afecarvalho@ipt.pt

Coisas “a ter em conta”

Segurança e “boas práticas”

afecarvalho@ipt.pt

Design e implementação de APIs

afecarvalho@ipt.pt

Design e implementação de APIs

Fazer uma API com operações de leitura e escrita é fácil. Fazer uma *boa* API com operações de leitura e escrita é uma arte.

Isto aplica-se principalmente a APIs REST, visto que REST é bastante liberal na sua especificação, o que significa que *não há regras bem definidas* que devem ser seguidas.

Existe, no entanto, um conjunto de “boas práticas” que foram descobertas ao longo dos tempos, mas nenhum deles deve ser tomado como “absoluto”.

Os próximos *slides* enumeram algumas delas.

URLs - Endpoints

Endpoints são os “links” (URLs) para uma “função” (recurso) de uma API, combinados com o método HTTP (GET, PUT, POST, DELETE, PATCH) correspondente.

1. **Todos os endpoints devem ser escritos em minúsculas**. A razão disto é porque existe menos problemas relativamente a URLs que sejam *case-sensitive*.
2. **Deve-se evitar usar verbos nos URLs**, especialmente “get”, “create”, “update”, ou “delete”. Se se usa um verbo como estes, deve-se usar um dos métodos HTTP.
3. **Endpoints devem representar recursos**. Por *recurso*, é algo como “agente”, “piloto”, “categoria”, “utilizador”, etc. Se eu tenho um endpoint como “/agentes”, sei que provavelmente vai ser uma lista de agentes. “/agentes/5” um único. Como se fossem pastas e ficheiros.

afecarvalho@ipt.pt

Comparação

Evitar	Preferir
GET /agentes/getAgentes	GET /agentes
GET /agentes/getAgente/5	GET /agentes/5
PUT /agentes/updateAgente/5	PUT /agentes/5
DELETE /agentes/deleteAgente/5	DELETE /agentes/5
POST /agentes/createAgente	POST /agentes
GET /agentes/getMultas/5	GET /agentes/5/multas
POST /agentes/addMulta/5	POST /agentes/5/multas

afecarvalho@ipt.pt

Endpoints - Métodos HTTP (GET)

O GET só se deve usar para **leituras** de dados.

1. Se eu estou a *obter* dados, estou à espera que o dado seja o mesmo (a não ser que seja alterado externamente). **Múltiplos GETs a um endpoint devem devolver o mesmo dado** e não devem causar alterações ao recurso (alterações como "registos de acessos" são válidas, por exemplo).
2. **Deve-se evitar usar o GET para fazer atualizações a recursos**, pela razão explicada acima.
3. Por favor, **NUNCA se usa o GET para eliminar dados**! (E sim, já vi isto a ser feito uma vez).

afecarvalho@ipt.pt

Endpoints - Métodos HTTP (POST vs. PUT)

O POST e o PUT são um bocado ambíguos. **Regra geral, o POST cria, o PUT atualiza.**

Na especificação do HTTP, o PUT é indicado como sendo "idempotente": **múltiplas chamadas ao endpoint com os mesmos dados resultam no mesmo resultado** (como múltiplos "save" de um ficheiro com o mesmo conteúdo, só atualizam realmente a data do ficheiro).

Por esta razão, **o PUT só deve ser usado quando é *seguro* eu chamar o mesmo método múltiplas vezes com os mesmos dados**. Não é seguro, por exemplo, se a alteração resulta sempre em objetos diferentes (ex: incrementar um contador).

Para tudo o resto, deve-se usar o POST (até para atualizações).

afecarvalho@ipt.pt

Endpoints - Métodos HTTP (DELETE)

O DELETE, tal como o nome implica, **só serve para apagar (ou "desativar") dados**.

É de esperar que, assim que invocar um endpoint que usa DELETE, que futuras invocações dêem erro (ex: 404 Not Found, 409 Conflict), já que o recurso não existe, ou foi "desativado".

É também de esperar que no GET do recurso associado, o resultado seja um 404 Not Found (por exemplo), ou um registo alterado de tal forma a que indique que esteja "desativado".

Mais uma vez (e por questões de segurança) não se usa o GET para apagar dados!

Status Codes

O HTTP define vários status codes. Regra geral, usamos os códigos entre o 200 e o 599.

- 200 - 299 - Resultado com sucesso
- 300 - 399 - Redirects, ou resultados provenientes de *caches*
- 400 - 499 - Erros resultantes de problemas com o pedido
- 500 - 599 - Erros para quando a coisa corre mesmo mal

Apesar de existirem muitos, só precisamos de saber cerca de 10 deles: 200, 201, 204, 400, 401, 403, 404, 409, 500.

afecarvalho@ipt.pt

Status Codes - Casos de “Sucesso”

Quero...	Então uso o status code...
Indicar que a operação correu bem, e vão ser disponibilizados dados.	200 (OK), com os dados no corpo da resposta.
Indicar que foram criados dados (regra geral, como resposta a um POST), como a “criação de um agente”.	201 (Created), e indicar no cabeçalho “Location” o URL para o novo recurso. 200 (OK) também é aceitável. Opcionalmente, deve-se disponibilizar o recurso criado no corpo da resposta.
Indicar que a **eliminação** correu bem.	204 (No Content). 200 (OK) também é aceitável.
Indicar que a **atualização** correu bem.	200 (OK), com os dados atualizados. 204 (No Content) também é aceitável, caso não se devolvam dados.

