Andrey Araujo Masiero

Arquitetura de software

Senac

Dados Internacionais de Catalogação na Publicação (CIP)
(Simone M. P. Vieira – CRB 8ª/4771)

Masiero, Andrey Araujo
Arquitetura de software / Andrey Araujo Masiero. – São Paulo : Editora Senac São Paulo, 2021. (Série Universitária)
Bibliografia.
e-ISBN 978-65-5536-676-1 (ePub/2020)
e-ISBN 978-65-5536-677-8 (PDF/2020)

1. Engenharia de software : Análise de sistemas 2. Informática : Desenvolvimento de software 3. Arquitetura de software : Desenvolvimento de software I. Título. II. Série.

21-1292t CDD – 001.6425
BISAC COM051230

Índice para catálogo sistemático
1. Análise de sistemas 001.6425

ARQUITETURA DE SOFTWARE

Andrey Araujo Masiero

Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional
Abram Szajman

Diretor do Departamento Regional
Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento
Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial
Luiz Francisco de A. Salgado
Luiz Carlos Dourado
Darcio Sayad Maia
Lucila Mara Sbrana Sciotti
Luís Américo Tousi Botelho

Gerente/Publisher
Luís Américo Tousi Botelho (luis.tbotelho@sp.senac.br)

Coordenação Editorial/Prospecção
Dolores Crisci Manzano (dolores.cmanzano@sp.senac.br)

Administrativo
grupoedsadministrativo@sp.senac.br

Comercial
comercial@editorasenacsp.com.br

Acompanhamento Pedagógico
Mônica Rodrigues dos Santos

Designer Educacional
Hágara Rosa da Cunha Araujo

Revisão Técnica
Gustavo Moreira Calixto

Preparação e Revisão de Texto
Camila Lins

Projeto Gráfico
Alexandre Lemes da Silva
Emília Corrêa Abreu

Capa
Antonio Carlos De Angelis

Editoração Eletrônica
Michel Iuiti Navarro Moreno

Ilustrações
Michel Iuiti Navarro Moreno

Imagens
Adobe Stock Photos

E-pub
Ricardo Diana

Editora Senac São Paulo
Rua 24 de Maio, 208 – 3º andar
Centro – CEP 01041-000 – São Paulo – SP
Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP
Tel. (11) 2187-4450 – Fax (11) 2187-4486
E-mail: editora@sp.senac.br
Home page: http://www.livrariasenac.com.br

Sumário

Capítulo 1
Arquitetura de sistemas de software, 7

1 Arquitetura monolítica, 10
2 Arquitetura em camadas, 11
3 Arquitetura orientada a serviços (SOA – *service-oriented architecture*), 16
Considerações finais, 22
Referências, 22

Capítulo 2
Padrões de projeto orientado a objetos, 25

1 O que é um padrão de projeto orientado a objetos?, 26
2 Tipos de padrões, 29
3 Aplicações de software voltadas à web e a dispositivos móveis, 31
Considerações finais, 32
Referências, 32

Capítulo 3
Padrões de projeto Singleton e MVC, 33

1 Padrão de projeto Singleton, 34
2 Padrão MVC (model-view-controller), 38
Considerações finais, 40
Referências, 40

Capítulo 4
Arquitetura baseada em componentes, 41

1 Visão geral sobre componentes, 42
2 Princípios e uso de componentes, 45
Considerações finais, 48
Referências, 49

Capítulo 5
Desenvolvimento de componentes, 51

1 Elaboração de um componente, 52
2 Aplicação em solução baseada na web, 53
Considerações finais, 57
Referências, 58

Capítulo 6
Integração de software, 59

1 API e o desafio na integração de software, 60
Considerações finais, 72
Referências, 72

Capítulo 7
Aspectos de segurança na arquitetura de software, 73

1 Princípios da segurança da informação, 74
2 Aplicação dos princípios de segurança na arquitetura de software, 77
Considerações finais, 80
Referências, 80

