

CETEJ35 - Java Web - JAVA_XXX (2024_01)

[Meus cursos](#) / [CETEJ35 - Web \(2024_01\)](#) / [Semana 07: 14/10 a 20/10](#) / [API Reativa](#)

API Reativa

✓ **Feito:** Ver

A fazer: Gastar pelo menos 20 minutos na atividade

A fazer: Passar pela atividade até o fim

Fecha: segunda-feira, 2 dez. 2024, 00:00

Nesta aula vamos construir uma nova aplicação que funciona como uma API, recebendo dados usando a arquitetura REST. Para fazer isso, vamos usar uma nova tecnologia do Spring - o Spring WebFlux.

INTRODUÇÃO À REATIVIDADE

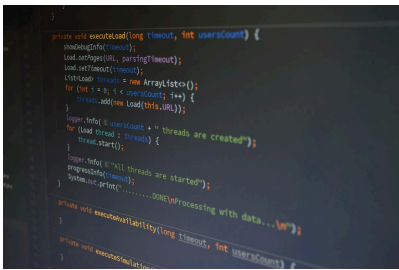
O código abaixo mostra um método em uma classe anotada com `@Controller`. Esse método é executado quando a URL `/excluir` é acessada (linha 59). Quando o método recebe a solicitação, o processador escala uma *thread* em um processo para tratar essa solicitação. Por padrão, as solicitações são *síncronas*. Isso significa que a *thread* bloqueia a continuidade do método até que a operação solicitada seja completada. A linha 64 exemplifica isso muito bem.

```
59     @GetMapping("/excluir")
60     public String excluir(
61         @RequestParam String nome,
62         @RequestParam String estado) {
63
64         var cidadeEstadoEncontrada = repository.findByNomeAndEstado(nome, estado);
65
66         cidadeEstadoEncontrada.ifPresent(repository::delete);
67
68         return "redirect:/";
69     }
```

Na linha 64, o código verifica a existência de dados em um banco de dados. Isso envolve: (i) recuperar o objeto responsável pela verificação; (ii) acessar o banco de dados; (iii) realizar a consulta; (iv) mapear os dados recebidos em uma entidade; e (v) retornar o resultado para o método. Enquanto essa verificação é feita, o método fica bloqueado.

Na prática, isso significa que o usuário que fez a solicitação deve aguardar o processo ser completado para só então receber o retorno. O processamento nesse exemplo pode ser rápido, mas esse nem sempre é o caso. Além disso, nesse exemplo, o usuário não depende do retorno, pois ele solicitou uma exclusão. Nesse caso, o usuário só precisa que os dados sejam excluídos. Ainda assim, ele precisa ficar esperando o processo terminar.

Esse é um exemplo ideal para o uso de uma operação *assíncrona*. Em uma operação assíncrona, a solicitação de verificação feita na linha 64 seria enviada ao banco de dados e, imediatamente, o restante do processo no método `excluir()` seria liberado, retornando uma resposta imediata ao usuário.



Note que isso não significa que a linha 64 seria executada mais rapidamente, mas sim, que o processador iria cuidar da execução dessa tarefa em algum momento. Isso torna seu aplicativo mais *responsivo*. Isso é similar ao conceito de **Promise** em JavaScript, para aqueles familiarizados com a linguagem JS.

Esse não é um recurso novo em Java, mas gerenciar concorrência costumava ser mais complicado antes da adição de [CompletableFuture](#), no Java 8. Em Java, o termo *reatividade* está relacionado com a reação a eventos. Esses eventos acontecem em nível de processamento, em vez de aplicações, como estamos acostumados a ver em aplicações distribuídas.

