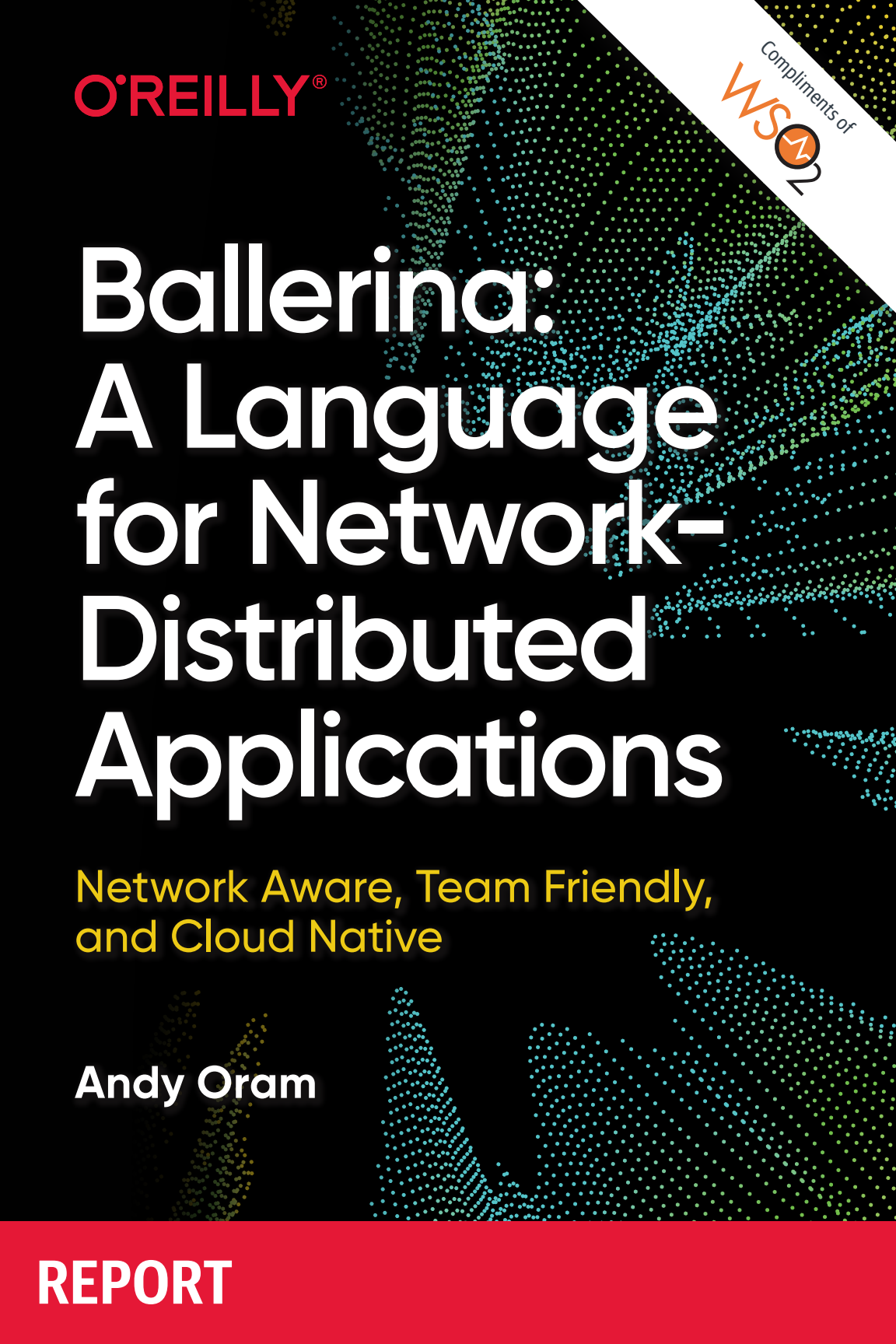
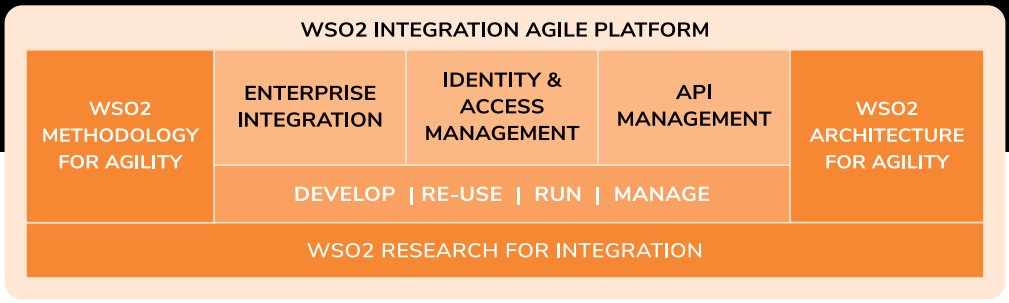

O'REILLY®
Compliments of
WSO2
Ballerina: A Language for Network-Distributed Applications
Network Aware, Team Friendly, and Cloud Native
Andy Oram
REPORT

Open Source Integration and API Management for Digitally Driven Organizations

All digital transformation is now API-driven, and integration technologies underpin their evolution. You can deliver faster, lower-risk integration projects with WSO2's open source Integration Agile Platform — including API Management, Enterprise Integration, ESB, and Identity Management technologies. To help make your integration an agile process, WSO2 provides unique technologies, methodologies, and architectures to speed up your transformation.



