FACULDADE DE TECNOLOGIA DE SÃO PAULO

ARQUITETURA EM CAMADAS

SÃO PAULO

2019

FACULDADE DE TECNOLOGIA DE SÃO PAULO

|  |  |
| --- | --- |
| Aluna | |
| Nº matrícula | Nome |
| 17200061 | Camilla Martins de Sales |
| Data: 19/02/2019 | |

|  |  |
| --- | --- |
| Disciplina: | Engenharia de Software III |
| Prof.: | Victor Troitiño |

SÃO PAULO

2019

**Arquitetura de software em camadas para iniciantes**

Por Pedro Bertoleti -22/05/2015

**O que é arquitetura de software embarcado e por que é importante?**

Para responder essas perguntas, considere o caso a seguir. Para ilustrá-lo, o protagonista chama-se João, que é um desenvolvedor de software embarcado.

O chefe de João lhe solicitou que fosse desenvolvido um sistema embarcado bare-metal e está muito ansioso para fazê-lo. Logo de cara, no exato instante em que ele pisou fora da sala do chefe, sua mente parecia efervescer! Um turbilhão de ideias de como desenvolver o sistema surge e a única coisa que João deseja é chegar no seu computador e mandar ver no código para dar vida às suas ideias.

No auge de sua empolgação, ele conseguiu desenvolver o código-fonte de uma única vez, como um relâmpago. Porém, durante a escrita, ele não se preocupou em absolutamente nada quanto a comentários, modularização de código, identação de código, etc. Para agilizar ainda mais seu desenvolvimento, ele tomou uma sábia decisão para o momento: pegar na internet exemplos de como fazer tal parte do sistema. Assim, haverá uma coisa a menos para desenvolver! E, num ímpeto incontrolável, copiou e colou sem dó nem piedade o trecho de código da internet no código dele, sem ao menos questionar se funcionava ou ainda verificar se tinha alguma lógica (afinal, se está na internet deve funcionar, né?). João, para mostrar que é muito eficiente, antes mesmo dos testes, modificou as rotinas e as deixou mais rebuscadas, afinal toda obra deve ser uma obra prima, certo?

Enfim, para dar vazão às suas ideias e terminar o projeto o quanto antes (e passar a impressão de ser um funcionário eficiente), ele simplesmente foi lá e fez.

É chegada a hora da verdade: o primeiro teste! Agora João se sente como Tony Stark prestes a dar vida à armadura do homem de ferro, afinal ele fez algo que lhe foi solicitado e fez rápido. Eis que o pior acontece: tudo que se é possível imaginar acontece, exceto funcionar. João desconfia de tudo: do hardware, do compilador, do Windows e, se bobear, até do que os seus olhos estão vendo. Em desespero, sai correndo mexendo em código e arrumando. Em resumo: João gastará no mínimo duas vezes o tempo previsto para realizar a tarefa.

Parece exagero a história, não? Acedite, se você não se preocupar com arquitetura de software, a grande maioria do que foi citado ocorrerá em 10 de 10 projetos que você desenvolver. Desenvolver rápido e metralhando código sem pensar e planejar e utilizando códigos de terceiros sem questionar ou ao menos testá-los isoladamente pode parecer eficiente em tempo (fazer certo visando somente o tempo de desenvolvimento), mas não é nada eficaz (ou seja, não é feita a coisa certa). Logo, se o João fosse também eficaz, tudo ficaria melhor.

Então vem a pergunta: o que é afinal arquitetura de software?

* Definição formal: A arquitetura de software de um sistema consiste na definição dos componentes de software, suas propriedades externas, e seus relacionamentos com outros softwares;
* Na prática: Arquitetar um software é organizá-lo de forma a ser o mais eficiente E eficaz possível.

Ou seja, o segredo de um bom software é a organização, mesmo quando importa-se códigos de terceiros (neste caso, cuidado redobrado e muito critério são necessários!). Utilizar uma arquitetura de software bem definida num projeto é fazer um software não necessariamente da maneira mais fácil e confortável, mas sim da forma que tenha melhor performance e que seja fácil de dar manutenção e evoluí-lo. Sim, todo projeto evolui (ganha funcionalidades e novas aplicações). Acredite, se não for definida uma boa arquitetura de software num projeto, a sensação de evoluí-lo é como se estivesse tentando sair da areia movediça: quanto mais você mexe, pior fica.

Meus amigos, aqui temos a primeira realidade do mundo dos sistemas embarcados: organização é tudo!

