiscountStrategy discountStrategy =
          (double discount, double price) -> price - discount * price;
```

Or we can let java type inference figure out the argument types :

```
DiscountStrategy discountStrategy = (discount, price) -> price - discount * price;
```

Also using `var` is valid

```
DiscountStrategy discountStrategy =
    (var discount, var price) -> price - discount * price
```

In this way **method references, anonymous classes and lambda expression** have been homogenized. This is one step closer to the functional paradigm where functions are first class citizens.

Function Signature	Lambda expression	Example

`R f()`	`f=()-> R;`
	`f=()-> { return R;};`
		`f=()->5;`
		`f=()->{return  5;} ;`

`R f(T1 a)`	`f=(a)-> R`	`f=a->a+5;`
		`f=(Double a)->5;`
		`f=(Double a)->{return a+5;} ;`
		`f=(a)->{return a+5;} ;`

`R f(T1 a, T2 b)`	`f=(a,b)-> R`	`f=(Double a, Double b)-> a+b;`
		`f=(Double a, Double b)-> a+b;`
		`f=(Double a, Double b) ->{return a+b;} ;`
		`f=(Double a, Double b) ->{return a+b;} ;`

`void  f()`	`f=( )-> R`	`f=()->{   }`
`void  f(T1 a)`	`f=(T1 a)-> R`	`f=(Double a)->{   }`

Built-in Functional Interfaces in Java

Java contains a set of functional interfaces designed for commonly occurring use cases, so we do not have to create our own functional interfaces for every single use case. Those are residing in the Package java.util.function. Here are the most important and most used :

`Function<T, R>` (docs.oracle)

The most common by far will be the `Function<T, R>`, the declaration of which you can see using the decompiler :

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

`Function<T, R>` represents functions and methods that look like this `R f(T t)`, taking one argument and returning one result. We cannot use this for our `discountStrategy` because `discountStrategy` has two arguments.

Visually this could be represented by just an arrow: $T \xrightarrow{f} R$

`BiFunction<T, U, R>` (docs.oracle)

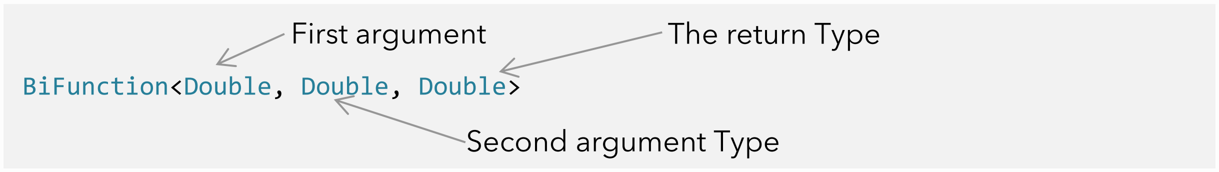
Luckily, java provides another build in functional interface for our case called `BiFunction` with the following declaration

```java
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

Using this we can define a local function using lambda expression and assign it to a `BiFunction` like this:

```java
BiFunction<Double, Double, Double> discountStrategy =
                            (discount, price) -> price - discount * price;
```

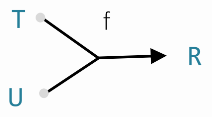
The `BiFunction<Double, Double, Double>` represents a function Type that has the following signature `R f(T t1, U t2)`.



```java
BiFunction<Double, Double, Double>
```

And we can assign a lambda expression directly to a `BiFunction <>`

If you wanted to visualize this could be a join:



Consumer<T> (docs.oracle)

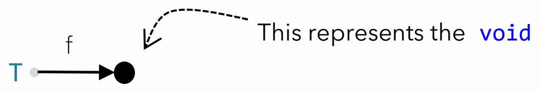
The `Consumer<T>` Functional interface represents functions that have this signature `void f(T t)`. In functional programming we usually do not like the `void` because it is not a Type like all the others and this forces us to treat it differently. `Consumer<T>` is just `Function<T, void>` but because void is not a Type we have to invent `Consumer<T>`. As an example of usage look at the following, in which we "consume" a string returning nothing:

```java
Consumer<String> print = (String x)-> System.out.println(x);
```

When you return void is an indication that you create a side effect of some sort. Using the method reference we can shorten the declaration :

```java
Consumer<String> print = System.out::println;
```

If you wanted to visualize this:

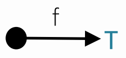


Supplier<T> (docs.oracle)

The `Supplier<T>` Functional interface represents functions that have this signature `T f()` that don't take any arguments. A simple example would be:

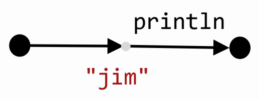
```java
Supplier<String> getName =()->"jim";
```

visualize this as a value coming out of nothing:



`Supplier` is the dual of `Consumer` and in this way we can compose them and annihilate them into a `void`.

```java
Consumer<String> print1 = System.out::println;
Supplier<String> getName =()->"jim";

print.accept(getName.get());// prints Jim
```



Predicate<T> (docs.oracle)

The `Predicate<T>` Functional interface represents functions that have this signature `boolean f(T t1)` that returns a `boolean`. Predicates are perfect for Filtering items in various data structures with Arrays probably the most common case. For example, filtering products based on price:

```
Predicate<Product> highPriced = product -> product.getBasePrice() > 1000;

var premiumProducts = Stream
                                .of(new Product(100), new Product(1100))
                                .filter(highPriced);
```

Summury Table

`Consumer<T1>`	`void    f(T1 a){…}`
`BiConsumer<T1,T2>`	`void    f(T1 a, T2 b){…}`
`Supplier<TResult>`	`TResult f(){…}`
`Function<T1,TResult>`	`TResult f(T1 a){…}`
`BiFunction<T1,T2,TResult>`	`TResult f(T1 a, T2 b){…}`
`Predicate<T>`	`boolean f(T1 a){…}`

Remember :

1. The `Consumer<>` only have inputs.
2. The `Function<,>` is the general case.
3. If you only want to return a result`:TResult` you must use a `Supplier<TResult>`.

