Queensland University of Technology

CAB402 - Assignment 2

Report

Name : Jamie Martin (N10212361)

Due Date : 31st May 2021 [11.59 p.m]

Table of Contents

1.0 Introduction

Elixir is a dynamic, functional programming language designed and developed over the past 10 years as an abstraction of the Erlang programming language. Elixir extends upon the Erlang language; a programming language developed by Ericsson in the late 80s to solve problems experienced in telephony switches and routers regarding availability, robustness, and concurrency. The stringent requirements made most languages of the time unfit for purpose. Thus, Erlang - and thus onwards Elixir - were built out of necessity; providing better solutions for concurrent, distributed, and fault-tolerant applications that most other languages at the time had left unanswered [1, 2, 3].

As an extension of Erlang, Elixir was designed to make the Erlang language more extensible and approachable; offering a Ruby-like syntax, and semantic abstractions for many of the rather complex language features Erlang offers [2].

Erlang, and in turn Elixir, are multi-paradigm languages, providing solutions to programming paradigms that most developers face using both imperative and declarative languages. These paradigms include:

- Concurrency
- Fault-tolerance
- Distributed Computing [2, 3]

Most languages forego answering these as core concepts of a language, if at all. For languages that do - for instance Golang and it's concurrency model - all paradigms aren't necessarily answered, nor are they answered in the same fashion . The implementation of the same paradigms in different languages can often differ. Occasionally, being an after-thought resulting in a sub-optimal implementation [3].

This report seeks to delve into those paradigms discussing their purpose; accompanying with code examples to give the reader a better understanding of the potential benefits of using Elixir for software applications.

2.0 Language Syntax

2.1 Functional Programming

Functional programming focuses on developing software as functional processes, as opposed to functions that are object-orientated. Unlike sequential functions that are object-oriented, functional languages treat workflows as a tree of expressions; creating a composition of functions that execute through said tree. Elixir facilitates this through its functional syntax; a tree-like structure that promotes isolation, enabling developers to create software that is concise and maintainable. In programming functionally, this often forces the developer to program code that is more easily testable compared to imperative languages. This is aided by the concept that every action is merely a function [3, 4, 5].

Example 1

A small example below depicts the use of functions to expose and abstract logic.

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
  # Math.sum(2, 3) => 5

  def square(a), do: a * a
  # Math.square(2) => 4
end
```

Example 2

In functional programming, the use of classes is absent, as structs are instead passed as values to be acted upon.

```
defmodule Square do
  defstruct w: 0, h: 0
```

```
  def area(%Square{w: w, h: h}) do
    w * h
  end
  # Sqaure.area(%Square{w: 10, h:15}) => 150
end
```

2.2 Immutability

Immutability refers to the immutable state of a variable. In Elixir's case, variables are invariant - in that their state cannot be mutated. Instead, a copy-on-write or transformation process occurs in which the new variable is rebound. Rather than the data structure being manipulated, the new reference applied on the right hand side is bound to the left hand side, with the old reference being left for garbage collection. A variable can instead be seen as a label to a value as opposed to space for a value [6].

Immutability results in code that is less prone to side effects, as state is no longer shared between functions, instead passed by messages. A side effect refers to the observable effect manipulating state can have on a system when working with mutable state shared across multiple operations, whereby changing said state in one operation can in turn break another operation, causing execution or runtime failures in code, often unintentionally [3, 4, 5].

2.3 Pattern Matching

Elixir provides a pattern-matching functionality for value assignments, in which an assignment instead matches or binds to the left hand operator. This enables a form of offensive programming, in that Elixir now only binds function calls that match. Multiple function definitions of the same name can be defined; Elixir will pick the first function that matches the given arguments [3, 7].

Example 1 [7]

```
defmodule Sum do
  def sum([head | tail], n) do
    sum(tail, n + head)
  end
```

```
  def sum([], n) do
    n
  end
end
```

The example above demonstrates the use of pattern matching on function calls. The example when called sums the given numbers in an array of numbers. Initially the pattern will be attempted to match the first definition. This first `sum()` requires that the array provided has at least one value for head. If the array is empty, the second `sum()` function will be called.

Example 2 [7]

