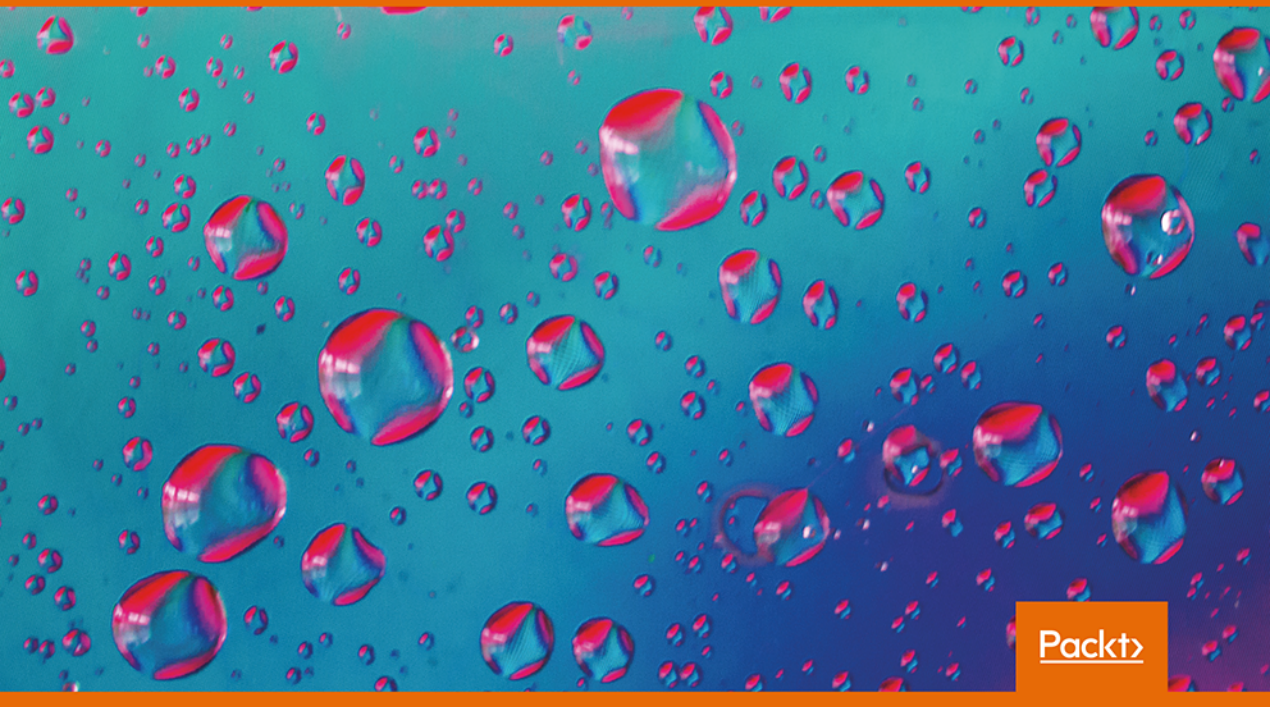
Helion

Java EE 8

Wzorce projektowe i najlepsze praktyki



Packt›

Rhuan Rocha, João Purificação

Tytuł oryginału: Java EE 8 Design Patterns and Best Practices

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-5503-3

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jee8wp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 6

Wzorce reaktywne

W tym rozdziale przyjrzymy się wzorcom reaktywnym i ich implementacjom. Napiszemy, jak z ich pomocą tworzyć lepsze aplikacje. Zajmiemy się też wyjaśnieniem, jak pewne ogólne koncepcje programowania reaktywnego wspomagają sam proces programowania. Po przeczytaniu rozdziału będziesz mógł stosować wzorce reaktywne w zgodzie z najlepszymi praktykami Javy EE 8.

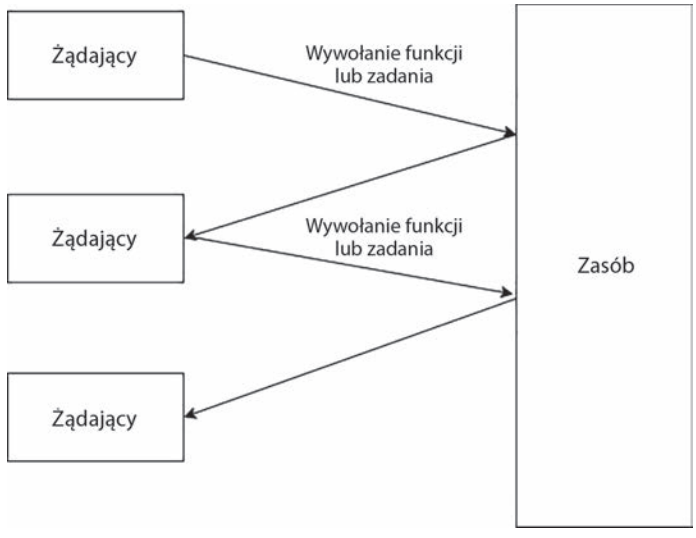
W tym rozdziale poruszone zostaną następujące tematy:

- pojęcie programowania reaktywnego;
- zdarzenia w mechanizmie CDI;
- implementacja zdarzenia CDI;
- asynchroniczne metody EJB;
- implementacja asynchronicznej metody EJB;
- asynchroniczne usługi REST;
- implementacje asynchronicznych usług REST.

Bardzo długo wiele aplikacji przetwarzało wszystkie żądania w sposób synchroniczny. W takim systemie użytkownik żąda zasobu i czeka na odpowiedź. Każdy element procesu jest wykonywany jeden po drugim.

Diagram na następnej stronie przedstawia działanie procesu, gdy ten jest wykonywany w sposób synchroniczny.

W procesie synchronicznym każde wywołanie funkcji lub zasobu odbywa się w sposób sekwencyjny krok po kroku. Gdy pewne zadania wykonują się dłużej, blokują na ten czas cały proces. Przykładem zadania, które może potrwać dłużej, jest chociażby komunikacja wejście-wyjście związana z odczytem danych z dysku lub źródła danych.



