Multitier Languages for Microservice Architectures

Simon Schönwälder[1], Pascal Weisenburger[1], and Guido Salvaneschi[2]

[1] Technische Universität Darmstadt, Germany
{schoenwaelder, weisenburger}@cs.tu-darmstadt.de
[2] Universität St.Gallen, Switzerland
guido.salvaneschi@unisg.ch

Summary Throughout the last years, *Microservice Architectures (MSA)* became increasingly popular. This architectural style fosters the design of maintainable and scalable applications by composing distributed systems of small services that are independently develop- and deployable. Such microservices preferably communicate loosely using lightweight mechanisms. Yet, the approach potentially induces complexity overhead, as it forces developers to deal with a number of low level programming details such as communication protocols, data formats, or interface incompatibility.

Multitier programming by contrast allows developing the different parts *(tiers)* of a distributed application within a single compilation unit, abstracting away the complexity associated with distribution. This includes details such as remote communication, data conversion, callback-based flow, or coping with multiple technologies. The compiler splits the code of a distributed application into the corresponding tiers, adds all necessary network communication, and generates the deployment units.

We propose to combine the advantages of both worlds. Our approach provides language-level features to address modular design and development, but compiles down to microservices in order to retain scalability, fault tolerance, and ease of deployment.

Multitier Languages for MSA

Microservices ensure that different functionalities within a distributed application can be implemented independently and deployed in a highly decoupled way [2, 3, 5]. However, this comes with a complexity overhead on developers, for instance because they explicitly have to implement inter-service communication. Separation into services can also lead to developers losing sight of the overarching system goals, making it difficult to understand what is happening to any particular request [7]. We observe that regular languages already provide abstractions to structure large projects in a way that supports decoupling, hence proper maintenance from separate teams within organizations. Traditional languages provide little support for distribution though, leaving developers alone with the aforementioned complexity. Our approach aims at abstracting away the complexity of common approaches by providing multitier language-level features [13] for microservice architectures. Teams can develop concise MSAs using the native abstractions of a single multitier language.

Our solution adopts language abstractions like modules and interfaces to retain the fundamental advantages of MSA, such as support for *independent teams*, *separate business domains*, and *separation of interfaces*. The developer is spared from low-level programming details like communication, which is handled by the multitier compiler. Simultaneously, we aim to preserve crucial operational characteristics of MSAs like *replication*, *data ownership*, and *service heterogeneity*. To implement our approach, we adopt the *ScalaLoci* multitier language [10, 11, 12] and propose a containerization extension designed as a macro-based Scala compiler plugin for it to provide language-level MSA support. We package, ship and orchestrate the tiers of a ScalaLoci application into (Docker) containers, which are widely used for MSA [1, 3, 8, 9].

Multitier Containerization at Work

We now present an example of a distributed booking system in ScalaLoci to illustrate our solution. The system consists of three services: *UIGateway*, *RoomService*, and *HasherService*. It allows users to book a room by issuing a `/book` request, which will then be made persistent in an automatically set-up database. For the sake of brevity, we omit boilerplate not essential for comprehension.

ScalaLoci applications are composed of `@multitier` objects that can contain an arbitrary number of `@peer` declarations which define the tiers of the application. The annotation `@containerized` is introduced by our extension and labels a `@multitier` object as a domain module whose peers shall be subject to containerization. ScalaLoci `@peer` declarations are thus service declarations in our approach, and each `@containerized` object constitutes its own module domain to which all its contained `@peer` services belong.

```
 1 @multitier sealed trait Api{
 2     // Service declarations
 3     @peer type Service
 4     @peer type UIGateway <: Service
 5     @peer type HasherService <: Service
 6     @peer type RoomService <: Service
 7
 8     val bookingResponseStream : Evt[Response] on RoomService
 9     lazy val id: UUID on Service = placed { UUID.randomUUID() }
10
11     def book(booking : Booking) : Unit on RoomService
12     def calcHash() : UUID on HasherService
13 }
```