Capítulo 8
Avaliação de arquiteturas, 81

1 Métodos de avaliação de arquiteturas, 82

Considerações finais, 90

Referências, 90

Sobre o autor, 93

Capítulo 1

Arquitetura de sistemas de software

Antes de começar efetivamente o projeto de um software, é necessário definir qual será a estrutura e a organização do projeto. O resultado desse passo é chamado de arquitetura de software. Esse é o primeiro estágio de qualquer projeto que deseja obter sucesso (SOMMERVILLE, 2011). Nas metodologias tradicionais de desenvolvimento de software, como no caso do Cascata, antes de qualquer coisa, era definida toda a arquitetura do software. Isso envolvia desde a organização de pastas do código-fonte até quantos servidores, além de quais padrões de projeto, seriam necessários para concluir o sistema com êxito (ENGHOLM JUNIOR, 2017; SOMMERVILLE, 2011).

Porém, com o uso das metodologias ágeis na construção de projetos de softwares, esse passo tornou-se incremental. Agora acontece sempre que existe a necessidade de entregar uma nova funcionalidade ou componente do sistema que não seja contemplado pela arquitetura atual. Dessa maneira, é importante determinar uma arquitetura que seja flexível e escalável dentro do projeto. O custo da refatoração de uma arquitetura de software durante a execução do projeto é bem caro (SOMMERVILLE, 2011).

Sendo assim, quando se fala em arquitetura é preciso analisar o problema da melhor maneira possível e imaginar qual é a melhor estrutura para atender à implementação do projeto. Isso guia um conjunto de decisões difíceis que devem ser tomadas. Definir uma arquitetura não tem uma receita pronta. É necessária certa experiência e estudos para entender quais componentes serão utilizados. Serão serviços de cloud computing? Qual a vantagem dessa decisão para o projeto? Os servidores serão ou não virtualizados? As aplicações serão distribuídas? Até que ponto utilizaremos essa estratégia? (SILVEIRA *et al.*, 2011).

Escolher os frameworks que irão compor o arcabouço de soluções do projeto; usar ferramentas de mapeamento de objetos relacionais (ORM) ou *stored procedures*; optar por projetos baseados em componentes ou baseados em ações. Todas essas são questões que descrevem o problema da estrutura do projeto de software, pois são todas questões arquiteturais que podem levar o projeto ao sucesso completo ou à ruína (SILVEIRA *et al.*, 2011).

A arquitetura deve identificar quais são os principais componentes estruturais do sistema e como será feito o relacionamento entre cada um. Assim, é considerada o elo crítico entre o projeto e a engenharia de requisitos (ENGHOLM JUNIOR, 2017; SOMMERVILLE, 2011). Pode-se dizer que existem dois níveis de abstração da arquitetura de software (SOMMERVILLE, 2011):

- **Arquitetura em pequena escala:** esse tipo de arquitetura concerne aos programas de maneira individual. Os programas, nesse caso, são os componentes que fazem parte da arquitetura. Nesse tipo de arquitetura, o foco é a decomposição do programa em componentes menores e mais definidos quanto à sua responsabilidade no conjunto.

- **Arquitetura em grande escala:** o foco desse tipo de arquitetura está em sistemas corporativos. Tais sistemas são complexos e incluem o uso em conjunto de outros sistemas, frameworks, componentes, etc. Geralmente, esses sistemas partilham da estratégia de sistemas distribuídos.

Depois do que foi exposto, pode-se dizer que a arquitetura está diretamente relacionada com a robustez e o desempenho do projeto de software, além de determinar a capacidade de distribuir o sistema em múltiplos lugares. Outro ponto em que a arquitetura ajuda bastante no projeto é na definição e no funcionamento dos requisitos não funcionais, que estão atrelados à arquitetura do projeto (SOMMERVILLE, 2011).

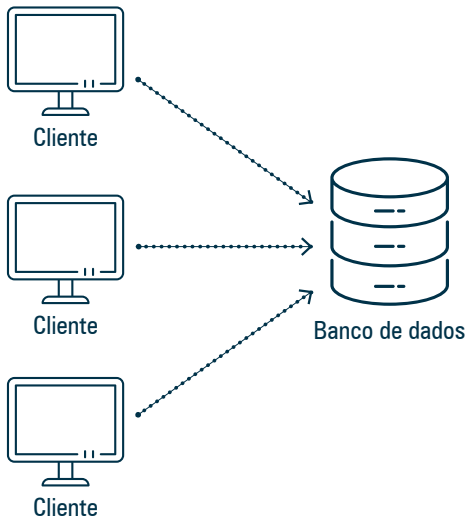
A documentação de uma arquitetura, em geral, é realizada através de um diagrama de blocos simples. Cada bloco representa um componente ou um conjunto de componentes. Quando representar um conjunto de componentes, este pode ser modelado em novos diagramas para atender a cada componente em específico. Sendo assim, cada componente deve ter seu próprio documento de arquitetura (ENGHOLM JUNIOR, 2017; SOMMERVILLE, 2011).

