**Também é uma forma de evitar sql injection**

**O que significa Consulta Parametrizada?**

Uma consulta parametrizada é um tipo de consulta SQL que requer pelo menos um parâmetro para execução. Um espaço reservado é normalmente substituído pelo parâmetro na consulta SQL. O parâmetro é passado para a consulta em uma instrução separada.

## explicando consulta parametrizada

Um dos principais motivos para o uso de consultas parametrizadas é que elas tornam as consultas mais legíveis. O segundo e mais convincente motivo é que as consultas parametrizadas ajudam a proteger o banco de dados contra ataques de injeção de SQL.

A seguir, é apresentado um exemplo de uma consulta parametrizada do ADO.NET:

SELECT LastName FROM Contacts WHERE ContactID = @ContactID;

@ContactID é o parâmetro para esta consulta, que pode ser definido em uma instrução subsequente semelhante à seguinte:

command.Parameters.Add (novo SqlParameter ("@ ContactID", theContactID));

### Parametrização

Normalmente quando pensamos em uma aplicação enviando consultas para execução no banco de dados, imaginamos algo extremamente simples e antiquado, em que a consulta inteira é um texto único. E essa visão simples é o que nos faz injetar os parâmetros da consulta (123) no corpo do comando (SELECT).

Por exemplo:

╔═══════════╗ ┌───────────────────────────┐ !!! ╔════════════════╗

║ Aplicação ║->--->-│SELECT ... WHERE id = '123'│->--->-║ Banco

╚═══════════╝ └───────────────────────────┘ ╚════════════════╝

Mas a realidade é muito mais complexa. Um fator é o contexto da sessão (como o estado da transação, configurações de sessão, objetos temporários…) que afeta tudo que é executado nela. E outro, tão importante quanto, é a estrutura completa da consulta, que pode ser enviada com muito mais informações que são usadas para transmitir parâmetros separados do corpo da consulta, assim como seus tipos de dados e nulidade. O exemplo a seguir mostra algumas dessas informações (em uma representação parecida com json por familiaridade, mas a comunicação costuma ser feita por protocolos de rede dedicados a esse propósito):

┌─────────────────────────────────┐

╔═══════════╗->--->-│{"q": "SELECT ... WHERE id = $1",│->--->-╔════════════════╗

║ Aplicação ║ │─────────────────────────────────│ ║ Banco de dados ║

╚═══════════╝->--->-│"params": [ { valor: "123", │->--->-╚════════════════╝

│ tipo: "integer", │

│ null: false } ] } │

└─────────────────────────────────┘

Dessa forma, mesmo que seja usado um parâmetro com valor malicioso (por exemplo, ' or '1' = '1), ele não corre o risco de ser interpretado como parte do comando, já que a) não foi injetado no corpo do comando; b) o tipo de dados e nulidade podem ser informados e usados como validação extra. Isso é chamado de [parametrização](https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html) de consultas e é a forma mais confiável de evitar injeções de SQL. Em alguns SGBDs, consultas parametrizadas podem ser até mais rápidas que consultas de texto simples por economizar parte do tempo de compilação que antes era desperdiçado com o parse dos parâmetros injetados. Contudo, existem alguns pontos muito importantes que precisamos garantir, mesmo parametrizando as consultas.

Infelizmente muitos softwares não têm esse conceito de consultas propriamente parametrizadas. Mas vários desses têm o conceito de consultas preparadas, que podem ser usadas também para parametrizar as consultas. Exemplos clássicos são [PHP](https://www.php.net/manual/en/security.database.sql-injection.php) e [MySQL/MariaDB](https://bobby-tables.com/php), que são a razão pela qual uma grande parte das recomendações na internet mencionam “prepared statements”. Mas um prepared statement é a solução para um problema distinto (alto tempo da etapa de compilação e otimização de consultas recorrentes), que traz suas próprias desvantagens (maior uso de memória no servidor), então não deveria ser usado sempre. Outros softwares, como o PostgreSQL, têm todas as quatro combinações de consultas parametrizada+não-preparada, parametrizada+preparada, não-parametrizada+preparada, e não-parametrizada+não-preparada.

Ainda outros não têm a possibilidade de consultas não parametrizadas, então toda consulta é parametrizada, por exemplo o MongoDB. Mas ainda é importante garantir que eles nunca interpretem os parâmetros, caso contrário isso abre oportunidade para outros tipos de injeção, como o $where que permite [injeção de javascript](https://nullsweep.com/a-nosql-injection-primer-with-mongo/).