The WSO2 Difference

First-in-the-industry integration agile methodology

Our agile transformation methodology helps IT transform to an integration agile model. You get faster releases and quicker responsiveness to the business.

Unique open source technology and licensing

Our approach to open source ensures the benefits of faster innovation, community contributions, freedom from cloud lock-in, and the value-add of support patch binaries.

Broadest integrated platform

No need to mix and match different API management, integration, identity, analytics, or microservices technologies from multiple vendors. We offer a common architecture across all.

FORRESTER®

WSO2 Named a Leader in The Forrester Wave™: API Management Solutions, Q4 2018 Report

wso2.com

Ballerina: A Language for Network-Distributed Applications

Network Aware, Team Friendly, and Cloud Native

Andy Oram

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Ballerina: A Language for Network-Distributed Applications
by Andy Oram

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Ryan Shaw
Development Editor: Jeff Bleiel
Production Editor: Deborah Baker
Copyeditor: Octal Publishing, LLC
Proofreader: Christina Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition
2019-08-06: First Release

978-1-492-06113-7

[LSI]

Table of Contents

CHAPTER 1
Introduction

Ballerina, a recently developed programming language released under a free and open source license, has followed the same evolutionary path taken by other computer languages. History shows that new concepts in computing—such as object orientation or design patterns—at first are cobbled onto existing languages. Thus, early graphics-rendering libraries followed the convention that "the first argument in the parameter list is the function's object." And the classic *Design Patterns* (Addison-Wesley), by Erich Gamma, John Vlissides, Richard Helm, and Ralph Johnson (a group referred to as the "Gang of Four"), implemented patterns such as iterators manually in C++, the broadly recognized language of the day. But newer languages incorporated these ideas into their syntax—and thereby brought the concepts into everyday use.

Now comes *Ballerina*. A fairly conventional procedural language in constructs and flow control, it endeavors to incorporate the best practices we know in the computer field about distributed computing, web programming, microservices integration, and Agile or DevOps-oriented development.

The report is not a tutorial on Ballerina, because I do not need to repeat information offered by Ballerina's own numerous online resources, including examples of selected features and of large-scale constructs, a language specification (the current version of Ballerina is 0.990), and an API reference. This report focuses on providing a context for understanding what Ballerina offers and how it solves modern development problems.

CHAPTER 2
Integrating the Environment

Ballerina's designers surveyed the needs of cloud-native, distributed programs and added language features to meet those needs more easily. Consider some of the differences in programming found in modern applications:

- Programs often run on multiple cores and are divided across multiple systems, connected through RESTful interfaces, message queues, or remote procedure calls (RPCs).
- Programs expose themselves online as web services and call out to other web services.
- Errors or failures from communicating services must be handled gracefully.
- Programs need to deal with untrusted data and authenticate themselves repeatedly to services.
- Databases are tightly integrated into program activity.
- Data is often structured, instead of coming in as byte streams that need to be parsed.
- Performance is a constant concern and must be measured obsessively. The environments in which these programs run are monitoring them, collecting metrics, and logging information—a set of tasks known as *observability*.

All modern languages provide libraries to handle these tasks, but Ballerina builds many of the tasks more directly into the language. It was created by WSO2, an API-first integration company, because its

managers saw how the computer field had moved in these directions and wanted to cut down development time. According to Ken Oestreich, vice president of product marketing at WSO2, "Development is becoming more about integration, and integration is becoming more code based."

Ballerina also comes with tools that work well with modern development practices, such as multiteam project management, Continuous Integration (CI) based on test suites, and other innovations that have been loosely grouped under the term DevOps. Ballerina offers plug-ins for the Visual Studio Code and IntelliJ IDEA integrated development environments (IDEs) as well as its own IDE. The Ballerina IDE provides a valuable visualization tool: sequence diagrams (which we'll look at in "Sequence Diagrams" on page 23).

As a fully fledged general programming language, Ballerina will probably compete most directly with Go in the distributed-computing market and with Node.js in the web-application market.

We now turn to the features Ballerina offers that make it network aware, team friendly, and cloud native.

A Network-Aware Language

Modern applications operate over networks, often mixing a range of network activities such as:

- Creating RESTful clients and servers
- Giving access to third-party services such as Salesforce, and acting as services themselves
- Making database queries
- Completing asynchronous calls
- Producing and consuming streaming data
- Exchanging data in common formats such as JSON
- Message-passing
- Logging
- Creating performance and reliability metrics

High-level Ballerina features make all these things easier to use and work with. Services are first-order objects, just as functions are in many other languages.

Thus, the HTTP module, in addition to all the usual activities such as GET and other verbs, allows you to create a service (known as a *listener*) or a client. The HTTP module also provides standard HTTP caching. A server in Ballerina can set the headers that control caching in each response it sends, such as how long to leave the resource in the cache.

Connectors to a variety of services, including Salesforce and the Twilio mobile phone interface, are provided through convenient modules. Access to relational databases is fairly conventional, supporting SQL statements with placeholders, transactions, and simple calls following the create-read-update-delete (CRUD) model.

Integration with such services typically relies on conveniently incorporating configuration information into Ballerina. To log in to Salesforce, for instance, you need a URL for the site you're logging in to, plus some client information such as ID and password. To log in to GitHub, instead of a URL you pass an organization name and a repository name. To attach to a database, you need the hostname, port, and so on.

The regular I/O module provides an interface for writing to and reading from streams. Several message-passing schemes, such as ActiveMQ, Kafka, and JMS, are supported through other modules.

