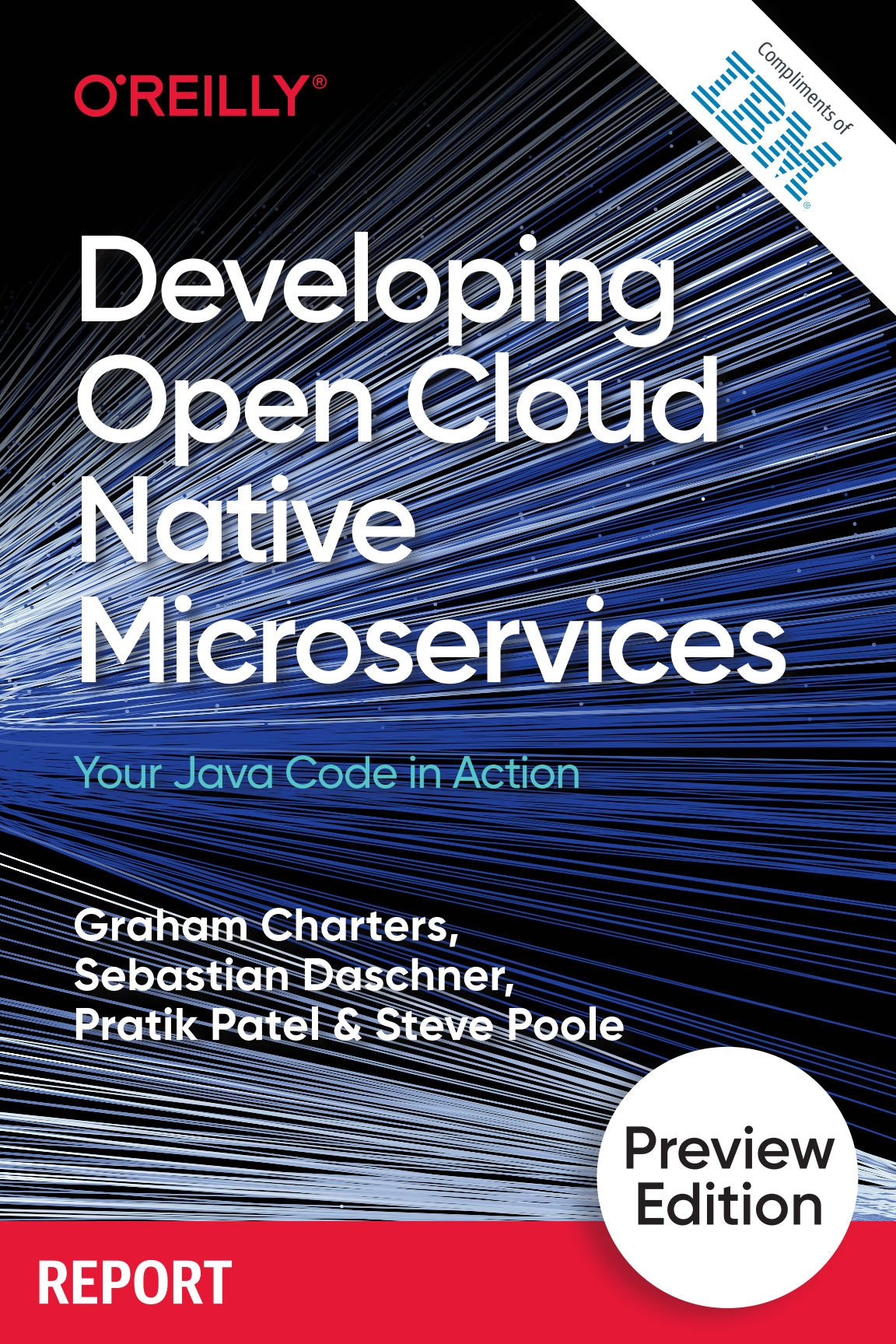
O'REILLY®
Compliments of
Developing Open Cloud Native Microservices
Your Java Code in Action
Graham Charters, Sebastian Daschner, Pratik Patel & Steve Poole
Preview Edition
REPORT

Developing Open Cloud Native Microservices

Your Java Code in Action

This Preview Edition of *Developing Open Cloud Native Microservices*, Chapter 3, is a work in progress. The final book is currently scheduled for release in August 2019 and will be available at oreilly.com once it is published.

Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole

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

Developing Open Cloud Native Microservices
by Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole

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*.

