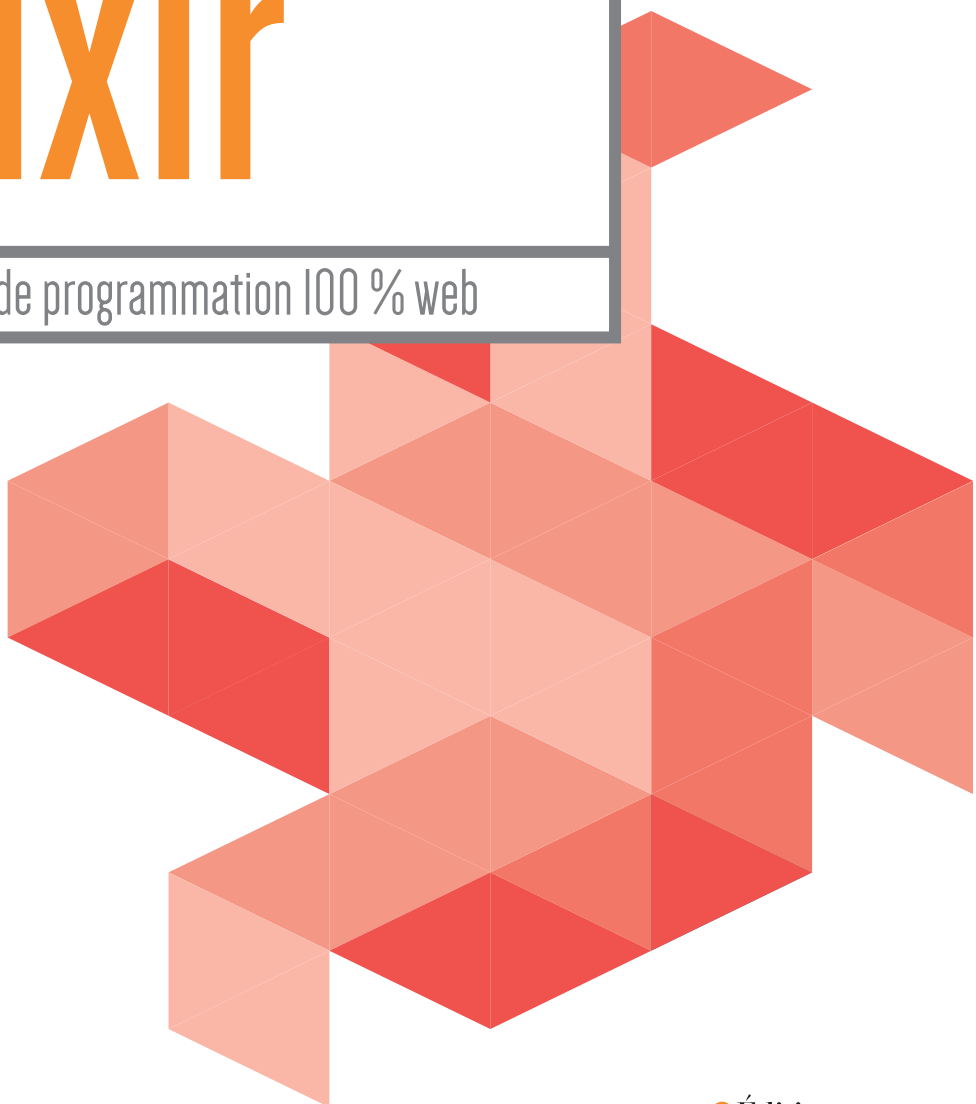
Vincent Le Goff

Elixir

Un langage de programmation 100 % web



Éditions EYROLLES

DANS LA MÊME COLLECTION

H. BERSINI, K. HASSELMANN. – **L'intelligence artificielle en pratique avec Python.**
N°0101094, 2e édition, 2023, 176 pages.