Existem três grandes tipos de arquitetura: monolítica, em camadas e orientada a serviços. Dentro de cada grande tipo, existem variações para acompanhar a evolução dos projetos. Cada projeto trabalha com requisitos particulares e que precisam atender ou resolver determinado problema. Assim, novos padrões arquiteturais sempre surgem, mas é possível, na maioria das vezes, classificá-los entre esses três tipos.

1 Arquitetura monolítica

Quando pensamos em uma arquitetura monolítica, geralmente falamos de um sistema desenvolvido para desktops. Sistemas monolíticos tendem a conter toda sua regra de negócio em um único local. A figura 1 exemplifica esse tipo de arquitetura.

Figura 1 – Arquitetura monolítica criada para que os clientes acessem um banco de dados centralizado



Na figura 1, é apresentada a composição de uma arquitetura monolítica junto a um banco de dados centralizado. Existem três clientes na figura, e toda vez que é feita uma consulta a alguma informação, todos os clientes acessam a mesma base de dados. Apesar do acesso centralizado aos dados, cada cliente possui todas as regras de negócios, de manipulação da interface do usuário e de acesso à parte de infraestrutura, como o banco de dados e a rede de computadores.

Clientes assim são, em geral, mais velozes, pois não necessitam de consultas a servidores e permitem uma gestão de operações off-line. Mesmo com a chegada dos sistemas web, muitas empresas adotavam esse tipo de cliente também no navegador, através das tecnologias Applet e, mais recentemente, JavaFX, no universo Java. Apesar de esses

clientes serem velozes para operação dos usuários, sua manutenção e, principalmente, a atualização dos sistemas eram difíceis (SILVEIRA *et al.*, 2011).

Isso acontece porque é necessário fazer a atualização de cada um dos clientes de maneira individual, máquina a máquina. Esse processo pesado de atualização levou à denominação cliente gordo, ou *fat client*, em inglês. Dentro do aplicativo cliente, estava contida toda a inteligência do sistema para execução das operações. Mas essa não é a única preocupação nesse tipo de arquitetura. O principal problema é que isso permite a instalação de clientes maliciosos que podem cortar a conexão com o banco de dados e derrubar o sistema inteiro de uma só vez (SILVEIRA *et al.*, 2011; SOMMERVILLE, 2011).

Para contornar essas situações, ferramentas de autenticação do cliente e procedimentos criados dentro do banco de dados eram formas de manter o código mais seguro e minimizar o acesso indesejado ao banco de dados por algum cliente malicioso (SILVEIRA *et al.*, 2011). A solução para esse tipo de arquitetura surgiu nos anos 1990, quando foi introduzida a arquitetura em camadas.

2 Arquitetura em camadas

Hoje em dia, quando surge o termo "arquitetura de software" em algum grupo de desenvolvedores, um pensamento único vem à tona para todos: camadas. Mas, no mundo da tecnologia, é sempre recomendado o uso das terminologias em inglês. Muitas vezes, traduzimos os termos para uma única palavra, como é o caso de "camadas", e, ao fazermos esse tipo de simplificação, não deliberadamente, mas por limitações do próprio idioma, podemos perder nuances do significado das coisas.