Ballerina simplifies integration of all these services by letting you store a configuration in the native *map* data type. A map is a key/value store, sometimes known as an *associative array*, *hashtable*, or *dictionary* in other languages. In a map, for example, you could define a key of "author" with a value of "Andy Oram" (or an array of multiple authors). All members of a map must be the same data type.

When you use a service such as Kafka, with its particular features such as topics and polling, you can specify them in a map and execute all of the necessary information using convenient Ballerina syntax.

Ballerina also offers simple arrays and a *record* data type, which is a structure containing fields of varying data types. In a record, you probably would define a string to hold the value "Andy Oram".

Asynchronous calls are crucial in any kind of distributed programming, so Ballerina takes major pains to hide their implementation from the programmer. When a program launches a call across the

network, Ballerina executes the network operations without blocking. All you need is the `start` keyword to launch a function asynchronously. (You can even specify an anonymous function, which is a function defined on the fly.) The program returns with a value that you assign to an object called a *future*. Then, you issue a `wait` call on this value to retrieve the result of the asynchronous function.

The ease of issuing asynchronous functions facilitates the building of complex concurrent programs such as master/worker relationships, pipelines, publish/subscribe models, and brokers.

All of the most common forms of authorization over the web are built into Ballerina: basic web authorization, JSON Web Token (JWT), and OAuth2. These are part of a fairly comprehensive approach to security, described later in this report.

Ballerina also makes data exchange easy by providing two popular formats, JSON and XML, as native data types. In addition, comma-separated values (CSV) data is fairly easy to parse as normal strings. Functions for reading and writing files or network content in these formats as well as for extracting data from the formats and manipulating data in those formats are built into standard libraries.

Observability features such as logging and metrics are built into Ballerina and integrated with common free and open source tools. This makes it easy to log error messages to a central repository, trace events such as failures, and view statistics about program behavior.

Ballerina supports the two common types of web logging (access logs and trace logs) and can issue messages at commonly recognized levels (debug, warning, trace, and so forth).

Metrics are also native to Ballerina, which automatically records information on network activity. Within your program, through the *observe* interface, you can filter network information and choose what to collect. The key concept behind tracing in Ballerina is the span, which has a start and stop that you specify within your program. You can also take a snapshot of the collected statistics and use counters. Some examples of graphs produced by the *observe* interface appear in "Observability in Ballerina" on page 24.

You can monitor statistics at the application level, such as how many SQL queries or how many HTTP GETs have been executed. Ballerina uses the Prometheus toolkit for monitoring and alerting. It adheres to the OpenTracing standard, plugging in to standard dis-

tributed tracing tools such as Jaeger. Through distributed tracing, you can view performance issues and network failures from a series of program runs.

A Team-Friendly Language

About a decade ago, the term "DevOps" was coined to reflect some groundbreaking ways in which organizations were changing the methods they used to created software. In DevOps, changes are tested quickly and deployed through automated methods (often in the cloud) instead of going through a traditional test and deployment cycle. The concept of DevOps draws a lot on earlier concepts such as Agile programming and is concerned with organizational structures and planning as well as with particular technologies. Indeed, the technologies on which DevOps is based—CI, automated configuration tools, distributed version control—predated the invention of the term. As a relatively new language coming into being when this combination of practices has become mature and well-established, Ballerina is designed around their practice.

Ballerina has its own build system based on the idea of a *project*, a way to organize development efforts that is already familiar from IDEs. As stated earlier, Ballerina also has plug-ins for popular IDEs. There is also a Ballerina repository where you can upload projects and share them with others: your own team members, other collaborating teams, or the general public.

A module for CI and testing is also built into Ballerina. You can add assertions and trigger test runs within a program.

A Cloud-Native Language

Cloud environments come in numerous varieties, both Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). They encourage the intensive use of network-aware features and DevOps-related team development practices, which were discussed earlier in this report.

In addition, as many legacy applications move to the cloud, new ones tend to be built around microservices and containers. Ballerina is designed for both. It currently has modules that deploy programs on Docker, Kubernetes, or AWS Lambda. You can build a program for one of these container frameworks and start it inside the frame-

work with minimal configuration. Along with Kubernetes, you can integrate your program into the Istio service mesh.

Other Notable Language Features

Developers concerned with network-aware, team-friendly, and cloud-native programming will be particularly interested in several other language features.

Compiler extensions
: This allows you to add new modules and annotations. This feature lies behind the Docker, Kubernetes, and AWS Lambda deployment options mentioned in "A Cloud-Native Language" on page 7.

Security
: In addition to the authentication modules mentioned earlier, Ballerina offers an encryption module and a powerful feature called *taint checking* that is particularly important in the world of networking. Taint checking traces the flow of data through each function and determines whether the data might have dangerous content. Typical dangers are SQL injection (when a malicious visitor includes database commands in data that is meant to be sent to a database) and HTML injection (when a malicious visitor includes code in a comment or other content uploaded to a public website). The colorful word "taint" derives from the introduction of the concept into the Perl language, where it has been since the mid-1980s.

  Tainting marks any data received from outside as unsafe: data retrieved from a database, entered by a user in a GET or POST request, taken from another service, entered on a command line, and so on. You must explicitly "untaint" the data, which you generally do after running it through a check. For instance, you can match against a regular expression to check data accepted by a user for suspicious SQL before entering it into a database, or for dangerous HTML tags before posting it on a website. You must also put code into sensitive functions to issue errors and fix or reject data that is still tainted. Because a failure to untaint the data is caught by the compiler, the developer can fix unsafe practices before running the program.