Wraz z rozwojem aplikacji internetowych pojawiła się konieczność jednoczesnego przyjmowania i obsługi dużej liczby żądań. W takiej konfiguracji przetwarzanie synchroniczne zaczęło ujawniać swoje słabości, bo nie radziło sobie z wystarczająco szybkim przygotowywaniem odpowiedzi przy wielu żądaniach. Aplikacje internetowe zaczęły więc coraz intensywniej korzystać z procesów asynchronicznych, co w wielu przypadkach pozwalało na przyspieszenie zwracania odpowiedzi.

W procesie asynchronicznym wywołania funkcji lub zasobów mogą odbywać się w sposób zrównoleglony bez potrzeby oczekiwania na zakończenie jednego zadania przed rozpoczęciem następnego. W takiej sytuacji zlecenie długotrwałego zadania wejścia-wyjścia nie opóźni rozpoczęcia prac nad następnym — oba zadania mogą się wykonywać równolegle.

Programowanie reaktywne to styl programowania wykorzystujący elementy funkcyjne. Obsługuje asynchronicznie strumienie danych wysyłane przez żądających. Zasoby nasłuchują strumienia i reagują na pojawiające się w nim dane. Poszczególne zadania działają jak funkcje i nie obsługują zmiennych poza swoim zakresem, co pozwala funkcjom na zrównoleglone działanie bez żadnej synchronizacji lub ryzyka efektów ubocznych.

W programowaniu reaktywnym mamy do czynienia z elementami, które reagują na zdarzenia, więc jeśli użytkownik żąda zasobu, zgłasza zdarzenie do odpowiedniego strumienia. Po pojawieniu się zdarzenia w strumieniu danych zadanie reaguje, uruchamiając swój algorytm przetwarzania. W ten sposób aplikacja internetowa może z łatwością w tym samym czasie obsługiwać wiele żądań bez problemów ze skalowalnością. Całość opiera się na czterech filarach. Oto one.

- **Elastyczność** — reagowanie na popyt. Aplikacja z łatwością korzysta z procesorów wielordzeniowych i wielu serwerów do przetwarzania żądań.
- **Odporność** — reagowanie na porażki. Aplikacja z łatwością reaguje i powraca do pełnej sprawności po błędach programowych, sprzętowych lub sieciowych.

- **Sterowanie komunikatami** — reagowanie na zdarzenia. Aplikacja składa się z działających asynchronicznie i nieblokująco menedżerów zdarzeń zamiast z wielu synchronicznie działających wątków.
- **Responsywność** — reagowanie na użytkowników. Aplikacja oferuje bogate interakcje przy bardzo niskim czasie oczekiwania.

Java EE 8 oferuje odpowiednie narzędzia pozwalające programiście na zastosowanie programowania reaktywnego w tworzonych aplikacjach. Narzędziami tymi są: zdarzenia z CDI, asynchroniczne wywoływanie metod komponentów EJB i asynchroniczne usługi REST.

Zdarzenia w CDI

Gdy w środowisku programistów rosła popularność programowania reaktywnego, język Java i środowisko Javy EE musiało zaoferować narzędzia pozwalające na podążanie za nowym trendem. W Javie EE skupiono się przede wszystkim na ułatwieniu tworzenia programów w stylu funkcyjnym i ułatwieniu obsługi asynchroniczności. Efektem tych prac są między innymi zdarzenia dostępne w CDI — kod może uruchomić zdarzenie w sposób synchroniczny i blokujący lub asynchroniczny i nieblokujący.

W CDI zdarzenie (obiekt `Event`) to mechanizm, który korzysta z wzorca Obserwator, co pozwala na przekazanie zdarzenia w celu obróbki do dowolnych innych komponentów działających w sposób asynchroniczny (nieblokujący) lub synchroniczny (blokujący). W tym rozdziale skupimy się na wersji asynchronicznej zdarzeń, bo ona stanowi część paradygmatu programowania reaktywnego.

Implementacja zdarzenia w CDI

Jako przykład implementacji zdarzenia w CDI wyobraźmy sobie asynchroniczny CDI i aplikację pozwalającą na przesłanie trzech typów plików (obsługiwanych na podstawie rozszerzenia): ZIP, JPG i PDF. W zależności od otrzymanego w żądaniu rozszerzenia uruchomi się jedno zdarzenie, a obserwator zapisze plik na dysku w asynchronicznym procesie. Każde rozszerzenie będzie posiadało własnego obserwatora i własny algorytm zapisu pliku. W tym przykładzie użyjemy następujących klas i interfejsów:

- `FileUploadResource` — klasa reprezentuje zasób, który otrzymuje wszystkie żądania związane z przesyłaniem plików i uruchamia odpowiednie zdarzenie na podstawie rozszerzenia pliku;
- `FileEvent` — komponent bean z danymi pliku przekazywany do zdarzenia;
- `FileHandler` — interfejs dla wszystkich obserwatorów, czyli wszystkie klasy reagujące na `FileEvent` będą musiały implementować `FileHandler`;
- `JpgHandler` — implementacja `FileHandler`, która zapisuje na dysku plik JPG; klasa jest obserwatorem reagującym na `FileEvent` zawierający plik JPG;

- `PdfHandler` — implementacja `FileHandler`, która zapisuje na dysku plik PDF; klasa jest obserwatorem reagującym na `FileEvent` zawierający plik JPG;
- `ZipHandler` — implementacja `FileHandler`, która zapisuje na dysku plik ZIP; klasa jest obserwatorem reagującym na `FileEvent` zawierający plik JPG;
- `Jpg` — to kwalifikator wskazujący, że obserwatorzy `JpgHandler` powinni reagować na to zdarzenie;
- `Pdf` — to kwalifikator wskazujący, że obserwatorzy `PdfHandler` powinni reagować na to zdarzenie;
- `Zip` — to kwalifikator wskazujący, że obserwatorzy `ZipHandler` powinni reagować na to zdarzenie;
- `FileSystemUtils` — klasa narzędziowa związana z obsługą systemu plików.