```
defmodule Math do
  def sum(i, n) do
    case i do
      [head | tail] -> sum(tail, n + head)
      [] -> n
    end
  end
end
```

Another example demonstrates the use of the built-in `case` syntax. Example 1 and 2 are both functionally the same, however Example 2 could be argued as more concise.

Pattern matching promotes a form of defensive programming without much unnecessary boilerplate. This enables programmers to focus on the domain logic, and build guarded, multi-functional functions. Pattern matching will be utilised in further examples to demonstrate its use cases and monadic capabilities.

3.0 Language Features

3.1 OTP (Open Telecom Platform)

The Open Telecom Platform (OTP) is an integral component of the Erlang programming language. Not only does it contain a set of libraries for interfacing with the BEAM virtual machine, it is a set of design principles that encompasses a large proportion of Erlang itself. Elixir inherits OTP, enabling programmers to utilise these abstractions of commonly used concepts for use in Elixir-built systems. The OTP package has a myriad of utilities, however, focus here will be put into concurrency, supervision, and distribution, discussing their purpose and what paradigms each solves [3, 8, 9, 10, 11].

3.2 Concurrency

Concurrency refers to the programming paradigm of managing multiple executing computations at the same time. Unlike parallelism - another programming paradigm - concurrency results in one being able to manage and communicate these executions, as opposed to only being able to execute multiple processes at once. As well as this, multiple concurrent executions can run on the same CPU core [3, 11, 12].

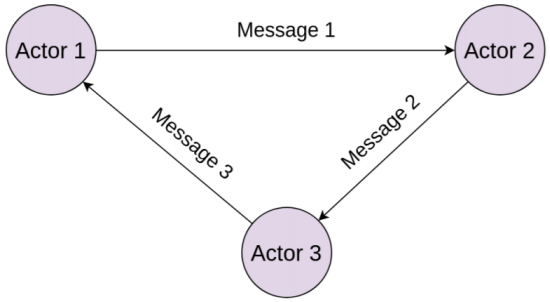
“

- **Concurrency** is the composition of independently executing things, typically functions
- **Parallelism** is the simultaneous execution of multiple things, possibly related, possibly not

” - Rob Pike [11]

In Elixir, concurrency is handled by processes spawned on the Erlang Runtime System; the BEAM virtual machine. These processes are akin to threads in that they are executions able to be handled in parallel albeit much cheaper than native threads. Despite being akin to threads, these processes explicitly do not share state. Instead, the state of each process is internal and communicated by message passing, by which state is copied between processes. This implementation of concurrency follows the Actor model, in which ‘actors’ (processes) control

and execute upon internal state; acting as a mailbox, receiving communications from other ‘actors’, executing upon that message, and then passing a message on to another ‘actor’ for consumption. On top of this, actors are also able to create new actors [3, 13, 14].



Simple actor system example [15]

Example [15, 16]

```
defmodule Bank do
  def start do
    spawn(__MODULE__, :loop, [0])
  end

  def loop(state) do
    receive do
      {:deposit, caller, amount} ->
        send(caller, "Deposited #{amount}")
        loop(state + amount)

      {:withdraw, caller, amount} ->
        send(caller, "Withdrew #{amount}")
        loop(state - amount)

      {:balance, caller} ->
        send(caller, "Balance is #{state}")
        loop(state)

      _ ->
        loop(state)
```

```
    end
  end
end
```

In the above example, basic concurrency models are demonstrated using a process to handle a very simple banking application. The module defined in the example creates a process with some internal state, that provides some actions to manipulate said internal state. In the case of this example, the internal state refers to the amount stored in the bank account. Messages can be passed to this process to deposit and withdraw an amount, as well as retrieve the balance. Here pattern matching is used to define operations by which can be called as a Map argument.

Command Line