afecarvalho@ipt.pt

Status Codes - Casos de “Insucesso”

Quero...	Então uso o status code...
Indicar que existem problemas de validação nos dados recebidos (pelo servidor).	400 (Bad Request) e indicar o que está errado, se possível.
Indicar que o cliente precisa de se autenticar (login) para usar o recurso.	401 (Unauthorized).
Indicar que o cliente **autenticado não pode executar a operação.**	403 (Forbidden). Se possível (e seguro), indicar o porquê.
Indicar que o recurso solicitado **não existe.**	404 (Not Found).
Indicar que o pedido não pode continuar por conflitos (ex: problemas de chaves, duplicações).	409 (Conflict). Se possível, indicar o porquê.

afecarvalho@ipt.pt

Status Codes - Casos de “Erro”

Quero...	Então uso o status code...
Indicar que “oh, f***-**, isto correu mesmo mal!”	500 (Internal Server Error). Só se usa isto se o pedido do cliente aparenta estar correto. Não se deve indicar o porquê do erro (geralmente porque é um problema no nosso código…). Este é o erro mais geral. Crashes, timeouts com bases de dados, falta de espaço em disco, problemas de configuração ou falta de dados na BD, quedas de meteoritos, … a lista continua.

O que colocar nas respostas - menos é mais

Não devemos devolver informação que não é necessária nos nossos endpoints. Isto vai **melhorar o desempenho do nosso servidor** (especialmente se existirem muitas entidades relacionadas), e também vai **reduzir o tamanho do JSON** -> menos tempo de transferência, menos memória, menos processamento -> melhor desempenho.

Isto aplica-se especialmente nos endpoints que “listam dados”: Como estes endpoints devolvem dados de várias entidades, se colocarmos dados de entidades relacionadas, teremos que fazer mais queries à BD, e o JSON ficará maior.

Se possível, só se deve colocar “entidades relacionadas”, especialmente se forem múltiplas (“1-N” ou “M-N”), em endpoints que devolvem 1 entidade.

afecarvalho@ipt.pt

Exemplos de endpoints

Quero...	Então faria...
Adicionar um agente	POST /api/agentes Dados do agente no corpo do pedido.
Obter uma lista de agentes	GET /api/agentes
Obter um agente, pelo seu ID (ex: 5)	GET /api/agentes/5
Obter as multas de um agente pelo seu ID (ex: 5)	GET /api/agentes/5/multas GET /api/multas?idAgente=5
Atualizar um agente pelo seu ID (ex: 5)	PUT /api/agentes/5 Dados do agente no corpo do pedido.

afecarvalho@ipt.pt

Exemplos de endpoints

Quero...	Então faria...
Atualizar apenas o stock de um produto pelo seu ID (ex: 5)	PUT /api/produtos/5/stock Dados sobre o novo stock no corpo do pedido.
Apagar um agente pelo seu ID (ex: 5)	DELETE /api/agentes/5
Marcar uma mensagem como lida (ou não) através do seu ID (ex: 5)	PUT /api/messages/5/read Dados sobre o estado de leitura no corpo do pedido.
Executar uma ação (ex: aprovar, rejeitar) num processo através do seu ID (ex: 5)	POST /api/processes/5/actions Dados sobre a ação (aprovar, rejeitar, delegar, etc.) no corpo do pedido. Usa-se POST porque o pedido vai avançar de estado de cada vez que se chama.

afecarvalho@ipt.pt

Exemplos de endpoints

Quero...	Então faria...
Fazer uma pesquisa simples de agentes.	GET /api/agentes Parâmetros de pesquisa colocados na query string do URL
Fazer uma pesquisa “complexa” (ex: muitos parâmetros, valores complexos, etc.)	POST /api/agentes/search Parâmetros de pesquisa colocados no corpo do pedido (Nota: isto vai contra os guidelines, mas guidelines não são leis)

afecarvalho@ipt.pt

Tratamento de erros

Quando “tudo corre mesmo mal”...

Quando as coisas correm mal, em linguagens como Java ou .NET, é gerado um “stack trace” com informação do erro, e onde ocorreu o erro. Esta informação é útil para corrigir o problema. No entanto...