E. SARRION. – **JavaScript – Vue.js côté client et Node.js/MongoDB côté serveur.**
N°0100836, 2022, 240 pages.

E. BOURREAU, G. FLEURY, P. LACOMME. – **Introduction à l'informatique quantique.**
N°0100653, 2022, 424 pages.

E. GASSARA. – **Docker/Kubernetes.**
N°0100569, 2022, 200 pages.

Y. BENZAKI. – **Les data sciences en 100 questions/réponses.**
N°67951, 2020, 126 pages.

K. NOVAK. – **Administration Linux par la pratique – Tome 2.**
N°67949, 2020, 418 pages.

C. DELANNOY. – **Le guide complet du langage C.**
N°67922, 2020, 876 pages.

J. LORIAUX, T. DEFOSSEZ. – **Emailing : développement et intégration.**
N°67849, 2020, 160 pages.

B. BARRE. – **Concevez des applications mobiles avec React Native.**
N°67889, 2e édition, 2019, 224 pages.

S. RINGUEDE. – **SAS.**
N°67631, 4e édition, 2019, 688 pages.

C. BLAESS. – **Développement système sous Linux.**
N°67760, 5e édition, 2019, 1080 pages.

T. PARISOT. – **Node.js.**
N°13993, 2018, 472 pages.

Retrouvez nos bundles (livres papier + e-book) et livres numériques sur
http://izibook.eyrolles.com

Vincent Le Goff

Elixir

Un langage de programmation 100 % web

Éditions
EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Mise en pages : Gaël Thomas

Depuis 1925, les éditions Eyrolles s'engagent en proposant des livres pour comprendre le monde, transmettre les savoirs et cultiver ses passions !

Pour continuer à accompagner toutes les générations à venir, nous travaillons de manière responsable, dans le respect de l'environnement. Nos imprimeurs sont ainsi choisis avec la plus grande attention, afin que nos ouvrages soient imprimés sur du papier issu de forêts gérées durablement. Nous veillons également à limiter le transport en privilégiant des imprimeurs locaux. Ainsi, 89 % de nos impressions se font en Europe, dont plus de la moitié en France.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.

© Éditions Eyrolles, 2023, ISBN : 978-2-416-01175-7

Table des matières

PREMIÈRE PARTIE

Syntaxe du langage Elixir

1

Premiers pas avec Elixir

Sur le chemin de votre nouveau réseau social, jeu multijoueur ou service de streaming avec plusieurs millions d'utilisateurs chaque jour, il y a un début, comme pour tout. Dans le cas présent, nous allons d'abord devoir comprendre la syntaxe du langage Elixir avant de nous attaquer, petit à petit, à des questions plus complexes. La bonne nouvelle est que tout ce que vous trouverez dans cette première partie vous servira, et même beaucoup !

Pourquoi utiliser Elixir ?

Il existe beaucoup de langages de programmation et leur nombre augmente presque chaque jour, ou du moins chaque semaine. Bien sûr, certains disparaissent également. Certains parviennent à rester populaires pendant des décennies, alors que d'autres ne feront jamais surface. Parmi tous ces langages pourquoi choisir d'apprendre Elixir plutôt qu'un autre ?

La raison principale se trouve dans les promesses du langage, dont certaines vont être détaillées ici.

Une disponibilité permanente

C'est bien de créer un logiciel, mais c'est encore mieux s'il peut communiquer avec son environnement. Par « environnement », il faut comprendre ici les logiciels qui tournent sur votre machine, votre réseau local ou même d'autres machines accessibles par Internet. La base d'Elixir, Erlang, a été créée pour remplir des objectifs de disponibilité dans un univers connecté. Aujourd'hui, cet univers connecté prend de plus en plus d'importance.

Mais la disponibilité n'est pas seulement l'aptitude à communiquer en permanence.

La tolérance aux erreurs

Il n'est pas nécessaire d'être ingénieur en astrophysique pour reconnaître qu'un logiciel, ou un système informatique, ne se comporte pas toujours comme il le devrait. Les bogues sont monnaie courante et peuvent faire qu'un logiciel se ferme plus ou moins brutalement, parfois c'est toute la machine qui cesse de fonctionner… Sans parler des pannes de courant, des problèmes de réseau Internet, de logiciels tellement gourmands en ressources qu'ils perturbent le bon fonctionnement des autres programmes.