Functors

« Thinking is necessary in order to understand the empirically given,
and concepts and "categories" are necessary as indispensable elements of thinking »
A. Einstein

The Idea:

In Java the most famous functional programming idea is to use `Stream.map` to replace iterations instead of for loops in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section.

"Practically a Functor is anything that has a valid `.map()` method"

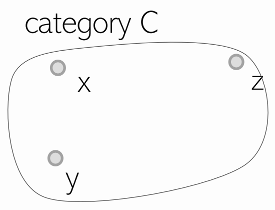
Functors can be considered the core concept of category theory.

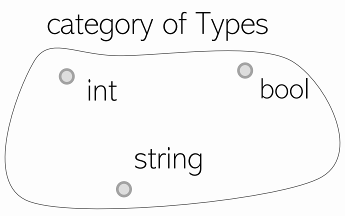
Categories

Category Theory is a mathematical discipline with a wide range of applications in theoretical computer science. Concepts like *Category*, *Functor*, *Monad*, and others, which were originally defined in Category Theory, have become pivotal for the understanding of modern Functional Programming (FP) languages and paradigms. The fundamental concept of category theory is unsurprisingly the concept of a **Category**.

A **Category** *C* consists of the following mathematical entities:

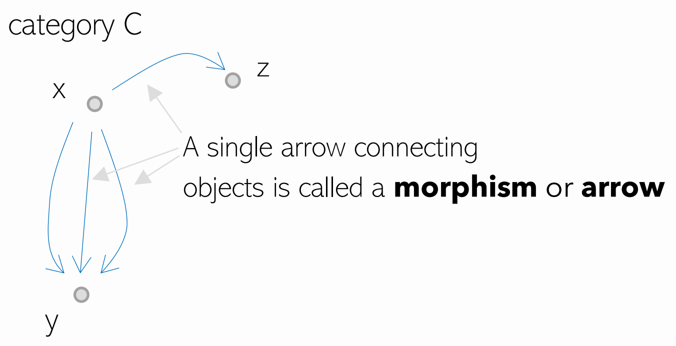
1. A collection of ***objects***. Any object-oriented programmer would find this as a great way to start a definition.





Our favourite category in programming is the **category of Types** : int, bool, char, etc. There are many interesting categories in programming and in this book, we will explore some of them.

2. A collection of **arrows** between objects (also called **morphisms**).


category C
x
z
y
A single arrow connecting objects is called a **morphism** or **arrow**

For our **Type** category, any arrow from **int** → **bool** for example represents **functions** that take an integer and return a Boolean. Some Java lambda expression examples of arrows might be the following:

```
var isEven =a->a%2;
```

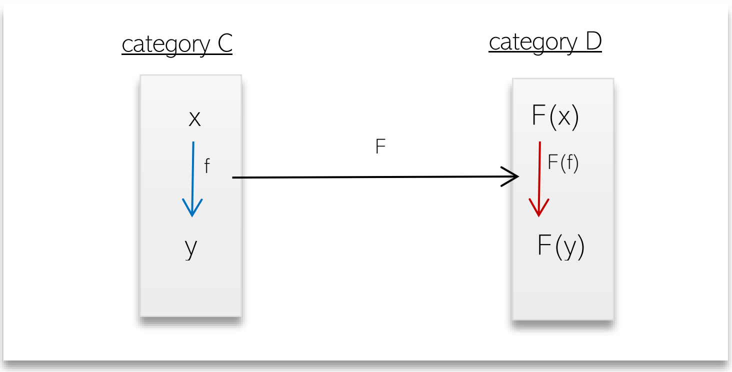
Or this one:

```
var isLessThan10 =a->a<10;
```

The basic focus of category theory ***is the relations between objects*** and not the objects per se, in contrast with the Set theory that primarily focuses on sets of objects. Functional programmers quickly endorsed this unique perspective of category theory. Now let us move to the core idea of this chapter. Functors.

2 Functors

In mathematics, a **functor** is a map between categories.



This Functor F must fulfil two requirements.

1. Should map each object x in *C* with an object F(x) in *D*,
2. Should map each arrow f in *C* with an arrow F(f) in *D*

For object-oriented programming, the best metaphor for functors is a ***container***, together with a **mapping** function. The Java Stream as a **data structure**, ***together*** with its **map** method is a Functor.

2.1 The Identity Functor

We will start by looking at the minimum structure that qualifies as a functor in Java:

```
public class Id<T>
{
    T value;
    public Id(T v) {  this.value = v;  }

    public <R> Id<R> map(Function<T,R> f)  {
        return new Id<R>(f.apply(this.value));
     }
}
```

Run This: Fiddle

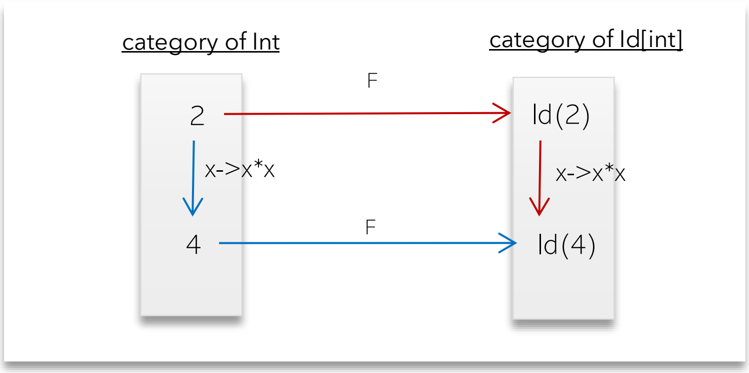
This is the minimum construction that we could call a functor because it has exactly two things