```
iex(1)> pid = Bank.start
#PID<0.109.0>
iex(2)> send pid, {:balance, self()}
{:balance, #PID<0.107.0>}
iex(3)> flush()
"Balance is 0"
:ok
iex(4)> send pid, {:deposit, self(), 100}
{:deposit, #PID<0.107.0>, 100}
iex(5)> flush()
"Deposited 100"
:ok
iex(6)> send pid, {:balance, self()}
{:balance, #PID<0.107.0>}
iex(7)> flush()
"Balance is 100"
:ok
iex(8)> send pid, {:withdraw, self(), 25}
{:withdraw, #PID<0.107.0>, 25}
iex(9)> flush()
"Withdrew 25"
:ok
iex(10)> send pid, {:balance, self()}
{:balance, #PID<0.107.0>}
iex(11)> flush()
"Balance is 75"
:ok
```

As shown in the above Command Line, once the process has been started, messages can be sent via built-in language syntax to manipulate the internal state of the specific process. More processes of the same module could be created, and manipulated, each storing their own bank account internal state. This could then be built upon, creating another process to handle the distribution and management of bank accounts simultaneously based on an identifier of sorts.

Concurrency can not only increase a system's efficiency and performance, but also make it easier to manage complex systems that require a multitude of operations occurring at the same time. Leading to less headaches for developers tackling problems where deadlocks or livelocks can be a real issue. That is not to say that concurrency alleviates all cases of deadlocks or livelocks. It is simply a set of tools to effectively manage the parallel executing and communication of operations [3, 11, 12].

3.3 Fault-tolerance

Fault-tolerance refers to the ability of a system being able to handle failure events and continue operating in some capacity. In the event of a fault, instead of taking down the whole system, fault tolerant applications can recover; resetting affected components of the application whilst other components of the system keep running [3, 17, 18].

Elixir follows a ‘let it crash’ philosophy, advising against the practice of defensive programming, simply letting the process crash when it runs into an error. Instead, Supervisors act as a manager for one or more processes, shifting the burden of recovering from the process itself to the process’ supervisor. Thus resulting in the supervisor being able to restart the process if a particular process is failing, without affecting the rest of the system. The OTP package exposes common abstractions for these supervisor methodologies, reducing the necessary boilerplate to implement these concepts [3, 17].

Example [17]

bank.ex

```elixir
defmodule Bank do
  use GenServer

  # -- Client API --

  def start_link() do
    GenServer.start_link(__MODULE__, 0, name: __MODULE__)
  end

  def deposit(amount) do
    GenServer.cast(__MODULE__, {:deposit, amount})
  end

  def withdraw(amount) do
    GenServer.cast(__MODULE__, {:withdraw, amount})
  end

  def balance() do
    GenServer.call(__MODULE__, :balance)
  end

  # -- GenServer Callbacks --

  def init(balance) do
    {:ok, balance}
  end

  def handle_cast({:deposit, amount}, balance) do
    {:noreply, balance + amount}
  end

  def handle_cast({:withdraw, amount}, balance) do
    {:noreply, balance - amount}
  end
```

```
  def handle_call(:balance, _from, balance) do
    {:reply, balance, balance}
  end
end
```

Utilising GenServer, inherited from the OTP package, we can abstract away some of the boilerplate involved in the initial example shown in **3.1 Concurrency**. This involves removing the main `loop(state)` function, which is now handled by GenServer.

Provided an instance has been opened, functions under the Client API comment are now able to be utilised.

Command Line

```
iex(1)> Bank.start_link()
{:ok, #PID<0.138.0>}
iex(2)> Bank.deposit(400)
:ok
iex(3)> Bank.withdraw(100)
:ok
iex(4)> Bank.balance()
300
```

supervisor.ex

```
defmodule Bank.Supervisor do
  use Supervisor

  def start_link() do
    Supervisor.start_link(__MODULE__, name: __MODULE__)
  end

  def init(_init_arg) do
    children = [
      %{
        id: Bank,
        start: {Bank, :start_link, []}
      }
```

```
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

Here the Supervisor module is utilised to manage the Bank GenServer process. The Supervisor spawns and manages the bank account process, ensuring its availability. Any failures resulting in the Bank process to fail, would prompt the Supervisor to attempt to restart the Bank process. This can be seen in the command line output below.

Command Line

```
iex(1)> Bank.Supervisor.start_link()
{:ok, #PID<0.154.0>}
iex(2)> Bank.balance()
0
iex(3)> Bank.deposit(100)
:ok
iex(4)> Bank.balance()
100
iex(5)> Bank.deposit("100")
:ok
iex(6)>
05:33:08.424 [error] GenServer Bank terminating
** (ArithmeticError) bad argument in arithmetic expression
    :erlang.+(100, "100")
    (bank 0.1.0) lib/Bank.ex:29: Bank.handle_cast/2
    (stdlib 3.15) gen_server.erl:695: :gen_server.try_dispatch/4
    (stdlib 3.15) gen_server.erl:771: :gen_server.handle_msg/6
    (stdlib 3.15) proc_lib.erl:226: :proc_lib.init_p_do_apply/3
Last message: {:"$gen_cast", {:deposit, "100"}}
State: 100

nil
iex(7)> Bank.balance()
0
```

Notice the Bank process becomes re-available despite the termination after a bad argument was sent to the `Bank.deposit()` function. The internal state of the process is re-initialised, however this could be caught by the Supervisor and used when initialising the new process.

