Chapter 1

This chapter discusses an overview of **Ballerina**. It is a language an open-source programming language used to implement and deploy distributed **services** and **cloud-native** applications. The language resources can be accessed here.

The *Ballerina* programming language and its runtime stack were created to address the pain points developers faced while integrating independent components into a complex application. According to [1], *Ballerina* makes it possible to write resilient programs that integrate and orchestrate across distributed endpoints. From a pure programming language perspective, it is a *compiled*, *statically* and *strongly* typed programming language.

Installation

To install *Ballerina*, go to the Download page and follow the instructions according to your operating system. Once the installation completes, make sure it is added to your PATH environment variable. You can check the successful installation of *Ballerina* by printing its version `bal --version`. At the time of this write-up, the version is *Swan Lake Update 7*.

Documentation and Resources

The main website of the language contains a lot of resources and documentation on the language. The key concepts and syntax are presented here. More resources include a blog and a collection of examples for a variety of functionality in *Ballerina*. You are strongly encouraged to start your journey by reading the documentation on the website. *Ballerina* also has a Playground where you can test a piece of code online.

The Git Protocol

Although the Git protocol is not part of the language, we present it here as a support for collaboration in a project.

Overview

This document provides a quick tutorial to the Git protocol for those unfamiliar with it. Git is a decentralised version control system. It manages the different versions a document goes through during its lifetime (from creation to deletion and through the various updates applied to the document). To get a good grasp of Git, I encourage you to read the Pro Git book. It offers detailed information about Git from installation to complex workflows.

Setup

Once you successfully install Git on your machine, you need to configure your user details for all future repositories. It is recommended to configure at least the user name and email address.

The command `git config --global user.name "MyFirstname MyLastName"` assigns a name for the current user. Every subsequent commit will attach that name and thus give credit to the author of the changes being committed. `git config --global --user.email "emailaddress"` will link the valid email address to the current user. For more configurations, please chack the documentation.

Initialisation

You can initialise a Git repository in two possible ways. First, you can change directory to an existing folder that does not a Git repository yet. Type the command `git init` and a directory tree (and some additional information) will be created for you. The second way of initialising a repository is by cloning an existing repository as follows `git clone url [local-name]`. This command will

copy a repository at location url and create a folder local-name if you provided that argument or use the same folder name as the original repository.

General Usage

Your general usage of the Git protocol consists in adding a new file to working tree and committing changes. The command `git add file-name` will add file-name to the directory in the working tree. To commit recent changes, use the command `git commit -m "commit message"`. The best practices in how to draft a commit message abound on the Internet. Please look them up and adopt them. It proves very useful in the context of a collaboration to structure commit messages following an agreed upon format.

Other usual commands that you might need include `git diff` and `git status`.

Branching

Branching can prove quite handy when using the Git protocol. It can help you explore different lines of thoughts without collapsing them. By default the master branch is created in the repository after initialisation. To list all local branches type the command `gir branch`. To include the remote branches, simply add the argument a `git branch -a`.

To create a new branch, use the command `git branch new-branch-name`. Please note that this command does not switch to the newly created branch. There are several variants of the command, please check the documentation for further exploration. Finally, to switch to another branch, use the command `git checkout branch-name`.

Collaboration

A way to setup collaboration in Git is to create a copy of the repository, which all collaborators can synchronise their versions with. Because Git is

decentralised, collaborators might asymmetrical views in their local working directories.

To add a remote branch to your repository, use the command `git remote add branch-name bracnh-url`. You can now merge a remote branch wuth your current branch as follows `git merge remote-bracnh-name`. Commonly, you can pull from and push to a remote branch by using the `git pull` and `git push` commands. Caution! With these two commands you will have to specify the local and remote branches involved.

A Glance at Ballerina

A **Ballerina** program can be executed in two possible ways: either running the **main** function or as a **service**. We will discuss the service approach in the next chapter. The example below shows a simple program with the **main** function.

```
import ballerina/io;

public function main(string name) {
    io:println("Hello ", name, "!");
}
```

To execute the program, type the command `bal run prg.bal -- Tom` (assuming the program is saved as prg.bal). Note that when the **main** function might throw an error, the signature changes. In the example below, the program throws an error when the name is an empty string. Note that the ? means that an error is an optional return value from the main function.

```
import ballerina/io;

public function main(string name) returns error? {
    if (string:length(name) > 0) {
      io:println("Hello You ", name, "!");
  } else {
    return error("Error! The name should not be an empty string");
  }
}
```

We can make the program more modular by defining a function that generates a greeting string.

```
import  ballerina/io;

public function main(string name) returns error? {
  string|error the_res = say_hello(name);

  if the_res is string {
      io:println(the_res);
  } else {
      return the_res;
  }
}

function say_hello(string name) returns string|error {
    if (string:length(name) > 0) {
        return "Hello " + name;
    } else {
        return error("Error! The name should not be an empty
string");
    }
}
```

The **Ballerina** programming language comes with a rish type system, including integers, floating point numbers, boolean, etc. We encourage the reader to check the documentation for more details about the type system. In the example above, the say_hello function accepts a **string** as argument and returns a **union** type of **string** and **error**. Similarly, the type **error?** is a union of **error** and ().