Implementacja klasy FileUploadResource

Klasa `FileUploadResource` jest klasą wykorzystującą JAX-RS do wykonania usługi typu REST pozwalającej na przesłanie plików o rozszerzeniach JPG, PDF i ZIP. Poniżej przedstawimy zarówno kod tej klasy, jak i kod kwalifikatorów służących do wyboru odpowiedniego obserwatora, który ma zareagować na zdarzenie.

Komponent bean wysyłany w zdarzeniu

Klasa `FileEvent` to komponent bean przesyłany jako część zdarzenia. Obiekt tej klasy otrzymają obserwatorzy.

```
import java.io.File;

public class FileBean {

   private File file;

   private String mimeType;

   public FileBean(){}

   public FileBean(File file, String mimeType){

      this.file = file;
      this.mimeType = mimeType;

   }

   public File getFile() {
      return file;
   }

   public void setFile(File file) {
      this.file = file;
   }
```

```
    public String getMimeType() {
        return mimeType;
    }

    public void setMimeType(String mimeType) {
        this.mimeType = mimeType;
    }
}
```

Kwalifikator wybierający obserwatora JpgHandler jako obsługującego zdarzenie

Poniższy kod definiuje kwalifikator `Jpg` pozwalający w przyszłości wskazać, kto ma obsłużyć zdarzenie.

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER,
    ElementType.FIELD})
public @interface Jpg {
}
```

Kwalifikator wybierający obserwatora PdfHandler jako obsługującego zdarzenie

Poniższy kod definiuje kwalifikator `Pdf` pozwalający w przyszłości wskazać, kto ma obsłużyć zdarzenie.

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER,
    ElementType.FIELD})
public @interface Pdf {
}
```

Kwalifikator wybierający obserwatora ZipHandler jako obsługującego zdarzenie

Poniższy kod definiuje kwalifikator `Zip` pozwalający w przyszłości wskazać, kto ma obsłużyć zdarzenie.

```
import javax.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE, ElementType.PARAMETER,
   ElementType.FIELD})
public @interface Zip {
}
```

Klasa FileUploadResource

Poniższy kod przedstawia klasę `FileUploadResource` będącą usługą REST wykorzystującą JAX-RS.

```
import javax.enterprise.event.Event;
import javax.enterprise.util.AnnotationLiteral;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import java.io.File;
import java.util.Objects;

@Path("upload")
public class FileUploadResource {

   @Inject
   Event<FileEvent> fileEvent;

   @Consumes("application/pdf")
   @POST
   public Response uploadPdf(File file) {

      FileEvent fileEvent = new FileEvent(file, "pdf");

      Event<FileEvent> pdfEvent = this.fileEvent.select(
         new AnnotationLiteral<Pdf>() {});

      pdfEvent.fireAsync(fileEvent)
         .whenCompleteAsync((event, err) -> {

            if (Objects.isNull(err))
               System.out.println("PDF zapisany");
            else
               err.printStackTrace();
         });

      return Response.ok().build();
   }

   @Consumes("image/jpeg")
   @POST
```

```
    public Response uploadJpg(File file) {

        FileEvent fileEvent = new FileEvent(file, "jpg");

        Event<FileEvent> jpgEvent = this.fileEvent.select(
            new AnnotationLiteral<Jpg>() {});

        jpgEvent.fireAsync(fileEvent)
            .whenCompleteAsync((event, err) -> {

                if (Objects.isNull(err))
                    System.out.println("JPG zapisany");
                else
                    err.printStackTrace();
            });

        return Response.ok().build();
    }

    @Consumes("application/zip")
    @POST
    public Response uploadZip(File file) {

        FileEvent fileEvent = new FileEvent( file, "zip" );

        Event<FileEvent> zipEvent = this.fileEvent.select(
            new AnnotationLiteral<Zip>() {});

        zipEvent.fireAsync(fileEvent)
            .whenCompleteAsync((event, err) -> {

                if (Objects.isNull(err))
                    System.out.println("ZIP zapisany");
                else
                    err.printStackTrace();

            });
        return Response.ok().build();
    }
}
```

Powyższy kod zawiera trzy metody — `uploadPdf(File file)`, `uploadJpg(File file)` i `uploadZip(File file)` — wywoływane odpowiednio, gdy użytkownik przesyła plik o rozszerzeniu PDF, JPG lub ZIP. Klasa posiada właściwość `fileEvent` typu `Event<FileEvent>`. To klasa odpowiedzialna za uruchomienie zdarzenia zgodnie z kwalifikatorem. Oto fragment, który odpowiada za wybór odpowiedniego obiektu `Event` na podstawie kwalifikatora:

```
Event<FileEvent> zipEvent = this.fileEvent.select(new AnnotationLiteral<Zip>() {});
```

Alternatywnym sposobem określenia odpowiedniego zdarzenia do uruchomienia jest wskazanie kwalifikatora w momencie wstrzykiwania obiektu za pomocą adnotacji `@Inject`, ale w ten sposób zdarzenie zawsze dotyczyłoby jednego typu. Użycie metody `select(Annotation... var)` pozwala na dynamiczne decydowanie o rodzaju zdarzenia. Oto przykład zapisu zdarzenia w sposób statyczny:

```
@Inject
@Pdf // Kwalifikator.
Event<FileEvent> pdfEvent;
```

Gdyby użyć powyższego kodu, `pdfEvent` zawsze powodowałoby utworzenie zdarzenia, na które reagowaliby tylko obserwatorzy oznaczeni kwalifikatorem `@Pdf`.

Aby wywołać zdarzenie asynchroniczne, musimy użyć metody `fireAsync(U var)` zwracającej wartość typu `CompletionStage`. Oto fragment kodu wywołujący metodę oraz przygotowujący funkcję wywołania zwrotnego informującą o zakończeniu obsługi zdarzenia.

```
zipEvent.fireAsync(fileEvent)
   .whenCompleteAsync((event, err) -> {

      if (Objects.isNull(err))
         System.out.println("ZIP zapisany");
      else
         err.printStackTrace();
   });
```

Implementacja obserwatorów

W momencie uruchomienia zdarzenia pewne elementy zareagują na nie i uruchomią zadania związane z przetworzeniem danych zawartych w zdarzeniu. Elementy te nazywamy **obserwatorami**. Korzystają one z obserwatora tworzącego między obiektami związek typu jeden-do-wielu. Do takiej sytuacji dochodzi, gdy jeden obiekt jest źródłem informacji, a pozostałe są jej obserwatorami. Gdy źródło wyśle zdarzenie, otrzymają je wszyscy obserwatorzy, którzy wcześniej zgłosili chęć jego otrzymania.