Concurrency
: Besides the many forms of concurrency mentioned earlier, such as asynchronous execution of network calls, Ballerina includes constructs to run subprocesses or multiple threads on the cores of a local system.

Error checking
: This is very rich in Ballerina. As in Java, any function can declare and return an error. Ballerina simplifies error handling by allowing a function to return a variety of data types. This is often used for error handling. A successful function returns the value it was meant to generate (a string, integer, tuple, or other data structure). But a function that encounters a problem simply returns an error object. This works through a language feature called a *union*, a well-known element of the original C language. However, in modern C there is very little use for unions. In Ballerina, they are extremely valuable for error checking. Typically, the caller checks the type of the value returned by each function and takes action to recover if an error is returned.

  Ballerina also makes it easy to defer error checking. Any statement can be preceded by the `check` keyword, which means, "If the following statement produces an error, return from this function immediately and pass the error to the caller." We can use `check` to pass errors up the call stack as far as we want, handling them where we find it convenient.

  The traditional practice of obsessively checking for errors on every operation—I've seen advice that you should even check for errors from a call closing a file—litters the source code with boilerplate `if` and `else` clauses. The code is cleaner if you pass errors on and check for them in a high-level function. Of course, this choice has downsides, too: you might run the application for a while after it has failed, and you lose opportunities for fine-grained logging and error messages. Ballerina gives you the option of handling errors at the point you find appropriate.

CHAPTER 3

Using Ballerina: A Sample Application

This section describes how to use Ballerina efficiently to produce sleek, readable code. The example presented comes from a small but fully functional service that carries out one tiny part of an online retail store: finding information about products in a customer order. Here are some of the features illustrated by this small application:

- Cooperating services
- Asynchronous calls to external services
- RESTful web communication
- Database access
- Error handling
- Format conversion, particularly involving JSON

The application is made up of the following parts:

Store service
: This receives an order ID over the web and returns a record containing various information such as the name, price, and inventory stock of each product. To do so, this service queries the Order service.

Order service
: This receives an order ID from the Store service and returns a record containing information about each product in the order.

To do so, this service queries a database, the Product service, and the Inventory service.

Product service
: This receives a product ID from the Order service, and returns a record containing the name and price of that product. To do so, this service queries the same database as the Order service.

Inventory service
: This receives a product ID from the Order service, and returns a record containing the stock (number of available items for that product). To do so, this service queries the same database as the Order and Product services.

Database
: We're using MySQL for this example, but the queries are very simple and could be run on any database with an SQL interface.

For such a small example, you could easily imagine combining some services. In particular, the Store service doesn't seem to add anything to the information returned by the Order service. But think of these services as part of a much larger application doing retail sales. We can assume that a real-life Store service would do many more things than we do here. So we break off services that perform one simple function, hiding details such as database calls from high-level services. Figure 3-1 shows the services and the flow of calls among them.

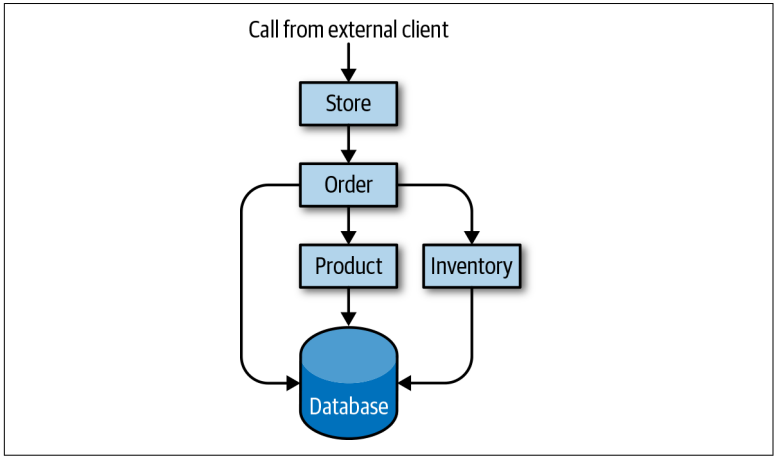


Figure 3-1. Services and their interaction

You can find the complete code available for download from *http://bit.ly/ballerina_report*.

In the rest of this section we'll walk through the most interesting parts of the code, leaving out some declarations and repeated code that you can take for granted. Along the way, I'll point out areas where Ballerina streamlines or facilitates coding. (Note that sometimes a long line in the code is split across two or more lines in the example so that it can fit the width of the page.)

The Store Service

We'll start with the top-level service, the Store. Here, as in the other two services, the preferred format for in-memory data is the record, which is well suited to database interaction. The code has to do a lot of translation back and forth between maps, records, and the JSON objects that are the most common way to transmit data over a RESTful web service.

So, first we'll define two simple records that contain information of interest to the customer. `Product` holds the string and price, whereas `Inventory` indicates how many are available:

```
type Product record {|
    int id;
    string name;
    float price;
|};

type Inventory record {|
    int productId;
    int stock;
|};
```

Being a Ballerina record, each data structure lists a data type and name for each field. The vertical bars indicate that each record is closed, meaning that we don't plan to add more fields dynamically. The `id` and `productId` fields correspond to a typical primary key in a relational database: a unique, arbitrarily chosen, positive integer.