**Afinal, como organizar um software embarcado?**

Pois bem, a história acima serviu para ilustrar o que NÃO se deve fazer. Vejamos agora uma das formas corretas de como se organizar um software embarcado.

Em 99,999% das vezes, um software embarcado terá que interagir com baixo nível (interagir com periféricos ou outros sistemas embarcados, a nível de bit e bytes) e alto nível (reportar dados, status e realizar operações para uma camada de aplicação, mais próxima do usuário final). Logo, o primeiro passo para se adotar uma boa arquitetura de software embarcado é dividir de forma clara e objetiva as camadas do mesmo, fazendo, claro, uma comunicação clara e objetiva entre as camadas (por meio de funções / métodos / mensagens bem definidas e versáteis).

**Arquitetura de software em camadas**

As camadas mais “baixas” (que interagem a nível de byte e bit, conversam com periféricos, lidam com pilhas de comunicação mais robustas) são chamadas de camadas mais especialistas, enquanto as camadas acima, que conversam mais com aplicações, são chamadas camadas menos especialistas. Observe a figura 1, a qual ilustra um exemplo de divisão de camadas da comunicação TCP/IP.

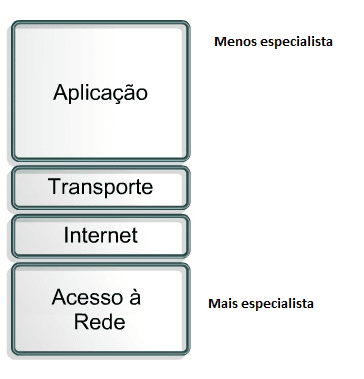


Figura 1 - Camadas simplificadas do TCP/IP

Desta figura podemos concluir, por exemplo, que a camada de aplicação (a camada menos especialista desse diagrama) não precisa “conhecer” como as camadas abaixo funcionam, porém ela se comunica por meio de mensagens claras e objetivas com a camada imediatamente inferior (Transporte). Assim como a camada de acesso à rede está preocupada em manter o canal de comunicação estável e seguro (do ponto de vista elétrico, sem risco de colisões de dados e coisas do gênero), ela precisa atender às solicitações da camada superior. Em resumo, é como se uma operação/trabalho fosse dividido em vários programas diferentes, cada qual com sua finalidade e objetivos.

**Vantagens de separar o software em camadas**

Mas afinal, quais são as grandes vantagens deste formato de organizar um software? Veja a seguir:

* Separando em camadas, pode-se fazer o desenvolvimento de um sistema em etapas. Com isso, além da organização, ganha-se tempo e performance (por mais poderosos que microcontroladores sejam, ganhar microssegundos faz toda a diferença no mundo dos embarcados!);
* Desenvolvendo desta maneira, o software das camadas tende a ser mais limpos, facilitando o entendimento futuro do software;
* Caso algum bug surgir, neste tipo de organização de software é mais fácil encontrá-lo e eliminá-lo, pois como as camadas são bem “isoladas” (focadas em funcionalidades específicas), fica relativamente simples descobrir em qual camada o bug está (por conseqüência, o tempo de debug diminui);
* Por último e não menos importante: organizando em camadas é possível substituir uma camada inteira por outra sem comprometer o sistema todo. Um exemplo disso é o próprio TCP/IP. Se for necessário trafegar os dados via Wi-Fi ou cabo (ethernet, por exemplo), basta alterar a camada mais especialista, já que as camadas acima comunicam-se com mensagens / métodos / funções que independem do canal de transmissão. Com isso, o código ganha versatilidade.

Sim meus amigos, se as camadas forem bem definidas, será quase como brincar de Lego!

Portanto, se o João da história contada no início desse artigo tivesse dividido o software dele em camadas, talvez não tivesse tantos bugs e, caso tivesse, facilmente identificaria onde eles estão.

Ok, tudo muito bonito, mas e no código? Como fica este tipo de arquitetura de software em camadas? Aqui começa a mão na massa em termos de código. Note que até agora somente foi arquitetado/pensado quais camadas o sistema terá, quais serão as suas funcionalidades e quais as funções/métodos/mensagens necessárias.

Esta é a segunda grande realidade do mundo do software embarcado: escrever código é a menor de suas preocupações. O mais importante é pensar em como tudo funcionará.