1. A "**constructor**" that maps a value `T v` to an object `Id<T>`
2. and it has a **mapping** method `map<T1>(Function<T, T1> f)` that lifts (maps)functions f

```
public class Id<T>
{
    T value;                                    This is requirement 1
    public Id(T v) {  this.value = v;  }
                                                    This is requirement 2
    public <R> Id<R> map(Function<T,R> f)  {
        return new Id<R>(f.apply(this.value));
     }
}
```

Because it's the minimal functor structure it goes by the name **<u>Identity functor</u>**.

Let us see a simple example where we have two integers 2 and 4 (here we take for simplicity the category of integers as our initial category C) also in this category there is the function `square  = x->x*x` that maps 2 to 4.



If we apply the Id(_) constructor we can map each integers to the Id[int] category. For example 2 will be mapped to **Id(2)** and 4 maps to **Id(4)**, the only part missing is the correct lifting (mapping) of the function f **Id[f]** to this new category. It's easy to see that the correct mapping is:

```
public <R> Id<R> map(Function<T,R> pattern)  {
        return new Id<R>(pattern.apply(this.value));
}
```

Because intuitively preserves the structure of the numbers (we will discuss this more next):

```
Function<Integer,Integer> square =x->x*x;

Id.of(value).map(square).getObject() == Id.of(square.apply(value)).getObject();
```

Run This: Fiddle

The Id<T> is very simple and doesn't do many things, but allows us for example to chain computations using sequential .map() computations :

```
var nameUpperCase=new Id<>(new Client(1,"jake"))
                         .map(x->x.getName())
                         .map(x->x.toUpperCase())
                         .getObject();
```

Run This: Fiddle

2.1.1 Maybe Functor Example with java Optional

The core example case upon which we are going to build the various ideas in this book is a simple retrieval of some Client from a Repository that matches a specific Id. The mock repository with an in-memory storage of the Clients in a simple Array might look like this:

```
public class MockClientRepository   {

    private Client[] clients;

    public MockClientRepository() {
        this.clients = new Client[]{
                new Client(1, "jim",1),
                new Client(2, "john",1)};
    }

    public Optional<Client> getClientById(int id) {
        return Stream.of(clients)
                .filter(client -> id == client.getId())
                .findAny();
    }
}
```

```
Optional<T> findAny();
```

Run This: Fiddle

The getClientById use the filter of the stream which has an Optional<> return type.If there is no client for a specific id, this should return Optional.empty.

Now we can use this repository in some Spring MVC controller:

```
public class ClientController {

    MockClientRepository ClientRepository = new MockClientRepository();

    public String getClientNameById(int Id) {
        return ClientRepository
                .getClientById(Id)
```

```
                    .map(client -> client.getName())
                    .orElse("no client found");
        }
    }
```

Run This: Fiddle

2.1.2 Optional in Vavr.io aka Option

The usual name for this functor F=1+X (disjoint union with a base point) in most functional languages is called **Maybe,** in java Optional and in Vavr.io is called **Option** which is another common naming for Maybe that comes closer to the OOP paradigm. Let us agree that **Maybe** is the name of the concept and **Option** an implementation of Maybe. It is the same thing. For the rest of this book **we will mostly use vavr.io Option** monad in our code examples.

we can get a Vavr Option by a java Optional, using the `Option.ofOptional`

The `.ofOptional` converts an `Optional` to an `Option`

```
Option<Client> optionClient = Option.ofOptional(
                                      Optional.of(new Client(1, "jim", 1))
                                    );
```

The converse transformation from `Option` to `Optional` is also available :

```
Optional<Client> optionalClient = optionClient.toJavaOptional();
```

this pair of functions allows you to jump between the two types at will.

$$\begin{tikzcd} \mathtt{Optional\langle T\rangle} \arrow[r, bend left, dashed, "\mathtt{.ofOptional()}"] & \mathtt{Option\langle T\rangle} \arrow[l, bend left, dashed, "\mathtt{.toJavaOptional()}"] \end{tikzcd}$$

With no surprise there is the standard `map` method now there is a `match` method that is called `fold:`

```
Option<Client> maybeClient = Option.some(new Client(1, "jim", 1));
Option<String> maybeName = maybeClient.map(Client::getName);

var clientName = maybeName.fold(
```

Pattern match

```
                () -> "no client ",
                (name) -> name
        );
```

Also, you can use the `getOrElse` methods like the Java `Optional`

```
String nameOrAlternativeMessage = maybeName.getOrElse(() -> "no client ");

String nameOrAlternativeMessage = maybeName.getOrElse("no client ");

String nameOrAlternativeMessage = maybeName.get();
```

Run This: Fiddle

2.1.3 Maybe Functor Example with Vavr.io Option

We can refactor now the `Client` repository example from the previous section by having the repository return an `Option<Client>`

```
public class MockOptionClientRepository {

    private Client[] clients;

    public MockOptionClientRepository() {
        this.clients = new Client[]{
                new Client(1, "jim",1),
                new Client(2, "john",1)};
    }

    public Option<Client> getClientById(int id) {

        var t = Stream.of(clients)
                .filter(client -> id == client.getId())
                .findAny();

        return Option.ofOptional(t);
    }
}
```

