Java 8: Completable Futures and Asynchronous Pipelines

By

Nolan Smith

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

August, 2015

Nashville, Tennessee

Approved:

Douglas Schmidt, Ph.D.

Jules White, Ph.D.

ACKNOWLEDGEMENTS

First, I'd like to thank Professor Douglas Schmidt for always encouraging my interests and directing my work. He has had enormous influence on my ability to think critically about design and implementation, as well as on my work ethic and dedication to the task at hand.

I would also like to thank Professor Jules White for helping in the origination of the idea for the application, and for being interested and supportive throughout my time working on the project.

I would lastly like to thank Professor Xenofon Koutsoukos and the Vanderbilt Engineering School for allowing me to join the graduate program to pursue my interests as a student.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Context

It is a commonly known trend that computing hardware grows in density and capacity at a rapid pace. However, developer competency in leveraging the advanced platforms' capabilities does not keep pace. As multiple core architectures and their performance benefits become a mainstream expectation reaching even mobile and wearable technologies, developer communities in every environment must learn to utilize the full gamut of hardware power available to them. Building tools and abstractions to support the developer learning process will secure the trend of increasingly powerful and responsive computational systems and applications [3].

1.2 History

Operating systems have long supported native threading for multicomputing purposes, and "green threading", or when threads are created and managed by a virtual machine (VM), has also been well established for years. Offloading long running computations to a secondary thread in order to prevent blocking a primary thread is a commonly observed technique in various areas of development. For example, such a technique drives many GUI applications in which user input may come at any unknown time. If the thread responsible for accepting user input is blocked, the application may feel unresponsive. Effectively managing threading and asynchrony is therefore critical to the usability of the application.

Further, though, modern applications tend to incorporate digital information from any number of disjoint web services, such as geographical location, price of goods, current events, and many others. As such, network requests, which are inherently indeterminate in running time and therefore are most effectively handled asynchronously, are being used liberally in today's applications. Because of the meteoric rise in the availability and utility of these services and APIs, the use of asynchronous code has become commonplace. As with most low-level abstractions, though, directly managing thread communication and scheduling is time consuming and verbose, and introduces inconsistencies across platforms and code bases. These asynchronous programming issues are compounded when operations depend on the results of a previously scheduled operation, or multiple operations, and such operations may fail independently.

Asynchronous solutions are often constructed using callbacks, which are a functions invoked upon completion of a given operation. Pipelines of operations require nesting callback functions which obfuscates the intent of each call individually, and can lead to a confusing tangle of code that is difficult to understand and build upon. Further, errors can cause the pipeline to fail in any phase, but relaying specific failure information back to the calling thread is difficult and requires extra code to selectively fail the remainder of the operations. This problem is compounded when certain phases are optional to the overall success of the pipeline.

1.3 Statement

The goal of this project is to present a solution free of common asynchronous pitfalls using native Java code in a client-server application setting. Further, the proposed solution should be easily read and understood by programmers of intermediate experience. The resulting application focuses on many Java 8 features, namely the CompletableFutures abstraction, the Streams API, and lambda expressions. These features play a pivotal role in unifying a well established language's interface for performing long running operations in complex dependency networks efficiently, while maintaining code readability and understanding.

1.4 Implementation

In order to display the flexibility of the Java 8 specific features regarding asynchronous computation, an example application was implemented. The application solution represents a network of dependencies between a number of real world, freely available web service requests in order to plan an eventful weekend at an input destination city.

Firstly, the application queries the Sabre Flights API to find a round-trip flight between the input origin and destination cities that is within the input budget, departing on the nearest Friday, and returning on the nearest Sunday. The timing and cost of the flight itinerary then serve as dependencies for when and how many ticketed events can be scheduled using the StubHub EventSearch API. Independently, the weather for the appropriate days is retrieved using the OpenWeatherMap API, and the call serves as an example of how the pipeline may be manipulated without affecting the outcome given the proper dependencies are upheld. The daily weather serves as a dependency for a query to the Google Places API for interesting things to do during the weekend outside of ticketed events.

While the example application is written primarily to convey the usage of the primary asynchronous programming abstractions available in Java 8, it also provides a general structure to a solution that could extend into other environments. The interfaces of the asynchronous method calls are critical to the interaction between phases of the pipeline, and could extend to any asynchronous operation chain to reduce code complexity and maintain modularity in an inherently dependent context.