Para exemplificar uma divisão de camadas eficiente e eficaz, consideremos o seguinte exemplo: um sistema embarcado que comunica-se com um módulo bluetooth, capaz de conectar-se a um dispositivo alvo (fornecendo-se um MAC address), enviar e receber dados e ter uma interface amigável de entrada e saída de dados (um teclado e display).

Percebe-se que temos três camadas neste sistema:

Camada menos especialista: camada que contém rotinas de interface, ou seja, rotinas de display, teclado e ações a serem realizadas;

Camada intermediária: contém o tratamento das solicitações das camadas menos especialistas e mais especialistas, além do gerenciamento das mensagens recebidas e das que devem ser enviadas às outras camadas (organização das mesmas em filas tipo FIFO, por exemplo);

Camada mais especialista: composta do “rádio” (canal de transmissão do bluetooth) e comandos ao hardware bluetooth (bem como interpretar suas respostas).

Ou seja, nosso terminal bluetooth, de forma simplista, teria a seguinte estrutura:

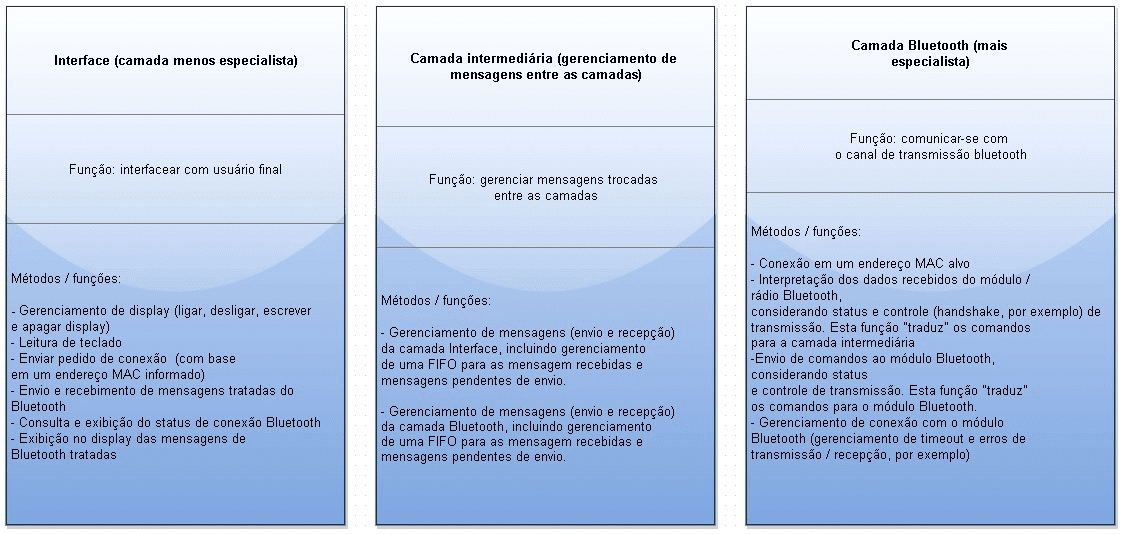


Figura 2 - Estrutura do Bluetooth

De forma simplista, aí está a organização em camadas de um terminal serial deste nível. Há pontos importantes a serem ressaltados, como por exemplo:

* Cada módulo ficou focado em uma tarefa macro / em um tipo de gerenciamento. Logo pode-se dividir-se o desenvolvimento em etapas (o que minimiza o tempo de desenvolvimento);
* Se, no futuro, for necessário substituir o canal de transmissão de bluetooth para ethernet, por exemplo, basta substituir a camada mais especialista. Assim como se for necessário alterar o tipo de display, basta alterar somente a camada menos especialista.

Em termos de código, que é o objetivo deste tópico, há dois caminhos:

Em compiladores e IDEs mais modernos, é possível separar o projeto em vários arquivos de código-fonte. Isto é ótimo nesta metodologia de software em camadas, pois neste caso bastaria fazer cada camada em um arquivo fonte (ou um arquivo fonte e um de cabeçalho, dependendo da linguagem escolhida);

Em compiladores e IDEs mais antigos, é necessário fazer o famoso “código linguição” (tudo em um só fonte). Neste caso, deve-se agrupar as camadas em funções, tomando cuidado redobrado com os nomes das funções e documentação/comentários.

Sim, parece besteira, mas dar nomes intuitivos às funções faz toda a diferença na hora de desenvolver, dar manutenção e evoluir. Recomendo fortemente adotar um padrão de nomenclatura e seguí-lo em todo o código, isso torna as coisas mais fáceis para você e para quem for trabalhar no código no futuro.