Importante: consultas parametrizadas são efetivas contra injeções de dados somente quando os parâmetros nunca são injetados posteriormente e nem interpretados no destino. Se algum componente intermediário fizer a injeção, tornando a consulta em um comando de texto único, então as garantias são perdidas. Por exemplo, um ORM pode aceitar consultas parametrizadas, mas ao enviar elas para o banco de dados, o driver faz a injeção arriscada silenciosamente:

┌─────────────────────────────────────┐ !!! ┌──────────────────────┐ !!! ╔════════════════╗

╔═══════════╗->--->-│{"query": "SELECT ... WHERE id = $1",│->--->-│ SELECT ... WHERE ... │->--->-║ Banco de dados ║

║ Aplicação ║ │─────────────────────────────────────│ / │ id = '' or '1' = '1' │ ╚════════════════╝

╚═══════════╝->--->-│"params": [ { valor: "' or '1' = '1",│->-/ └──────────────────────┘

│ tipo: "integer", │

│ null: false } ] } │

└─────────────────────────────────────┘

Isso também significa que, quanto mais componentes entre o desenvolvedor e o banco de dados, maior o risco de injeção de SQL, já que qualquer um pode fazer a injeção e fazê-la incorretamente. Alguns frameworks, bibliotecas e ORMs fazem essa injeção silenciosa apesar de apresentar APIs parametrizadas para o desenvolvedor (como algumas implementações de API-DB de Python e PHP); alguns drivers de comunicação com bancos de dados também o fazem (como o psycopg2 ainda hoje e alguns drivers JDBC); e até os próprios bancos de dados podem fazer internamente a injeção de parâmetros antes da etapa de parse da consulta.

>>> Framework >>> ORM >>> driver >>> protocolo >>>

┌────────────┐ ┌────────────┐ !!! ┌──────────┐ !!! ┌──────────┐ !!! ╔════════════════╗

╔═══════════╗->--->-│{"q": "...",│->--->-│{"q": "...",│->--->-│SELECT ...│->--->-│SELECT ...│->--->-║ Banco de dados ║

║ Aplicação ║ │────────────│ │────────────│ / └──────────┘ └──────────┘ ╚════════════════╝

╚═══════════╝->--->-│"p": [...] }│->--->-│"p": [...] }│->-/

└────────────┘ └────────────┘

Em resumo, injeções de SQL não podem ser mitigadas apenas pelo uso de uma camada adicional de software – na verdade, camadas adicionais aumentam o risco. Também é impossível garantir que a aplicação esteja imune, já que o erro em qualquer ponto dessa esteira de comunicação será propagado até o banco de dados de forma quase que indetectável.

## Conclusão

Além de consultas parametrizadas, muitas outras estratégias também devem ser usadas, como whitelist de valores, análise de código da aplicação, análise de logs do servidor de banco de dados, auditoria, uso de valores com tipos de dados, entre outros. Concatenação de texto nunca deveria ser usada com valores, mas sim deixada apenas para casos em que for estritamente necessária, como para adicionar cláusulas a mais na estrutura da consulta.

Por fim, injeções de dados devem ser tratadas com muita atenção durante o desenvolvimento do software, que deve passar por rigorosas validações de segurança em todos os componentes. Injeções são mais um [espectro](https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)) que nunca deixará de assombrar o mundo da tecnologia.

## SQL Injection

## Mesmo as melhores soluções contra injeções de dados ainda precisam ser usadas com muito cuidado, como as consultas parametrizadas que usamos contra SQL Injections.

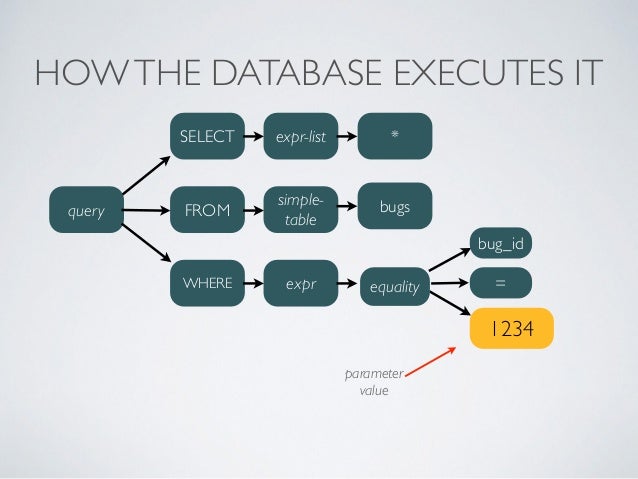
## SQL Injection

A injeção de dados mais popular e a que mais nos interessa agora é a injeção de SQL. Ela acontece quando um comando SQL é construído concatenando textos, especialmente quando isso inclui dados de origem insegura, causando o vazamento de dados ou a execução de outros comandos inesperados.