IMPORTANTE

No caso da arquitetura em camadas, temos dois tipos de separação. A primeira separação de camadas acontece em um nível lógico. Essas

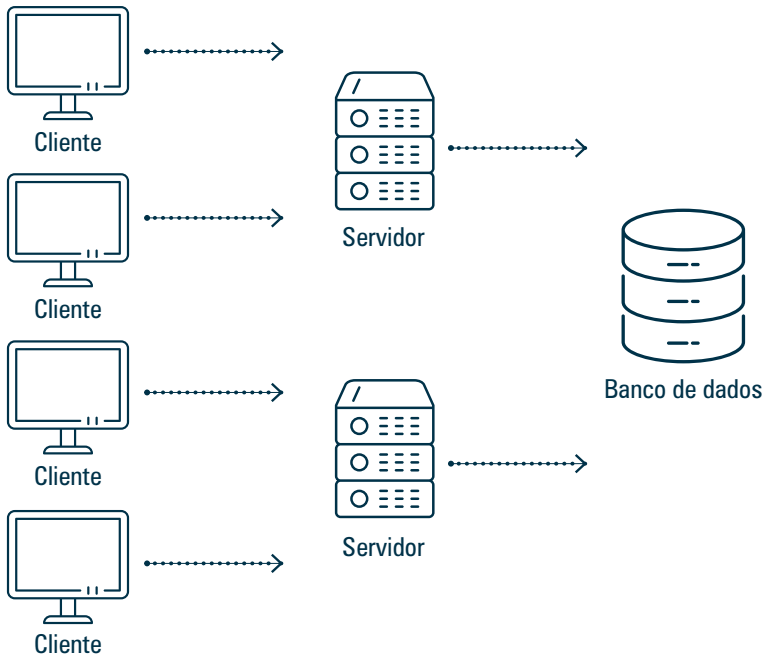
camadas são chamadas de *layers*. O segundo tipo de camada acontece com uma separação física; nesse cenário, existem as *tiers*. Apesar de ambas serem traduzidas como "camadas", existe uma sutil diferença entre os dois tipos (SILVEIRA *et al.*, 2011).

Vamos entender melhor o que acontece entre os dois tipos de camadas. As *layers* estão relacionadas com a separação do código, com a forma como são agrupadas as responsabilidades desse conjunto de componentes ou programas. Assim, podemos reduzir o acoplamento entre cada parte do sistema, o que facilita sua manutenção, permitindo que alterar uma parte não afete as demais. A separação dos *layers* de um sistema deve ser bem estruturada, pois, caso contrário, podem existir problemas de manutenção e até mesmo de desempenho (FOWLER, 2002; EVANS, 2004; SILVEIRA *et al.*, 2011).

Já as *tiers* tratam da separação física da arquitetura. O exemplo mais simples que pode ser apresentado nesse caso é a separação do servidor de aplicação com o servidor de banco de dados. A evolução do cenário tecnológico tornou possível o uso de componentes distribuídos em serviços de cloud computing e até mesmo em servidores virtualizados dentro de um mesmo hardware. Esse avanço tecnológico, contudo, dificulta a visualização dos limites entre uma *tier* e outra (SILVEIRA *et al.*, 2011).

Durante o projeto arquitetural, a atenção à divisão das *tiers* deve ser dobrada. O excesso de camadas físicas pode gerar um problema catastrófico ao sistema. A justificativa sobre a quantidade de *tiers* no projeto surge com o argumento de que o sistema terá uma maior disponibilidade, escalabilidade, entre outros pontos. Contudo, é muito importante que fatores como a quantidade de chamadas remotas executadas entre as *tiers* e os demais componentes sejam bem dimensionados, pois isso pode comprometer o desempenho do sistema e até mesmo sua escalabilidade, justamente uma das vantagens desse tipo de divisão (SILVEIRA *et al.*, 2011; SOMMERVILLE, 2011). A figura 2 apresenta a arquitetura em camadas de uma aplicação web corporativa simples.

Figura 2 – Arquitetura de software com múltiplas camadas físicas (*multitiers*)