Et alors le système plante. Avec un peu de chance, vous trouverez des solutions pour résoudre ce type d'erreur, créer des copies de sauvegarde des fichiers sur lesquels vous étiez en train de travailler, enregistrer là où vous en étiez dans un jeu vidéo ou encore récupérer les pages web ouvertes… Mais le fait demeure : il s'agit d'une interruption de service – parfois longue –, sur laquelle vous n'avez pas toujours de contrôle.

Le langage Erlang, sur lequel est basé Elixir comme mentionné précédemment, part du principe qu'il est inévitable que des erreurs se produisent, mais que ces dernières ne devraient pas entraîner une interruption de service. Dans le pire des cas, une opération rencontre une erreur inattendue, indique qu'elle ne peut continuer et le système réessaye.

L'idéal ne serait-il pas d'avoir un système sans erreur ? Sans doute, mais je n'ai encore jamais rencontré un tel système, capable de survivre à une erreur de connexion.

Mais… si je fais tourner mon programme sur mon ordinateur et que l'électricité est coupée, n'y a-t-il rien à faire pour maintenir cette fameuse disponibilité ?

La disponibilité présente certes des limites. En revanche, Erlang permet, très simplement, de partager un système sur plusieurs machines. L'avantage de ce système est que si l'une d'elles disparaît, Erlang peut alors confier la tâche à une autre machine. Dans un tel cadre, il faudrait que toutes les machines soient indisponibles pour faire tomber le système entier… ce qui peut arriver cependant si l'alimentation électrique est coupée pour toutes les machines par exemple !

En extrapolant, même si le logiciel développé tourne sur une centaine de machines dispersées dans le monde entier, il est toujours possible qu'une météorite affecte la disponibilité globale du service. Mais toutes proportions gardées, cette notion de partage des tâches entre plusieurs machines joue un grand rôle dans la gestion des erreurs. Même si une partie du programme plante pour une raison ou une autre, il peut être redémarré et reprendre sa mission sans affecter les autres parties.

La distribution

Comme nous venons de le voir, si un logiciel peut travailler avec plusieurs machines, il peut survivre à de nombreuses erreurs. Chaque machine possède les informations nécessaires pour faire fonctionner le logiciel dans son ensemble : si l'une d'elles s'interrompt, une autre peut prendre sa place et les tâches sont alors réparties entre les machines restantes.

Elixir, grâce à Erlang, offre une gestion extrêmement transparente des opérations distribuées. Un programme écrit en Elixir est composé de très nombreuses opérations qui peuvent être exécutées sur une machine ou bien envoyées pour être exécutées par une autre. Le système reste identique, qu'il s'exécute sur une seule machine ou sur plusieurs.

Une maintenance efficace

Enfin, il est nécessaire de souligner un autre point, répondant aux objectifs d'Erlang (et d'Elixir) dès la création : un système en place devrait pouvoir être maintenu (mis à jour) sans provoquer d'interruption de service.

Vous avez certainement déjà vécu la situation où vous êtes en train de consulter sur votre site web préféré, quand apparaît soudain un message vous informant que le système est en cours de maintenance et que vous devrez revenir dans quelques heures pour continuer. Le monde dans lequel nous vivons étant de plus en plus connecté, la plupart des services tentent d'être toujours disponibles, même durant leur maintenance. Cependant, la mise à jour d'un service, qui nécessite généralement son redémarrage, peut déconnecter de très nombreux utilisateurs. Erlang a fait beaucoup d'efforts pour permettre la mise à jour du service sans affecter les utilisateurs et préserver sa disponibilité.

C'est une promesse assez audacieuse, comme vous pouvez l'imaginer ! Elle suppose un minimum d'implication de la part du développeur. Mais c'est aussi cette promesse, en particulier, qui retient l'attention de plus en plus de développeurs.