CHAPTER 2

SOLUTIONS IN ASYNCHRONOUS PROGRAMMING

2.1 Callbacks and Handlers

Historically, asynchronous solutions hinged upon callbacks and handlers, which in general can be described as way to invoke code designated to run upon the completion of a typically long running operation occuring remotely. For example, a network request might also pass along an function indicator or closure that will have access to the result of the request. This technique is particularly effective when all the required information is retrieved from the single asynchronous call being made. However, problems arise when these methods are chained and the callbacks are nested.

Firstly, error handling must be done at each stage, and each stage must acknowledge each possible exception that may have been thrown in an earlier method of the pipeline. This lengthens and decentralizes the code necessary for handling exceptions. It is also difficult to relay the specific failure back to the calling thread, making appropriate cleanup difficult. Lastly, it is difficult to selectively fail or retry operations.

Secondly, code becomes difficult to test as invoking the asynchronous calls individually requires mocking the effect of each of the previous pipeline phases. The code is structured in a way that is dependent on the calling order, with relevant data being passed as parameters, rather than as singular units of computation capable of existing by themselves. This tends to lead to more traditional symptoms of poor design, and code becomes unmaintainable and unreadable.

2.2 Lazy Computation and Caching

While lazy computation is not a direct solution to asynchronous programming pitfalls, it is often used as an optimization to reduce the number of long running operations needed within an application session. Computing lazily means that data is not generated or retrieved until it is needed within a pipeline. A common example is iteration over random values to be used in later phases; Rather than computing a collection of random values of appropriate size upfront, random values are computed and supplied each time a phase of the pipeline requires one. This can be generalized into more common results caching of data requested from the internet, which can reduce the total number of network requests made in a given session. This technique does not act as a solution to the challenges presented by asynchronous programming, however lazy computation is an important consideration in asynchronous system design as unnecessary requests can be foregone.

2.3 "Future" and "Chainable Future" Abstractions

Many environments support "future" abstractions, which serve as a wrapper around a value that represents the result of an operation that may or may not be completed. These abstractions have been paramount to concurrent and asynchronous systems, but often plague the system with similar issues to a callback or handler based solution. That is, scheduling futures to interact with, or depend upon, one or many other futures leads to tangles of dependent code.

More recently, a chainable version of the future abstraction has gained traction in asynchronous environments that is able to invoke an operation or chain of operations immediately upon the given value's completion. In addition to eliminating the need to hand-synchronize multiple dependent asynchronous calls, it is often possible to run different phases of the pipeline in different threads, allowing independent calls to run concurrently. This departure allows natural pipelining of operations that can exist individually, and promotes sound design while maintaining the performance of an asynchronous solution. This is the abstraction upon which Java 8's CompletableFuture is based, and serves as the primary design element for the weekend planner application.

CHAPTER 3

JAVA 8

3.1 Executors

While the Executor and Executor Service interfaces are not new in Java 8, they are fundamental to asynchronous and concurrent Java code, and are used heavily in the weekend planner application. Both interfaces and their implementing classes serve to manage threading within a Java application, often acting as a scheduler managing an underlying thread pool. The power of the Executor family lies in the flexibility it affords the programmer by allowing thread pool implementations to be substituted with no change to the client code. Employing executors effectively reduces cluttered code accumulated in handling thread creation directly or even using a thread pool implementation.

A secondary, Java 8 specific benefit of using executors is that each method of the CompletableFuture class can specify an executor upon which to schedule the specified operation. This allows the executor implementation to handle the scheduling of background tasks with no infrastructure code necessary. When an executor is not specified, however, the operation completes in the same context in which the CompletableFuture completed. This knowledge affords programmers fine control over the progression of a pipeline while keeping code succinct.

3.2 CompletableFutures

Java 8's CompletableFuture is the manifestation of the "Chainable Future" abstraction in Java, and is the core element of the work presented in the weekend planner application. The CompletableFuture class provides a way to construct a pipeline of typically asynchronous operations. A few of the methods of the CompletableFuture class play a particularly important role in the weekend planner application and are presented here in detail.

The first is the “thenApply” method. The thenApply method takes a CompletableFuture, or completion stage as part of a pipeline, and transforms the CompletableFuture value once it is completed. It then returns a CompletableFuture of the transformed value. In this way the thenApply method works analogously to the “map” function of the Streams API, which is discussed in a later section.