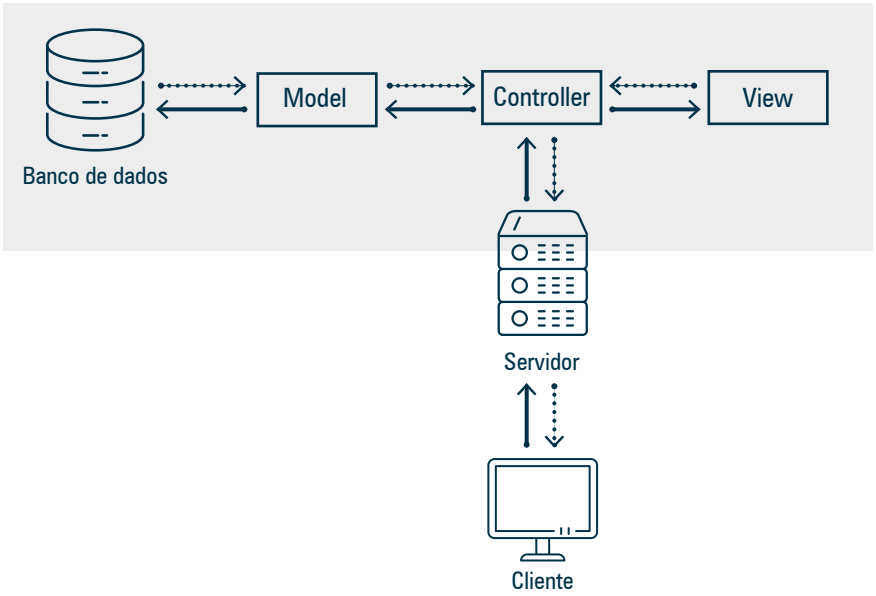
Na figura 2, são apresentados quatro clientes, que podem acessar qualquer um dos dois servidores de aplicação (ambos com a mesma versão do sistema), que, por sua vez, acessam a mesma base de informação. Assim, temos uma pequena evolução da arquitetura monolítica com a arquitetura de camadas. Claro que não basta apenas duplicar os servidores e o projeto de arquitetura em camadas estará pronto. Para manter ambos os servidores funcionando adequadamente, ferramentas de balanceamento de carga, protocolos, clusters, entre outros elementos, são utilizados. Saindo do mundo físico e indo para o mundo lógico, existem muitos padrões de arquitetura em camadas, como o model-view-controller (MVC) ou o proposto por Eric Evans (2004) em quatro camadas em domain-driven design. Vamos nos aprofundar mais no modelo MVC, por ser considerado um dos mais populares e também a base para os demais modelos.

2.1 Padrão model-view-controller (MVC)

Com o surgimento das servlets no Java EE em 1997, o desenvolvimento web ganhou uma nova cara, e seu desempenho em relação às linguagens que utilizam o CGI como base era acima da média. As servlets traziam as abstrações sobre o protocolo HTTP de forma transparente aos desenvolvedores. Na parte da interface gráfica do usuário, as views existiam através um mecanismo de templates. Nas tecnologias web, a grande evolução se deu com a criação do padrão arquitetural model-view-controller, o popular MVC (SILVEIRA *et al.*, 2011).

O padrão funciona com a divisão lógica do sistema em três camadas. A camada model é responsável por toda a lógica do negócio. A view cuida principalmente da apresentação ao usuário. E, por fim, a camada controller faz a intermediação entre as chamadas dos clientes e as camadas view e model. A figura 3 apresenta o fluxo entre o cliente e as camadas MVC.

Figura 3 – Diagrama do padrão arquitetural em camadas MVC



O processo de comunicação no padrão MVC é iniciado a partir de uma requisição feita no cliente. A requisição é enviada ao servidor, que acessará a aplicação. O servidor direciona a entrada das informações ao controller, que irá identificar o que o cliente está solicitando. Imagine que essa solicitação seja uma lista de produtos cadastrada no banco de dados, que será exibida ao cliente no formato de uma tabela interativa.

Nesse momento, o controller solicita as informações à camada model; esta é responsável por acessar o banco de dados e efetuar a busca das informações desejadas pelo cliente. Os dados recuperados no banco de dados pela model são encaminhados ao controller em resposta ao pedido solicitado. Para trabalhar nas informações visuais, a camada controller invoca a view, passando os dados recuperados pela model. Com os dados em mãos, a view trabalha na construção de uma apresentação visual ao cliente. Assim que finalizada, ela devolve ao controller, que redireciona ao cliente a resposta através do servidor de aplicação.