CDI udostępnia mechanizm tworzenia obserwatorów reagujących na zgłaszane zdarzenia. W przedstawionym przykładzie dla każdego rodzaju zdarzenia musimy utworzyć obserwatora odpowiadającego za jego obsługę. Wszystkie klasy obserwatorów będą implementowały ten sam interfejs — `FileHandler` — zawierający tylko jedną metodę — `handle(FileEvent file)`. Zwróć uwagę, że typem parametru metody jest `FileEvent`, czyli dokładnie ten sam typ, którego używamy do zgłoszenia zdarzenia. Oto implementacja interfejsu `FileHandler`:

```
import java.io.IOException;

public interface FileHandler {
   public void handle(FileEvent file) throws IOException;
}
```

Poniższy kod przedstawia klasę `JpgHandler` będącą implementacją interfejsu `FileHandler`. Klasa odpowiada za zapis pliku JPG w systemie. Metoda klasy zostanie wywołana, gdy dojdzie do zgłoszenia zdarzenia oznaczonego kwalifikatorem `@Jpg`.

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;
```

```
public class JpgHandler implements FileHandler {

   @Override
   public void handle(@ObservesAsync @Jpg FileEvent file) throws IOException {

      FileSystemUtils.save(file.getFile(), "jpg", "jpg_" +
         new Date().getTime() + ".jpg");
   }
}
```

Metoda klasy zawiera w parametrze zarówno adnotację `@ObservesAsync`, jak i `@Jpg`. To zapis stosowany przez CDI, który wskazuje, że metoda obserwuje zdarzenia `FileEvent`, ale powinna być uruchamiana tylko wtedy, gdy zdarzenia te są oznaczone kwalifikatorem `@Jpg`.

Poniższy kod przedstawia klasę `PdfHandler` również implementującą `FileHandler`. Tym razem klasa odpowiada za zapis pliku PDF. Metoda klasy zostaje wywołana, gdy dochodzi do zgłoszenia zdarzenia oznaczonego kwalifikatorem `@Pdf`.

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;

public class PdfHandler implements FileHandler {

   @Override
   public void handle(@ObservesAsync @Pdf FileEvent file) throws IOException {

      FileSystemUtils.save(file.getFile(), "pdf", "pdf_" +
         new Date().getTime() + ".pdf");
   }
}
```

Metoda klasy zawiera w parametrze zarówno adnotację `@ObservesAsync`, jak i `@Pdf`. W ten sposób informuje, że metoda obserwuje zdarzenia `FileEvent`, ale powinna być uruchamiana tylko dla zdarzeń oznaczonych kwalifikatorem `@Pdf`.

Poniższy kod przedstawia klasę `ZipHandler` również implementującą `FileHandler`. Tym razem klasa odpowiada za zapis pliku ZIP. Metoda klasy zostaje wywołana, gdy dochodzi do zgłoszenia zdarzenia oznaczonego kwalifikatorem `@Zip`.

```
import javax.enterprise.event.ObservesAsync;
import java.io.IOException;
import java.util.Date;

public class ZipHandler implements FileHandler {

   @Override
   public void handle(@ObservesAsync @Zip FileEvent file) throws IOException {

      FileSystemUtils.save(file.getFile(), "zip", "zip_" +
         new Date().getTime() + ".zip");
   }
}
```

Metoda klasy zawiera w parametrze zarówno adnotację `@ObservesAsync`, jak i `@Zip`. W ten sposób informuje, że metoda obserwuje zdarzenia `FileEvent`, ale powinna być uruchamiana tylko dla zdarzeń oznaczonych kwalifikatorem `@Zip`.

Asynchroniczne metody komponentów EJB

Tworzenie zdarzeń i przekazywanie ich do komponentów, które na nie reagują, to dobry mechanizm do rozwiązywania wielu różnych rodzajów problemów. Czasem jednak chcemy po prostu wywołać metodę pewnej klasy bez konieczności blokowania procesu aż do momentu jej zakończenia.

Asynchroniczna metoda EJB to mechanizm, który pozwala klientowi wywołać metodę i od razu uzyskać z powrotem sterowanie (bez oczekiwania). Wartość, która zostanie w przyszłości uzyskana jako wynik metody, znajdzie się w zwróconym obiekcie `Future<T>`. Co więcej, klient może często nawet sterować wykonaniem metody, na przykład może ją anulować. Obiekt poza wynikiem oferuje właściwości pozwalające na sprawdzenie zakończenia prac, zgłoszenia wyjątku lub anulowania przetwarzania.

Różnice między zdarzeniami i asynchronicznym wywołaniem metody

Zarówno zdarzenia CDI, jak i asynchroniczne wywołania metod EJB mają podobną charakterystykę, czyli oferują nieblokujące wykonanie zadania. W przypadku wywołania asynchronicznego klient ma możliwość anulowania procesu i monitorowania postępów. Główną różnicą między dwoma rodzajami wykonywania zadań jest to, że metoda asynchroniczna działa na zasadzie jeden-do-jednego między wywołującym i wywoływanym. Wywołujący dokładnie wie, co zostanie wywołane i że zadanie będzie dotyczyć tylko prac związanych z tą metodą. W przypadku zdarzeń istnieje związek typu jeden-do-wielu, bo wywołujący tylko zgłasza zdarzenie, a całą resztą zajmują się obiekty nasłuchujące. W przypadku zdarzeń wywołujący nie jest w stanie dowiedzieć się w łatwy sposób o wyniku działań, a samo wykonanie zadań z racji zastosowania wzorca Obserwator może się opóźnić. Metoda asynchroniczna EJB nie stosuje obserwatora, zwraca wynik i najczęściej jest uruchamiana od razu.