We now return an `Option<Client>`

Run This: Fiddle

Now in order to fold this into a string that contains either the name of the client or a static error we will use the `.map` () and `.fold` () methods of `Option`:

```
 public class ClientOptionController {

   MockOptionClientRepository ClientRepository = new MockOptionClientRepository();

    public String getClientNameById(int Id) {
        return ClientRepository
                .getClientById(Id)
                .map(client -> client.getName())
                .fold(
                        () -> "no client ",
                        (name) -> name
                );
    }
}
```

Run This: Fiddle

2.2 Combining `Future` and `Option` - `Future<Option<T>>`

If we rewrite the case of the previous section using the Vavr's `Option<>` instead of java's Optional<> we have the following computation inside the controller:

```
public class ClientFutureOptionController {

public CompletableFuture<String> getClientNameByIdAsync(int Id) {
 return ClientRepository
     .getClientByIdAsync(Id)
     .thenApply(clientOption -> clientOptional.map(client -> client.getName()))
     .thenApply(clientOption -> clientOptional.fold(
                        () -> "no client ",
                        (name) -> name
                ));
    }
 }
```

Run This: Fiddle

We can also use some static functional extensions (that move the expressions one level deeper, `mapT, foldT)`

```
public class OptionExtensions {

public static <T, T1> Function<Option<T>, Option<T1>> mapT(Function<T, T1> fn) {
        return optional -> optional.map(fn);
    }

public static <T, T1> Function<Option<T>, T1> foldT(Supplier<? extends T1> leftMap
per, Function<? super T, ? extends T1> rightMapper) {
        return optional -> optional.fold(leftMapper, rightMapper);
    }
}
```

Run This: Fiddle

These static methods can shorten the expressions :

```
public CompletableFuture<String> getClientNameByIdAsync(int Id) {
        return ClientRepository
                .getClientByIdAsync(Id)
                .thenApply(mapT(Client::getName))
                .thenApply(foldT(
                        () -> "no client ",
                        (name) -> name
                    ));
    }
```

Run This: Fiddle

Monads

«Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem? »
-*Brief, Incomplete and Mostly Wrong History of Programming Languages*,

3 Monads

3.1 Validation Monad

In this this section we are going to take a brief look at the validation type. The Validation behaves like the Either but also **has an operation to accumulate validation errors**. Firstly, we could rewrite the validation example to use `Validations` instead of Either. In the place of Left we use the `va invalid lid` and in the place of a Right we use a `valid`:

```
Function<String, Validation<TestError, String>> isValidName = (name) ->
              CharSeq.of(name).replaceAll(VALID_NAME_CHARS, "")
                  .transform(seq -> seq.isEmpty()
                      ? Validation.valid(name)
                      : Validation.invalid(
                           new TestError("Name contains invalid characters"))
                   );

Function<Integer, Function<String, Validation<TestError, String>>>
 isValidMaxLenght = maxLenght -> str ->
                           str.length() <= maxLenght ?
                            Validation.valid(str) :
                            Validation.invalid(new TestError("failed max length")
                       );


String nameIfBothValidOrElseError = isValidMaxLenght.apply(3).apply("Nick")
                            .flatMap(name -> isValidName.apply(name))
                            .fold(
                                    (error) -> "error: " + error.getMessage(),
                                    (name) -> name
                            );
```

Run This: Fiddle

Clean Architecture in Java

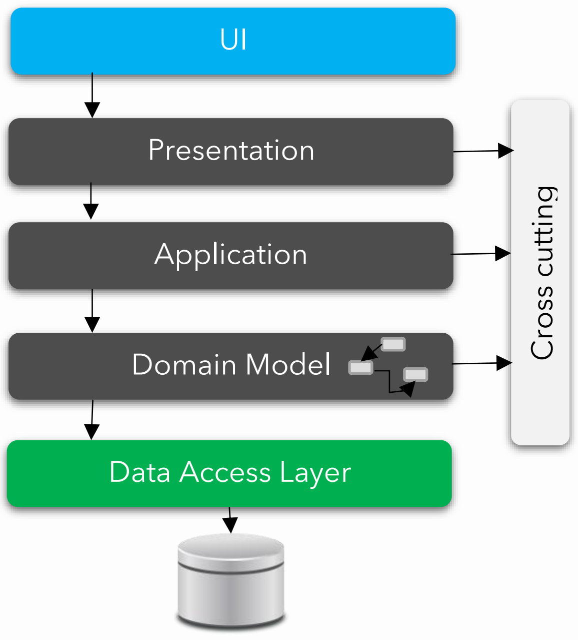
In this Section we are going to take a brief look at the possibility of a **Functional Software Architecture** and how can be integrated with the latest trends in **modern software architectures**. This is not a deep dive in Architectural Design but a discussion behind the architecture of the sample spring boot web Application.