Listing 1: Api of BookingSystem's services

First, we make the functionalities of the services available through a common trait (Listing 1) were we declare the three services (Lines 4-6). These services share a common supertype `Service` that holds a service `id`. The components that we define later extend this trait with the specific service application logic. The `on` keyword specifies the location of data (Lines 8 and 9) and functions (Lines 11 and 12), e.g., `bookingResponseStream` is placed on the *RoomService* service. As an example of inter-service communication, the *UIGateway* may subscribe to the `bookingResponseStream` residing on the *RoomService*. The ScalaLoci compiler then automatically creates the communication code for the remote access, and the *UIGateway* receives all emitted events from *RoomService*.

Additionally, we use the case class `Booking` to implement room bookings, and the trait `Response` with two subtypes `Confirmed` and `Declined` to indicate booking success or failure.

```
@multitier trait UIGatewayImpl extends Api{
    @peer type UIGateway <: Service {type Tie <: Multiple[RoomService]}

    // logic of UIGateway
    def main() : Unit on UIGateway = on[UIGateway] {
        val routes = get { path("book") {
            (prename, surname, roomNr, from, until) =>
                val booking = Booking(prename, surname, Integer.parseInt(roomNr), from, until)

                // call the book function on a random connected RoomService
                getRandomService[RoomService] call book(booking)

                // subscribe to the response stream on RoomService
                bookingResponseStream.asLocalFromAllSeq
                    .map(_._2)
                    .filter(_.booking.roomNr == booking.roomNr)
                    .observe{
                        ... // handle and return Confirmed or Declined response
19                 }
20             }
21         }
22         ... // startup and manage HTTP server
23     }
24 }
```

Listing 2: Implementation of UIGateway

We now show the service implementation. Listing 2 contains the logic for the *UIGateway* service. It extends the previously defined `Api` trait so that it can access remote data not defined within its own scope. Listing 2 also defines the connection type to other services: the *UIGateway* service can connect to an arbitrary number of *RoomServices*, including zero (Line 2). Alternatively, `Tie<:Multiple[Service]` would allow all services to connect with each other.

The `main` method waits for requests at `/book` and it is in the *UIGateway* service. Upon receiving a request, the *UIGateway* service constructs a `Booking` object and remotely invokes the `book` method of the *RoomService* (Line 11). The compiler generates the necessary remote communication to serve this method call. Because there may be multiple connected *RoomServices*, we choose a random one to answer the request. The *UIGateway* service waits for a response by subscribing to the *RoomService*'s `bookingResponseStream`. The `asLocal` keyword (and variants like `asLocalFromAllSeq`) allows the read-only access to an object located on a different service – the compiler takes care of the remote communication. The boilerplate code for checking parameter validity, handling the request response back to the user (Line 18), and for starting the HTTP server (Line 22) is omitted for the sake of brevity.

```
@multitier trait HasherServiceImpl extends Api{
  @peer type HasherService <: Service {type Tie <: Multiple[RoomService]}

  // logic of HasherService
  // for simplicity we just return a random UUID here instead of real hash calculations
  def calcHash() : UUID on HasherService = placed{ UUID.randomUUID() }
}
@multitier trait RoomServiceImpl extends Api{
  @peer type RoomService <: Service {type Tie <: Optional[UIGateway] with Single[HasherService]}

  // logic of RoomService
  val bookingResponseStream : Evt[Response] on RoomService = placed{ Evt[Response] }

  def book(booking : Booking) : Unit on RoomService = placed{
    (remote call calcHash).asLocal.onComplete {
      case Success(hash) =>
        val hashedBooking = booking.withHash(hash)
        //getBookingDb returns a db helper for RoomService's local db
        getBookingDb.insertDocumentObserved("bookings",
          hashedBooking.toBsonSeq,
          new Observer[Completed] {
            override def onError(_): Unit = bookingResponseStream fire Declined(hashedBooking)
            override def onComplete(): Unit = bookingResponseStream fire Confirmed(hashedBooking)
          }
        )
      case Failure(_) => bookingResponseStream fire Declined(booking)
    }
  }
}
```