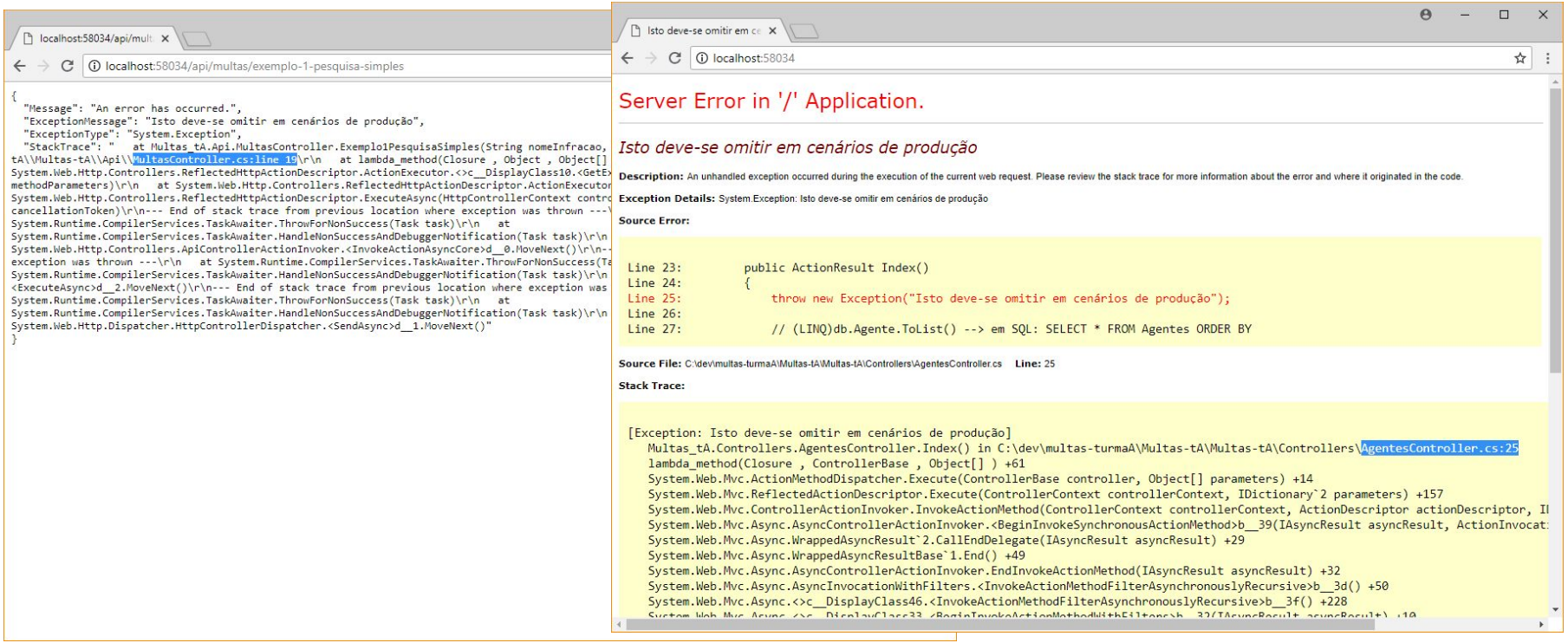
Em cenários de produção, **é má ideia expor detalhes técnicos sobre erros** “inesperados” da nossa aplicação.

A razão para isto é que **os “stack traces” têm informação sobre o código fonte das nossas aplicações**.

Esta informação pode ser usada por atacantes para descobrir vectores de ataque.

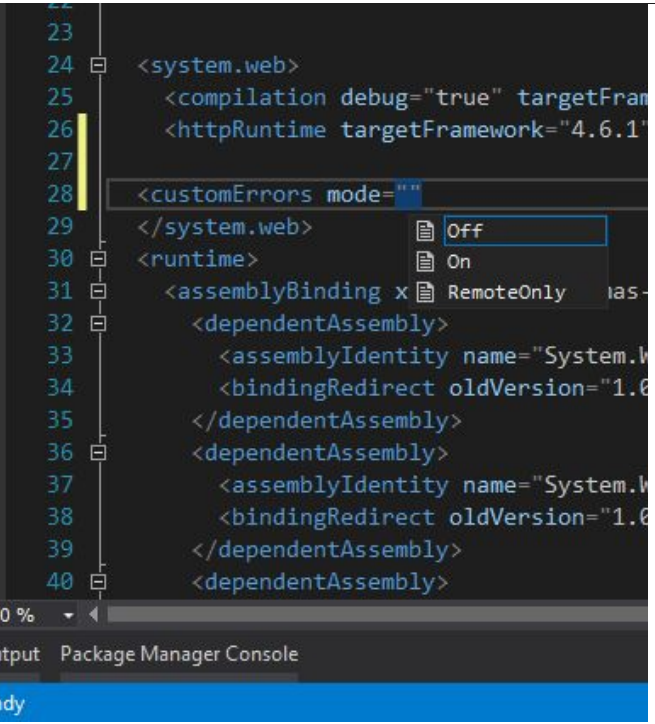
afecarvalho@ipt.pt

O que se pretende evitar é isto:



afecarvalho@ipt.pt

Solução - Em ASP.NET



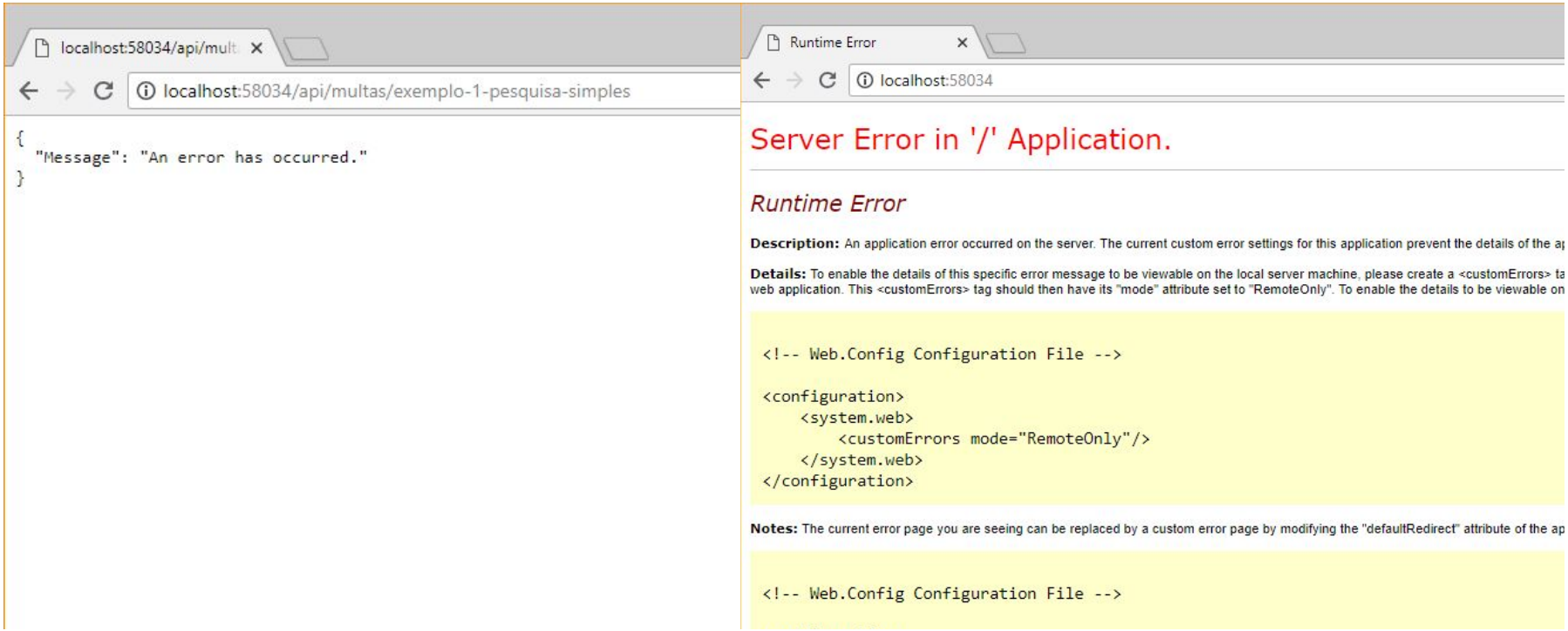
No ficheiro “Web.config”, existe uma secção chamada “system.web”.

Aqui, podemos adicionar um elemento XML chamado “customErrors”. O “mode” do “customErrors” aceita:

- **Off** - Os detalhes dos erros **são sempre mostrados**. **(Default)**
- **On** - Os detalhes dos erros **são sempre ocultos**.
- **RemoteOnly** - Os detalhes dos erros **só aparecem se o site for acedido em “localhost”.**

afecarvalho@ipt.pt

<customErrors mode=”On” />

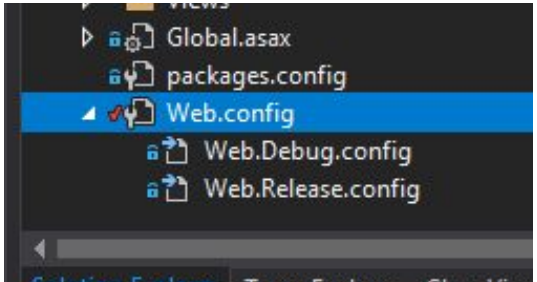


afecarvalho@ipt.pt

Ter o melhor de dois mundos...

Quando estamos a desenvolver, é útil termos acesso aos erros da nossa aplicação. No entanto, desativar os relatórios de erro é uma boa ideia em cenários de produção. Para evitar “esquecimentos”, pode-se fazer uma de duas coisas:

1. **Usar o modo “RemoteOnly” do “customErrors”** - Na maior parte dos casos, o site não é visualizado no mesmo servidor de onde é servido, logo é como se fosse “On”. Em desenvolvimento, é como se estivesse desligado.
2. **Usar o “Web.Release.config”** para mudar a definição do “customErrors” quando se publica o site (os ficheiros do projeto explicam como).



Web.(Debug / Release).config

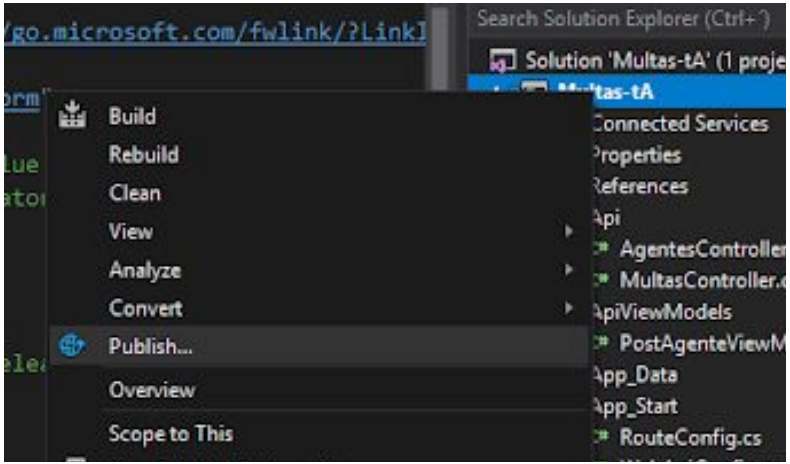
Estes ficheiros **servem como overrides** para quando se quer publicar um site num servidor. **Regra geral, usa-se o "Release", a não ser que se queira ter informação de debug"** (ex: para despistar algum problema em produção).