Por exemplo, se o endpoint de API http://example.com/app/tabelaView?id=123 tiver a seguinte consulta no backend:

String query = "SELECT \* FROM tabela WHERE ID='" + request.getParameter("id") + "'";

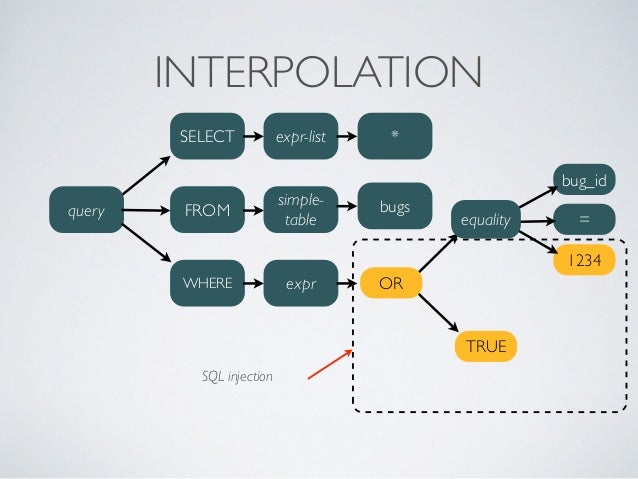
Que gera o plano de execução similar a:

[](https://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies)

Então essa API é suscetível a requisições maliciosas como http://example.com/app/tabelaView?id=' or '1'='1 que, pela concatenação acima, gera o comando abaixo:

SELECT \* FROM tabela WHERE ID='' or '1'='1'";

O plano de execução é similar e retornará a tabela completa:

[](https://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies)

Formas de evitar sql injection

## Mitigações

Existe uma longa lista de ações de mitigação que são usadas para evitar injeções de SQL. Mas é de vital importância compreender quando usar e quando não usar cada uma delas, já que algumas falham em proteger os sistemas contra injeções de dados, outras aumentam a complexidade do seu código, algumas introduzem bugs tão perigosos quanto as injeções que tentam evitar e ainda outras causam uma falsa sensação de segurança ao mover a responsabilidade para outro componente (normalmente menos confiável). Em resumo, é importante dissolver vários dos [mitos](https://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies) que cercam as injeções de SQL. Então vamos ver aqui apenas algumas das mitigações mais comuns.

### **Quoting e escaping**

Quoting é a ação de adicionar aspas simples, duplas ou outros limitadores de texto em torno do dado que está sendo concatenado, na tentativa de que ele seja interpretado corretamente pelo SGBD. Como visto no exemplo da API, mesmo com aspas simples em torno do campo injetado é possível contornar esse mecanismo quando o dado contém aspas. Portanto, essa estratégia não é uma mitigação válida contra injeção de SQL.

Escaping é a ação de adicionar caracteres especiais para que as aspas do dado não sejam interpretadas como fim do dado, por exemplo tornando ' em \' e \ em \\. Com isso, a consulta anterior não mais retornaria o conteúdo completo da tabela, mas sim um erro de tipo de dados ou uma resposta vazia:

SELECT \* FROM accounts WHERE custID='\' or \'1\'=\'1'";

Então aparentemente o problema foi solucionado através do uso simultâneo de quoting e escaping. Contudo, essa solução não é ideal, por diversas razões.

Um erro em qualquer função de quoting ou escaping invalida completamente a estratégia. E erros dessa natureza não são incomuns, com novos bugs sendo descobertos diariamente e atualmente mais de 1350 bugs de quoting listados na base de dados de detalhes de CVE <https://www.cvedetails.com/google-search-results.php?q=quoting&sa=Search> e 7300 bugs de escaping na mesma base de dados <https://www.cvedetails.com/google-search-results.php?q=quoting&sa=Search>. Muitos desses bugs são em drivers, bibliotecas, ORMs e frameworks populares, especialmente provenientes de plataformas comunitárias como PyPI, RubyGems e módulos de npm.