Implementacja asynchronicznej metody EJB

W przykładzie zastosujemy ten sam scenariusz, co we wcześniejszym przykładzie dotyczącym zdarzeń CDI. Aplikacja będzie umożliwiała przesłanie plików trzech typów (rozszerzeń): ZIP, PDF i JPG. W zależności od typu będzie obsługiwała i zapisywała pliki w różnych folderach. W trakcie realizacji przykładu wykonamy następujące klasy:

- `FileUploadResource` — klasa reprezentuje zasób, który otrzymuje wszystkie żądania związane z przesyłaniem plików i wywołuje odpowiedni komponent EJB na podstawie rozszerzenia pliku;
- `JpgHandler` — komponent EJB, który zapisuje na dysku plik JPG; klasa zawiera metodę asynchroniczną;
- `PdfHandler` — komponent EJB, który zapisuje na dysku plik PDF; klasa zawiera metodę asynchroniczną;
- `ZipHandler` — komponent EJB, który zapisuje na dysku plik ZIP; klasa zawiera metodę asynchroniczną;
- `FileSystemUtils` — klasa narzędziowa związana z obsługą systemu plików.

Implementacja komponentów EJB

Aby skorzystać z asynchronicznych metod EJB, musimy utworzyć bean sesyjny i skonfigurować go tak, aby stosował metody asynchroniczne. Poniższy kod przedstawia implementację klasy `PdfHandler`, czyli bean sesyjny odpowiedzialny za zapis plików typu PDF.

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class PdfHandler {

   @Asynchronous
   public Future<String> handler(FileBean file) throws IOException {

      return new AsyncResult(
         FileSystemUtils.save(
            file.getFile(),
            "pdf",
            "pdf_" + new Date().getTime() + ".pdf"));

    }
}
```

Powyższy kod zawiera w klasie `PdfHandler` metodę `handler(FileBean file)`. Użycie adnotacji `@Asynchronous` zamienia zwykłą metodę w metodę asynchroniczną.

Oto ogólna sygnatura metody `handle(FileBean file)`.

```
@Asynchronous
public Future<String> handler(FileBean file) throws IOException {
   // Logika biznesowa.
}
```

Należy zwrócić uwagę, że metoda zwraca typ `Future<T>`. W tej konkretnej implementacji używamy obiektu klasy `AsyncResult`, która implementuje interfejs `Future`. W przykładzie jako wynik wykonania metody zwracamy ścieżkę do zapisanego pliku. Oto fragment realizujący opisane zadanie.

```
return new AsyncResult(
   FileSystemUtils.save(
      file.getFile(),
      "pdf",
      "pdf_" + new Date().getTime() + ".pdf"));
```

Teraz zdefiniujemy klasę o nazwie `JpgHandler`, która odpowiadać będzie za zapis w systemie pliku typu JPG.

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class JpgHandler {

   @Asynchronous
   public Future<String> handler(FileBean file) throws IOException {

      return new AsyncResult(
         FileSystemUtils.save(
            file.getFile(),
            "jpg",
            "jpg_" + new Date().getTime() + ".jpg"));

    }
}
```

Metoda w zasadzie jest taka sama jak metoda `PdfHandler`, ale zapisuje plik w innym folderze, stosując inny szablon nazw. Także i tym razem obiekt `Future` będzie po wykonaniu zadania zawierał ścieżkę do zapisanego pliku.

Ostatnia z klas obsługi dotyczy zapisywania plików typu ZIP i nosi nazwę `ZipHandler`.

```
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;
import java.util.concurrent.Future;

@Stateless
public class ZipHandler {

   @Asynchronous
   public Future<String> handler(FileBean file) throws IOException {
```

```
            return new AsyncResult(
               FileSystemUtils.save(
                  file.getFile(),
                  "zip",
                  "zip_" + new Date().getTime() + ".zip"));

        }
    }
```

Także i ta klasa działa jak dwie poprzednie, choć stosuje nieco inny wzorzec plików i zapisuje je w innym folderze. Podobnie jak wcześniej, obiekt `Future` będzie po wykonaniu zadania zawierał ścieżkę do zapisanego pliku.

Implementacja klasy FileUploadResource

Klasa `FileUploadResource` to klasa zasobu wykorzystująca JAX-RS do otrzymania usługi typu REST obsługującej przesyłanie plików o rozszerzeniach JPG, PDF lub ZIP. Oto kod klasy.

```
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import java.io.File;
import java.io.IOException;

@Path("upload")
public class FileUploadResource {

   @Inject
   private PdfHandler pdfHandler;

   @Inject
   private JpgHandler jpgHandler;

   @Inject
   private ZipHandler zipHandler;

   @Consumes("application/pdf")
   @POST
   public Response uploadPdf(File file) throws IOException {

      FileBean fileBean = new FileBean(file, "pdf");

      pdfHandler.handler(fileBean);

      return Response.ok().build();
   }

   @Consumes("image/jpeg")
   @POST
   public Response uploadJpg(File file) throws IOException {
```

```
        FileBean fileBean = new FileBean(file, "jpg");

        jpgHandler.handler(fileBean);

        return Response.ok().build();
    }

    @Consumes("application/zip")
    @POST
    public Response uploadZip(File file) throws IOException {

        FileBean fileBean = new FileBean(file, "zip");

        zipHandler.handler(fileBean);

        return Response.ok().build();
    }
}
```

W powyższym kodzie klasy poniższy fragment odpowiadał za wstrzyknięcie klas `PdfHandler`, `JpgHandler` i `ZipHandler` za pomocą adnotacji `@Inject`.

```
@Inject
private PdfHandler pdfHandler;

@Inject
private JpgHandler jpgHandler;

@Inject
private ZipHandler zipHandler;
```

Gdy do aplikacji trafi żądanie, metoda odpowiedzialna za jego przetworzenie pobiera plik i buduje obiekt typu `FileBean`. Następnie metoda ta wywołuje asynchroniczną metodę EJB, aby zapisać plik. Trzy kolejne bloki kodu przedstawiają operacje wywoływania metody asynchronicznej.