Podem (e devem) ser usados para **mudar todo o tipo de configurações**, incluindo as **connection strings das bases de dados,** como fazer o **tratamento de erros**, entre outras coisas.

Quando se usa o "Publish…" do projeto, o Visual Studio vai aplicar os ficheiros com base no "Publish Profile" (default: "Release").

afecarvalho@ipt.pt

Publish (disponível como right-click no projeto)



afecarvalho@ipt.pt

Erros “Funcionais”

Erros “Funcionais”

Este tipo de erros está relacionado com a lógica da nossa aplicação. São situações que “a lógica do negócio” indica como sendo casos de erro:

- Registos inexistentes na base de dados (ex: um agente com um determinado ID não existe)
- Erros de validação (ex: campos obrigatórios, formatos)
- Potenciais conflitos que poderiam ser gerados (ex: é um requisito que não se podem criar 2 agentes com o mesmo nome)
- Problemas de autorização (ex: não se pode aceder a multas que foram “passadas” por outros agentes).

Como tratar erros “funcionais”?

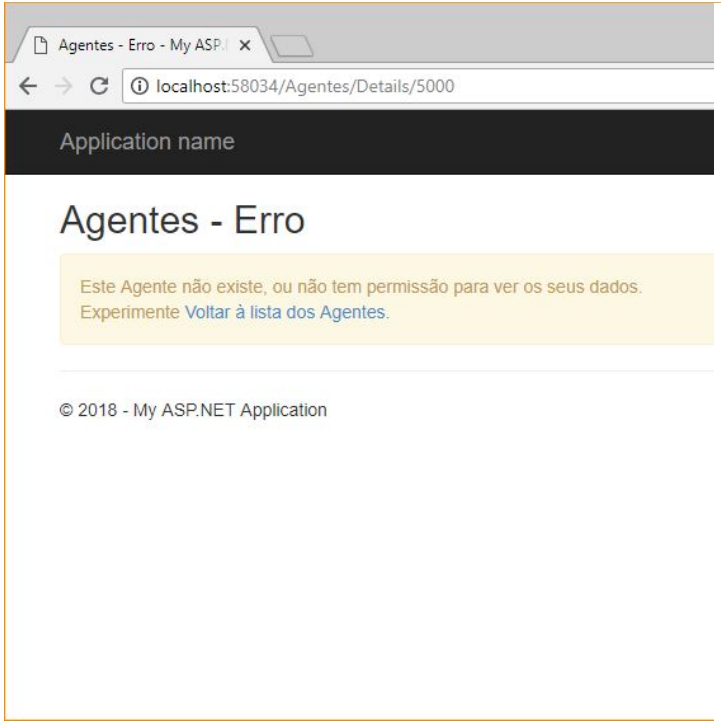
Isto não é “preto no branco” - depende de quem faz os requisitos e de quem desenvolve a aplicação. **O mais importante é não confundir o utilizador!**

Deve-se **mostrar informação suficiente (se possível) para indicar o que correu mal** (e se possível, como corrigir), **mas não tanto que possa ser um “buraco de segurança”**.

Na minha opinião, e pela minha experiência, omitir erros (ex: fazer o redirect para outra página quando alguma coisa não existe), vai confundir o utilizador (especialmente se ele segue um link proveniente de um e-mail ou site de terceiros)...

É mais correto mostrar uma página adequada com alguma informação “amigável”.

Um erro “agradável”



Eis um exemplo mais correto (na minha opinião) de tratar de um erro quando um Agente não existe:

- **Informamos o problema, de forma clara**, mas ao mesmo tempo introduzindo alguma ambiguidade (por questões de segurança, alguém pode estar a tentar extrair dados que não deve ter acesso...)
- **Disponibilizo uma forma de “voltar a um sítio seguro”** (a lista dos Agentes).

afecarvalho@ipt.pt

Questões de segurança

Questões de segurança

Tudo o que está nos slides seguintes é meramente bom senso.

Infelizmente, ainda se encontram demasiadas aplicações web (e ainda por cima, novas) que são vulneráveis a este tipo de ataques…

Felizmente, as ferramentas modernas (como ASP.NET MVC) são mais seguras *by default*, e é muito mais difícil "abrir-se buracos" quando eles não existem.

Isto significa que, para muitos dos casos seguintes, **não há desculpa para fazer as coisas mal**.

Anyway…

afecarvalho@ipt.pt

0. NUNCA confiar no utilizador (e nos seus dados)

Seja por estupidez, seja por más intenções, **os utilizadores são a causa de 99.5%(*) dos problemas nas aplicações**, sejam elas web, móveis, desktop, etc..