4 A Clean Functional Architecture Example

4.1.1 Clean Architecture with Spring Boot

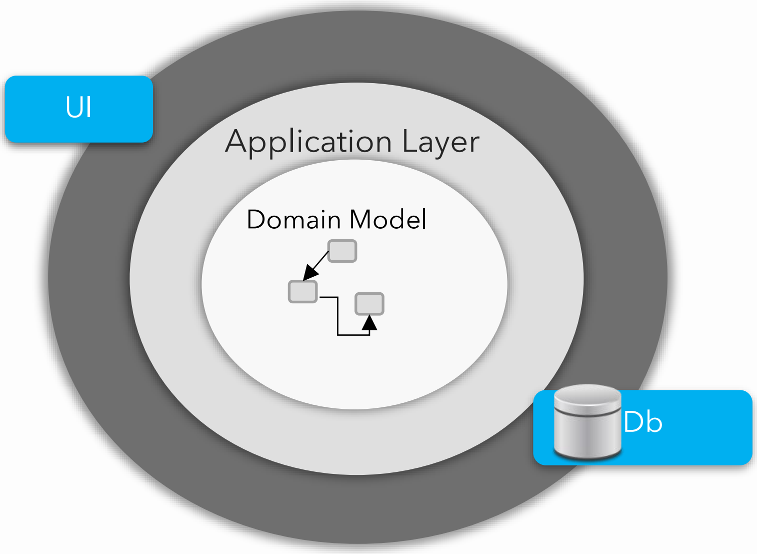
The architecture of web applications depends on the scale and scope of the application. The Previous decade the dominant architecture for small and medium web applications was the **Layered Architecture.** A Layered Architecture, organize the project structure into four main categories:

1. Presentation-UI
2. application
3. domain
4. and infrastructure



Under the weight of ideas such as Domain -Driven Design from Eric Evans and Clean Code from "Uncle Bob" - Robert C. Martin the focus of the Architecture became the separation of concerns of the **Domain** from all the technicalities such as technologies, tools and implementation details.

The domain and its Use Cases now has been placed in the **centre** of the architecture design with minimal dependencies. This led to a series of evolutionary transformations of the Layered structure (Hexagonal Architecture, Pipes and adapters , DDD etc) leading at to the concept of Clean architecture.



With Clean Architecture, the **Domain** and **Application** layers are at the centre of the design. This is known as the **Core** of the system.

The **Domain** layer contains enterprise logic and types and the **Application** layer contains business logic and types. The difference is that enterprise logic could be shared across many systems, whereas the business logic will typically only be used within this system.

***Core** should not be dependent on data access and other infrastructure concerns*, so those dependencies are inverted. This is achieved by adding interfaces or abstractions within **Core** that are implemented by layers outside of **Core**. For example, if you wanted to implement the Repository pattern you would do so by adding an interface within **Core** and adding the implementation within **Infrastructure**.

All dependencies flow inwards, and **Core** has no dependency on any other layer. **Infrastructure** and **Presentation** depend on **Core**, but not on one another.

4.2 A Functional Applications Architecture

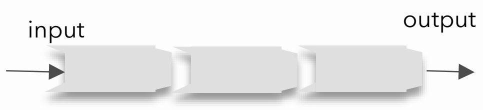
Where does Functional Programming fit in the larger Architectural view? In the article form Scott Wlaschin "A primer on functional architecture" he recognizes three principles of functional programming that can be applied to the architectural level:

1. The first is that **functions** are standalone values. In a functional architecture, the basic unit is also a function, but a much larger business-oriented one that he calls a **workflow**.

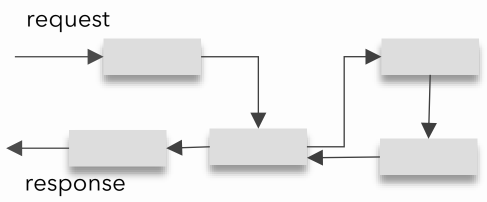
2. Second, **composition** is the primary way to build systems. Two simple functions can be composed just by connecting the output of one to the input of another.

3. functional programmers try to use **pure functions** as much as possible.

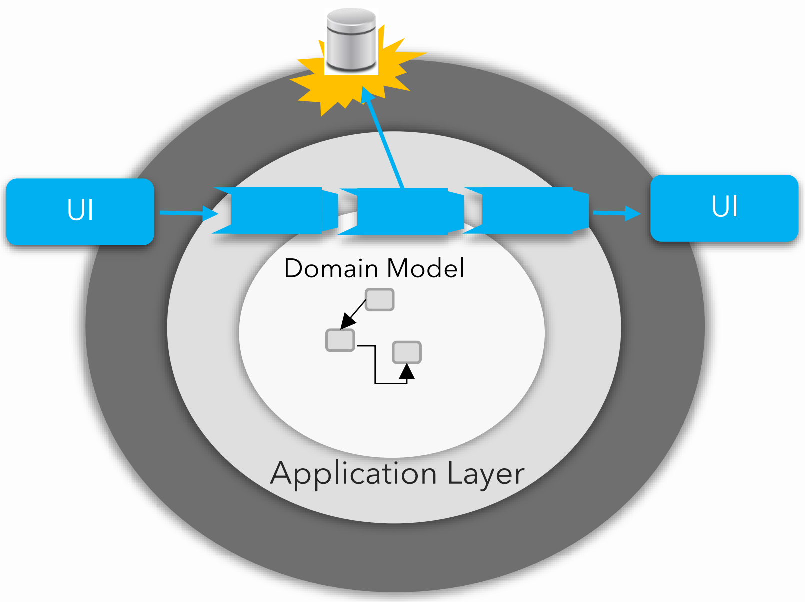
The general idea is that you write as much of your application as possible in an FP style chain computation using the Task, Option, Either, Validation Monads or their combinations and avoiding side effects and coupling. This would lead to the usual pipeline computation style:



Instead of the chaotic object oriented:



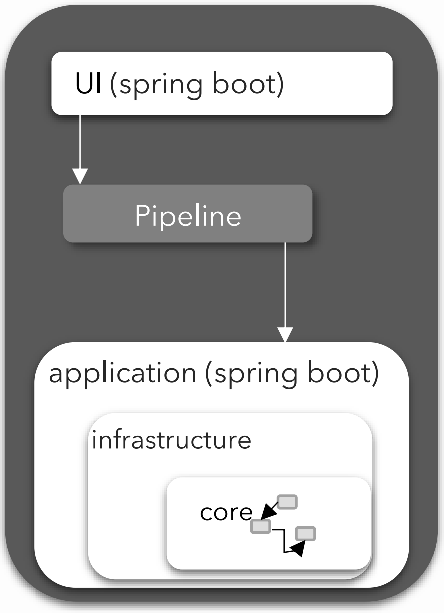
The idea for a functional architecture as laid out by Wlaschin is a pipeline that goes through the layers of the Architecture with minimal coupling and side effects.