As well, the language has a collection of data structures, including **lists**, **records**, **mappings**. A **list** can be implemented in various ways, e.g., **arrays** and **tuples**. The example below illustrates the usage of arrays and tuples. The print_names function prints each name in an array passed as argument. The compute_area function takes a tuple as rectangle and compute the area.

```
import  ballerina/io;

public function main() {
    print_names(["Paul", "Peter", "James"]);
    io:println(compute_area([2.0, 3.0]));
}

function print_names(string[] all_names){
  foreach string single_name in all_names {
    io:println(single_name);
  }
}

function compute_area([float, float] rect) returns float {
  return rect[0] * rect[1];
}
```

Another data structure is mapping keys to values, which can be implmented as a **map** or a **record**. A **map** is an associative array that maps keys to values. All keys are of type of **string**. On the other hand, a **record** contains a collection of named fields, each of a specific field. The exampke below illustrates the usage of a **map** and a **record**.

```
import ballerina/io;

type CourseDetail record {|
    int code;
    string designation;
    int credit;
|};

public function main() {
  CourseDetail c_detail = {
      code: 1234,
      designation: "Philosophy",
      credit: 20
  };

  print_course_details(c_detail);

  map<string> the_songs = {
    "You are the one": "Let it rain",
    "Subcity": "Crossroads",
    "Smoke and Ashes": "New Beginning"
  };

  print_albums(the_songs);
}

public function print_course_details(CourseDetail c_det) {
  io:println("printing course details -- begin");
  io:println(c_det.code);
  io:println(c_det.designation);
  io:println(c_det.credit);
  io:println("printing course details -- end");
}

public function print_albums(map<string> all_songs) {
    foreach string single_song in all_songs.keys() {
        io:println(single_song, " : ", all_songs[single_song]);
    }
}
```

Exercises

Exercise 1

Write a **Ballerina** program that displays "Hello DSA class!"

Exercise 2

Write a program that randomly selects a number between 10 and 60 and compute its factors. You will print each identified factor.

References

[1] Jewel, T. (2018). Ballerina Microservices Programming Language: Introducing the Latest Release and "Ballerina Central"

Chapter 2

This chpater discusses how to implement services in **Ballerina**. Services an communicame using various communication paradigms. In this chapter, we will focus on **inter-process communication** (IPC).

Service Essentials

A service is a component that runs independently. In Chapter 1, we discuss the main function which is one of the ways to run a **Ballerina** program. A service builds on a listener (e.g., HTTP listener) and uses two special functions. First, the **init** function which initallises the state of the service. For example, if the service uses a data structure, it can be initialised in the **init** function. The second type of special function is a **resource function**, which implements the various calls between a client and the service.

The example below illustrates the key parts of a service. First, it imports the http module. Then, it stores the port number. We use a configurable variable, which will be discussed. Next, we define a Person record and a table of Person examples. The service keyword starts the block where the service is defined. We have three resource functions. They will answer two HTTP GET calls and an HTTP POST. The first resource function returns all the people in the group table. It answers the call `curl  http://localhost:8080/people`. Note that we use curl for quick demo purposes. The second resource function returns a person given his/her id. If the person does not exist, the NOT_FOUND response will be sent back to the client. You can test the function wth `curl http://localhost:8080/people/1`. Note that here we use a path parameter. finally, the third resource function adds a new person to the group of people. you can test the function with `curl -X POST http://localhost:8080/people -H "Content-Type: application/json" -d '{"id":"4", "name":"James Dean"}'`

```
import ballerina/http;

configurable int port = 8080;

type Person readonly & record {|
    string id;
    string name;
|};

table<Person> key(id) group = table [
        {id: "1", name: "Jonathan Pikes"},
        {id: "2", name: "Vincent Foster"},
        {id: "3", name: "Sarah Tommy Robson"}
];

service / on new http:Listener(port) {
    resource function get people() returns Person[] {
        return group.toArray();
    }

    resource function get people/[string id]() returns
Person|http:NotFound {
        Person? a_person = group[id];
        if a_person is () {
            return http:NOT_FOUND;
        } else {
            return a_person;
        }
    }

    resource function post people(@http:PayLoad Person new_person)
returns http:Response {
        group.add(new_person);
        http:Response post_resp = new;
        post_resp.statusCode = http:STATUS_CREATED;
        post_resp.setPayload({id: new_person?.id});
        return post_resp;
    }
}
```