Além disso, a consulta fica poluída com os dados enxertados e consequentemente, seu tempo da etapa de parsing aumenta. E adicionar mais rodadas de quoting e escaping não diminui os riscos, mas intensifica esses efeitos colaterais.

Então é importante perceber que quoting e escaping nunca serão a solução ideal, então devem ser vistos mais como um plano B. Felizmente existem soluções mais seguras, confiáveis e leves [https://cheatsheetseries.owasp.org/cheatsheets/SQL\_Injection\_Prevention\_Cheat\_Sheet.html](https://tureba.org/dispon%C3%ADveis.html).

**XSS Residente**

Falhas de XSS Residente ocorrem sempre que uma aplicação recebe dados não confiáveis e os envia ao navegador sem validação ou filtro adequados. Permite aos agentes de ameaça executarem scripts no navegador da vítima que podem obter sessões do usuário, desfigurar sites, ou redirecionar o usuário para sites maliciosos [OWASP 2013]. Diferente da injeção de SQL que busca executar instruções SQL maliciosas em consultas no banco de dados ocorrendo fora da vista dos usuários, o XSS Residente busca executar marcações maliciosos ou códigos Java Script em valores que serão mostrados em uma página WEB. Este tipo de ataque busca enganar o usuário que tem confiança em um site levando-o a executar uma ação ou exibição de informações para um outro site não confiável [Snyder, Myer e Southwell 2010]. Na Figura 3, verificam-se os passos de um ataque XSS Residente, neste caso os agentes de ameaça são considerados qualquer pessoa que possa enviar dados para a aplicação, tanto usuários internos, externos e ate mesmo administradores. Os vetores de ataque são compreendidos por scripts que são enviados ao interpretador do navegador. Considera-se que praticamente qualquer fonte de dados pode ser um vetor de ataque inclusive fontes internas do banco de dados, como no caso do XSS Residente. As vulnerabilidades de segurança exploradas pelo Cross-Site Scripting são as mais comumente localizadas em aplicações WEB já que elas ocorrem quando os dados fornecidos pelos usuários são inclusos pela aplicação sem a devida filtragem e validação. Os impactos técnicos que o ataque pode causar são a execução de scripts no navegador da vitima visando o sequestro da seção do usuário, a pichação de sites, inserção de conteúdo hostil e redirecionamento dos usuários. Os impactos no negócio que o ataque XSS Residente poderá causar são a exposição publica de maneira negativa da imagem da empresa, expondo para seus clientes uma vulnerabilidade grave no sistema que usam, bem como violação da confidencialidade dos dados que são trafegados através do sistema [OWASP 2013]. Figura 3. Fluxo de um ataque Cross-Site Scripting (XSS Residente). Fonte:  
OWASP Top 10, 2013. As vulnerabilidades relacionadas ao XSS Residente estão ligadas a não filtragem de todas as entradas enviadas pelos usuários bem como a não avaliação da segurança das informações na entrada antes que sejam incluídas na pagina de saída. Sem um filtro apropriado as entradas fornecidas serão consideradas conteúdos ativos no navegador, existem algumas ferramentas que podem localizar algumas falhas de XSS automaticamente, entretanto cada aplicação possui paginas de saída diferenciadas utilizando interpretadores diversos juntamente com o navegador como JavaScript, Flash, Silverlight e ActiveX causando maior dificuldades para uma detecção automatizada assim necessitando de uma analise manual, bem como testes de invasão para analise das vulnerabilidades [OWASP 2013]. Prevenir os ataques por XSS consistem na separação de dados inseguros do conteúdo ativo do navegador, assim deve-se efetuar uma filtragem total dos dados inseguros baseados no contexto HTML (corpo, atributo, JavaScript, CSS ou URL) [OWASP 2013].  
3. Materiais e Métodos Para a realização do experimento o ambiente foi configurado com o S.O. (sistema operacional) CentOS 6.4 32bits, contendo o servidor de aplicação GlassFish 41 rodando uma Servlet2 Java que executa uma consulta SQL3 no banco de dados PostgreSQL. Para Riggs e Krosing (2011) o PostgreSQL é um avançado servidor de banco de dados SQL que possui amplo leque de plataformas disponíveis. Tendo como um grande ponto positivo possuir código aberto o que permite a personalização bem como uma licença  
1 GlassFish é um servidor de aplicações Java em sua plataforma empresarial. Suporta todas as especificações da API Java EE. É através de um servidor de aplicação que se tem um ambiente para instalação e execução de aplicações Java específicas, sem precisar instalar a aplicação nos computadores clientes.  
2 Servlet é uma classe Java usada para estender as funcionalidades de um servidor; semelhante a um servidor, ele gera dados HTML e XML para a camada de apresentação de uma aplicação WEB.  
3 SQL ou Structured Query Language, ou Linguagem de Consulta Estruturada, é a linguagem padrão utilizada em bancos de dados relacionais sendo simples e de fácil utilização.