UI
UI
Domain Model
Application Layer

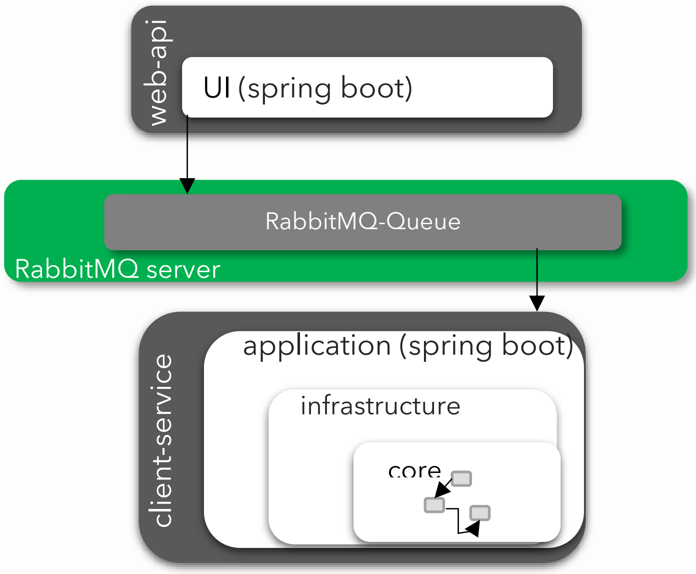
4.3 Download and Setup the Project

All projects target **openjdk-14.0.2** and most of them import vavr.io library through maven **io.vavr:vavr:1.0.0-alpha-3**. You can open and run the projects using IntelliJ community edition. There are 3 different versions:

- A Sample monolith application Spring Boot WebAppExample where everything is in a single Spring boot project (**ui, application, infrastructure, core**) communicating through a Pipeline design pattern implementation (PipelinR)



- A Distributed Project that consists of two sub-systems (**ui**) and (**application**, **infrastructure, core**) communicating through RabbitMQ
- And a Distributed Project that consists of two sub-systems as before, but now (**application**, **infrastructure, core**) are all in different **modules**.



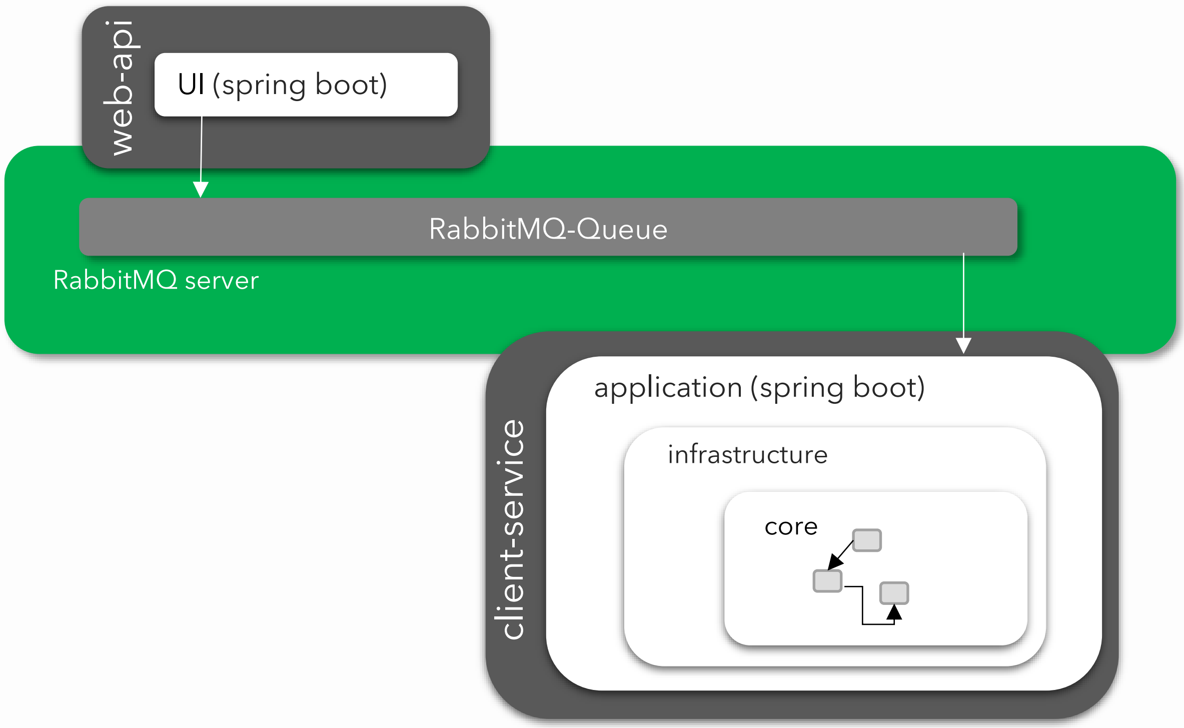
Here we will focus mainly on the last Distributed Project example of the architectural evolution. Nonetheless, we will also briefly explore the Mediator and a Pipeline design pattern.