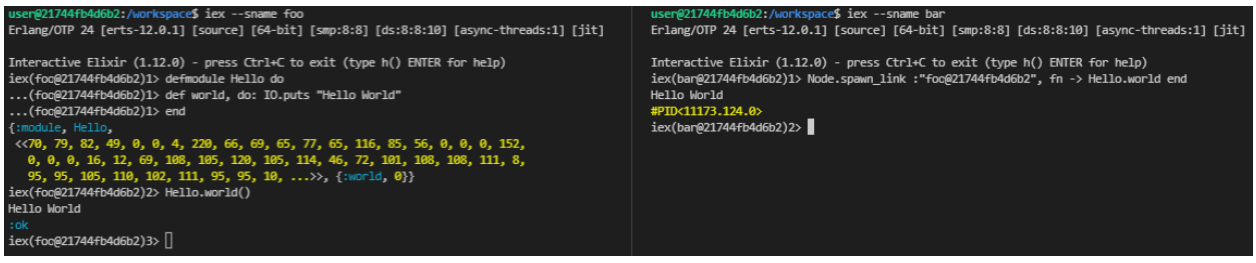
Providing tooling for supervised fault-tolerant systems enables programmers to focus less on the architecture of the system and more on the domain logic. Common abstractions of this propel that idea forward, removing unnecessary boilerplate from the equation.

3.4 Distributed computing

Distributed computing refers to a collection of independent systems able to communicate and operate together. Paradigms in Erlang and Elixir such as isolated state and processes, provide capability for a distributed runtime. In communicating via message-passing, a message can be passed from a process on one system to another. This leads to systems that are no longer geospatially restricted, as two systems can be running and interacting with one another in entirely different locations. This is all included in the Erlang OTP package, so minimal logic is required to get this functioning [3, 17].

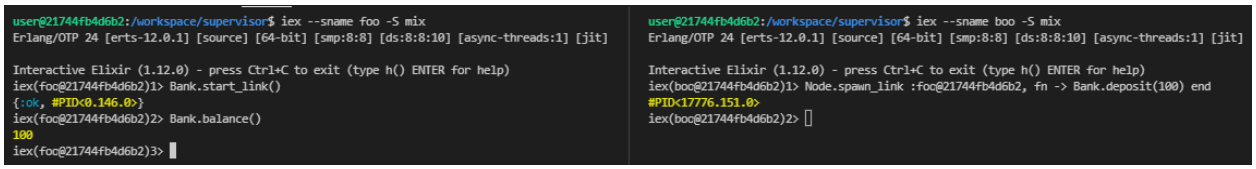
Below is a simple example of how we can utilise the OTP package for distributed computing. Starting two named interactive Elixir environments, a module can be defined on one, and then called from another. This is done by using the `Node.spawn_link()` function, which tells the receiving node to execute the function provided as an argument.

Example [18]



To receive something from another executing node, some send and receive logic can be defined to do so. This enables one node to execute logic on another, and retrieve its value for use in the node's own operations. In this next example, we'll utilise the bank account GenServer functionality built in 3.3 Fault-Tolerance to communicate with another isolated Elixir environment running locally.

Example 2 [19]



In the above example, on the left hand side the interactive elixir runtime is started with the name `foo`. A Bank instance is then initialised with the default value of 0. On the right hand side, an interactive elixir runtime starts with the name `bar`. Then, `Node.spawn_link` is used to tell `foo` to deposit 100 in its bank account. Moving to the left side, `Bank.balance()` is called to show that the balance has in fact been manipulated.

Not only can these distributed concepts and methods introduce geo-redundancy in shared systems, but also greater isolation between uncommon logic and systems, resulting in highly-available, highly-fault tolerant, distributed systems. In utilising distributed systems, hardware no longer becomes a non-issue in regards to performance or failures of system hardware, provided distributed methods have been implemented. Distributed systems enable systems to interact and make calls to other systems effectively; reducing the added complexity of distributed requirements programmers may face [3, 18].