Como pode ser observado, cada camada possui funções que interagem com outras camadas. Logo, tais funções devem ter escopo global.

É plenamente válido (e recomendo em muitos casos) o uso de bibliotecas consagradas para “conversar” com hardwares específicos (de preferência bibliotecas fornecidas por fabricantes dos próprios hardwares). Porém, tais bibliotecas não foram desenvolvidas seguindo sua arquitetura e nomenclatura, logo deve-se tomar muito cuidado ao colocá-las no seu sistema. Com isso, temos a terceira realidade do mundo dos sistemas embarcados: critério na importação de código terceiro é fundamental.

**Conclusões**

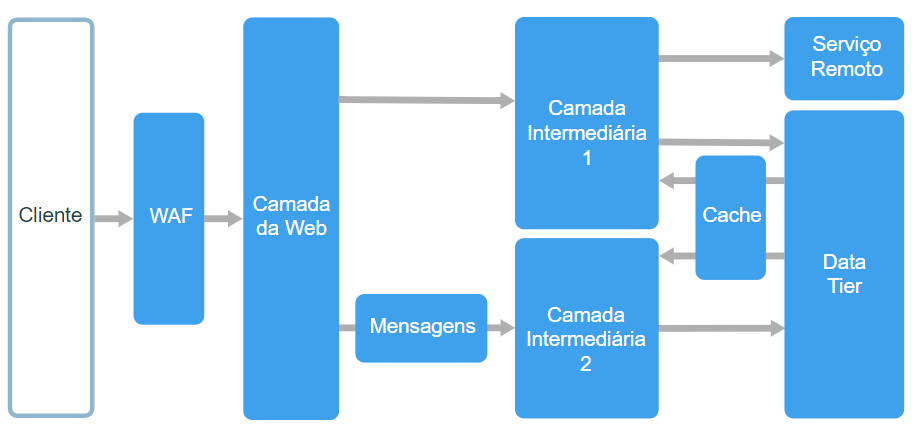
Resumindo tudo o que foi dito, uma boa arquitetura de software é aquela que permite seu software ficar organizado de forma que seja possível desenvolvê-lo, mantê-lo e evoluí-lo de forma bem definida. Para isto, foi apresentada a forma de arquitetura de software em camadas.

As camadas vão da menos especialista (mais próxima do usuário final) até a mais especialista (a camada que “conversa” com o hardware, a nível de bits e bytes, sendo que em caso de evolução ou alteração de software, basta alterar ou substituir camadas específicas sem comprometer o resto do sistema.

**Estilo de arquitetura de N camadas**

29/08/2018

Uma arquitetura de N camadas divide um aplicativo em camadas lógicas e camadas físicas.



Camadas são uma maneira de separar responsabilidades e gerenciar dependências. Cada camada tem uma responsabilidade específica. Uma camada superior pode usar os serviços em uma camada inferior, mas não o oposto.

As camadas são separadas fisicamente, sendo executadas em computadores separados. Uma camada pode chamar diretamente a outra camada ou usar mensagens assíncronas (fila de mensagens). Embora cada camada possa ser hospedada em sua própria camada, isso não é necessário. Várias camadas podem ser hospedadas na mesma camada. Separar fisicamente as camadas melhora a escalabilidade e a resiliência, mas também aumenta a latência da comunicação de rede adicional.

Um aplicativo tradicional de três camadas tem uma camada de apresentação, uma camada intermediária e uma camada de banco de dados. A camada intermediária é opcional. Aplicativos mais complexos podem ter mais de três camadas. O diagrama acima mostra um aplicativo com duas camadas intermediárias, encapsulando áreas diferentes de funcionalidade.

Um aplicativo de N camadas pode ter uma arquitetura de camada fechada ou um arquitetura da camada aberta:

* Em uma arquitetura de camada fechada, uma camada somente pode chamar a próxima camada imediatamente abaixo.
* Em uma arquitetura de camada aberta, uma camada pode chamar qualquer uma das camadas abaixo dela.

Uma arquitetura de camada fechada limita as dependências entre camadas. No entanto, ele poderá criar o tráfego de rede desnecessário se uma camada simplesmente passar as solicitações para a próxima camada.