Development Editor: Michele Cronin
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2019: First Edition

Revision History for the First Edition
2019-07-15: First Release

978-1-492-05272-2

[LSI]

Table of Contents

Preface

Prerequisites for Reading This Book

This book is primarily aimed at readers with some knowledge of the Java programming language who wish to understand how to get started with creating cloud native Java applications. Readers wihout an understanding of Java can still benefit from the book, as many of the principles will hold regardless of programming language or framework.

Why This Book Exists

This book exists to help the Java developer begin their journey into cloud native. There is much to learn on this voyage, and this book is intended to provide initial guidence to important high level concepts and start the reader along a well-trodden and proven technical direction.

What You Will Learn

By the end of this book you will understand the unique challenges that arise when creating, running and supporting cloud-native microservice applications. This book will help you decide what else you need to learn when embarking on the journey to the cloud and how modern techniques will help with deployment of new applications in general.

The book will briefly explain the important characteristics to consider when designing an application for the Cloud and cover the key principles for microservices of distribution, data consistency, con-

tinuous delivery etc. that not only are important for a cloud application but which will support the operational and deployment needs of modern 24x7, highly available Java based applications in general.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Developing Open Cloud Native Microservices* by Graham Charters, Sebastian Daschner, Pratik Patel, and Steve Poole (O'Reilly). Copyright 2019 IBM Corporation, 978-1-492-05272-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Foundation

In this chapter, we're going to lay the foundation for developing cloud native microservice applications. First of all, we will focus on how to implement the business logic in plain Java with as little coupling to framework-specific APIs as possible. We will then look at the edges of the applications that communicate with other applications and databases. We'll see how to persist our business objects and how to implement HTTP-based services.

When it comes to implementing the communication boundaries, there are two main approaches to developing cloud native microservices: contract-first and implementation-first. With contract-first, the service API (the contract) is defined, for example using Swagger or OpenAPI, and this is used to generate the service implementation skeleton. With implementation-first, the service is implemented and then the contract generated (e.g., using runtime or tools-based OpenAPI generation). Both are valid approaches, but we mainly see developers using implementation-first, which is the approach taken in this chapter.

Rapidly Developing Service Implementations

Let's dive into the implementation of our applications. One of the most important aspects enterprise developers should focus on is implementing the business logic. Not on cross-cutting concerns, integration, observability, or anything else for now, but only what adds value to the application and its users. That is, at the core of our microservices, we at first start from a plain Java view, and only

model and implement components that have a direct relation to our business use case.

The convenience of Enterprise Java is that the programming model puts little weight on top of our individual classes and methods, thanks to the declarative approaches of both Jakarta EE and MicroProfile. Typically, the classes of our core domain logic are simple Java classes, enhanced with a few annotations, and that's it.

This is why in this book we start with only covering Java and CDI (Context and Dependency Injection), then gradually add more specifications as our application requires some more cross-cutting concerns. However, with this plain approach you can already achieve a lot.

Implementing Domain Classes Using CDI

In our coffee shop application, one of the entry points for our uses case, sometimes also referred to as *boundary*, is a class called `CoffeeShop`. This class implements functionality to order a cup of coffee, or retrieve the status of previous orders:

```
public class CoffeeShop {

    @Inject
    Orders orders;

    @Inject
    Barista barista;

    public List<CoffeeOrder> getOrders() {
        return orders.retrieveAll();
    }

    public CoffeeOrder getOrder(UUID id) {
        return orders.retrieve(id);
    }

    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        OrderStatus status = barista.brewCoffee(order);
        order.setStatus(status);

        orders.store(order.getId(), order);
        return order;
    }

    public void processUnfinishedOrders() {
        // ...
```

```
        }
    }
```

The `CoffeeShop` class exposes the use cases for ordering a new coffee, retrieving a list of all orders, or a single one, and for processing unfinished orders. It defines two dependencies, `Orders` and `Barista`, to which it delegates the further execution.

As you can see, the only Enterprise Java-specific declarations are the injections of our dependencies via `@Inject`. Dependency injection, as well as inversion of control in general, is one of the most useful patterns for developing our applications. We developers are not required to instantiate and wire dependent components, including all their transitive dependencies, which means we can focus on efficiently writing the business domain logic. We define the dependencies in a way of "we need to use this component in our class" without regarding the instantiation. The life cycle of our instances, or beans, is managed by CDI.

You can mix and match injecting CDI beans of different scopes. The injection framework makes sure that all combinations work as desired.