Wywołanie asynchronicznej metody EJB w celu zapisu pliku PDF

Poniższy fragment kodu ilustruje zapis pliku PDF za pomocą wywołania asynchronicznej metody EJB:

```
FileBean fileBean = new FileBean(file, "pdf");
pdfHandler.handler(fileBean);
```

Wywołanie asynchronicznej metody EJB w celu zapisu pliku JPG

Poniższy fragment kodu ilustruje zapis pliku PDF za pomocą wywołania asynchronicznej metody EJB:

```
FileBean fileBean = new FileBean(file, "jpg");
jpgHandler.handler(fileBean);
```

Wywołanie asynchronicznej metody EJB w celu zapisu pliku ZIP

Poniższy fragment kodu ilustruje zapis pliku PDF za pomocą wywołania asynchronicznej metody EJB:

```
FileBean fileBean = new FileBean(file, "zip");
zipHandler.handler(fileBean);
```

Asynchroniczna usługa REST

Zdecydowanie rośnie liczba aplikacji oferujących API typu REST, a także liczba aplikacji korzystających z tego rodzaju usług. Aplikacje używające zasobów typu REST również muszą mieć możliwość asynchronicznego (czyli bez blokowania) przetwarzania procesów.

Asynchroniczna usługa REST to asynchroniczny proces ułatwiający obsługę wątków. W przypadku nadejścia żądania do serwera może zostać utworzony nowy wątek, aby obsłużyć nieblokujące zadanie, na przykład operacje na systemie plików. JAX-RS obsługuje asynchroniczne przetwarzanie zarówno w API po stronie serwerowej, jak i po stronie klienckiej, ale asynchroniczna usługa REST dotyczy części serwerowej. Część kliencka konsumuje zasoby usługi, więc ją najczęściej nazywamy asynchronicznym konsumentem REST.

API klienckie może oczywiście korzystać z asynchronicznego wywołania, które zwraca obiekt typu `Future<T>` tuż po zgłoszeniu żądania. Klient może śledzić postępy wykonania i odpowiednio na nie reagować. Strona serwerowa zwraca obiekt typu `CompletionStage<T>`, co pozwala użyć metody wywołania zwrotnego wykonywanej w zależności od etapu. W przykładowej implementacji przedstawionej dalej w tym rozdziale użyjemy wywołania wykorzystujące programowanie reaktywne.

Implementacja asynchronicznej usługi REST

Także przy implementacji asynchronicznej usługi REST użyjemy tego samego przykładowego zadania, co w przypadku zdarzenia CDI i asynchronicznej metody EJB. Przykładowa aplikacja umożliwiać będzie przesłanie trzech typów plików: ZIP, JPG i PDF. W zależności od rozszerzenia plik będzie trafiał do różnych folderów w systemie plików. W przykładzie skorzystamy z następujących klas:

- `FileUploadResource` — klasa reprezentuje zasób współpracujący z procesem asynchronicznym, który otrzymuje wszystkie żądania związane z przesyłaniem plików i wywołuje odpowiedni komponent EJB na podstawie rozszerzenia pliku;
- `JpgHandler` — komponent EJB zawierający metodę, która zapisuje na dysku plik JPG;
- `PdfHandler` — komponent EJB zawierający metodę, która zapisuje na dysku plik PDF;
- `ZipHandler` — komponent EJB zawierający metodę, która zapisuje na dysku plik ZIP;

- `FileSystemUtils` — klasa narzędziowa związana z obsługą systemu plików;
- `FileUploadClient` — klient API zaimplementowany dzięki JAX-RS, który w sposób asynchroniczny wywołuje usługę REST.

Implementacja EJB

Zaimplementujemy trzy komponenty EJB o nazwach `PdfHandler`, `JpgHandler` i `ZipHandler`, których zadaniem będzie obsługa zapisu przesłanego pliku w odpowiedniej lokalizacji.

Pierwsza z klas, `PdfHandler`, odpowiada za zapis plików PDF.

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class PdfHandler {

   public String handler(FileBean file) throws IOException {
      return FileSystemUtils.save(
         file.getFile(),
         "pdf",
         "pdf_" + new Date().getTime() + ".pdf" );
   }
}
```

Druga z klas, `JpgHandler`, odpowiada za zapis plików JPG.

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class JpgHandler {

   public String handler(FileBean file) throws IOException {
      return FileSystemUtils.save(
         file.getFile(),
         "jpg",
         "jpg_" + new Date().getTime() + ".jpg" );
   }
}
```

Ostatnia z klas, `ZipHandler`, odpowiada za zapis plików ZIP.

```
import javax.ejb.Stateless;
import java.io.IOException;
import java.util.Date;

@Stateless
public class ZipHandler {
```

```
    public String handler(FileBean file) throws IOException {
       return FileSystemUtils.save(
          file.getFile(),
          "zip",
          "zip_" + new Date().getTime() + ".zip" );
    }
}
```