We can also define an `Order` record, which stores an array of products and other useful information for a sale:

```
type Order record {|
    int id;
    float total;
    boolean processed = false;
```

```
    Product[] products?;
|};
```

The `Order` record contains a flag to indicate whether the order has been processed. This flag will be set by the Store service after successfully receiving all the data about the products. At the start, when each `Order` is created, this flag is explicitly set to `false` as the default value. The record ends with a simple array of products using the `Product` type defined earlier. The question mark in that field allows an `Order` to be defined without that field. This is helpful in case we need to create an order without any products and then create the array later.

HTTP Service

Now we'll define a service to accept order requests from some outside source (not shown in this example). A RESTful service is a hierarchy of directories on the web. For instance, if we run the Store service on port 9799 of the local host, we expose its base path like this:

```
http://localhost:9799/StoreService
```

To process an order (suppose it has an ID of 524), anyone with the proper access can issue the following, which turns into an HTTP GET request to a service named `processOrder`:

```
http://localhost:9799/StoreService/processOrder?orderId=524
```

Similarly, the Order service runs on port 9797 of the local host, exposing its base path like this:

```
http://localhost:9797/OrderService
```

This site might offer typical services for creating (the HTTP POST command), reading (GET), updating (PUT), and deleting (DELETE) orders, along with other special functions. Anyone with the proper access can read information on order 260 by invoking the following, which issues a GET request to the `getOrder` service:

```
http://localhost:9797/OrderService/getOrder?orderId=260
```

We'll soon see how the Store service makes use of the Order service's RESTful interface. But now, we're going to set up the Store service itself. First an annotation, which begins with an at-sign (`@`), to invoke features from the `http` runtime:

```
service StoreService on new http:Listener(9090) {
    @http:ResourceConfig {
        methods: ["GET"],
        path: "/processOrder"
    }
```

As explained in "A Network-Aware Language" on page 4, Ballerina makes it easy to work with services, including third-party services, by placing configuration information in a map as shown. In this map, the `methods` key indicates that our service handles only GET requests, and the `path` key lists the path we saw earlier in a URL.

Now we can define the service containing a resource function that outsiders invoke to process an order. As is typical for REST, we have an endpoint on the web and a function to process requests made to that endpoint:

```
    resource function processOrder(http:Caller outboundEP,
                                   http:Request req) returns error? {
```

We've chosen to name the function that accepts requests with the same name as the path where we accept requests, `processOrder`. The `http` runtime will invoke the function and pass it the client's endpoint and request. The rest of the function unpacks the request, invokes the Order service to get product information, and returns that information. Because the function's job is to return information to a caller over the network, the function doesn't need to return any value. However, it returns an `error` data type if something goes wrong.

The function starts by getting the parameters from the request, using `getQueryParams`, a resource function from the `http` runtime. This function returns a map, which is suitable because the query can define fields with any name, such as `orderId`, and set them to arbitrary values:

```
        map<any> qParams = req.getQueryParams();
```

Now we use a built-in resource function of the `int` data type to convert the string in the `orderId` parameter to an integer. If the parameter is not an integer, the `convert` function returns an error. Thus, the resulting `orderID` can be either the data type we want (an integer) or an `error` data type. The syntax of the following call partly shows how error checking is built into Ballerina. Any function can return a variety of data types, but this Ballerina feature is normally used as we do here—to return an error when necessary:

```
int | error orderId = int.convert(qParams["orderId"]);
```

Before invoking the Order service, we'll do two error checks. We can proceed only if the `orderId` is an integer and if it's greater than zero. The type-check operation called `is` returns true if the variable on the left side is an instance of the data type on the right side:

```
if (orderId is int && orderId > 0) {
```

Having passed this check, we run the function `getOrder`, which we define later in the file, and which will contact the Order service. We'll get back either JSON-formatted data or an error:

```
json | error retrievedOrder = getOrder(untaint orderId);
```

Because `getOrder` passes the `orderID` over the network to another service, we need to deal with taint checking. As explained earlier, Ballerina traces the flow of data through the function. Anything that comes from an untrusted source, such as a file or network connection, is considered "tainted." If you try to pass tainted data as an argument to a call that goes over the network, the call will fail at runtime. Taint checking flags the problem at compile time instead.

You can also define functions that must be invoked with untainted data by adding the `@sensitive` keyword to a parameter. The syntax for taint checking and defining sensitive functions is trivial in Ballerina, but because it enforces taint checking, programmers are forced to avoid some of the dangers of dealing with external input.

The preceding `if` statement checked to make sure `orderId` is an integer, so we know it can't contain dangerous characters. We can safely include the `untaint` keyword so that Ballerina will untaint the variable, and the `getOrder` function can use it for a network call.

For now, let's assume that the guards are satisfied and that we have issued the `getOrder` call. As with the `convert` function we just saw, our `getOrder` function can return either a valid item of data or an error. A valid item of data will be in JSON, the favored format on the web and a built-in Ballerina data type. But before unpacking the JSON, we'll check the return value in a manner that's typical for all languages:

```
if (retrievedOrder is error) {
    log:printError("error in retrieving order details.",
                   err = retrievedOrder);
    respond(outboundEP, "error in retrieving order details.",
```

```
                    statusCode = 500);
        }
```

We're now nested within two `if` statements: the first checked whether the `orderId` was valid, whereas the one shown here checks whether we received an error back from our call. The code just shown demonstrates logging, which is crucial for tracking what goes wrong with network services or with applications in general. Logging is easy in Ballerina, and provides functions to log different things at typical levels such as debug, warn, and error.