Listing 3: Implementation of RoomService and HasherService

Listing 3 analogously contains the logic for *RoomService* and for *HasherService*. Like before, the services define their interconnections. `Optional` ties denote that a *RoomService* can run without a *UIGateway* – even if the connection fails, the service keeps running. On the other hand, *RoomService* relies on the functionalities of the *HasherService*. For this reason, the tie is defined as mandatory using `Single`. The *HasherService* shuts down and restarts if the connection attempt fails. Line 6 in Listing 3 shows the implementation of *HasherService*'s `calcHash` function, which simply returns a random UUID to keep our example concise. Lines 12 and following contain the logic of *RoomService*. The `bookingResponseStream` identifier is bound to an event stream. The `book` function, called by *UIGateway* (Listing 2, Line 11), handles the booking process. First, the *RoomService* requests a hash from its connected *HasherService* by invoking its `calcHash` function. After receiving the hash, *RoomService* sets the hash as the booking's confirmation code, and writes the booking object to its connected local database. The database setup is in Listing 4. We omit the boilerplate code for `insertDocumentObserved` – it does what its name suggests. Depending on the outcome of the database write, the `insertDocumentObserved` function fires either a `Confirmed` or a `Declined` response to the `bookingResponseStream`. The response is ultimately received by the *UIGateway* (Listing 2, Line 14). We purposely used two ways of inter-service communication – remote function calls and subscription to remote events – to demonstrate both options.

```
 1  @containerized @multitier object BookingSystem extends UIGatewayImpl with HasherServiceImpl with
        RoomServiceImpl
 2
 3  @gateway("""{ "ports":"80" }""")
 4  object UIGateway extends App {
 5    multitier start new Instance[BookingSystem.UIGateway](
 6      listen[BookingSystem.RoomService] {...}
 7    )
 8  }
 9  @service("""{ "replicas":"3", "localDb":"mongo" }""")
10  object RoomService extends App {
11    multitier start new Instance[BookingSystem.RoomService](
12      connect[BookingSystem.UIGateway] {...} and
13      connect[BookingSystem.HasherService] {...}
14    )
15  }
16  @service
17  object HasherService extends App {
18    multitier start new Instance[BookingSystem.HasherService](
19      listen[BookingSystem.RoomService] {...}
20    )
21  }
```

Listing 4: Starting the Application

Finally, Listing 4 depicts the startup code for the application. In this example, we separate the application logic into different traits, but we only define a single `@containerized` domain module that integrates all service functionalities (Line 1). Hence all services will be deployed together in a single domain (*BookingSystem*).

The deployment process of the application is controlled using `@service` and `@gateway`. Every declared `@service` object is deployed into its own Docker image (corresponding to a container). The container uses the respective object as its main entry point. The object extends `App` and is therefore executable. We create one `@service` object per `@peer` running said peer with `multitier start`. This solution ensures that when the deployed container starts up, it also starts up the respective peer logic defined in Listings 1-3. Thus, `@peer` allows one to define a service and its logic, and `@service` defines how to deploy the service. We specifically denote the *UIGateway* object with `@gateway` instead of `@service`, because it is externally reachable over port 80. For the *RoomService*, we instruct the compiler to generate 3 replicas of this service plus a local mongoDb database in a dedicated container (Line 9) which is used by the `book` method in Listing 3. In the example, all connections are defined at startup, but ScalaLoci also supports connections established or destroyed at runtime.