Le *no downtime* (« sans interruption » en français) a beaucoup gagné en popularité ces dernières années. Il existe plusieurs outils pour y parvenir et Erlang n'est pas le seul à offrir une alternative. Mais le fait qu'Erlang ait été conçu dans ce but dès le début, bien avant que cela ne devienne une nécessité pour beaucoup de services modernes, offre certains avantages.

Avantages

Comme nous venons de le voir, Erlang (et Elixir) propose donc de vous permettre de créer un service (un site web, un jeu vidéo multijoueur, un service de streaming, etc.) disponible en permanence, avec une grande réactivité et une garantie de mettre à jour ledit service sans déconnecter le moindre utilisateur. Il propose également de créer un système facile à distribuer : si votre site web reçoit 100 utilisateurs par jour et que ce nombre augmente, même brutalement, il n'est pas difficile d'ajouter de nouvelles machines qui peuvent aisément gérer la charge supplémentaire, sans interruption du service existant.

Inconvénients

Erlang est un langage puissant. Il existe depuis un moment et maintient certaines de nos infrastructures les plus critiques, celles qui ne doivent sous aucun prétexte s'interrompre. Mais ne comparons pas Erlang à une baguette magique : comme mentionné précédemment, il offre de nombreuses possibilités qui seront développées au fil de cet ouvrage, mais nécessite

également un bon investissement de la part du développeur. Si ce dernier décide de créer un service indisponible, il peut le faire, c'est sa responsabilité.

En outre, Elixir a été conçu pour simplifier le développement de logiciels avec Erlang. Cela étant, Elixir reste un langage de programmation et il ne figure pas encore parmi les plus simples à utiliser. Mais après avoir lu cet ouvrage et mis en pratique son contenu, vous serez en mesure d'utiliser Elixir pour développer au quotidien.

Un bref historique du langage

Vous comptiez apprendre un langage, Elixir, et je vous parle d'un autre, Erlang ! Il est temps de clarifier tout cela et d'aborder l'histoire de ces langages. Je vous propose un historique bref et très simplifié, car pour une introduction à Elixir, ne mentionner qu'Erlang reste assez rare dans les ressources que l'on peut trouver.

En 1986, la compagnie de télécommunications Ericsson publie sa première version du langage de programmation Erlang, avec de nombreux outils et une plate-forme compatible. Nous n'entrerons pas dans le détail ici, nous verrons tout cela en temps utile dans les prochains chapitres. Un point mérite d'être souligné cependant ici : Ericsson avait pour objectif de créer un système hautement disponible, résistant aux erreurs, distribué et facile à maintenir. Cela vous rappelle-t-il quelque chose ?

À l'époque, Ericsson avait besoin de ces caractéristiques pour développer un réseau téléphonique. De nos jours, cela peut sembler simpliste, mais un réseau téléphonique se doit d'être disponible en permanence. Les utilisateurs seraient en effet très mécontents si, au milieu de leur conversation téléphonique, un opérateur leur annonçait que le réseau doit être mis à jour et qu'ils doivent rappeler dans une heure ou deux. Il en va de même si un problème informatique ou mécanique cause un brusque retard dans la transmission du son d'un point à un autre. Ericsson aurait peut-être pu trouver une parade purement électronique. Cependant, la société de télécommunications a développé Erlang, qui répondait à ces problématiques.

Si Erlang a initialement été créé pour un réseau téléphonique, son champ d'application est en réalité bien plus vaste. Aujourd'hui, on peut se souvenir de son origine, des besoins soulevés et en être surpris (le *no downtime* était déjà une préoccupation au milieu des années 1980 !). Bien sûr, Erlang aurait pu tout simplement être considéré comme un langage de programmation pour la téléphonie, conçu pour répondre à un besoin spécifique et n'ayant aucune utilité dans les autres domaines. Mais cela n'a pas été le cas et Erlang est un langage encore très utilisé aujourd'hui.