The `CoffeeOrder` which represents the entities of our domain, is written using plain Java only, for now. It's a POJO (plain old Java object) containing properties for the order ID, type, and status:

```
public class CoffeeOrder {

    private final UUID id = UUID.randomUUID();
    private CoffeeType type;
    private OrderStatus status;

    // getters & setters ...

}
```

The `CoffeeType` and `OrderStatus` types are Java enums which define the available types of drinks (`ESPRESSO`, `LATTE`, `POUR_OVER`) and order statuses (`PREPARING`, `FINISHED`, `COLLECTED`), respectively.

The components that implement our business logic are also the ones that should be tested well. Writing test cases is beyond the scope of this book. However, with the approach of plain-Java first we can effi-

ciently develop test cases that cover the majority of our business logic using test frameworks such as JUnit and mocking framworks such as EasyMock or Mockito.

Scopes

Besides dependency injection, CDI also enables us to define the scope of beans. The scope of the bean determines its lifecycle, for example, when it will be created and when it will be destroyed. Instances of the shown `CoffeeShop` class are created with an implicit *dependent* scope, that is the scope is dependent on the scope of whoever uses them. If, for example, a *request*-scoped HTTP endpoint is injected with our `CoffeeShop` bean, the `CoffeeShop` instance lifecyle will also exist within the same *request* scope.

If we have the need to define a different scope, for example for a class that exists only once in our application, we annotate our class accordingly. The following example shows the *application*-scoped `Orders` class:

```
@ApplicationScoped
public class Orders {

    private final ConcurrentHashMap<UUID, CoffeeOrder> orders =
            new ConcurrentHashMap<>();

    public List<CoffeeOrder> retrieveAll() {
        return orders.entrySet().stream()
                .map(Map.Entry::getValue)
                .collect(Collectors.toList());
    }

    public CoffeeOrder retrieve(UUID id) {
        return orders.get(id);
    }

    public void store(UUID id, CoffeeOrder order) {
        orders.put(id, order);
    }

    public List<CoffeeOrder> getUnfinishedOrders() {
        return orders.values().stream()
                .filter(o -> o.getStatus() != OrderStatus.COLLECTED)
                .collect(Collectors.toList());
    }
}
```

The `Orders` class is responsible for storing and retrieving coffee orders, including their status. The `@ApplicationScoped` annotation declares that there is to be one instances of the `Orders` bean. No matter how many injection points we have in our application—`CoffeeShop` is one of them—they will always be injected with the same instance.

Be aware of the default concurrency management of EJB singletons if you're using them instead of *application*-scoped CDI beans. CDI beans don't manage concurrency, and it's the developers' responsibility to make them thread-safe.

The most commonly used scopes that are available in CDI are *dependent*, *request*, *application*, and *session*. If these capabilities are, for some reason, not enough, developers can write their own scopes and extend the features of CDI. In a typical enterprise application, however, this is seldom required.

Configuration

A typical application contains a few concerns that we might need to configure, such as how to look up and access external systems, how to connect to databases, or which credentials to use. The good news is that in a cloud native world, we can externalize a lot of different kinds of configuration from the application-level to the environment. That is, we don't have to configure and change the application binaries, but instead, can have the different configuration values injected from the environment, for example containers or container orchestration, such as from Kubernetes ConfigMaps.

As a developer, we want to focus on configuration that relates to the application business logic. Depending on your business, it might sometimes be required that applications behave differently in different environments.

In general, we want to be able to inject configuration values with minimal developer effort. We just covered dependency injection earlier, and ideally, we'd like to have a similar way of injecting configured values into our code.

With CDI we could write CDI producers that lookup our configured values and make them available. However, there's an even easier method, if we use MicroProfile Config.

MicroProfile Config

MicroProfile Config defines functionality that allows developers to easily inject configured values, in a very similar way that we're used to from CDI. For example, it ships with default config sources for environment variables that we can use right away. Available environment values in our systems are loaded and ready to be injected without any further developer effort.

Let's assume that we want to enhance our coffee-shop example to define default coffee drinks, if the clients don't explicitly mention, which type of coffee they'd like to have. That is, we set some default `CoffeeType` in our `CoffeeOrders` if the provided type is empty.