Complete Example

In this section, we present a complete example of a service that implements a course management where we fulfil all CRUD operations. In the example, we use a query parameter rather than a path parameter. We also implemented concurrency constructs to allow the service to interact with multiple clients.

```
import ballerina/io;
import ballerina/http;

type CourseDetail record {|
    int code;
    string designation;
    int credit;
|};

isolated service /courses on new http:Listener(8080) {
    private int counter;
    private CourseDetail[] all_courses;

    function init() {
        self.all_courses = [];
        self.counter = 0;
    }

    isolated resource function get all() returns CourseDetail[] {
        io:println("handling a get request to /courses ...");

        lock {
            return self.all_courses.clone();
        }
    }

    isolated resource function post create(@http:Payload CourseDetail
new_course) returns json {
        io:println("handling a post requestto /courses ...");

        lock {
            self.all_courses.push(new_course.clone());
        }

        lock {
            self.counter += 1;
            return {code: new_course.code, id: self.counter.clone()};
        }
    }

    isolated resource function get course(int code) returns
CourseDetail? {
        io:println("handling a get request /courses/course ...");

        lock {
            foreach CourseDetail c in self.all_courses {
                if c.code == code {
```

```
                    return c.clone();
                }
            }

            return ();
        }
    }

    resource function delete course(int code) returns http:Ok {
        io:println("handling a delete request to /courses/course
...");

        // all_courses = all_courses.filter(isolated function
(CourseDetail val) returns boolean {
        //     return (val.code != code).clone();
        // });

        lock {
            CourseDetail[] new_courses = [];
            foreach CourseDetail c in self.all_courses {
                if c.code != code {
                    new_courses.push(c.clone());
                }
            }
            self.all_courses = new_courses.clone();
        }

        http:Ok ok = {body: "Operation Completed!"};
        return ok;
    }
}
```

Using an API

Another way of implementing a service is to start by defining the interface using an API. We can use a RESTful API to implement an IPC. Ballerina offers an integration with openapi.

```yaml
openapi: "3.0.1"

info:
  title: A Virtual Learning Application API
  version: "0.0.1"

servers:
  - url: http://localhost:8080/vle/api/v1


paths:
  /users:
    get:
      summary: Get all users added to the application
      operationId: get_all
      description: Returns all users registered for the application
      responses:
        '200':
          description: "A list of users"
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/User"
        default:
          $ref: "#/components/responses/ErrorResponse"
    post:
      summary: Insert a new user
      operationId: insert
      description: Create a new user and add it to the collection of
users
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: User successfully created
          content:
            application/json:
              schema:
                properties:
                  userid:
                    type: string
```

```
                    description: the username of the user newly
created
        default:
          $ref: "#/components/responses/ErrorResponse"
```

With the API file defined, you can generate the skeleton of the service by typing the command `bal openapi -i /path/to/api/file -o output/path`. The skeleton will declare the service and the resource functions. It is then your task to implement the logic of each resource function.

Exercises

- Implement the client application to interact with the service in the second section.
- Complete the API above and implement the corresponding service.

Chapter 3

This chapter discusses a new communication paradigm: **remote invocation**. The principle is as follows. A service implements one or several functions (or methods) which can be invoked remotely by clients. The signatures of the functions are provided in a separate contract document following a specific syntax.

In the **Ballerina language** one can implement a remove invocation using gRPC. In the remainder of this chapter we discuss how to implement a remote invocation in **Ballerina** using gRPC.