Bien des années plus tard, en 2012, José Valim publie Elixir. Il s'agit d'un langage basé sur Erlang qui utilise la même plate-forme mais qui est bien plus simple à lire, à apprendre et à utiliser pour développer. Elixir est un traducteur qui intervient entre votre programme et Erlang. Cela veut aussi dire qu'en programmant avec Elixir, vous bénéficiez de la robustesse d'Erlang, ce qui est très utile.

Cet historique est extrêmement simplifié et contestable. En voulant faire beaucoup de simplifications, on frôle l'inexactitude. Comme dit précédemment, mentionner uniquement Elixir

et Erlang pour commencer est un raccourci : Erlang n'est pas tout jeune et il propose de nombreux outils que nous allons découvrir par la suite.

Installation

Commençons par installer Elixir.

Installer… un langage de programmation ? Eh oui.

Votre ordinateur utilise des logiciels. Cela ne signifie pas qu'il sache pour autant comment ils sont écrits. Prenons un exemple très simplifié : votre ordinateur ne comprend qu'une langue très basique, le binaire, constituée uniquement de zéros et de uns. Mais quand on rédige un programme informatique, il est assez rare d'écrire directement dans ce langage. On préfère écrire le programme d'une façon plus compréhensible pour nous.

Cependant, votre ordinateur ne sait pas comment lire ce programme écrit de façon « plus compréhensible pour nous ». Il connaît son langage et ne va pas apprendre tous les langages de programmation qui peuvent exister. C'est pour cela que nous avons besoin d'un logiciel : il faut convertir notre programme « humainement compréhensible » en une version « informatiquement compréhensible ».

Remarque

Notre programme est écrit dans le langage de programmation ciblé par le logiciel (Elixir, en l'occurrence) mais il ne sera pas converti en binaire directement. Là encore, il s'agit d'une simplification volontaire de ma part. Nous verrons par la suite ce qu'il en est réellement.

Sous Linux

Il existe différentes distributions sous Linux, et donc autant de possibilités pour installer Elixir. Voici les commandes à utiliser pour Debian (et Ubuntu). Si vous êtes sur une autre distribution, jetez un coup d'œil à la page suivante : https://elixir-lang.org/install.html.

```
wget https://packages.erlang-solutions.com/erlang-solutions_2.0_all.deb && sudo dpkg
-i erlang-solutions_2.0_all.deb
sudo apt-get update
sudo apt-get install esl-erlang
sudo apt-get install elixir
```

La première ligne de commande télécharge et installe une clé qui permet à `apt-get` d'accéder à la dernière version d'Erlang. `apt` est ensuite mis à jour, puis Erlang et ses dépendances sont installés. Enfin, la dernière ligne de commande installe Elixir.

Sous Windows

La méthode recommandée pour Windows est d'utiliser l'installateur officiel, disponible à l'adresse https://elixir-lang.org/install.html. Cliquez sur Distributions, puis faites défiler jusqu'à Windows. Cliquez alors sur le lien Download the installer afin de télécharger l'installateur. Exécutez-le et suivez les différentes étapes en conservant les options par défaut.

Remarque

Elixir peut également être installé via Chocolatey si celui-ci est présent sur votre système.

Pour ma part, je préfère passer par Windows Subsystem for Linux (WSL) qui permet d'exécuter une machine virtuelle (sous Linux) au sein même de Windows et offre une grande facilité d'échange entre votre système Windows et votre machine virtuelle Linux. J'utilise WSL au quotidien pour le développement et il est facile d'accéder à Elixir depuis ma console Ubuntu.

Sous macOS

Le site officiel d'Elixir propose deux options si vous êtes sous macOS.