Have a look at our updated `CoffeeShop` class:

```
public class CoffeeShop {

    @Inject
    @ConfigProperty(name = "coffeeShop.order.defaultCoffeeType",
            defaultValue = "ESPRESSO")
    private CoffeeType defaultCoffeeType;

    // ...

    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        setDefaultType(order);
        OrderStatus status = barista.brewCoffee(order);
        order.setStatus(status);

        orders.store(order.getId(), order);
        return order;
    }

    private void setDefaultType(CoffeeOrder order) {
        if (order.getType() == null)
            order.setType(defaultCoffeeType);
    }

    // ...
}
```

The `CoffeeType` is resolved from the environment variable `coffeeShop.order.defaultCoffeeType`. Some operating systems don't support dots in their variable names, which is why MicroProfile

Config supports multiple ways of replacing the dots with underscores. We could thus define this very variable as `COFFEE_SHOP_ORDER_DEFAULTCOFFEETYPE`, also. If that environment variable is not set in the running application, the value will default to `ESPRESSO`.

The `CoffeeType` enum defines multiple values that can be resolved by the string representations and so we can choose the `ESPRESSO` string representation as the default.

Persisting Service Data

We just saw how to implement the main business logic components into our applications. Besides stateless processing logic, most applications require to persist state, usually the domain entities that represent the core of our business.

There are many database technologies to choose from. In this chapter we want to focus on relational database management systems (RDBMS), which arguably offer a straightforward way of persisting domain objects that, from experience, cover the majority of use cases.

Java Persistence API

In the Enterprise Java world, the Java Persistence API (JPA) is one of the most widely used technologies to persist domain entities. It offers an effective, declarative way to map types and their properties to relational database tables. JPA integrates well with models that are built following the concepts of domain-driven design. Persisting entities doesn't introduce much code overhead and doesn't overly constraint the modeling. This enables us to construct the domain model first, focusing on the business aspects, and integrating the persistence concerns afterwards.

JPA's main concepts are the entity beans, which represent the individual persisted domain entities, and the entity manager, which is responsible for storing and retrieving entity beans.

Mapping Domain Models

JPA enables us to directly map our domain entities as well as aggregates to the database. In order to do that we define domain types such as the `CoffeeOrder` as JPA entity beans:

```
@Entity
@Table(name = "orders")
public class CoffeeOrder {

    @Id
    private String id;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    @Column(name = "coffee_type")
    private CoffeeType type;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    private OrderStatus orderStatus;

    // getters & setters

}
```

The `@Entity` annotation defines the `CoffeeOrder` as an entity bean. Each entity bean is required to define the notation of identity, that is, contain an ID property, annotated with `@Id`. The individual fields are usually mapped to database columns which are also configured declaratively, using the corresponding annotations. Full examples can be found in the JPA documentation.

The `CoffeeOrder` example will persist our coffee order to the `orders` table, with the enumerations persisted as string representations.

Managing Persisted Entities

The `EntityManager` is the main entry point which manages the persistence of our entities. Our business logic invokes its functionality during the processing of a coffee order use case:

```
@ApplicationScoped
public class CoffeeShop {

    @PersistenceContext
    EntityManager entityManager;

    // ...
```

```
    @Transactional
    public CoffeeOrder orderCoffee(CoffeeOrder order) {
        order.setId(UUID.randomUUID().toString());
        setDefaultType(order);
        OrderStatus status = barista.brewCoffee(order);
        order.setOrderStatus(status);

        return entityManager.merge(order);
    }

}
```

The `merge` operation make the coffee order a managed entity, meaning JPA will manage its storing to, and retrieval from, the database.

The `@Transactional` annotation states that the `orderCoffee` is to be executed within a transaction. The default for the annotation is to require a transaction so if there isn't one already, a new one will be started. If a new transaction was started, once this method returns the container will automatically commit the transaction and the coffee order will be persisted to the database. The merge operation make the coffee order to cause the coffee order to be persisted.

If we define aggregate entities that not only contain primitive types or value objects but references to other entities, the persist operations are invoked on the root entities and cascaded to the referenced sub-entities. The difference between entities and value objects, such as strings, enumerations, or currency values, is the notation of identity. It doesn't make a difference to (most) businesses which instance of a 10 EUR bill we refer to, but it does make a difference which coffee order has just been completed successfully. The latter needs to be identified individually and thus represents a domain entity.

The `EntityManager` implements the DAO (Data Access Object) pattern. Depending on the complexity of the invocations made on the `EntityManager` type it is often not necessary to encapsulate them in a separate DAO-like type.

Integrating RDBMS Systems

Our basic example shows the persistence configuration that is required on the project code level. In order to integrate the database into our application, we need to define the datasource—that is, how to connect to the database.

Ideally, we can abstract the detailed configuration from our application configuration. As we saw earlier, environment-specific configuration should not be part of the application code but be managed by the infrastructure.