muito tolerante quanto a sua instalação, uso e distribuição. Atualmente se encontra na versão estável 9.3.3, o desenvolvimento desta versão possibilitou a expansão da confiabilidade, disponibilidade e integridade das informações, informa a Comunidade Brasileira de PostgreSQL (2013). No que diz respeito à segurança, o PostgreSQL se destaca já que isso sempre foi uma das preocupações da equipe de desenvolvedores do SGBD4. Supri a necessidade dos utilizadores mesmo em aplicações complexas, que exijam o controle de grandes volumes de dados, ou em casos de tratamento de informações críticas. Para aumentar a segurança das informações que trafegam pela rede entre o SGBD e os clientes, é possível criptografar a troca de informações entre o cliente e o servidor através do SSL (Secure Sockets Layer) que é nativo do PostgreSQL para uso nas conexões [PostgreSQL 2006]. Outro aspecto importante é o mecanismo para criação de cópias de segurança e sua restauração em caso de falhas, já que devido a importância das informações contidas nos bancos de dados não se pode correr o risco de perdê-las por uma falha de hardware, queda de energia ou qualquer outro tipo de ameaça a consistência, integridade e disponibilidade dos dados [Lucas 2006]. Dentro do banco de dados foram criadas duas tabelas: a cliente e usuário. Conforme a Figura 4, ambas criadas dentro do banco de dados chamado tcc. A tabela cliente contém os campos “cpf”, “nome”, “endereco” e “celular”. Já a tabela usuário armazena informações referentes ao usuário do sistema, uma vez que para acessar os dados do sistema é necessário estar autenticado e ter permissão de acesso. As informações contidas nessa tabela são “nome”, “email” e “senha”, além de um  
identificador único de usuário chamado “idusuario”. Após a criação das tabelas, foi criada uma view (visão) chamda vw\_clientes que faz a seleção de um cliente com base em um CPF informado por parâmetro. Além disso, foi criado um usuário no banco de dados PostgreSQL chamado appweb com permissão apenas de seleção (SELECT) dos registros da view vw\_clientes, não tendo nenhuma permissão nas tabelas usuário e cliente; além disso, esse usuário tem acesso permitido somente a partir do endereço local (localhost). Paralelamente foi criado uma function (função) chamada noxss visando converter script potencialmente perigosos em entidades HTML buscando prevenir a execução do XSS Residente

Importante falar sobre isso

## OWASP

O [OWASP](https://owasp.org/) (Open Web Application Security Project) é uma fundação com o objetivo de melhorar a segurança de sistemas, em especial aplicações web, através do uso de softwares abertos, ferramentas, recursos, treinamentos e conscientização sobre o uso dos dados.

Ele mantém o [OWASP Top Ten](https://owasp.org/www-project-top-ten/), que é um documento destinado a desenvolvedores e que relata os dez maiores riscos de segurança para as aplicações web. No topo dessa lista está a [injeção](https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.html) de dados.

Uma injeção de dados é feita quando dados são inseridos em comandos que serão executados. E quando eles causam uma execução inesperada dos comandos, isso pode trazer consequências desastrosas para a confiabilidade, integridade e segurança dos sistemas, como demonstrado pela famosa tirinha [XKCD](https://xkcd.com/) [“Exploits of a mom”](https://xkcd.com/327/), onde uma injeção maliciosa é responsável pela perda dos registros de alunos de uma escola. Injeções podem afetar tudo: desde protocolos de rede, comandos SQL, arquivos XML, arquivos multimídia, serviços de autenticação como LDAP, cabeçalhos SMTP e HTTP, comandos do sistema operacional, consultas NoSQL e diversos outros.

**Conexões**

A conexão padrão é composta por propriedades básicas que indicam o servidor, o título do banco ao qual o programa deve se conectar e informações de autenticação (usuário e senha). Veja abaixo:

Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;

Acima podemos ver a estrutura de uma conexão padrão com as propriedades exibidas na ordem em que foram descritas. Esta estrutura é descrita como padrão por ser o suficiente para a maior parte dos projetos. Porém a mesma definição de padrão pode variar de acordo com a tecnologia utilizada para desenvolvimento ou serviço de dados.