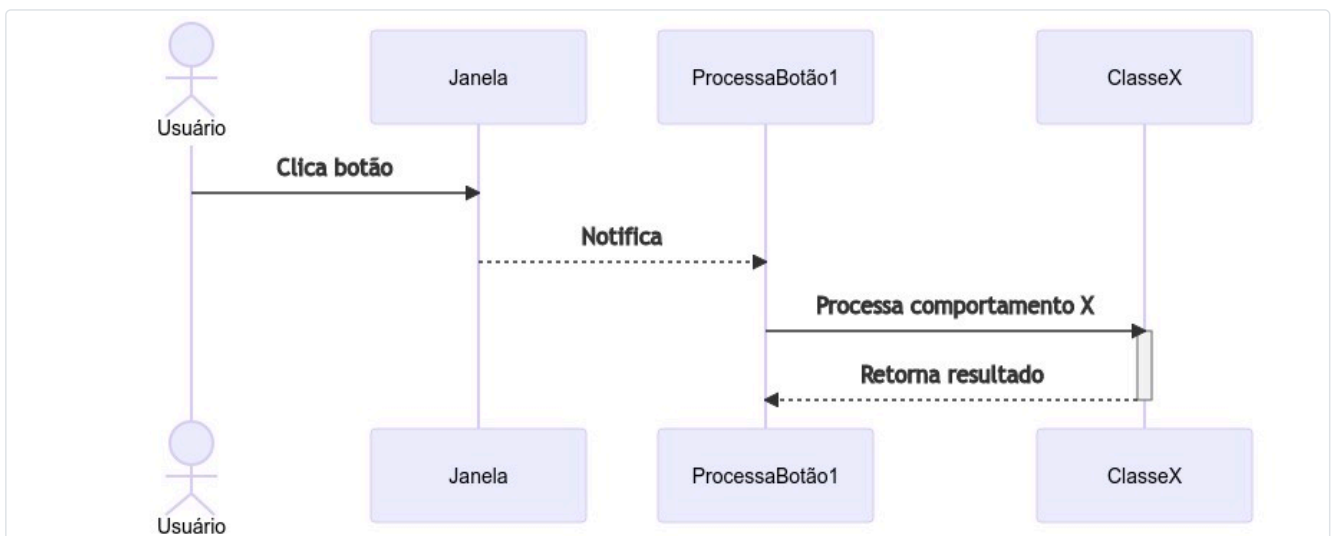
Até agora, temos usado o [Spring MVC](#) para desenvolver aplicações. O Spring MVC é uma das tecnologias fundamentais do [Spring Framework](#). O Spring MVC permite desenvolver aplicações Web baseadas na [API Servlet](#). A API Servlet é uma especificação de como tratar solicitações em um container Web. Usar a API Servlet tal como é implementada por containers como [Tomcat](#) ou servidores de aplicação como [Glassfish](#) não costumava [ser uma tarefa trivial](#).

O Spring MVC facilita muito a vida do desenvolvedor fornecendo mecanismos para implementar aplicações Web de forma rápida e simples, como temos visto até agora nesse curso. Contudo, o Spring MVC não implementa reatividade. Para isso, precisamos de outro projeto - o [Spring WebFlux](#).

O Spring WebFlux usa uma estrutura muito similar ao Spring MVC para o desenvolvimento de aplicações Web. A similaridade facilita a adoção do Spring WebFlux. Contudo, ele introduz diferenças significativas em como o processamento de uma solicitação é realizada. Essas diferenças são necessárias para permitir um processamento *não bloqueante*. A principal diferença está no uso do [padrão publisher-subscriber](#) (editor-assinante).





O padrão *publisher-subscriber* é muito conhecido em computação, e pode ser melhor introduzido por meio do [padrão de projeto observer](#). Um exemplo clássico do uso desse padrão é uma janela de uma aplicação qualquer (Figura abaixo). Toda vez que um botão é clicado, ele dispara um evento (*publisher*). Esse evento é publicado para os assinantes (*subscribers*). Outra forma de representar a mesma coisa é dizer que *a janela notifica os responsáveis pelo comportamento do botão*. Esses assinantes são classes responsáveis por processar determinados comportamentos do botão. Observe que a notificação é feita de forma assíncrona. Assim, a janela não fica bloqueada enquanto o comportamento do botão é processado.



[◀ Verificação de aprendizado - Integração](#)

Seguir para...

[Atividade II WebConf ▶](#)



 [Contate o suporte do site](#) 

Você acessou como RAFAEL ROCHA DA SILVA PROENCA (Sair)
CETEJ35 - Web (2024_01)

- Tema
- Adaptable
- Boost
- Clássico
- Campus
- Apucarana
- Campo Mourão
- Cornélio Procopio
- Curitiba
- Dois Vizinhos
- Francisco Beltrão
- Guarapuava
- Londrina
- Medianeira
- Pato Branco
- Ponta Grossa
- Reitoria
- Santa Helena
- Toledo
- UTFPR
- Ajuda
- Chat UTFPR
- Calendário Acadêmico
- Biblioteca
- e-Mail
- Nuvem (OwnCloud)
- Produção Acadêmica
- Secretaria Acadêmica
- Sistemas Corporativos
- Sistema Eletrônico de Informação - SEI
- Suporte ao usuário
- Criação de curso
- Comunidade
- Português - Brasil (pt_br)
- Deutsch (de)
- English (en)
- Português - Brasil (pt_br)

Resumo de retenção de dados

Baixar o aplicativo móvel.

 [Dê um feedback sobre este software](#) 

Universidade Tecnológica Federal do Paraná - UTFPR
Suporte ao usuário