Next, the “thenCompose” method plays a significant role in chaining asynchronous method calls. The difference between thenCompose and thenApply is that, where thenApply would transform a CompletableFuture into a nested CompletableFuture, thenCompose would flatten the nesting and return a typical CompletableFuture.

The “thenCombine” method is another flexible and heavily used CompletableFuture functionality in the weekend planner application. The thenCombine method takes two CompletableFuture parameters and combines them according to the parameter function as soon as they have both completed. The method is particularly helpful when a disjoint dependency, that is, a dependency that is not integral to previous phases of the pipeline, is introduced.

Finally, the “handle” method of the CompletableFuture plays an important role in exception handling in CompletableFuture operation pipelines. The handle method makes use of two parameters, one representing a successful result of the prior phases in the pipeline, and one representing a Completion Exception that might have been generated. The Completion Exception object acts as a wrapper around the thrown exception with additional functionality for failing through the remainder of the pipeline after a given phase fails. The handle method allows the success and exception cases to be treated individually, and in this way is useful for handling the exceptions of an individual CompletableFuture operation as well as those of an entire pipeline. For example, in a client-server environment, handling an exception is done most effectively by conveying what call failed back to the client for appropriate user notification. Using the handle method, this can be implemented in a centralized manner at the end of the pipeline. However in a mission-critical application, if a critical phase fails, it may be important to retry the operation or fail out of the pipeline completely and invoke a backup routine. This could be implemented using the handle method at each critical phase.

3.3 Streams API

The Streams API is another significant Java 8 addition that is used within the weekend planner application. Its purpose is more general than the CompletableFuture abstraction in that the Streams API brings lazy computation, infinite series, and monadic, functional-like programming to Java. Pipelining of operations is supported, but the Streams API is not tailored to asynchronous and long running operations as is the case for CompletableFutures.

One of the major benefits of the Streams API in addition to the succinct functional syntax is the simplicity of parallelizing operations. In many cases, parallelizing a stream requires a single method call. Under the hood, this call will use a thread pool to execute the pipeline concurrently. The Streams API can be leveraged in combination with the CompletableFutures abstraction to generate multiple CompletableFutures in parallel and wait for the completion of one or each of them. This functionality can be an optimization for making multiple independent requests.

3.4 Lambda Expressions and Functional Interfaces

A third significant addition to Java 8 is a language feature to make anonymous inner class creation more succinct called lambda expressions. This feature is effective in reducing infrastructure code when used in combination with the CompletableFuture “supplyAsync” method as this method is parameterized with a Supplier. Because Supplier in Java 8 is an interface with a single method that returns, or “supplies”, a value of a given type, a lambda expression can be used to replace a Supplier with a typical method call that also returns type of T. In practice, this upholds the common trend of the additions discussed here to significantly reduce infrastructure code.

CHAPTER 4

WEEKEND PLANNER APPLICATION

4.1 Problem Definition

The primary goal of the weekend planner application is to provide an example of a way to organize a solution where dependent asynchronous programming is central to the functionality and performance. In order to deploy the Java 8 features in a non-trivial way, a full pipeline of dependent asynchronous operations was necessary. The problem of planning a weekend using freely available web services was a natural choice as a network of dependencies exists between the operations. At a high level the application retrieves real time flight information and schedules a parameterized number of different variants of the weekend plan. The plan incorporates real world ticketed event postings on StubHub in addition to low-cost or free places of interest such as museums and parks retrieved from the Google Places API. The dependencies between each pipeline phase is detailed below, and an overview of the architecture is given.

4.2 Implementation

4.2.1 Pipeline Dependencies

First, the Sabre Flights by Cities API was used to retrieve real time flight information. Retrieving the appropriate flight information depends upon the origin and destination city, as well as the budget input by the user. Further, the flight depends upon an implicit time constraint; The weekend in this context is considered to begin at 5:00pm on Friday in the origin city and end at any point on Sunday. Therefore the flight itinerary must have a departing flight between 5:00pm and 11:59pm on Friday in the origin city and must include a returning flight on Sunday. Further still, the Sabre Flights by Cities API is authenticated using an Open Authorization 2.0 token, which must first be generated by the server. This adds another, implementational dependency which must be handled by the application.