Por esta razão, **NUNCA, mas NUNCA, se confia nos dados que se recebem**, independentemente de como os recebemos (web services, formulários, ficheiros, etc.)

O uso do **"if (ModelState.IsValid)" em MVC é um começo, mas o input pode ser válido no seu formato, e mesmo assim, ser malicioso**.

Aplicar "regras de negócio", rotinas de "sanitização de inputs" e regras de controlo de acessos é essencial para mitigar este tipo de problemas.

1. “Reduzam” o volume de dados que recolhem

Não estou a falar relativamente ao GDPR (mas também se aplica).

Se têm um formulário que se dedica a fazer uma alteração a parte de um objeto (ex: alterar a quantidade de um produto em stock), não faz sentido estar a receber a informação toda sobre o produto (só o ID do produto, e o novo nº de itens em stock).

Isto ajuda a prevenir ataques de “overposting”, em que um atacante pode alterar dados “que não era suposto serem alterados” ao modificar o formulário de forma a inserir esses dados.

A solução para isto é, mais uma vez, usar View Models, só com os dados necessários, e fazer o update manualmente num objeto que veio da base de dados, após validação.

afecarvalho@ipt.pt

2. Cuidado com o XSS

HTML tem um problema: é tudo texto. Se o texto que recolhemos das caixas de texto não for corretamente tratado, arriscamo-nos a abrir o nosso site a ataques de XSS (Cross Site Scripting).

Estes ataques acontecem quando alguém escreve HTML (como uma tag <script>) dentro de uma caixa de texto, e:

- O texto não é “sanitizado” no servidor, antes do guardar na BD
- Não fazemos o HTML encode do texto, quando o lemos da BD para a View

O que acontece é que o HTML (e <script>) é interpretado pelo browser “vítima”. A partir daqui, o atacante tem controlo total sobre a página e os seus conteúdos.

XSS e ASP.NET MVC

Felizmente, o ASP.NET MVC tem alguns mecanismos que nos ajudam:

- **Inputs suspeitos como <script> são imediatamente barrados** (a não ser que se use o [ValidateInput(false)] nos controllers e/ou action methods.
- **Todo o texto que se coloca numa View** (ex: Html.DisplayFor, uso de variáveis, etc.) **é automaticamente codificado de forma segura** (<script> -> <script>), forçando o browser a interpretá-lo como texto.
- Só quando se usa a função @Html.Raw, é que o texto não é codificado quando é enviado para a view (usar, portanto, com EXTREMO cuidado).

afecarvalho@ipt.pt

XSS e JavaScript (ou, mais um prego no innerHTML)

Existe mais uma vantagem para usar o “textContent” ou “innerText” quando se usa JavaScript: O texto é automaticamente codificado de forma segura, mitigando assim os ataques de XSS quando o texto é “injetado” via JavaScript e DOM.

O “innerHTML” trata o texto “as-is”, o que significa que sites que usam “innerHTML” podem ser vulneráveis a ataques de XSS, se o texto não for validado antes de se colocar nos elementos do DOM.

O “innerHTML” só se deve usar quando temos controlo TOTAL sobre o que vamos colocar na página.

afecarvalho@ipt.pt

XSS e Frameworks de Front-End (React, Angular, Vue…)

Todas estas 3 frameworks fazem HTML-encode de todos os dados *by default*.

É difícil (ou pelo menos, chato, e o código resultante “salta à vista”) quando queremos usar HTML puro.

React, por exemplo, usa uma funcionalidade chamada “**dangerouslySetInnerHTML**” (o nome diz tudo…).

afecarvalho@ipt.pt

3. Formulários submetidos por terceiros (XSRF)

Só porque não posso aceder a um formulário, não significa que não posso submetê-lo. Basta encontrar alguém que possa fazê-lo por mim, e forjar uma página que seja carregada por alguém que possa aceder ao formulário (e sem que ela saiba).

Este tipo de ataque é conhecido como “Cross-Site Request Forgery”, ou XSRF. Este ataque funciona devido à forma como os browsers enviam cookies quando se submete um formulário.

Quando submeto um formulário para o site “X”, os cookies do site “X” (incluíndo os de autenticação) também são enviados, mesmo que o formulário esteja no site “Y”.

XSRF e ASP.NET MVC

Felizmente, o ASP.NET MVC tem proteção contra XSRF “out of the box”, e até a aplica nos seus templates:

- Nos **formulários** (de POST), adicionar um “**@Html.AntiForgeryToken()**”
- Nos **action methods dos controllers** (em POST), adicionar um “**[ValidateAntiForgeryToken]**”

A combinação destes dois passos num formulário protege-o deste ataque, porque adicionam alguma informação “random” que só pode ser validada pelo servidor.

Nota: Isto tem que ser adicionado a todos os formulários para que eles estejam seguros!

XSRF e ASP.NET MVC

A proteção é feita da seguinte forma:

1. Um cookie do tipo “HttpOnly” que não é acessível por JavaScript, com um token “random”
2. Um campo no formulário, com um valor “random”, e relacionado com o cookie (igual, um par de chaves, etc.)

Se um destes estiver em falta, ou não estiverem correctamente relacionados, ou forem “para outros utilizadores”, ou não tiverem um valor válido, o servidor rejeita o pedido automaticamente (isto é, nem sequer chega ao Controller).