After logging the error, we tell the caller about it. We have written a small function called `respond` that takes three arguments. Let's look at that function now, although it comes later in the source code:

```
function respond(http:Caller outboundEP, json | string payload,
        int statusCode = 200) {
```

Two of the arguments show interesting Ballerina features. The second argument accepts either JSON or a plain string as its data type. The third argument demonstrates that arguments to functions, like fields in records, can take defaults. If the `respond` function is invoked with just two arguments, it assumes everything went fine and returns an HTTP 200 status code indicating success. However, because we're invoking the function to report an error, we pass a third argument of 500 to indicate an internal server error.

Because the payload is sent over the web, we'll format it as JSON. You'll see throughout this application that we use JSON for data exchange among services. So we create a standard HTTP response and set the necessary fields:

```
http:Response res = new;
res.statusCode = statusCode;
res.setJsonPayload(payload, contentType = "application/json");
```

We finish the `respond` function by sending out the response. Naturally, that send can fail, too, and if it does, we log that information:

```
    error? responseStatus = outboundEP->respond(res);
    if (responseStatus is error) {
        log:printError("error in sending response.",
                       err = responseStatus);
    }
}
```

This useful little function is called by all four services in our application. For a large application, of course, we'd create a module of con-

venience functions and include it in each service. But we won't bother doing that here for one short function.

The `->` operator creates a network call. By default, observability is turned on by Ballerina, so anything run with the `->` operator is traced, producing metrics that you can view later. We'll review examples in "Observability in Ballerina" on page 24.

Let's go back to our `getOrder` function. We just finished error-handling in an `if` clause, so we'll enter an `else` clause that runs when there is no error. Here, we'll call the `respond` function to send back the order. We untaint the order so that it can go out over the network. We'll also omit the third argument so that we get its default value, 200, for success:

```
        else {
            respond(outboundEP, untaint retrievedOrder);
    }
```

The `processOrder` function finishes with some error-handling similar to what we've already seen, so we'll move on to something more interesting: calling out to another service.

The Store as Client

Now we need to write the code that lets us act as a client to the Order service. In one line, we can set up our endpoint on the web:

```
http:Client clientEP = new ("http://localhost:9091");
```

Remember that, to contact the Order service, we invoked a function named `getOrder`. Let's step through that function now:

```
function getOrder(int orderId) returns json | error {
```

We showed earlier the */OrderService/getOrder* endpoint. This is how we contact it, passing the `orderId` that we know is a valid positive integer:

```
var response =
    clientEP->get("/OrderService/getOrder?orderId=" + orderId);
```

We use the `var` keyword, instead of a precise data type, to cover different possible results: we might get either a valid HTTP response or an error from our `get` call. So next we check what we received:

```
    if (response is http:Response) {
```

Inside this `if` statement, knowing we have a valid response, we unpack it:

```
json payload = check response.getJsonPayload();
```

The `check` keyword encapsulates a potent error-handling technique, mentioned in "Other Notable Language Features" on page 8. If an error is returned from the statement that follows (in this case, a call to `getJsonPayload`), the `check` keyword causes the current function to terminate and send the error back up the stack. If execution proceeds normally, we know that `getJsonPayload` succeeded and can confidently refer to the `payload` variable, knowing it has valid data:

```
var productOrder = Order.stamp(payload.orderDetails);
var productInventory = Inventory[].stamp
                       (payload.inventoryDetails);
```

In these two lines, we change the JSON in the payload to our internal record format. Ballerina makes things easy here, too. Its `stamp` function accepts any data type it understands and stores the contents as another data type. It is very similar to `convert`.

We defined the `Order` and `Inventory` types in this file, so `stamp` can figure out their structure and convert the JSON to a record. The square braces `[]` define an array of `Inventory` records, so we can get information on multiple products.

But we also need to verify the results. Both `productOrder` and `productInventory` were defined as generic data through the `var` keyword, so we can't be sure that the content of the response was the correct data type unless we check:

```
if (productOrder is error) {
    log:printError("order data received in invalid.",
                   err = productOrder);
}
if (productInventory is error) {
    log:printError("inventory data received in invalid.",
                   err = productInventory);
}
```

We saw the `is` operation used earlier on a built-in data type, `int`, and now we use it on our own data types.

A concluding `if` statement does one more check that we have good data. In this block, we set the `processed` flag so future users will know we have processed the order, package up everthing in JSON, and return from the function:

```
if (productOrder is Order && productInventory is Inventory[]) {
    productOrder.processed = true;
    json finalPayload = {
         orderDetails: check json.convert(productOrder),
         inventoryDetails: check json.convert
                                  (productInventory) };
    return finalPayload;
}
```

The Order Service

Much of the code in this service is similar to the Store service: we set up an HTTP service, accept a request, and make some calls of our own to other services. What's special in the Order service is its use of asynchronous calls to run two queries simultaneously. You'll see how simple that is, and also make a database query and retrieve the results.

The database has a typical one-to-many table for which the key is the product ID and one of the columns is the order ID. By passing the order ID to a query, we can get back all the product IDs. But this is not enough for the Order service, because we also want other product information. We can get these from the Product and Inventory services.