The Functional styles and implementations are kept intact in all the above implementations. The point we try to make is that the Functional style can be used in any stage of the architectural design of a project.

4.4 Clean Architecture with Spring Boot and Vavr.io

The Distributed-SpringBoot-CleanArchitecture application **approximates a Functional architecture Style embedded in a Clean architecture**.

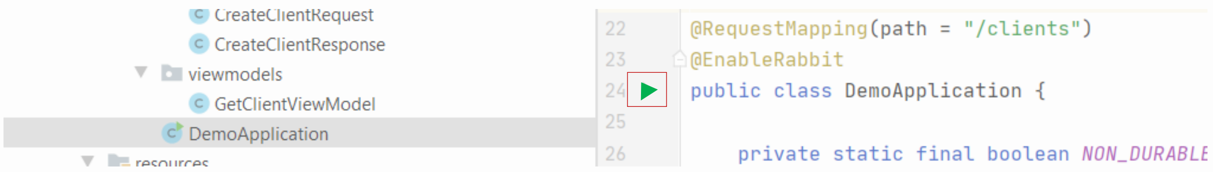
The application consists of four projects the three of them are libraries targeting the .NET standard 2.0 framework (application, core, infrastructure) and the "UI" Project named **web-api** is a Boot Spring MVC Web application Maven project generated by Spring Initializr. that is completely independent.



4.5 How to Run the project using IntelliJ community edition

This section is intended to guide you through the process of debugging the full sample. If you do not want to run the application, then you can just skip this section.

The system is consisted of two parts the web-api application and the clients-service application. You can open and run each one separately using the IntelliJ community edition.



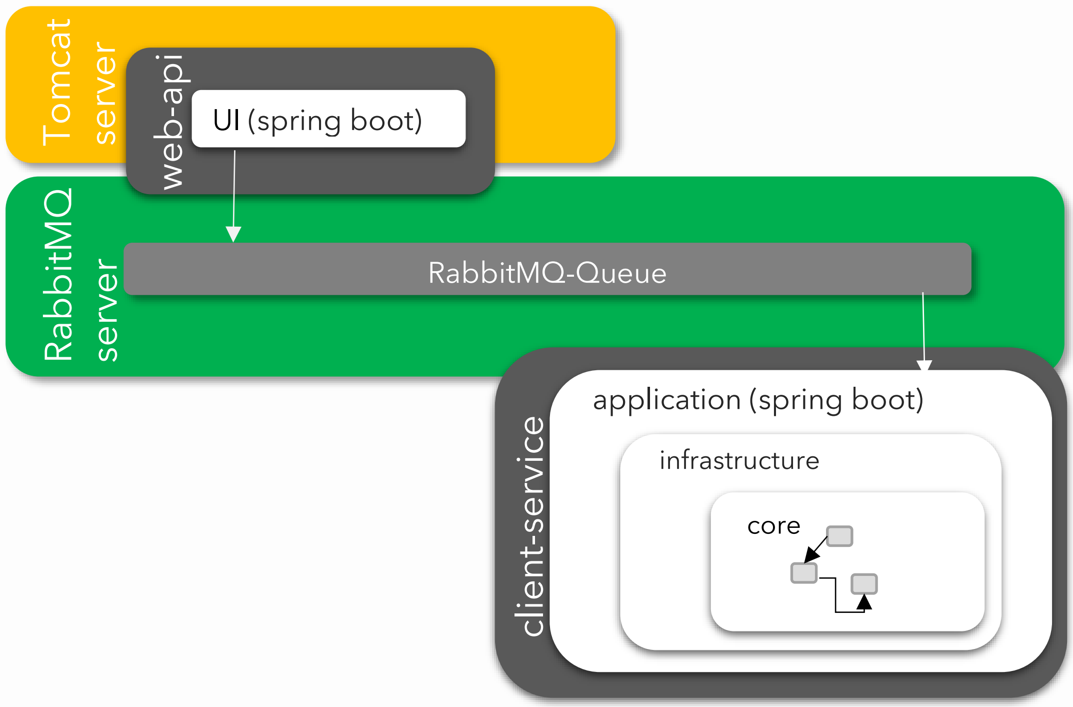
```
@RequestMapping(path = "/clients")
@EnableRabbit
public class DemoApplication {

    private static final boolean NON_DURABLE
```

o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''

RabbitMQ server

In order to be able to use the sample you must have installed the RabbitMQ server to the machine you plan to run the sample .There are various downloads on the RabbitMQ site. Unfortunately, you should also have installed Erlang. The RabbitMQ installation will ask you to install erlang if you have not.