A propriedade Server: É definida aqui através do domínio ao qual o servidor do banco de dados pode estar relacionado assim como pode ser um endereço ip ou nome de uma instância local (veremos mais a frente).

A propriedade Database: Esta propriedade deve receber o nome do banco de dados.

A propriedade User Id: também conhecida como nome do usuário ou username deve ser usada para identificar quem ou qual usuário está tentando se conectar através desta conexão.

A propriedade Password: Deve ser preenchida com a palavra-passe ou senha definida para o mesmo usuário que foi definido na propriedade User Id.

**Conexão confiável**

Neste modo de conexão o SQL Server autentica o usuário através de um token (referente à entidade do windows). Este formato é o formato padrão do SQL Server e considerado mais seguro que a autenticação do SQL Server (usuário e senha).

Uma conexão confiável pode ser estabelecida adicionando o parâmetro Trusted\_Connection: Ao adicionar este parâmetro você está permitindo que os usuários sejam autenticados através das credenciais do windows.

Server=myServerAddress;Database=myDataBase;Trusted\_Connection=True;

Utilizando este recurso é possível atribuir permissões a grupos de usuários do sistema que podem ser criados e configurados através do próprio windows (no caso do MSSQL Server). E posteriormente estes terem acesso ao servidor de dados.

**Conexão a uma instância**

Conectar à uma instância nada mais é que especificar através do nome a instalação do banco de dados que você deseja utilizar. Podemos ter várias versões do mesmo banco em uma única máquina e ao conectar sem declarar este nome, estamos nos conectando à instância padrão.

Ao adicionarmos a propriedade Server: Podemos além de declarar o nome ou endereço do servidor, a instância que desejamos nos conectar. Veja no exemplo abaixo:

Server=myServerName\myInstanceName;Database=myDataBase;User Id=myUsername;Password=myPassword;

**Utilizando uma porta diferente**

Partindo da string de conexão padrão existem casos onde o servidor do nosso banco de dados está configurado para permitir conexões a partir de uma porta "não padrão". No caso do SQL Server esta porta é atualmente a porta 1433.

Mas para apontar uma porta iremos utilizar poucas modificações na string de conexão padrão. Sendo necessário adicionar apenas uma vírgula após o nome do Servidor e logo em seguida a porta que desejamos. Veja o exemplo:

Server=myServerAddress, 1433;Database=myDataBase;User Id=myUsername;Password=myPassword;

**Conexão via endereço de IP (Internet Protocol)**

Nas conexões deste tipo não utilizamos a propriedade Server e sim Data Source para especificar onde está nosso banco de dados. Como valor este campo pode receber o endereço IP do servidor e por padrão terá a porta 1433. Caso a porta configurada seja outra podemos configurar conforme vimos no tópico acima.

A propriedade Network Library: Especifica o valor da conexão com o banco de dados. Podendo ser: DBNMPNTW (Pipes nomeados), DBMSRPCN (Multiprotocol/RC), DBMSVINN (Banyan vines), DBMSSPXN (IPX/SPX) e DBMSSOCN (TCP/IP) que é o padrão.

Na propriedade Initial Catalog: Devemos especificar o nome do banco de dados e os demais campos seguem como padrão. Veja o exemplo:

Data Source=190.190.200.100,1433;Network Library=DBMSSOCN;Initial Catalog=myDataBase;User ID=myUsername;Password=myPassword;

**Conclusão**

Este artigo desenvolveu definições para casos frequêntes do uso de strings de conexões. Veja que tratamos das propriedades mais comuns. Existem algumas outras propriedades que especificam atributos como criptografia e tempo de conexão.

Se você está utilizando este artigo como material de estudo sugiro que instale uma instância de banco de dados e configure de forma a testar um conjunto de configurações especificas para cada instância. Então poderá fazer comparações relativas ao resultado de cada conjunto.

**Referências**

[Strings de conexão | SQL SERVER](https://www.connectionstrings.com/sql-server/)

[SqlConnection.ConnectionString | Documentação da Microsoft](https://docs.microsoft.com/pt-br/dotnet/api/system.data.sqlclient.sqlconnection.connectionstring?view=dotnet-plat-ext-6.0)