4.0 Conclusion

The Elixir programming language creates an easy-to-approach abstraction of the Erlang Runtime and Open Telecom Platform, giving programmers tooling to create versatile, reliable systems.

Functional programming seems to be growing in popularity, both for its programming efficiency and performance. Functional programming offers unknown capabilities in software systems, and enables programmers to solve problems differently. The immutability and pattern matching that Elixir provides as a language can reduce headaches, providing built-in solutions for common problems programmers face.

Concurrent tooling gives programmers a collection of utilities to manage a multitude of executions simultaneously, whilst being able to communicate that with other processes. This creates processes capable of functioning in isolation, communicating only when necessary, and managing their own state, or the availability of others.

Fault-tolerance built into the language gives developers a standard library of support for common problems faced across many industries. The philosophy of ‘letting it crash’, enables developers to focus on domain logic, as opposed to defensive programming. In letting the program crash, and having tooling to manage so, you get rid of solving the problem of crashing, entirely.

The ability for distributed computing solves issues regarding the fatigue of hardware innovation. Distributed systems can now be distributed across hardware, enabling an application to easily fan-out in terms of operations as opposed to upgrading a single server. More servers can be provisioned as necessary, and operations can be isolated on a process, or hardware level.

These programming paradigms solved by Erlang in the late 1980s have benefits that are now being realised in web and embedded software applications; enabling concurrent, fault-tolerant, distributed systems and changing the way programmers now solve problems.

5.0 Personal Comments

As a programmer with a predominantly C-like background, the Ruby-like and functional style took me by surprise. Before this unit, I had barely even dipped by toe into the pool of functional programming. The Ruby-like syntax I had only seen before in Lua, and that was some time ago. The functional style, I had only really experienced in F#, which is still comparatively somewhat C-like. I can see though how abstracting Erlang - a rather complex language - to newer syntaxes (Ruby) could generate more adoption for the Elixir language.

The inclusion of immutability in a programming language sparks my interest. Having to deal with mutated state in the likes of C-like and frontend languages is never without its hurdles. Coupling this with pattern-matching to create guards for functionality as a language syntax feature is incredible. Too often does one have to resort to third party libraries for better tooling for defensive programming. Elixir solves the problem by both letting it crash, and giving tooling to ensure correct calls do not.

In regards to the functionalities Elixir exposes, I had worked before with concurrency in Golang so thinking concurrently and communicating via message-passing were familiar to me. That being said, the implementations that Elixir and Golang provide differ quite a bit. The difference is partly due to the other programming paradigms that Elixir addresses, as Elixir does not just provide tooling for concurrency. One of Golang's claim to fame is predominantly concurrency done simply, although it's still an optional feature. Elixir is more so concurrency by design by focusing best practices on programming concurrently.

The concept of being able to just 'let it crash' very much appeals to me. When it comes to defensive programming I'm often looking for the quickest way to do so. Anything that's outside of the domain logic seems a slight waste of time, especially in the nature of today's fast-paced climate. Giving tooling and methodology for this as part of the standard library is unlike anything I've seen before; and could definitely be a game changer for rapid development, giving programmers full focus on domain logic for rapid prototyping.

Being able to compute across multiple systems is astounding. I'm intrigued as to how it works in more detail, and to research some of its use cases for large scale operations. As Moore's Law is slowly becoming no more (pardon the pun), distributed computing is something that is definitely a necessity. Being able to span out your system architecture across multiple hardware systems, alleviates the issue of having to continually upgrade a single server. Obviously, the application has to be developed with distribution in mind, and isn't entirely "free". However, that cost is now left up to the programmer, and less of a restriction to have to worry about.

Overall, Elixir has been an interesting insight into functional programming and the benefits it can offer. Not only this, the tooling Elixir provides for developing reliable, performant applications is very appealing. There are definitely motivations to utilise Elixir for projects in the future.

6.0 References

[1] Elixir. The Elixir programming language [Internet]. [place unknown]: Elixir; 2021 [cited 2021 May 20]. Available from: https://elixir-lang.org/

[2] Däcker B. Early History of Erlang [Internet]. Stockholm: Ericsson; 2000 [cited 2021 May 20]. Available from: http://www.erlang.se/publications/bjarnelic.pdf

[3] Armstrong J. Making reliable distributed systems in the presence of software errors (Doctoral dissertation).