Implementacja klasy FileUploadResource

Klasa `FileUploadResource` to zasób zaprojektowany w taki sposób, aby umożliwić klientom przesyłanie plików na serwer z użyciem asynchronicznego żądania typu REST. Klasa korzysta z JAX-RS do utworzenia zasobu typu REST. Kod w zależności od rozszerzenia otrzymanego pliku deleguje zapis do odpowiedniego komponentu EJB. Oto kod klasy `FileUploadResource`.

```
import javax.annotation.Resource;
import javax.enterprise.concurrent.ManagedExecutorService;
import javax.enterprise.context.spi.CreationalContext;
import javax.enterprise.inject.spi.Bean;
import javax.enterprise.inject.spi.BeanManager;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import java.io.File;
import java.io.IOException;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

@Path("upload")
public class FileUploadResource {

   @Resource
   private ManagedExecutorService executor;

   @Consumes("application/pdf")
   @POST
   public CompletionStage<String> uploadPdf(File file) {

      BeanManager beanManager = getBeanManager();

      CompletableFuture<String> completionStage = new CompletableFuture<>();
      executor.execute(() -> {

         FileBean fileBean = new FileBean(file, "pdf");

         Bean<PdfHandler> bean = (Bean)
            beanManager.getBeans(PdfHandler.class).iterator().next();
         CreationalContext cCtx = beanManager.createCreationalContext(bean);
         PdfHandler pdfHandler = (PdfHandler)
            beanManager.getReference(bean, PdfHandler.class, cCtx);
```

```
            try {
               completionStage.complete(pdfHandler.handler( fileBean ));
            } catch (IOException e) {
               e.printStackTrace();
               completionStage.completeExceptionally(e);
            }

         });

         return completionStage;
      }

      private BeanManager getBeanManager(){
         try {
            // Ręczne wyszukiwanie JNDI dotyczące BeanManager CDI (JSR 299).
            return (BeanManager) new
InitialContext().lookup("java:comp/BeanManager");
         } catch (NamingException ex) {
            throw new IllegalStateException(
               "Błąd wyszukania BeanManager w JNDI");
         }
      }

      @Consumes("image/jpeg")
      @POST
      public CompletionStage<String> uploadJpg(File file) throws IOException {

         BeanManager beanManager = getBeanManager();

         CompletableFuture<String> completionStage = new CompletableFuture<>();
         executor.execute(() -> {
            FileBean fileBean = new FileBean(file, "jpg");

            Bean<JpgHandler> bean = (Bean)
               beanManager.getBeans(JpgHandler.class).iterator().next();
            CreationalContext cCtx = beanManager.createCreationalContext(bean);
            JpgHandler jpgHandler = (JpgHandler)
               beanManager.getReference(bean, JpgHandler.class, cCtx);

            try {
               completionStage.complete(jpgHandler.handler( fileBean ));
            } catch (IOException e) {
               e.printStackTrace();
               completionStage.completeExceptionally(e);
            }

         });

         return completionStage;
      }

      @Consumes("application/zip")
      @POST
      public CompletionStage<String> uploadZip(File file) throws IOException {
```

```
        BeanManager beanManager = getBeanManager();

        CompletableFuture<String> completionStage = new CompletableFuture<>();
        executor.execute(() -> {
           FileBean fileBean = new FileBean(file, "zip");

           Bean<ZipHandler> bean = (Bean)
              beanManager.getBeans(ZipHandler.class).iterator().next();
           CreationalContext cCtx = beanManager.createCreationalContext(bean);
           ZipHandler zipHandler = (ZipHandler)
              beanManager.getReference(bean, ZipHandler.class, cCtx);

           try {
              completionStage.complete(zipHandler.handler(fileBean));
           } catch (IOException e) {
              e.printStackTrace();
              completionStage.completeExceptionally(e);
           }
        });

        return completionStage;

     }
  }
```

Przedstawiona powyżej klasa `FileUploadResource` posiada właściwość `executor` z wstrzykniętym obiektem typu `ManagedExecutorService`. Wstrzykiwanie odbywa się za pomocą adnotacji `@Resource` z CDI. Uzyskany obiekt pozwala utworzyć nowy wątek i w nim wykonać niezbędne zadanie. Oto omówiony fragment kodu klasy:

```
@Resource
private ManagedExecutorService executor;
```

Dodatkowo klasa zawiera trzy metody otrzymujące żądanie, przygotowujące obiekt `FileBean` i wysyłające go do procesu EJB odpowiadającego za zapis pliku. Oto jedna z takich metod, `uploadPdf(File file)`, która odpowiada za obsługę przesłanego pliku typu PDF.

```
@Consumes("application/pdf")
@POST
public CompletionStage<String> uploadPdf(File file) {

   BeanManager beanManager = getBeanManager();

   CompletableFuture<String> completionStage = new CompletableFuture<>();
   executor.execute(() -> {

      FileBean fileBean = new FileBean(file, "pdf");
      // Pobierz EJB poprzez CDI.
      Bean<PdfHandler> bean = (Bean)
         beanManager.getBeans(PdfHandler.class).iterator().next();
      CreationalContext cCtx = beanManager.createCreationalContext(bean);
      PdfHandler pdfHandler = (PdfHandler)
         beanManager.getReference(bean, PdfHandler.class, cCtx);
```

```
            try {
               completionStage.complete(pdfHandler.handler(fileBean));
            } catch (IOException e) {
               e.printStackTrace();
               completionStage.completeExceptionally(e);
            }

         });

         return completionStage;
      }
```

Przedstawiony fragment klasy `FileUploadResource` obsługuje przesłane pliki w sposób asynchroniczny. Aby metoda mogła działać w sposób asynchroniczny, możemy albo użyć `AsyncResponse` (`@Suspended final AsyncResponse ar`) jako parametru metody, albo wskazać, że zwraca ona do klienta obiekt typu `CompletionStage<T>`. W przykładzie skorzystaliśmy z tego drugiego rozwiązania. Pozostaje jeszcze tylko utworzyć osobny wątek, aby zadanie realizować w sposób nieblokujący. Oto fragment metody `uploadPdf`, który odpowiada za utworzenie osobnego zadania zapisującego plik PDF.

```
      executor.execute(() -> {

         FileBean fileBean = new FileBean(file, "pdf");
         Bean<PdfHandler> bean = (Bean)
            beanManager.getBeans(PdfHandler.class).iterator().next();
         CreationalContext cCtx = beanManager.createCreationalContext(bean);
         PdfHandler pdfHandler = (PdfHandler)
            beanManager.getReference(bean, PdfHandler.class, cCtx);

         try {
            completionStage.complete(pdfHandler.handler(fileBean));
         } catch (IOException e) {
            e.printStackTrace();
            completionStage.completeExceptionally(e);
         }

      });
```