JPA manages the persistence of entities within persistence contexts. The entity manager of a persistence context acts as a cache for the entities and uses a single persistence unit which corresponds to a database instance. If only one database instance is used within the application, the entity manager can be obtained directly, as shown in the example, without specifying the persistence unit.

Persistence units are specified in the `persistence.xml` which resides under the `META-INF` directory of our project:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="coffee-orders" transaction-type="JTA">
    </persistence-unit>
</persistence>
```

This example configures a persistence unit that doesn't specify a data source, thus uses the default data source. Our application container is required to define a default datasource which is used here. With this approach we can decouple the infrastructure configuration from our application binary build. This means we only need to build the application once and can change the configuration as we take it through test, staging and into production—a best practice in the world of cloud native microservices.

If we use multiple datasources in our application, we define multiple persistence units and refer the datasources via JDNI:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="coffee-orders" transaction-type="JTA">
        <jta-data-source>jdbc/CoffeeOrdersDB</jta-data-source>
    </persistence-unit>
    <persistence-unit name="customers" transaction-type="JTA">
        <jta-data-source>jdbc/CustomersDB</jta-data-source>
    </persistence-unit>
</persistence>
```

In this case, we are required to qualify the `EntityManager` lookups with the corresponding persistence unit:

```
@PersistenceContext(unitName = "coffee-orders")
EntityManager entityManager;
```

Transactions

As introduced before, Java Enterprise makes it easy to execute business logic within transactions. This is required, once we make use of relational databases, and also once we want to ensure that our data is stored in an all-or-nothing fashion. In one way or another, the majority of enterprise applications require ACID (Atomic, Consistent, Isolated, Durable) transactions.

However, in a distributed system, a business use case might involve multiple external systems, databases, or backend services. Traditionally, these *distributed transactions*, have relied on the use of a two-phase commit (2PC) protocol to coordinate updates across the external systems. To achieve this consistency across distributed systems takes time and resources, and thus it comes at the cost of availability. However, in modern internet-scale systems, availability is often key and so other techniques based around the goal of *eventual consistency* are employed. This has resulted in the emergence of patterns such as Sagas and CQRS (Command Query Responsibility Separation). In moving to an eventual consistency model, a system becomes more loosely-coupled and responsive with the caveat that the data may be a little stale. For a more detailed understanding of these principals, we recommend you take a look at the literature on CAP Theorem.

However, local transactions, that only span a single service and a single database are typically required in order to guarantee data consistency. As we've seen in the example, the `@Transactional` annotation enables this functionality without requiring developers to write boilerplate code or extensive configuration. If required, we can further refine the behavior of how multiple, nested methods are executed. For example, methods that are executed within an active transaction can suspend the transaction and start a new transaction which is active during the execution of the method, or be part of an existing transaction. For further information, have a closer look at the semantics of the parameters of the `@Transactional` annotation and the JTA (Java Transaction API) specification.

In our `CoffeeShop` example, we used a CDI bean. An alternative approach, although less in favor nowadays, would have been to use a session EJB (Enterprise Java Beans). With session EJBs, by-default, all the EJB methods are transactional and so there would be no need for the `@Transactional` annotation.

Implementing REST Services

After seeing how our domain entities can be persisted, let's have a look at how to integrate other applications using HTTP-based communication. We're going to show how to implement HTTP-based services using JAX-RS, how to map entities to transfer objects, and how applications can benefit from the ideas behind Hypermedia

Once we have implemented our business logic and potentially implemented persistence, we have to further integrate our application into our system. Business logic that is not accessible from the outside mostly provides little value, and typically enterprise systems are required to provide endpoints to communicate with other systems.

The following shows how to implement REST services using Java Enterprise technology. We'll implement JAX-RS resource classes that handle the HTTP functionality and make our business logic accessible to outside of the application.

Boundary Classes

JAX-RS resource classes typically represent the boundaries, the entrypoints of our business use cases. Clients make HTTP requests and thus start a specific business process in the backend.