Esse é o fluxo básico de uma arquitetura em camadas que atende ao padrão MVC. Apesar de as camadas view e controller estarem bem claras para a grande maioria dos desenvolvedores, a model é um pouco mais complexa. O que realmente vai dentro dessa camada no projeto de um sistema? Podemos encontrar objetos/componentes referentes a classes de domínio, repositórios e objetos de acesso aos dados (DAO – data access objects). Entretanto, existem algumas arquiteturas que auxiliam um pouco mais nessa organização, como a arquitetura limpa, que traz um novo tipo de camada para tentar desmistificar o que está dentro da model.

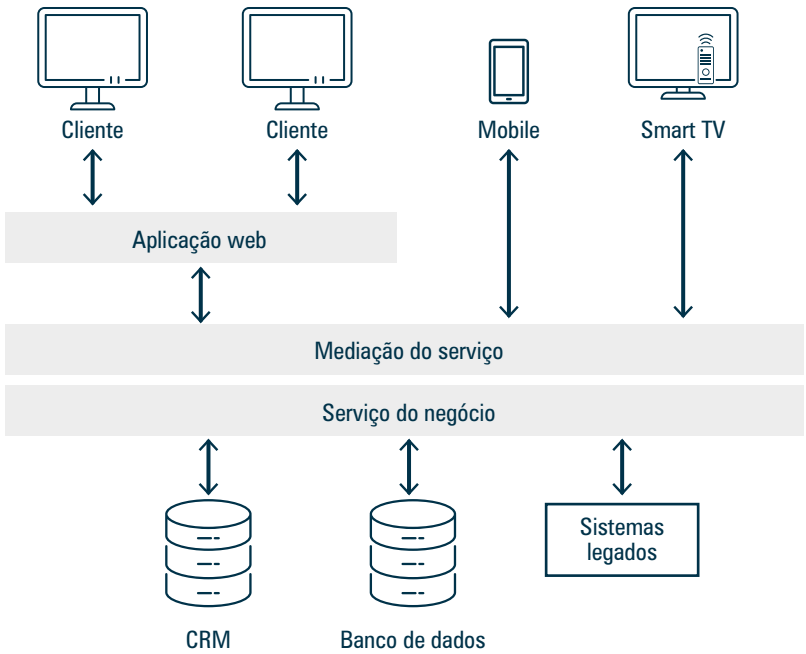
Os modelos apresentados até o momento são todos voltados à comunicação cliente-servidor, mas o que acontece quando a comunicação precisa ocorrer entre dois sistemas diferentes, sem intervenção de um cliente? Para essa situação, existe a arquitetura orientada a serviços.

3 Arquitetura orientada a serviços (SOA – service-oriented architecture)

Até agora, os modelos de arquitetura apresentados tinham o cenário de cliente-servidor muito bem definidos. Entretanto, com a evolução da tecnologia, houve a necessidade de se trabalhar com a comunicação direta entre os sistemas. Esse tipo de arquitetura tem como principal objetivo disponibilizar as funcionalidades do sistema através de serviços. Ela ficou conhecida como arquitetura orientada a serviços (SOA – service-oriented architecture).

A SOA tem como objetivo facilitar a integração entre sistemas, orientando a criação e a disponibilização de soluções modulares e fracamente acopladas, tendo como base o conceito de serviços. Alguns dos benefícios desse tipo de arquitetura são: facilidade de manutenção, reúso e controle de componentes, flexibilidade, qualidade e menor custo. Na figura 4, temos um exemplo da arquitetura orientada a serviços.

Figura 4 – Exemplo de uma arquitetura de software orientada a serviços



Na figura 4, temos dois clientes que estão conectados à aplicação web, que está conectada com uma camada de mediação do serviço. Também se conectam a essa camada um celular e uma smart TV. A camada de mediação apenas trata as requisições do serviço do negócio. O serviço dá acesso a todos os bancos de dados da empresa, como o de CRM (customer relationship management), ao banco de dados corporativo e aos sistemas legados. O que é possível inferir pelo diagrama é que o serviço atual funciona como uma porta de acesso ao sistema da companhia. Através dele, diversos outros sistemas conectam-se ao banco de dados da empresa. Perceba que os clientes não se conectam diretamente aos serviços: existe uma aplicação entre cliente e serviço que intermedeia essa conexão, fazendo a comunicação existir também entre sistemas.