Implementacja API klienckiego

Aby wysłać asynchroniczne żądanie przy użyciu programowania reaktywnego, użyjemy API klienckiego oferowanego przez JAX-RS. Oto kod klasy realizującej to zadanie.

```
      public class ClientAPI {
         private static final String URL =
            "http://localhost:8080/asyncRestService/resources/upload";
         private static final String FILE_PATH = "test.pdf";

         public static void main(String[] args) {
            Client client = ClientBuilder.newClient();
            WebTarget target = client.target(URL);
            try {
```

```
            CompletionStage<String> csf = target.request()
               .rx()
               .post(Entity.entity(new FileInputStream(
                  new File(FILE_PATH)), "application/pdf"), String.class);

            csf.whenCompleteAsync((path, err) -> {

               if (Objects.isNull(err))
                  System.out.println("Plik zapisano: " + path);
               else
                  err.printStackTrace();
            });

         } catch (FileNotFoundException e) {
            e.printStackTrace();
         }
      }
   }
```

Kod wysyła żądanie do serwera i pozwala na realizację innych prac, bo sterowanie do kodu wywołującego powraca niemalże natychmiast. Dodatkowo kod tworzy funkcję wywołania zwrotnego, która wykona się po zakończeniu asynchronicznego przetwarzania żądania. Oto część klasy `ClientAPI` realizująca przypisanie funkcji wywołania zwrotnego do realizowanego żądania.

```
csf.whenCompleteAsync((path, err) -> {

   if (Objects.isNull(err))
      System.out.println("Plik zapisano: " + path);
   else
      err.printStackTrace();

});
```

Podsumowanie

W tym rozdziale przyjrzeliśmy się paradygmatowi programowania reaktywnego i jego implementacji za pomocą mechanizmów Javy EE 8. Dodatkowo omówiliśmy wykonywanie w Javie EE 8 wywołań asynchronicznych wraz ze sterowaniem nimi i otrzymywaniem wyniku poprzez funkcje wywołań zwrotnych.

Zdarzenia w CDI to mechanizm stanowiący część specyfikacji CDI. Jest dostępny we wszystkich warstwach aplikacji. Okazuje się szczególnie przydatny w trakcie prac nad warstwą prezentacji. Wynika to z faktu, iż CDI skupia się głównie na tej warstwie i ma spore znaczenie w interakcjach i sesjach HTTP. Co ważne, zdarzenia pozwalają przypisać jednej akcji wielu reagujących na nią odbiorców.

Asynchroniczna metoda EJB nie stosuje paradygmatu programowania reaktywnego. Jest zwykłym procesem asynchronicznym pozwalającym na zmniejszenie czasu niezbędnego do odpowiedzenia klientowi. Ponieważ to mechanizm związany z komponentami EJB, zaleca się

stosowanie go w warstwie biznesowej. Zaletą tego rozwiązania jest dostęp do innych mechanizmów EJB, na przykład sterowania transakcjami.

Asynchroniczna usługa REST to mechanizm specyfikacji JAX-RS pozwalający na utworzenie usługi REST realizującej przetwarzanie w trybie asynchronicznym. W tym mechanizmie sterowanie wraca do procesu tuż po wysłaniu żądania, więc klient nie musi bezczynnie czekać na jego zakończenie. Mechanizm ten stosuje się tylko na poziomie warstwy prezentacji.

W następnym rozdziale przyjrzymy się bliżej wzorcom mikrousług i sposobom ich implementacji. Omówimy również kilka powiązanych z mikrousługami wzorców szczegółowych, takich jak wzorzec agregatora, wzorzec przedstawiciela, wzorzec łańcuchowania, wzorzec rozgałęziania i wzorzec asynchronicznego przekazywania komunikatów.

Skorowidz

M

O

P

R

S

Ś

T

U

V

W

Z

Ż

Java EE 8: zestaw świetnych narzędzi dla zawodowca!

Od profesjonalnych systemów informatycznych wymaga się wysokiej dostępności usług, łatwego wprowadzania niezbędnych zmian, skalowalności i możliwości przetworzenia ogromnej ilości danych. Java EE 8 znakomicie nadaje się do tworzenia aplikacji spełniających te wyśrubowane kryteria. Poza tym Java to język wieloplatformowy, jej kod jest otwarty, została dobrze przetestowana, a doświadczenie i wsparcie społeczności użytkowników okazują się nie do przecenienia. Wszechstronność i popularność Javy ma też mroczną stronę — programiści bardzo często muszą rozwiązywać istotne problemy, które zwykle dotyczą integracji usług, wysokiej dostępności i odporności systemu na błędy. Rozwiązaniem pozwalającym uniknąć wielu z tych kłopotów jest zastosowanie odpowiednich wzorców projektowych i przestrzeganie dobrych praktyk.

To książka przeznaczona dla programistów, którzy chcą tworzyć aplikacje biznesowe z użyciem wzorców projektowych, wzorców biznesowych i najlepszych praktyk. Zawiera wyjaśnienie podstawowych koncepcji Javy EE 8, opis jej warstw oraz omówienie najlepszych praktyk tworzenia aplikacji biznesowych. Przedstawia zasady łączenia wzorców projektowych i wzorców biznesowych w Javie EE 8, a także techniki optymalizacji rozwiązań z wykorzystaniem programowania aspektowego, programowania reaktywnego i mikrousług. Opisano tu szereg wzorców: integracyjne, reaktywne, bezpieczeństwa, wdrażania i operacyjne. Zaprezentowano również projekt MicroProfile, przydatny do tworzenia aplikacji dla architektury mikrousług.

W tej książce między innymi:

- korzyści ze stosowania wzorców projektowych
- wzorce warstw prezentacji i biznesowej
- wzorce dla aplikacji działających w chmurze
- implementacja wzorca SSO
- wzorce związane ze skalowalnością, z wydajnością i zarządzaniem aplikacją

Rhuan Rocha —jest doświadczonym programistą Javy EE. Obecnie wdraża rozwiązania Red Hat oparte na Red Hat Middleware. Wcześniej korzystał z Javy przy pisaniu oprogramowania dla firm i rządów.
João Purificação — jest inżynierem elektroniki. Dobrze zna języki C i C++, obecnie wykorzystuje Javę EE do projektowania aplikacji. Jest starszym architektem w firmie Resource IT z siedzibą w São Paulo.

ISBN 978-83-283-5503-3
9 788328 355033
Cena: 57,00 zł

Packt>