[4] Hughes J. Why functional programming matters. The computer journal. 1989 Jan 1;32(2):98-107.

[5] Hudak P. Conception, evolution, and application of functional programming languages. ACM Computing Surveys (CSUR). 1989 Sep 1;21(3):359-411.

[6] Ghilardi D. Immutability in Elixir [Internet]. [place unknown]: Dario Ghilardi; 2015 [cited 2021 May 20]. Available from: https://darioghilardi.com/immutability-in-elixir/

[7] Elixir School. Pattern Matching - Elixir School [Internet]. [place unknown]: Elixir School; 2021 [cited 2021 May 20]. Available from: https://elixirschool.com/en/lessons/basics/pattern-matching/

[8] Torstendahl S. Open telecom platform. Ericsson Review(English Edition). 1997;74(1):14-23.

[9] Laurent SS, Eisenberg JD. Introducing Elixir: Getting Started in Functional Programming. "O'Reilly Media, Inc."; 2016 Dec 22.

[10] Cesarini F, Vinoski S. Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems. "O'Reilly Media, Inc."; 2016 May 16.

[11] Pike R. Concurrency is not Parallelism [Internet]. [place unknown]: Heroku; 2012 [cited 2021 May 20]. Available from: https://vimeo.com/49718712, https://talks.golang.org/2012/waza.slide#1

[12] Roscoe B. The theory and practice of concurrency. Pearson; 2005.

[13] Sposito A. Elixir, processes and this thing called OTP [Internet]. [place unknown]: Plataformatec; 2018 [cited 2021 May 20]. Available from: http://blog.plataformatec.com.br/2018/04/elixir-processes-and-this-thing-called-otp/

[14] Zaiaev S. Actor model overview using Elixir language [Internet]. Tartu, Estonia: University of Tartu; 2020 [cited 2021 May 20]. Available from: https://courses.cs.ut.ee/MTAT.08.024/2020_spring/uploads/Main/B96281_5867_1.pdf

[15] Eftimov I. Understanding the basics of Elxir's concurrency model [Internet]. [place unknown]: Ilija Eftimov; 2019 [cited 2021 May 20]. Available from: https://ieftimov.com/post/understanding-basics-elixir-concurrency-model/

[16] Elixir School. Concurrency [Internet]. [place unknown]: Elixir School; 2020 [cited 2021 May 20]. Available from: https://elixirschool.com/en/lessons/advanced/concurrency/

[17] Fuentes G. Supervisors: Building fault-tolerant Elixir applications [Internet]. [place unknown]: AppSignal; 2017 [cited 2021 May 20]. Available from: https://blog.appsignal.com/2017/08/10/elixir-alchemy-supervisors-building-fault-tolerant-elixir-applications.html

[18] Waldo J, Wyant G, Wollrath A, Kendall S. A note on distributed computing. InInternational Workshop on Mobile Object Systems 1996 Jul 8 (pp. 49-64). Springer, Berlin, Heidelberg.

[18] Elixir School. OTP Distribution [Internet]. [place unknown]: Elixir School; 2020 [cited 2021 May 20]. Available from: https://elixirschool.com/en/lessons/advanced/otp-distribution/

7.0 Appendix

7.1 Code Examples

Functional Programming

/functional-programming/

This directory holds simple examples of functional programming in Elixir. Utilised in section 2.1 Functional Programming, giving a brief introduction for the functionality to come in more complex examples.

Pattern Matching

/pattern-matching/

This directory holds examples of utilising pattern matching provided by Elixir as a language syntax. Demonstrated in section 2.2 Pattern Matching to show the various ways that pattern matching can be exploited.

Concurrency

/concurrency/

This directory holds examples of utilising concurrency in Elixir for section 3.2 Concurrency. It utlises Elixir's built-in `spawn()` and `receive()` functions to create a process with internal state.

OTP

/otp/

This directory holds examples for utilising the OTP package to demonstrate functionality for section 3.3 Fault Tolerance. The code makes use of the OTP packages GenServer and Supervisor to build upon the 3.2 Concurrency example, making it more robust, easier to use, and fault-tolerant.