**Quando usar essa arquitetura**

Arquiteturas de N camadas normalmente são implementadas como aplicativos IaaS (infraestrutura como serviço), com cada camada em execução em um conjunto separado de VMs. No entanto, um aplicativo de N camadas não precisa ser IaaS pura. Muitas vezes é vantajoso usar serviços gerenciados para algumas partes da arquitetura, particularmente cache, mensagens e armazenamento de dados.

Considere uma arquitetura de N camadas para:

* Aplicativos Web simples.
* Migrar um aplicativo local para o Azure com o mínimo de refatoração.
* Desenvolvimento unificado de aplicativos em nuvem e locais.

Arquiteturas de N camadas são muito comuns em aplicativos locais tradicionais, portanto, é uma opção natural para migrar cargas de trabalho existentes para o Azure.

**Benefícios**

* Portabilidade entre nuvem e local e entre plataformas de nuvem.
* Curva de aprendizado menor para a maioria dos desenvolvedores.
* Evolução natural do modelo tradicional de aplicativos.
* Abertura para ambiente heterogêneos (Windows/Linux)

**Desafios**

* É fácil terminar com uma camada intermediária que faz exatamente apenas CRUD no banco de dados, acrescentando latência extra sem fazer qualquer trabalho útil.
* O design monolítico impede a implantação independente de recursos.
* O gerenciamento de um aplicativo IaaS requer mais trabalho que o de um aplicativo que usa apenas serviços gerenciados.
* Pode ser difícil gerenciar a segurança de rede em um sistema grande.

**Práticas recomendadas**

* Use o dimensionamento automático para controlar as alterações na carga. Consulte Melhores práticas de dimensionamento automático.
* Use mensagens assíncronas para desacoplar camadas.
* Armazenar em cache dados semiestáticos. Consulte Melhores práticas de cache.
* Configure a camada de banco de dados para alta disponibilidade usando uma solução como Grupos de disponibilidade AlwaysOn do SQL Server.
* Coloque um WAF (firewall do aplicativo Web) entre o front-end e a Internet.
* Coloque cada camada na própria sub-rede e use sub-redes como um limite de segurança.
* Restrinja o acesso a camada de dados, permitindo solicitações somente das camadas intermediárias.

**Arquitetura de N camadas em máquinas virtuais**

Esta seção descreve uma arquitetura de N camadas recomendada em execução em VMs.