Exemplo

https://app.pluralsight.com/player?name=mvc4-building-m7-security&mode=live&clip=8&course=mvc4-building

4. SQL Injection

SQL tem um problema, parecido com o HTML: As queries são texto, e é fácil manipular o input de forma a fazer com que a base de dados execute comandos “que não estávamos à espera”. Isto acontece quando:

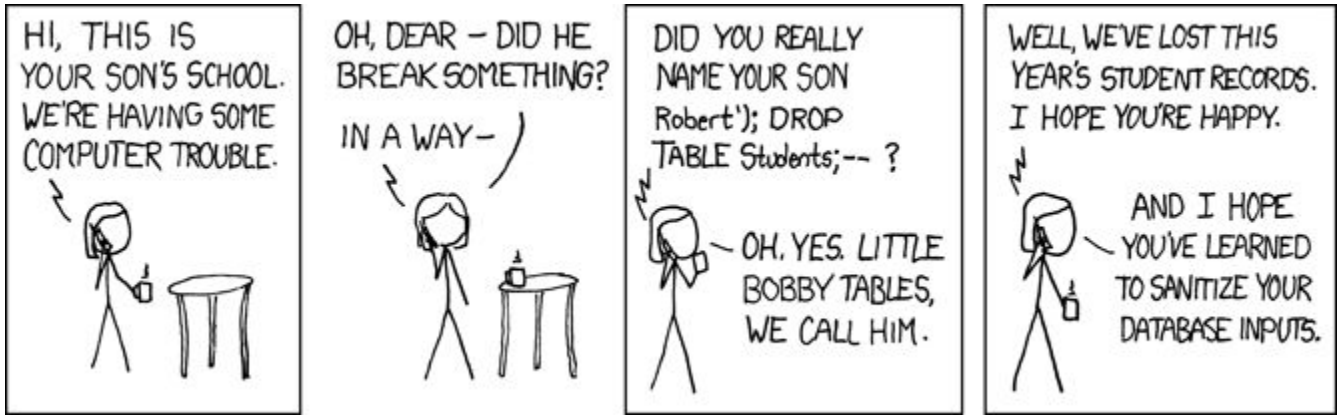
- Não se validam os inputs.
- Não se faz uso de “parameters” quando se está a executar comandos SQL numa base de dados. (Em vez disso, geralmente, concatenam-se variáveis…)

A Entity Framework trata disto por nós, porque usa parameters no SQL gerado.

Em suma: NUNCA CONCATENAR VARIÁVEIS COM SQL! SE O FIZEREM, ESTÃO CHUMBADOS/DESPEDIDOS!

afecarvalho@ipt.pt

SQL Injection



O problema é o **'); DROP TABLE Students; --**

Da forma como é concatenado no SQL, a **'** fecha a já existente, o que permite executar comandos arbitrários. O resto, é história...

https://xkcd.com/327/

afecarvalho@ipt.pt

5. Exposição de dados sensíveis

Isto aplica-se principalmente nas APIs, mas aplica-se também nas Views normais.

Às vezes, não queremos expor toda a informação da BD nos nossos outputs (JSON, XML, HTML). Isto pode depender de vários critérios: alguns utilizadores podem não ter acesso à informação, outros podem ter acesso a uma versão “censurada”, etc.

Por esta razão, **não se deve usar as classes da BD diretamente nos outputs** (elas expõem tudo). Mais uma vez, View Models, a aplicação de Linq, e o “if” são úteis para censurar os dados sensíveis no servidor.

Muitas fugas de dados são concretizadas porque os outputs não são validados...

6. Determinar “quem é” quando se “atualiza o próprio”

Estou a falar de exemplos como “atualizar o próprio perfil”, e distingo-o aqui porque já vi casos disto a ser mal feito… e com efeitos catastróficos.

Se o objetivo de um formulário/endpoint API é atualizar a informação do utilizador que está a usar a plataforma, então só se deve olhar para as credenciais (User ID), disponibilizados pela framework de autenticação (ex: ASP.NET Web API).

NUNCA, mas NUNCA, se deve ter um parâmetro a indicar o User ID, especialmente se não for validado, nestes cenários!

A razão para isto é porque já encontrei sites que não validam “quem eu sou”, e permitem que eu atualize dados “em nome de outra pessoa” (isto não é XSRF).

afecarvalho@ipt.pt

Determinar “quem é” o user - ASP.NET

Nos controllers MVC e Web API, e nas Views Razor, temos acesso à variável “User”. Lá dentro, temos informação como:

- User.Identity.Name - Indica o login do user (campo do UserName na BD)
- User.Identity.IsAuthenticated - Indica se o user está autenticado (pode ser anónimo)
- User.IsInRole() - Indica se o user está na role especificada

Com estas informações, é possível fazer queries à base de dados para obter o ID do user que está a usar os serviços, com Linq, com com os objetos do ASP.NET Identity.

Não há razão para não os usar, e fazer asneira como atualizar o perfil fiscal de outros, sabendo o email!

7. Não confiar no cliente - Parte 2

ASP.NET MVC dá a possibilidade de, com jQuery, adicionar validações *client-side* que impedem que o formulário seja submetido se não estiver válido.