The following shows a JAX-RS resources class that implements the HTTP handling for retrieving coffee orders:

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;
```

```
    @GET
    public List<CoffeeOrder> getOrders() {
        return coffeeShop.getOrders();
    }
}
```

The `@Path` annotation declares the class as a JAX-RS resource for handling the `/orders` URL path. There are annotations for all standard HTTP verbs, such as `@GET`, `@POST`, `@HEAD`, etc. These annotations declare a method as a JAX-RS resource method. Once a client makes an HTTP `GET` request to the `/orders` resource, the request will be handled in the `getOrders` method of this class.

JAX-RS automatically maps Java objects to HTTP requests and responses, respectively. It supports content negotiation, that is clients can instruct the servers which content type they use for request bodies, and which one they expect for the server's responses. This example resource class supports JSON, which is specified by the `@Consumes` and `@Produces` annotations. Developers can declare further content type capabilities by implementing and registering `MessageBodyWriter` and `MessageBodyReader` types in JAX-RS, but for most microservices, JSON is the format used. We may pay attention not to confuse and use the wrong import for the `@Produces` annotation, since CDI also defines on with the same name but different behavior.

In order to create some coffee orders, clients typically create new resources by sending `POST` requests to the backend's URLs. The following shows the JAX-RS resource method for creating new orders:

```
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;

    @Context
    UriInfo uriInfo;

    @POST
    public Response orderCoffee(CoffeeOrder order) {
        final CoffeeOrder storedOrder = coffeeShop.orderCoffee(order);
        return Response.created(buildUri(storedOrder)).build();
    }
```

```
        private URI buildUri(CoffeeOrder order) {
            return uriInfo.getRequestUriBuilder()
                    .path(OrdersResource.class)
                    .path(OrdersResource.class, "getOrder")
                    .build(order.getId());
        }
    }
```

The `orderCoffee` resource method will recieve the `POST` request with the coffee order as body in the JSON content type, and map it to an `CoffeeOrder` object. The resource method calls the business functionality of the boundary and returns a `Response` object, a wrapper object for HTTP responses with additional information, to indicate that the resources has been created successfully. The JAX-RS implementation will map this to the `201 Created` HTTP status code and the `Location` header field. We'll see later in the *Validating resources* section how other response codes can be returned when the input data is invalid.

This is all we need to do to implement HTTP endpoints on our side.

Mapping Entitites to JSON

By default, all properties of a business entity for which we define getter and setter methods are mapped to JSON. However, often we require some further control, how exactly the properties of an object are serialized, especially when the mapping slightly varies from what we represent in Java.

In Enterprise Java, there are two ways to map Java objects from and to JSON.

The first way is to declaratively map the properties, using a standard called JSON-B (Java API for JSON Binding). This is what we implicitly used in the previous examples. By default, this approach will map all properties for which an object defines getter and setter methods to JSON object fields. Nested object types are handled recursively.

The second way is to programmatically create or read JSON objects using a technology called JSON-P (Java API for JSON Processing). JSON-P defines methods that we can call directly to create arbitrary objects. This approach provides the biggest flexibility in how objects

are mapped. Let's look at an example how to programmatically map our coffee order type:

```
@Path("/orders")
public class OrdersResource {

    @Inject
    CoffeeShop coffeeShop;

    @Context
    UriInfo uriInfo;

    @GET
    public JsonArray getOrders() {
        return coffeeShop.getOrders().stream()
                .map(this::buildOrder)
                .collect(JsonCollectors.toJsonArray());
    }

    private JsonObject buildOrder(CoffeeOrder order) {
        return Json.createObjectBuilder()
                .add("type", order.getType().name())
                .add("status", order.getStatus().name())
                .add("_self", buildUri(order).toString())
                .build();
    }

    private URI buildUri(CoffeeOrder order) {
        return uriInfo.getRequestUriBuilder()
                .path(OrdersResource.class)
                .path(OrdersResource.class, "getOrder")
                .build(order.getId());
    }
}
```

The `JsonArray` type defines a JSON array of arbitrary elements. The `getOrders` resource method maps the individual coffee orders to `JsonObject` and aggregates them into the array type. We can see how the object builder allows us to compose objects with any desired structure. The produced JSON objects differ slightly from the declaratively mapped approach.

The question often arises as to when to use JSON-B or JSON-P. Typically, we find that the declarative approach of JSON-B is the simplest and covers most use cases.

If the default JSON-B serialization is not what you need then it also allows you to control how the Java objects are mapped to JSON, especially how and whether individual properties are being mapped.

The following example will slightly modify the mapping of our coffee order:

```
public class CoffeeOrder {

    @JsonbTransient
    private final UUID id = UUID.randomUUID();

    @JsonbTypeAdapter(CoffeeTypeDeserializer.class)
    private CoffeeType type;

    @JsonbProperty("status")
    private OrderStatus orderStatus;