IMPORTANTE

Um serviço deve definir quais e como são os dados acessados no sistema da companhia. Como esses serviços acontecem em uma camada física de rede, para que haja a comunicação entre os sistemas, eles são conhecidos como web services. Porém não confunda a definição da arquitetura SOA com uma simplificação de web services. Estes são utilizados dentro da arquitetura orientada a serviços. Os serviços devem ter sua implementação independente das aplicações que irão consumi-lo (SOMMERVILLE, 2011).

Uma arquitetura orientada a serviços, em geral, é desenvolvida e implementada em computadores física e geograficamente distribuídos. Para garantir que os serviços sejam capazes de se comunicar sem falhas, alguns protocolos são estabelecidos (SOMMERVILLE, 2011; ENGHOLM JUNIOR, 2017). Esses protocolos servem para que a comunicação entre componentes ocorra com sucesso. Cada protocolo pode utilizar um idioma para escrever suas mensagens, que são necessárias para que todo o processo ocorra com sucesso. A compreensão das

mensagens deve ser fácil tanto para seres humanos (desenvolvedores) quanto para máquinas (aplicações). No mundo dos serviços, existem três representações comumente encontradas para mensagens, que são XML, JSON e YAML.

O XML é uma representação utilizada em diversos serviços, contudo dentre todas as três mencionadas é a mais verbosa, e, por isso, nas tecnologias mais atuais tem diminuído a frequência de seu uso.

Exemplo do XML:

```
<endereco>
  <rua>
    Avenida Caminho do Mar
  </rua>
  <cidade>
    São Bernardo do Campo
  </cidade>
</endereco>
```

O formato JSON traz uma escrita mais leve, com um visual mais padronizado. Com o grande volume de aplicações em JavaScript através do Node.js, é o formato mais utilizado para comunicação de serviços por meio da internet hoje.

Exemplo do JSON:

```
{ endereco:
  {
  rua: Avenida Caminho do Mar,
  cidade: São Bernardo do Campo
  }
}
```

Por fim, temos o formato YAML, que é mais utilizado em arquivos de configuração de servidores e aplicações. Sua escrita se assemelha

muito à maneira como escrevemos nossos documentos no dia a dia. Contudo, para comunicação entre serviços, pode apresentar uma pequena perda de desempenho devido à falta de delimitadores.

Exemplo do YAML:

```
endereco:
rua: Avenida Caminho do Mar
cidade: São Bernardo do Campo
```

Os padrões podem influenciar, e muito, o tipo de idioma que será adotado para as mensagens. Um dos padrões para a criação de web services é o protocolo simples de acesso a objetos, ou SOAP (simple object access protocol).

3.1 Protocolo simples de acesso a objetos (SOAP – simple object access protocol)

O SOAP baseia-se em uma invocação remota de um método. Para tal, necessita especificar o endereço do componente, o nome e os argumentos desse método. Os dados são formatados em uma mensagem XML e são enviados e recebidos por meio do protocolo-padrão da internet, o HTTP. Componentes implementados como web services SOAP devem funcionar independentemente do sistema operacional e da linguagem de programação utilizados.

Nesse padrão de comunicação entre sistemas, existem dois atores que fazem a interação: o sistema que realiza o pedido ao serviço e o sistema que provê o serviço. Para que a comunicação seja efetiva, existem três componentes importantes nesse processo. São eles:

- **UDDI – universal description, discovery and integration (descrição, descoberta e integração universais):** protocolo para organização e registro do web service.
- **WSDL – web service description language (linguagem de descrição de serviços web):** linguagem utilizada para fazer a descrição de um serviço.
- **SOAP – simple object access protocol (protocolo simples de acesso a objetos):** protocolo estabelecido para que a troca de mensagens seja executada.

O WSDL é baseado em XML e descreve tudo que é feito pelo web service (seus métodos, nomes, parâmetros e retorno do processo). Essa especificação foi desenvolvida pelo consórcio da W3C. O WSDL pode ser compartilhado como arquivo ou acessado on-the-fly (durante a execução do processo/programa), utilizando o próprio navegador para isso.