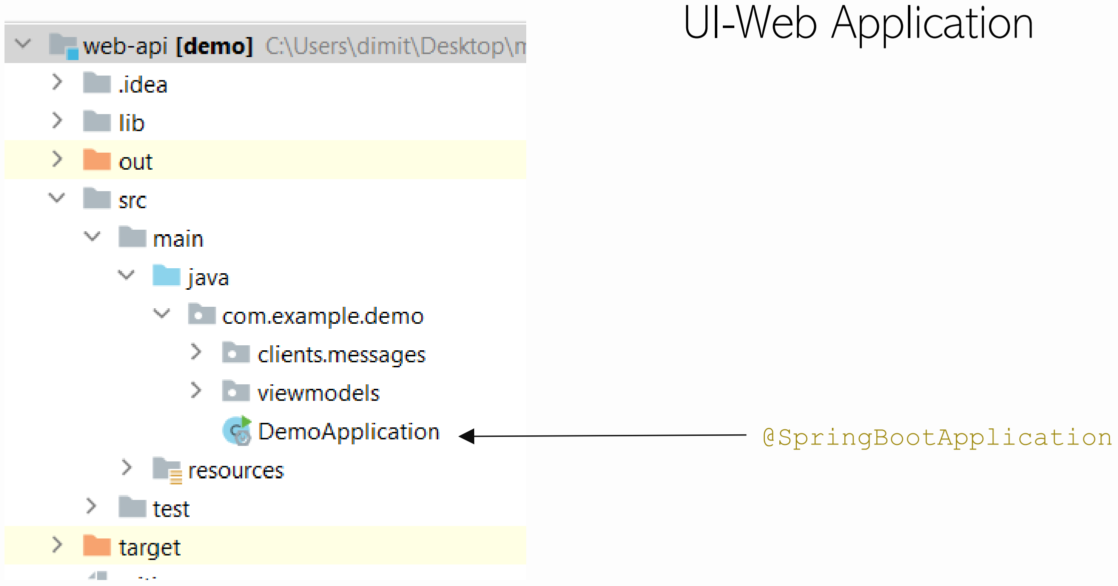
If the RabbitMQ server is not running you will get an error log from the web-api project when you try to run it.



4.6 Web API

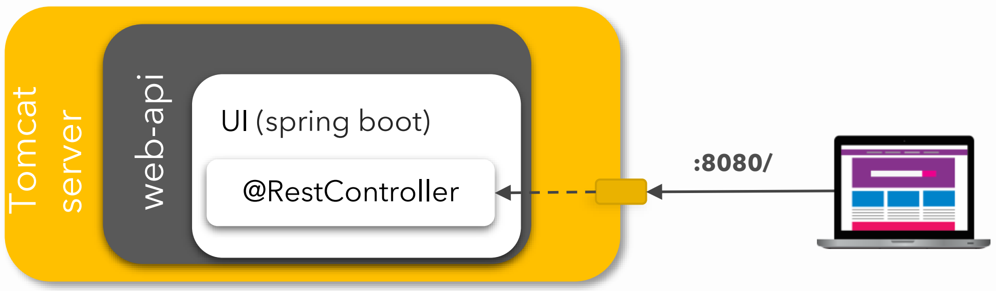
This web-api project is a single Spring MVC boot project. The project has a single `@RestController`. The web-api project have a single responsibility, to handle the HTTP requests received from the web browser and to return the HTTP responses.

UI-Web Application

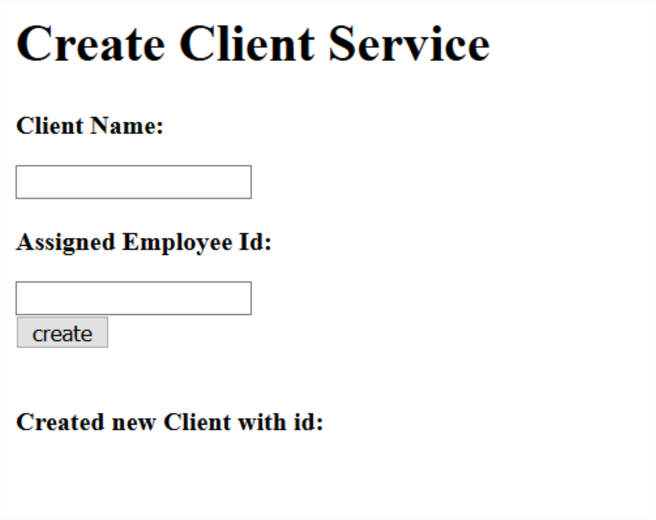


We will handle exceptions and errors that have occurred in the Domain or Data projects to effectively communicate with the consumer of APIs. This communication will use HTTP response codes and any data to be returned located in the HTTP response body.

When you run the application and you try to browse the **http://localhost:8080/** from any browser the Spring Application will serve the default index.html.



This looks like this. A very simple unstilled html page index.html



Create Client Service

Client Name:

Assigned Employee Id:

create

Created new Client with id:

In this SpringBootApplication, the routing is configured by Attribute Routings and more specific with the decorating @RequestMapping Attribute.

```
@SpringBootApplication
@RestController
@RequestMapping(path = "/clients")
public class DemoApplication {

    @GetMapping("/get/{clientId}")
    public ResponseEntity<SearchViewModel> get(@PathVariable Integer clientId)
    {
        ...
    }


    @PostMapping("/create")
    public ResponseEntity<CreateClientResponse> create(@RequestBody CreateRequest
createRequest)
    {
        ...
    }
}
```

Github: source

The above REST controller covers the /clients URL paths and we can call them from the front-end side. For example we can call the /clients/get/1 using the Fetch-API :

```
fetch(`/clients/get/${clientId}`, { method: 'GET' })
```

Github: source

This will reach the @RestController at the get method directed by the @GetMapping

```
@GetMapping("/get/{clientId}")
public ResponseEntity<SearchViewModel> get(@PathVariable Integer clientId)
{
        ...
}
```

Similarly for making a POST http request at `/clients/create/` using the Fetch-API we can write :

```
fetch(`/clients/create/`,
                {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify({name: '' , employeeId: 1})
                })
```

The data are send in the http `body` as a `Json` String

Github: source

This will reach the `@RestController` at the `create` method where the Json will be deserialized as `CreateRequest` instance thanks to the `@RequestBody` attribute

```
@PostMapping("/create")
public ResponseEntity<CreateClientResponse> create(@RequestBody CreateRequest crea
teRequest)
{
   ...
}
```

The http `body` json string will be deserialized into a `CreateRequest`

Of course, we could use alternatively jQuery, Angular, or any other JavaScript framework to make those http calls to the `@RestControllers`.