    // methods omitted
}
```

The `@JsonbTransient` annotation declares the `id` field as transient, that is, this field will be ignored by JSON-B and neither be read from nor written to JSON. `@JsonbProperty` allows for further customizing of the JSON object field name.

If the default mapping of a Java type doesn't work for us, we can always declare a custom type adapter, for example, using the `@JsonbTypeAdapter` annotation as shown above. You can have a look at the adapter for the coffee type enum in the further resources of this book.

These and a few other ways of customizing the JSON mapping built into JSON-B already fulfil a majority of cases. If more flexibility or control is required, we can switch to programmatically create or read JSON structures using JSON-P. This approach is especially helpful when dealing with Hypermedia REST resources, as we will see later in this chapter.

Validating Resources

Request data that is received from clients needs to be sanitized before it can be used further. For security reasons you should never trust the data that has come from and external source such as a Web form or REST request. In order to make it simple to validate input, Java Enterprise ships with the Bean Validation API that allows us to declaratively configure the desired validation. The good news for developers is that this standard integrates seemlessly with the rest of the platform, for example with JAX-RS resources.

To ensure that only valid coffee orders are accepted in our application, we enhance our JAX-RS resource method with Bean Validation constraints:

```
...

@POST
public Response orderCoffee(@Valid @NotNull CoffeeOrder order) {
    final CoffeeOrder storedOrder = coffeeShop.orderCoffee(order);
    return Response.created(buildUri(storedOrder)).build();
}
```

The `@NotNull` annotation ensures that we'll recieve properly populated orders. `@Valid` makes sure that the order itself is validated for potential subsequent validation constraints.

Let's have a look at our enhanced coffee order class, that defines what makes a valid order:

```
public class CoffeeOrder {

    @JsonbTransient
    private final UUID id = UUID.randomUUID();

    @NotNull
    @JsonbTypeAdapter(CoffeeTypeDeserializer.class)
    private CoffeeType type;

    private OrderStatus status;

    // ... getters & setters
}
```

The type of a coffee order must not be null either, that is, clients must provide a valid enumeration value. The value is automatically mapped by the provided JSON-B type adapter and would provide a `null` value in case an invalid value has been transmitted. Consequently, validation will fail for any invalid values.

JAX-RS integrates with Bean Validation in the way that means if any constraint validations fail, an HTTP status code of `400 Bad Request` is automatically returned. Therefore, the presented example is already sufficient to ensure that only valid orders can be sent to our application.

REST and Hypermedia

Representational State Transfer (REST) provides an architectural style of web services that often well-suits the needs and structure of web applications. The idea is to loosely-couple applications with interfaces that are accessible in a uniform way, located by URLs. There are a few REST constraints such as the use of uniform interfaces, identification of individual resources, that is the entities in our domain, or the use of Hypermedia as the engine of application state, commonly referred to as HATEOAS. The representations of the domain entities are modified in a uniform way, in HTTP for example using the HTTP methods such as GET, POST, DELETE, PATCH, or PUT.

Hypermedia adds linking related resources and resource actions, that are accessible by URLs. If some HTTP resources are somewhat related to others, they can link to these, providing full URLs that the client directly follow. In this way, the server guides the clients through the available resources using semantic link relations. The clients don't require knowledge of how the URLs are structured on the server. Its usage was originally described in the work by Roy T. Fielding and his dissertation, Architectural Styles and the Design of Network-based Software Architectures.

The following shows a basic example of a coffee order representation in the JSON format:

```
{
  "type": "ESPRESSO",
  "status": "PREPARING",
  "_links": {
    "self": "https://api.coffee.example.com/orders/123",
    "customer": "https://api.coffee.example.com/customers/234"
  }
}
```

The `_links` field of the representation gives some example links, to the coffee order resource itself and to the customer who created this order. If a client would like to know more about the customer, it would follow the provided URL in a subsequent GET request.

It's usually not sufficient to only follow links and read resources using GET, therefore it's possible as well to exchange information on how to modify resources, for example using POST or PUT request.

The following demonstrates an example Hypermedia response that uses the concept of actions. There are a few Hypermedia-aware content types that support these approaches, such as the Siren content type on which this example is based:

```
{
  "class": [ "coffee-order" ],
  "properties": {
    "type": "ESPRESSO",
    "status": "PREPARING"
  },
  "actions": [
    {
      "name": "cancel-order",
      "method": "POST",
      "href": "https://api.coffee.example.com/cancellations",
      "type": "application/json",
      "fields": [
        { "name": "reason", "type": "text" },
        { "name": "order", "type": "number", "value": 123 }
      ]
    }
  ],
  "links": [
    "self": "https://api.coffee.example.com/orders/123",
    "customer": "https://api.coffee.example.com/customers/234"
  ]
}
```

In this example, the server provides the client the ability to cancel a coffee order and describes its usage in the `cancel-order` action. A new cancellation means the client would `POST` a JSON representation of a cancellation containing the order number and a reason to the provided URL. In this way, the client only requires knowledge of the action itself, represented by the name `cancel-order`, and where the provided information originate from, for example the order number, which is provided, and the reason, which is only known by the client and, for example, may be entered in a text field in the UI.

This is one example of a content type that enables the use of Hypermedia controls. There is no real standard format that the industry has agreed upon. However, this Siren-based example nicely demonstrates the concepts of links and actions. Whatever content type and representation structure is being used, the projects need to agree and document on their usage. But as you can see, this way of structuring the web services requires far less documentation since the usage of the API is baked into the resource representations already.

This also greatly reduces the likelihood of API documentation becoming out of date with the code. We'll see later in Chapter 4 of this book how OpenAPI can further improve this.

One of the benefits of using HATEOAS is that the control of how the resources are accessed resides on the server side. The server owns the communication and is even free to change the URL structures, since the clients are no longer required to make any assumptions on how the URLs are being constructed.

The decoupling the communication also results in less duplication of business logic. Clients do not need to contain the logic for the conditions under which an order can be cancelled, they can simply display the functionality of cancelling orders if the corresponding action is provided in the HATEOAS response. Only the knowledge that is actually required to reside on the client side, for example how the reason of cancelling an order is being provided, needs to be implemented on the client side (i.e. UI decisions).

Let's come back to how we map our domain entities to JSON on the server side. As you can see, the JSON structures of Hypermedia resources can become quite complex, may result in complex Java type hierarchies if we were to us the declarative approach of JSON-B. For this reason, it's worth considering the programmatic approach using JSON-P to create these Hypermedia resources. The creation of coffee order representation, for example, can be factored out into a separate class with that single responsibility to remove redundancy in the JAX-RS resources, if required.

Conclusion

As you saw, we can already implement the vast majority of our enterprise application using plain Java and CDI. At its core, our business logic is written in plain Java with some dependency injection added to simplify defining dependent components. MicroProfile Config enables us to inject required configuration with minimal impact in the code. Besides that, what's left is mainly integration into our overall enterprise system, as well as non-functional requirements such as resiliency and observability.

We saw how to integrate persistence into our applications using JPA, and how to map domain entities to relational databases with minimal developer effort. Thanks to the previous specification work

being done in the JTA standard, we can define transactional behavior without obscuring the business code.

We can implement REST endpoints using the JAX-RS standard with JAX-RS resources. The declarative programming model allows us to efficiently define the endpoints with default HTTP bindings, and also to further customize the HTTP request and response mappings, if required.

Enterprise Java supports binding our entities to and from JSON, either declaratively using JSON-B, or programmatically using JSON-P. It depends on the complexity of the entity representations which approach makes more sense. The requests can be validated using Bean Validation which allows developers to specify the validation programmatically or declaratively, as well. Enterprise developers might want to have a look into the concepts behind Hypermedia, that allow further decoupling from the server, make the server resources discoverable, and the communication more flexible and adaptive to change.

About the Authors

Graham Charters is an IBM Senior Technical Staff Member and WebSphere Applications Server Developer Advocacy Lead based at IBM's R&D Laboratory in Hursley, UK. He takes a keen interest in emerging technologies and practices and in particular programming models. His past exploits include establishing and contributing to open source projects at PHP and Apache and participation in, and leading, industry standards at OASIS and the OSGi Alliance.

Sebastian Daschner is a Lead Java Developer Advocate for IBM. His role is to share knowledge and educate developers about Java, enterprise software, and IT in general. He enjoys speaking on conferences, writing articles and blog posts, producing videos, newsletters, and other content.

Pratik Patel is a Lead Developer Advocate at IBM. He wrote the first book on 'enterprise Java' in 1996, *Java Database Programming with JDBC*. He has also spoken at various conferences and participates in several local tech groups and startup groups. He hacks Java, iOS, Android, HTML5, CSS3, JavaScript, Clojure, Rails, and, well, everything except Perl.

Steve Poole is a long time Java developer, leader, and evangelist. He is a DevOps practitioner (whatever that means). He has been working on IBM Java SDKs and JVMs since Java was less than 1. A seasoned speaker and regular presenter at international conferences on technical and software engineering topics.