Setting Up the Order Service

Our Order service defines the same record types that we saw in the Store service, plus another little one to hold a product ID:

```
type OrderEntry record {|
    int productId;
|};
```

Like the Store, we implement a function that we expose on an HTTP endpoint of the same name. Our code creates an Order service just as we created a Store service:

```
@http:ServiceConfig {
    basePath: "/OrderService"
}
service OrderService on new http:Listener(9091) {
    @http:ResourceConfig {
        methods: ["GET"],
        path: "/getOrder"
    }
```

As we saw in the call made by the Store service, the endpoint that handles incoming requests is called `getOrder`. So is our function.

```
resource function getOrder(http:Caller outboundEP,
                           http:Request req) returns error? {
```

Asynchronous Invocation

I won't bother showing the initial code that retrieves data from the request, because it is essentially the same as the Store order. We get an array of product IDs and issue two asynchronous calls:

```
        if (productIds is int[]) {
            future<Order|error> productOrderFuture =
                start getProductsForOrder(productIds, orderId);

            future<Inventory[]|error> inventoryDetailsFuture =
                start getInventoryForOrder(productIds, orderId);
```

Two key elements of syntax consitute an asynchronous call: defining the return value as a `future` and launching the function through `start`. The `future` declaration might look complicated at first, but it's basically like many other return values we've seen in Ballerina:

```
            future<Order|error>
```

This just means that the call can return an Order or an error. The other `future` is similar, but retrieves an array:

```
            future<Inventory[]|error>
```

Having launched two functions, we could do other processing, but we don't have any other work to do, so we'll wait for the two calls to return. That entails simply calling `wait` on the two variables we just declared as `futures`:

```
map<Order|Inventory[]|error> result =
  wait { productOrder: productOrderFuture,
         inventoryDetails: inventoryDetailsFuture };
```

Let's dissect that compact call. We have seen the `map` keyword before. It allows both an Order and an Inventory to be returned, or an error:

```
map<Order|Inventory[]|error>
```

And after the `wait` keyword, we see exactly that the map has two fields, one with the key `productOrder`, and the other with the key `inventoryDetails` (which we know from the `start` statement to be an array):

```
{ productOrder: productOrderFuture,
  inventoryDetails: inventoryDetailsFuture };
```

Though it requires detailed code to extract JSON, check for valid formats, and return the data to the calling service, this code simply requires variations on things we've seen in other services. So we'll just spend a moment on the database call before moving on to new topics.

The Database Query

The activities in this section will be familiar to anyone who has accessed a database from a programming language using something like the Open Database Connectivity (ODBC) standard. We need to establish a connection to our database using several parameters such as the host running the database and our access information. In a production environment, of course, you would get your access information from your organization's secure access system. Cloud providers, in particular, have elaborate and secure systems for retrieving passwords, and these vary from vendor to vendor. Ballerina can also retrieve values from encrypted configuration files.

Ballerina interacts with the database using maps, as it does with other services. The following snippet shows how we issue our SQL query, using a question mark as a placeholder, as in other languages:

```
sql:Parameter param = {
    sqlType: sql:TYPE_INTEGER,
    value: id
};
var result =
  dbClient->select(
             "SELECT productId FROM ORDERS WHERE orderId = ?",
                    OrderEntry, param);
```

The following code shows how Ballerina extracts data from the results of an SQL query, and how it fills an array. After we define `productIds` as an empty array, we retrieve one row at a time from the query results and add it to the array, which expands automatically to accept each new product ID:

```
    int[] productIds = [];
    foreach var row in result {
        var productId = check row.productId;
        productIds[productIds.length()] = <int> productId;
    }
    return productIds;
```

All the interesting features of the Store application have now been shown. Certainly, the Order service has a lot more work to do: it must define the `getProductsForOrder` and `getInventoryForOrder` functions that we invoked asynchronously. But those functions work very much like the `getOrder` function that the Store service defined to call the Order service, so we don't need to cover them. The Product and Inventory services will also be familiar to you after you've used the code covered so far.

Sequence Diagrams

Ballerina is a well-structured language, based on a nesting of function calls and control flow syntax that lends itself to visualization through sequence diagrams. Because network communication is so important in most Ballerina applications, the Ballerina developers encourage programmers to consider structure carefully, as well, and include it in the sequence diagrams generated by the Ballerina IDE.

Ballerina's client connectors, workers, and remote endpoints are represented as actors in each sequence diagram, and messages passed between them are represented as actions. For each network request, the diagram shows key information such as which function handles the call, what arguments are passed, and how the call fits into the structure of the calling function.

To some extent, thanks to the sequence diagrams, Ballerina code is self-documenting. They help in architectural development, debugging, and identifying interdependencies.

To illustrate how a sequence diagram can illuminate the workings of the Store service we saw in this chapter, let's view a snippet of code from the `getProductsForOrder` function and see how it turns up in a sequence diagram. The function generates a separate request to the Product service for each product in the order. Thus, the function issues requests in a loop, which starts as follows:

```
foreach var id in ids {
    http:Request req = new;
    int pId = check sanitizeInt(id);
    var result =
       check productServiceEP->get(
            "/ProductService/getProduct?productId=" + pId);
    var payload = result.getJsonPayload();
```

Figure 3-2 is a sequence diagram generated from the Order service by the Ballerina developer IDE, focusing on the GET request in `getProductsForOrder`. The diagram shows an unnamed client on the left, the Order service in the center, and the Product service on the right. The diagram contains the most important elements of each call—for instance, in the upper left you can see that the client sends IDs and an Order ID to the Order service. The diagram also shows how the Order service issues multiple GET requests to the Product service in a loop.