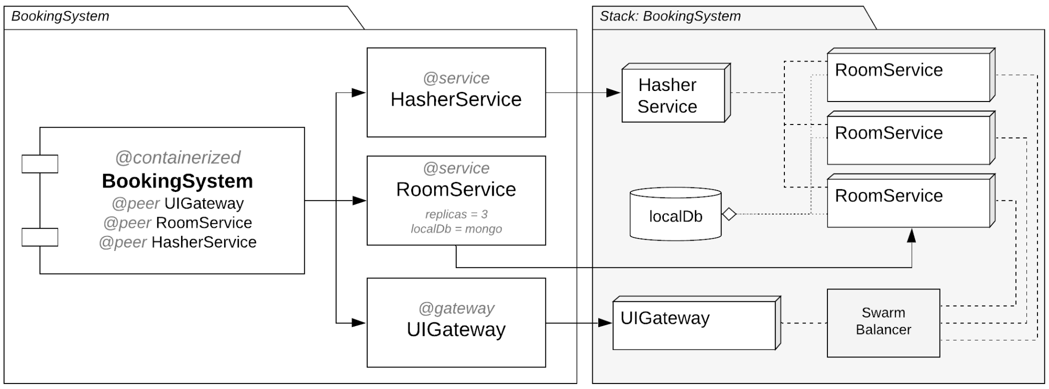

Figure 1: Deployment process for Listings 1-4

Deployment Structure Figure 1 shows the resulting architecture after the compiler automatically generates the communication and the containerization code. The domain of a `@containerized` object is modeled using Docker stacks. The example therefore produces a ready-to-deploy load-balanced Docker Swarm architecture (gray background) that contains one stack with three services. Dotted lines indicate communication flow. Each `@gateway/@service` object is deployed to a corresponding Docker image and it runs as a Swarm service. The affiliation of a service to a stack is determined by the `@containerized` module of the `@peer` it executes. The *RoomService* runs with three container instances and a dedicated local database. A more sophisticated MSA may contain multiple domains and stacks, or may integrate with other non-ScalaLoci microservice domains via custom API gateways using `@gateway`. One can thus use our solution for whole MSAs or just for single application domains achieving static safety (e.g., cross-service type checking), still respecting MSA's polyglot philosophy.

Outlook

The future work of this line of research unfolds along three directions. First, at the language level, we envision more advanced build-in support for loose service coupling through the use of open communication interfaces such as REST, clear separation of interfaces offered and used by single modules, and facilitated integration of heterogeneous external services developed with different technologies. A major focus for future work is modularity to support larger projects. A first step in this direction is the ScalaLoci module system [11].

Second, we plan to add direct support for data management to our multitier solution. The developer can then use dataflow abstractions [4] to express data retrieval and processing and the compiler automatically generates the required deployment [6] based on containers.

Finally, the current solution does also produce low-level output that can principally be used in any orchestration tool, it is mainly optimized for direct use with Docker Swarm. Hence, another direction encompasses extended support for sophisticated integration mechanisms and tools, such as direct support for Kubernetes or KNative, DevOps/CI integration tools, service monitoring, or FaaS/Serverless Computing.

References

[1] Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *14th IEEE International Symposium on Network Computing and Applications*, pages 27–34. 2015.

[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, 2017.

[3] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[4] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[5] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[6] Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. Language-integrated privacy-aware distributed queries. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[7] Umesh Ram Sharma. *Practical Microservices: A Practical Approach to Understanding Microservices.* Packt Publishing, 2017.

[8] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[9] M.V.L.N. Venugopal. Containerized microservices architecture. *International Journal Of Engineering And Computer Science*, 6(11):23199–23208, 2017.

[10] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[11] Pascal Weisenburger and Guido Salvaneschi. Multitier modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP)*, volume 134, pages 1–29, 2019.

[12] Pascal Weisenburger and Guido Salvaneschi. Implementing a language for distributed systems: Choices and experiences with type level and macro programming in scala. *The Art, Science, and Engineering of Programming*, 4(3), 2020.

[13] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Computing Surveys*, 2020.