O SOAP se tornou um dos mais conhecidos formatos de mensagens e protocolo utilizados por web services. O protocolo de comunicação tem como base o XML, que permite a comunicação por mensagens entre aplicações via HTTP. Em linhas gerais, uma mensagem SOAP é um documento XML comum que possui um elemento chamado envelope, que contém um cabeçalho e um corpo. O envelope é o elemento principal do XML, que representa a mensagem. O cabeçalho é um mecanismo genérico que permite a adição de características à mensagem SOAP. Já o corpo é o elemento que contém a informação a ser transportada entre os sistemas.

O uso de SOAP é bem complexo e pode consumir muitos recursos computacionais. Dessa maneira, um novo padrão arquitetural vem surgindo junto com o conceito de web API (application programming interface – interface de programação de aplicações), que pode ser definido como um conjunto de rotinas e padrões estabelecidos, implementados e documentados por uma determinada aplicação, mas que podem ser

utilizados por demais aplicações. Dessa maneira, nenhum detalhe de implementação fica exposto, sendo necessário apenas o desenvolvedor saber o que deve ser enviado e o retorno dessa rotina.

Esse tipo de implementação permite a interoperabilidade entre os sistemas, através da comunicação entre eles. São alguns exemplos de API: Twitter Developers; Marvel Developers; Google Developers; Facebook Developers.

O principal protocolo de comunicação utilizado em sistema web é o HTTP (hypertext transfer protocol – protocolo de transferência de hipertexto), aplicado na world wide web desde 1990, sendo chamado de HTTP/0.9. A versão HTTP/1.0 foi desenvolvida no período entre 1992 e 1996, possibilitando agora a transmissão de outros tipos de informação que não apenas texto.

Durante a evolução do protocolo HTTP/1.0, foi desenvolvida também a nova versão, a HTTP/1.1, que viabiliza conexões persistentes e o uso de servidores proxy, melhora a organização do cache e propicia mais métodos de requisição além do *POST* e do *HEAD* (presentes na versão HTTP/1.0).

IMPORTANTE

Nesse mesmo período, Roy Fielding, em sua tese de doutorado, descreveu um projeto de arquitetura de software construído para servir a aplicações distribuídas em rede. Essa arquitetura é conhecida como REST, do inglês "representational state transfer" (transferência de estado representacional). O REST foi lançado efetivamente nos anos 2000 para descrever interfaces de comunicação entre sistemas utilizando quaisquer tipos de representação, inclusive texto puro.

O REST é bem mais leve que seu antecessor, o SOAP, que necessita de diversas abstrações extra para conseguir estabelecer a comunicação entre os demais sistemas. São algumas vantagens do REST:

- Simples, leve, fácil de desenvolver e evoluir.
- Tudo é determinado como um recurso (resource).
- Cada recurso é identificado através de um endereço (URI – unified resource identifier).
- Os recursos se comunicam através de representações como: HTML, XML, JSON e YAML.
- Utiliza o protocolo HTTP.
- Os métodos mais comuns utilizados são: *GET*, *POST*, *PUT* e *DELETE*.

Hoje, grande parte dos frameworks e padrões sobre a arquitetura do sistema utiliza REST para definir como serão realizadas as trocas de mensagens.

Considerações finais

Neste capítulo, apresentamos o conceito de arquitetura de computadores e sua evolução, desde as arquiteturas monolíticas até as arquiteturas orientadas a serviços. A cada dia, novos padrões são apresentados pela indústria, e precisamos sempre acompanhar esses avanços. O objetivo é sempre diminuir a complexidade do desenvolvimento e facilitar a compreensão do projeto por parte da equipe de desenvolvedores.

Referências

ENGHOLM JUNIOR, Hélio. **Análise e design orientados a objetos**. São Paulo: Novatec, 2017.

EVANS, Eric. **Domain-driven design**: tackling complexity in the heart of software. Boston: Addison-Wesley Professional, 2004.

FOWLER, Martin. **Patterns of enterprise application architecture**. Boston: Addison-Wesley, 2002.

SILVEIRA, Paulo *et al.* **Introdução à arquitetura de design de software**: uma introdução à plataforma Java. Rio de Janeiro: Elsevier Brasil, 2011.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson; Boston: Addison-Wesley, 2011.