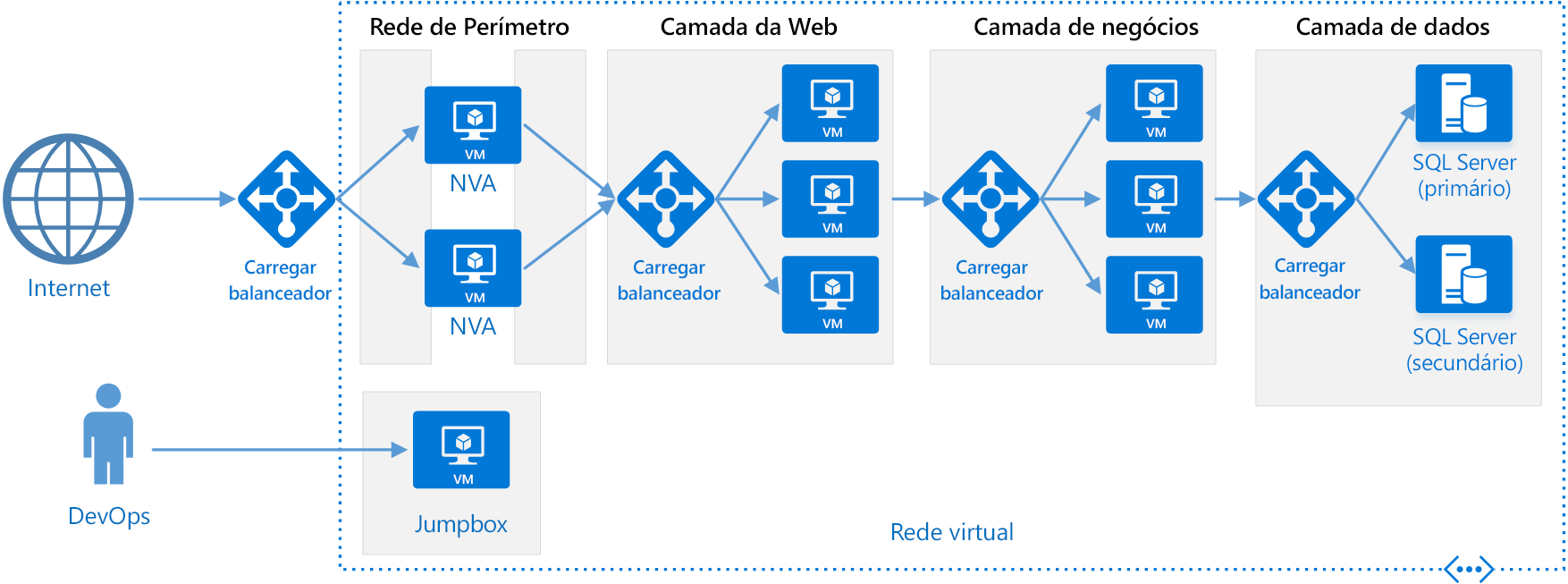


Diagrama físico de uma arquitetura de N camadas

Cada camada consiste em duas ou mais VMs, colocadas em um conjunto de disponibilidade ou um conjunto de dimensionamento de VM. Várias VMs oferecem resiliência no caso de falha da VM. Balanceadores de carga são usados para distribuir solicitações entre as VMs em uma camada. Uma camada pode ser dimensionada horizontalmente adicionando mais VMs ao pool.

Cada camada também é colocada dentro da própria sub-rede, o que significa que seus endereços IP internos ficam dentro do mesmo intervalo de endereços. Isso facilita a aplicação de regras NSG (grupo de segurança de rede) e o roteamento de tabelas para camadas individuais.

As camadas Web e comercial não têm estado. Qualquer VM pode lidar com qualquer solicitação para essa camada. A camada de dados deve consistir em um banco de dados replicado. Para Windows, é recomendável SQL Server, usando Grupos de Disponibilidade AlwaysOn para alta disponibilidade. Para o Linux, escolha um banco de dados com suporte para replicação, como Apache Cassandra.

NSGs (grupos de segurança de rede) restringem o acesso a cada camada. Por exemplo, a camada de banco de dados só permite o acesso da camada comercial.

**Considerações adicionais**

* Arquiteturas de N camadas não estão restritas a três camadas. Para aplicativos mais complexos, é comum ter mais camadas. Nesse caso, considere usar roteamento de camada 7 para rotear solicitações para uma camada em particular.
* As camadas são os limites de escalabilidade, confiabilidade e segurança. Considere ter camadas separadas para serviços com requisitos diferentes nessas áreas.
* Use Conjuntos de Dimensionamento de VM para dimensionamento automático.
* Procure locais na arquitetura em que você possa usar um serviço gerenciado sem refatoração significativa. Em particular, examine cache, mensagens, armazenamento e bancos de dados.
* Para maior segurança, coloque uma rede DMZ na frente do aplicativo. A DMZ inclui NVAs (soluções de virtualização de rede) que implementam a funcionalidade de segurança, como firewalls e inspeção de pacotes. Para obter mais informações, consulte Arquitetura de referência de DMZ da rede.
* Para alta disponibilidade, coloque duas ou mais NVAs em um conjunto de disponibilidade, com um balanceador externo de carga para distribuir solicitações de Internet entre as instâncias. Para obter mais informações, consulte Implantar soluções de virtualização de rede altamente disponíveis.
* Não permita o acesso direto a RDP ou SSH para VMs que estejam executando o código do aplicativo. Em vez disso, os operadores devem fazer logon em um jumpbox, também chamado de um host bastião. Esta é uma VM na rede que os administradores usam para se conectar às outras VMs. O jumpbox tem um NSG que permite que RDP ou SSH somente de endereços IP públicos aprovados.
* Você pode estender a rede virtual do Azure para a sua rede local usando uma VPN (rede privada virtual) site a site ou o Azure ExpressRoute. Para obter mais informações, consulte Arquitetura de referência de rede híbrida.
* Se a sua organização usar o Active Directory para gerenciar a identidade, você talvez queira estender seu ambiente do Active Directory para a VNet do Azure. Para obter mais informações, consulte Arquitetura de referência de gerenciamento de identidade.
* Se você precisar de disponibilidade mais alta do que a fornecida pelo SLA do Azure para VMs, replique o aplicativo em duas regiões e use o Gerenciador de Tráfego do Azure para failover. Para obter mais informações, consulte Executar VMs do Windows em várias regiões ou Executar VMs do Linux em várias regiões.

**Referências:**

<https://www.embarcados.com.br/arquitetura-de-software-em-camadas/>

<https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/n-tier>