Protocol Buffers

In gRPC, the contract between a service and its clients is expressed following **protocol buffers**, language-neutral, platform-neutral mechanisms for serialising structured data. You can access the documentation at https://protobuf.dev. The code below provides an example of a protocol buffers contract. First, we specify the version being used with the *syntax* keyword. Next, we import two protocol buffers packages: empty and wrappers. For functions that do not return a value, we can use the **Empty** type. As well, wrappers helps handle primitive types. Next, we define the **users** service and essentially its remote functions. For example, the **delete_user** function expects a *StringValue* as argument (the username) and returns nothing. Note that if an error occurs during the execution of the function, it will be raised. Besides the primitive types, complex objects (called message) can also be defined and used in function signature. For example the **SingleUserResponse** message is a complex object that comprises of a lastname, a firstname and an email.

```
syntax = "proto3";

import "google/protobuf/empty.proto";
import "google/protobuf/wrappers.proto";

service users {
  rpc create_user(CreateRequest) returns (CreateResponse);
  rpc update_user(CreateRequest) returns (SingleUserResponse);
  rpc view_user(google.protobuf.StringValue) returns
(SingleUserResponse);
  rpc delete_user(google.protobuf.StringValue) returns
(google.protobuf.Empty);
  rpc get_all_users(google.protobuf.Empty) returns (stream
SingleUserResponse);
  rpc get_some_users(stream google.protobuf.StringValue) returns
(stream SingleUserResponse);
}

message CreateRequest {
  string username = 1;
  string lastname = 2;
  string firstname = 3;
  string email = 4;
}

message CreateResponse {
  string userid = 1;
}

message SingleUserResponse {
  string lastname = 1;
  string firstname = 2;
  string email = 3;
}
```

Service Implementation

From the contract defined in the previous section, one should generate the stubs as follows: `bal grpc --input mycontract.proto --output stubs`. The command will generate the stubs the glue code that interact with underlying layers.

To implement the service, follow the steps below:

1. create a new service package;
2. copy the stub file in the folder of the new package;
3. implement the service.

An incomplete version of the service is provided below.

```
import ballerina/io;
import ballerina/grpc;

type UserDetail record {|
    string username;
    string firstname;
    string lastname;
    string email;
|};

listener grpc:Listener ep = new (9090);

@grpc:ServiceDescriptor {descriptor: ROOT_DESCRIPTOR, descMap:
getDescriptorMap()}
isolated service "users" on ep {
    private map<UserDetail> all_users = {};

    remote function create_user(CreateRequest value) returns
CreateResponse|error {
        io:println("executing remote operation create_user...");

        string new_user_name = value.username;
        UserDetail new_user = {username: new_user_name, firstname:
value.firstname, lastname: value.lastname, email: value.email};

        lock {
            if self.all_users.hasKey(new_user_name.clone()) {
                string user_already_exists_error_message = "A user
with username ${new_user_name} already exists";
                return error(user_already_exists_error_message);
            } else {
                self.all_users[new_user_name.clone()] =
new_user.clone();
                return {userid: new_user_name.clone()};
            }
        }

    }

    remote function update_user(CreateRequest value) returns
SingleUserResponse|error {
        io:println("executing remote operation update_user...");

        string new_user_name = value.username;
        UserDetail up_user = {username: new_user_name, firstname:
value.firstname, lastname: value.lastname, email: value.email};
```

```
        lock {
            if self.all_users.hasKey(new_user_name.clone()) {
                self.all_users[new_user_name.clone()] =
up_user.clone();
                return {firstname: up_user.firstname, lastname:
up_user.lastname, email: up_user.email}.clone();
            } else {
                string no_user_error_message = "no user with username
${new_user_name} exists";
                return error(no_user_error_message);
            }
        }
    }
```

Exercises

- Complete the implementation of the service and run it
- Implement a client and call the functions in the service

Chapter 4

This chapter discusses **indirect communication**, a communication paradigm where processes no longer know each other's address space and lifetime.

In the **Ballerina language** one can implement an indirect communication using Kafka. Apache Kafka is a distributed event streaming platform that combines the properties of a message queue and a publish/subscribe event system. The communicating entities are called a **consumer** and a **producer**. In the examples below, both producer and consumer exchange messages about a couse using the "dsa" topic.

Producer Implementation

```
import ballerina/io;
import ballerinax/kafka;

type CourseAssignment readonly & record {
    string deadline;
    string mode;
    string presentation;
};

public function main() returns error? {
  kafka:Producer prod = check new (kafka:DEFAULT_URL);

  io:println("Welcome to the Kafka tutorial...");

        CourseAssignment msg = {
                deadline: "29/01/2021",
                mode: "git-repo",
                presentation: "TBD"
        };

        check prod -> send({topic: "dsp", value: msg});
}
```

Consumer Implementation

```
import ballerina/io;
import ballerinax/kafka;

type CourseAssignment readonly & record {
    string deadline;
    string mode;
    string presentation;
};

listener kafka:Listener cons = new (kafka:DEFAULT_URL, {
    groupId: "group-id",
    topics: "dsp"
  });


service on cons {

  remote function onConsumerRecord(CourseAssignment[] assignments) {
      from CourseAssignment ca in assignments
      do {
        io:println(ca);
        };
    }
}
```

Exercise

- Start a kafka broker and make the consumer and the producer communicate;
- Add more message exchanges between the consumer and the producer.