1. Si vous utilisez Homebrew, il vous suffit d'entrer la commande suivante :

```
brew install elixir
```

2. Si vous utilisez Macports, saisissez :

```
sudo port install elixir
```

Une fois l'installation terminée

Quel que soit votre système d'exploitation, une fois l'installation terminée, vous devriez pouvoir entrer la commande suivante :

```
iex --version
```

Nous allons d'abord passer un certain temps dans la console (et même une grande partie du livre), il est donc utile de savoir comment l'ouvrir et l'utiliser. Sous Linux, vous n'avez probablement pas besoin d'aide. Sous Windows, en revanche, il y a bel et bien une procédure particulière. Je vous conseille d'opter pour la simplicité : appuyez sur la touche *Windows* de votre clavier afin d'afficher une barre de recherche. Entrez *cmd* et appuyez sur la touche *Entrée*.

Il existe plusieurs consoles sous Windows, choisissez celle que vous préférez. L'important est de pouvoir exécuter `iex`.

En entrant cette commande, Elixir est censé retourner sa version installée.

```
vincent@PC:~/code/bonjour$ iex --version

IEx 1.14.0 (compiled with Erlang/OTP 25)
```

Si ce n'est pas le cas, consultez le site officiel pour obtenir plus d'informations. Vous pouvez également effectuer une recherche web sur l'erreur rencontrée. Il est fort probable que ce soit une erreur connue et, espérons-le, facile à corriger.

À ce stade, vous avez accès à la commande `iex`, mais aussi à la commande `elixir`. Voyons en quoi elles diffèrent. L'installation d'Elixir vous donne accès à quatre commandes : `elixir`, `iex`, `mix` et `elixirc`. Nous allons nous concentrer sur `iex` et `mix` pour commencer. Notez que la commande `elixirc` permet de compiler manuellement des fichiers Elixir. La commande `elixir`, quant à elle, permet d'exécuter « à la volée » un module Elixir sans le compiler. Nous verrons ces deux commandes lors de l'exploration des modules.

Revenons à `iex`. Vous pouvez saisir cette commande sans aucun argument :

```
iex
```

Vous devriez alors voir le résultat suivant dans la console :

```
Erlang/OTP 25 [erts-13.0.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
[jit:ns]

Interactive Elixir (1.14.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

C'est… magnifique. Si si, je sais bien ce que vous vous dites. Cette console vous permet d'entrer du code Elixir et de voir le résultat directement. Nous l'utiliserons par la suite, mais pour l'heure, vous pouvez la refermer en appuyant deux fois sur les touches *Ctrl+C*.

Une première démonstration

Nous verrons la syntaxe du langage aux prochains chapitres. Avant de clore celui-ci, nous allons écrire un petit programme. Le programme traditionnellement utilisé, le fameux « Hello World », est de moins en moins utile, car dans Elixir (comme dans beaucoup d'autres langages), il suffit d'une seule ligne pour parvenir à ce résultat. Mais la tradition étant ce qu'elle est, sachons la respecter. Cela nous donnera l'opportunité de voir un peu de code incompréhensible, qui sera sans doute intéressant pour ceux ayant déjà une expérience de programmation. Bien entendu, pas d'affolement, nous allons détailler ce code dans les prochains chapitres.

Nous allons donc à présent créer un projet. Je vous invite à créer au préalable un répertoire qui contiendra nos futurs travaux, par exemple le répertoire `code`. Pour ce faire, voici les com-

mandes pour Linux (ou du moins, sous WSL). Adaptez en fonction de votre système, bien entendu, puis déplacez-vous dans le répertoire créé :

```
vincent@PC:~$ mkdir ~/code
vincent@PC:~$ cd ~/code
vincent@PC:~/code$
```

Pour créer notre premier projet, nous allons utiliser `mix` :

```
mix new bonjour --sup
```

La commande `mix` permet de manipuler le projet (ou des applications) comme nous allons le voir. `new` permet de créer un projet. On lui donne le nom du projet, ici `bonjour`, et on lui attribue l'option `--sup`. Je ne vais pas rentrer dans le détail de cette option ici, ce n'est pas le sujet ici.