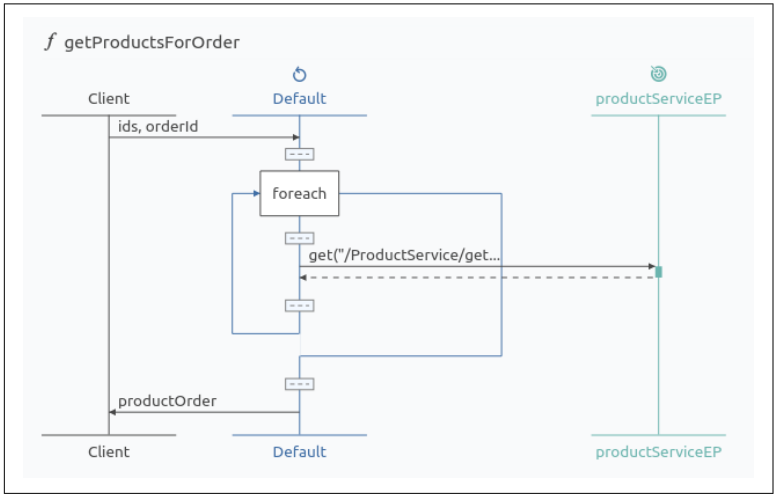


Figure 3-2. Sequence diagram of the Order service

Observability in Ballerina

As explained in "A Network-Aware Language" on page 4, Ballerina builds in many observability features. To illustrate these, we'll look at a few screenshots generated by runs of the Store service described in Chapter 3.

Figure 3-3 shows some typical metrics from successful runs. It was produced through the Grafana charting tool for system monitoring. The time periods in three of the four graphs—Throughput, Response Time Percentiles, and HTTP Status Codes—cover three minutes, from 4:29 to 4:32. The services seem to be humming along pretty well until slightly after 4:30:50. At that moment, something not visible in these screenshots (a service or network failure) brings everything to a halt.

Figure 3-3. Typical metrics

Because we have stopped displaying data right after the failure, we show an error rate of 0.09% at the right side of the top bar. Response Time Percentiles is the first graph to reflect the failure, with a sudden rise in response time before 4:31. The Throughput and HTTP Status Codes graphs quickly follow Response Time Percentiles in showing a total failure.

The highest activity in all three graphs is a red line that reflects a `getOrder` call. This makes sense because that call in the Order service is the most heavyweight part of the system. It must call the Product and Inventory services as well as issue a database query. The tiny legend at the bottom of each graph indicates that the call is handling a GET request from port 9091, which is the client endpoint defined by the Store service.

The `getOrder` call is actually not a CPU hog, probably because it spends most of its time waiting. Far more CPU time seems to be spent on two other services, as we see in the bar graph at the lower right. Here the large green bar belongs to a `checkInventory` call, and the large orange bar to a `getProduct` call, which are invoked in other services by the Order service.

Now we'll look at the life cycles of some calls, through traces produced by Jaeger. Figure 3-4 is a sprawling overview of calls triggered by the `processOrder` call in the Store service.

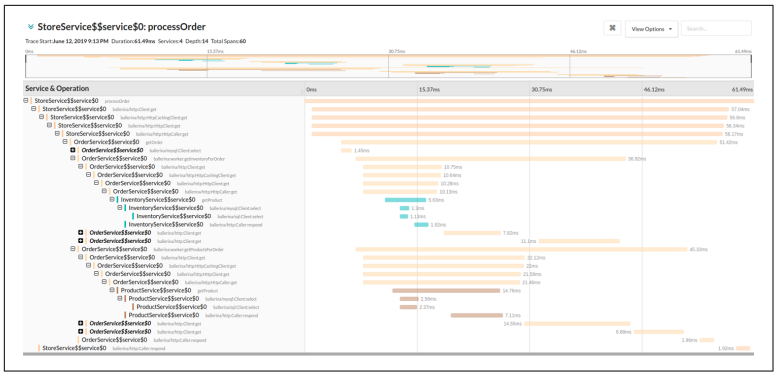


Figure 3-4. Functions cascading from a Store service request

The left side of the graph shows (in faint gray type) the functions called, in order from top to bottom. The right side of the graph shows when each call started and ended, with numbers to indicate how long it took.

A lot of calls involve the HTTP cache mentioned in "A Network-Aware Language" on page 4, even though our application doesn't use the cache. The graph shows that the two asynchronous calls made by the Order service do indeed run in parallel. It also shows that `getProductsForOrder` takes the most time, followed closely by `getInventoryForOrder`. This shows that we were justified in running them in parallel.

Conclusion

On the surface, Ballerina looks like many other C-style languages, making it a fast study. Where it differs from other such languages is in how it builds in modern computing concepts from the ground up as first-class language properties. What in other languages might require struggling with configuration files and command-line options (leading to the all-too-familiar phenomenon of using preprocessing and postprocessing scripts for deployment) can in Ballerina be done directly within the program. Thus, this language should speed development and reduce failures in modern cloud-native, distributed, integrated environments.

About the Author

As an editor at O'Reilly Media, **Andy Oram** brought to publication O'Reilly's Linux series, the groundbreaking book *Peer-to-Peer*, and the best seller *Beautiful Code*. Andy has also authored many reports on technical topics such as data lakes, web performance, and open source software. His articles have appeared in *The Economist*, *Communications of the ACM*, *Copyright World*, the *Journal of Information Technology & Politics*, *Vanguardia Dossier*, and *Internet Law and Business*. He has presented talks at many conferences, including O'Reilly's Open Source Convention, FISL (Brazil), FOSDEM (Brussels), DebConf, and LibrePlanet. Andy participates in the Association for Computing Machinery's policy organization, USTPC. He also writes for various websites about health IT and about issues in computing and policy.