Secondly, event postings at the destination city are retrieved using the StubHub EventSearch API. The retrieved flight timing information is used to limit the event postings query to the appropriate time frame. The cost of the flight is used to limit the maximum ticket price in the query, as well as to select events to plan for each trip variant. In all the logical dependencies of the event query are the timing and the cost of the flight. The event service also requires an authorization token, but one that can be generated directly from the user development account information and needn't be retrieved from a server. However, the token generation is brought into the pipeline by invoking "thenCombine" so that it may be computed as soon as there are available resources and therefore concurrently to a network request, and allows for flexibility in the way that the authorization token is generated.

The pipeline is then constructed to retrieve the weather from the OpenWeatherMap web service. While the weather for the weekend is independent from the previous phases of the pipeline, it is composed into the pipeline in order to show the modular nature of pipelined operations. To this end, retrieval of the weather can be rearranged to any point earlier in the call chain and the pipeline will still produce identical results. It should be noted that the OpenWeatherMap service is unauthorized.

At any point after the weather has been retrieved, the Google Places web service is queried for places of interest at the destination city. The weather conditions of each day determine the query parameter that limits the scope of the request to certain types of places. Clear or cloudy conditions will search for amusement parks, campgrounds, parks, stadiums, and zoos, while rainy and snowy conditions will search for aquariums, art galleries, bowling alleys, movie theaters, museums, and shopping malls. Because the places requests are made by day rather than for the entire weekend at once, the Streams API is employed in combination with CompletableFutures to make each day request run concurrently. Here again the authorization token can be constructed locally and does not require a network request, however the destination city must be converted into latitude and longitude coordinates via reverse geocoding. The Google Places API has a reverse geocoding web service which is combined into the pipeline with the places requests.

The final stage of the pipeline is responsible for identifying and sending either the well formed weekend plan back to the client, or a detailed and specific message about an exception that might have occurred at any given phase using the handle method. The server handles each individual exception by sending detailed information back to the client, and allows the client code to handle exceptions more specifically. The weekend planner application displays a message to the user detailing the exception information.

4.2.2 Architecture

The organization of the solution is important in maintaining sound modular structure in a densely dependent environment. The code is divided into packages based on purpose, which are detailed below.

- WeekendPlanner

  The WeekendPlanner package holds the three primary classes that drive the application. The first is the WeekendPlannerServlet that is directly responsible for handling client requests. The server will respond to HTTP GET requests with valid cities from the Sabre Flights by Cities API, and respond to POST requests with a planned weekend at the requested destination city. The WeekendPlannerServlet is where the most concise use of the CompletableFuture abstraction is focused. The second is the WeekendPlannerResponse, which serves as a container for the data aggregated throughout the pipeline, and is directly sent back to the client. The WeekendPlannerResponse is the structure that each phase of the pipeline transforms. Finally, the WeekendPlannerOps class serves as an interface between the WeekendPlannerServlet and the individual Ops classes. It is primarily responsible for maintaining the executor upon which the operations are run, and submitting each asynchronous task of the Ops classes to the executor.

- Ops

  The individual Ops classes are responsible for maintaining constants and operations directly relating to the member web service. For example, the PlacesOps class stores the query parameters for indoor and outdoor places. Each Ops class shares a common base, BaseOps, which ensures that each Ops class defines how the service may be authorized in addition to maintaining the endpoint of the service. In all, the Ops classes are an aggregate structure of a Retrofit web service, the authorization information for that service, and the operations necessary to invoke the service.

- Requests, Responses, and Model

  The requests, responses, and model packages all represent data that is being sent and received between the client, the server, and the various web servers. The requests and responses packages are the top level structures, while the model package holds the nested objects of each response. Web service JSON responses are deserialized into these objects in order to manipulate the data as part of the pipeline. While they are plain old Java objects, each class exists as an inner class of the relevant response object to reduce the number of files necessary.

- Services

  The services package holds a collection of interfaces necessary for the Retrofit library. Each interface defines only the request operations to each related web service and the various query parameters used in the requests.

- Utils

  Finally, the utils package provides various authorization and date-time manipulation methods. For example, the DateUtils class is responsible for determining the date of the nearest weekend in addition to other various timing calculations. These methods are general purpose, and do not fit logically into any of the existing classes.

REFERENCES

[1] Oracle Corporation. Java platform standard edition 7, 8 documentation, Accessed 2014, 2015.

[2] Brian Goetz and et al. *Java Concurrency in Practice*. Addison-Wesley, Upper Saddle River, NJ, 2006.

[3] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Uppser Saddle River, NJ, 1999.

[4] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action: Lambdas, streams, and functional-style programming*. Manning Publications, Shelter Island, NY, 2014.