```
vincent@PC:~/code$ mix new bonjour --sup
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/bonjour.ex
* creating lib/bonjour/application.ex
* creating test
* creating test/test_helper.exs
* creating test/bonjour_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd bonjour
    mix test

Run "mix help" for more commands.
vincent@PC:~/code$
```

`mix` va créer un répertoire du même nom que notre projet `bonjour`.

Une fois créé, déplacez-vous dans ce répertoire :

```
cd bonjour
```

Vous pouvez visualiser le contenu du dossier. Vous pouvez constater qu'il contient le dossier `lib`, dans lequel se trouve le code du projet, notamment le sous-dossier `bonjour`. Si vous ouvrez ce sous-dossier, vous verrez qu'il ne contient qu'un seul fichier : `application.ex`.

L'extension `.ex` (à ne pas confondre avec `.exe`) est utilisée pour un fichier qui contient le code Elixir. Vous pouvez ouvrir ce fichier avec un éditeur tel que Vim, Emacs, Sublime,

notepad++… Le Bloc-notes de Windows peut aussi être utilisé même s'il n'offre pas une expérience de développement très agréable.

Si vous ouvrez ce fichier, vous allez voir beaucoup de lignes de code probablement incompréhensibles pour vous !

```
defmodule Bonjour.Application do
  # See https://hexdocs.pm/elixir/Application.html
  # for more information on OTP Applications
  @moduledoc false

  use Application

  @impl true
  def start(_type, _args) do
    children = [
      # Starts a worker by calling: Bonjour.Worker.start_link(arg)
      # {Bonjour.Worker, arg}
    ]

    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: Bonjour.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Pour l'heure, le but n'est pas de comprendre, même vaguement, ce code, mais de créer un programme simple pour afficher un message à l'écran (« Bonjour »). À la ligne 10, juste sous la ligne `def start(_type, _args) do`, ajoutez la ligne suivante :

```
    IO.puts("Bonjour !")
```

Une fois cette modification faite, votre fichier devrait donc ressembler à :

```
defmodule Bonjour.Application do
  # See https://hexdocs.pm/elixir/Application.html
  # for more information on OTP Applications
  @moduledoc false

  use Application

  @impl true
  def start(_type, _args) do
    IO.puts("Bonjour !")
    children = [
      # Starts a worker by calling: Bonjour.Worker.start_link(arg)
      # {Bonjour.Worker, arg}
    ]
```

```
    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: Bonjour.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Il s'agit de votre première instruction avec Elixir, un peu de considération ! Sauvegardez le fichier et retournez dans la console. Entrez `mix run` pour lancer votre programme :

```
vincent@PC:~/code/bonjour$ mix run
Compiling 2 files (.ex)
Generated bonjour app
Bonjour !

vincent@PC:~/code/bonjour$
```

La commande `mix run` lance alors le programme, qui génère trois actions successives.

1. La compilation de votre code : la tâche de `mix` est de convertir vos fichiers `.ex` en une version compréhensible par Erlang. On entend souvent par « compilation » le fait de convertir un code humainement lisible en code compréhensible par l'ordinateur (binaire ou autre).
2. La création d'une application `bonjour`. Dites-vous qu'il s'agit de l'ensemble de votre code compilé (traduit).
3. Le lancement : enfin, le programme se lance et affiche `Bonjour !` comme nous le lui avons demandé.

La ligne ajoutée, `IO.puts("Bonjour !")`, peut vous sembler assez obscure. Nous allons voir dans les prochains chapitres ce qu'Elixir comprend… et pourquoi.

Vous vous demandez peut-être si vous avez réellement besoin de tout ce code pour créer un simple programme ? La réponse est : pas vraiment. Mais cela nous permet de travailler directement avec `mix`, de voir une structure simple pour nos projets et de saisir du code Elixir (même sans le comprendre, ce qui est utile). Nous reviendrons sur la syntaxe de base au prochain chapitre.

En résumé

- Erlang a été créé au milieu des années 1980 pour répondre à des besoins de télécommunications continues.
- Elixir est conçu sur Erlang et bénéficie donc de ses avantages.
- La haute disponibilité, la résistance aux erreurs et la distribution font partie des points forts d'Elixir.