E isso é útil, porque ajuda o utilizador a corrigir problemas rapidamente, e ajuda o servidor, porque "liberta carga" com pedidos errados.

No entanto, **eu posso desativar o JavaScript**, ou posso **forjar um pedido com dados inválidos**.

Por esta razão, **deve-se validar sempre no servidor! Validar no cliente é meramente por conveniência, no servidor é por sobrevivência!**

8. Passwords

Se possível, não guardem passwords. Existem outras formas de identificar um utilizador:

- Em grandes empresas, existem servidores como Active Directory, que permitem autenticar utilizadores. O .NET tem objetos para autenticar com ADs.
- Existe possibilidade de autenticar utilizadores por OAuth, e empresas como o Google, Facebook, Twitter, Microsoft permitem fazer isso.
- O ASP.NET Identity tem packages no Nuget com objetos que permitem autenticar utilizadores nestes web sites, e receber informação sobre qual foi o utilizador que se autenticou (como o e-mail, o nome, entre outras coisas).
- Existem às vezes também mecanismos *custom* de SSO (Single Sign On) nas empresas.

afecarvalho@ipt.pt

8. Passwords

Se tiverem que guardar palavras-passe… e se alguma vez a BD for atacada...

- **NUNCA GUARDEM PALAVRAS-PASSE EM TEXTO CLARO!** São facilmente roubadas.
- **NUNCA GUARDEM PALAVRAS-PASSE ENCRIPTADAS!** Se tiver acesso à chave de desencriptação, tenho acesso às passwords.
- **NUNCA GUARDEM HASHES SIMPLES (MD5, SHA-1, SHA-256)!** São vulneráveis a "ataques de dicionário".
- Só devem guardar hashes provenientes de algoritmos criados para tal: um algoritmo comum é o **Bcrypt**, que gera um hash que é o resultado da password e de uma string random, aplicada várias vezes (mais detalhes em GSRI e Sistemas Distribuídos).

O ASP.NET Identity faz isto corretamente por vocês. Não inventem.

afecarvalho@ipt.pt

9. Regras de passwords

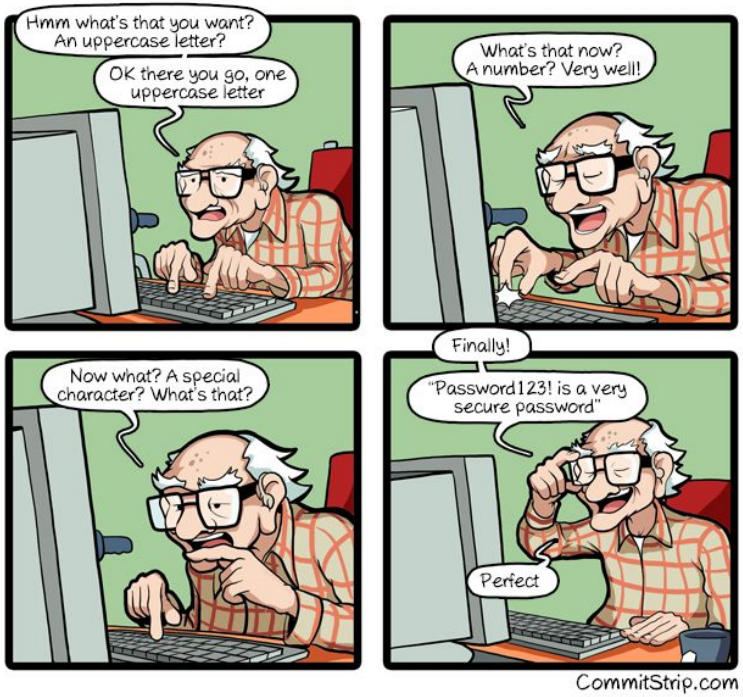
Evitem criar regras demasiado complexas para passwords. Parece contraproducente, mas é mais provável que as pessoas usem as mesmas passwords se se impõe demasiadas regras.

Se possível, evitem também ter “expiração” nas passwords.

Deve-se encorajar o uso de “password managers”--não impeçam o “copy-paste” no campo da password.

Uma password grande (ex: uma frase) é muito mais segura do que um conjunto de 8 caracteres com milhares de regras.

afecarvalho@ipt.pt



http://www.commitstrip.com/en/2018/04/27/security-security-security/

Have I Been Pwned

Este site, criado por Troy Hunt, contém informação (anonimizada, quando possível) sobre passwords e web sites que foram atacados e cujos dados foram expostos.

Está disponível em: https://haveibeenpwned.com/

Mais recentemente, também disponibiliza uma API que indica se uma password é vulnerável (ou foi usada) ou não.

Já existem web sites a usar isto, em vez de regras de passwords.

A API das passwords está disponível em: https://haveibeenpwned.com/API/v2

afecarvalho@ipt.pt

OWASP

OWASP

OWASP é um projeto sem fins lucrativos que tem como objetivo indicar as melhores práticas de segurança no desenvolvimento de aplicações web.

É maioritariamente conhecido pelo "OWASP Top 10", uma lista com os 10 tipos de problemas de segurança mais comuns hoje em dia nas aplicações web.

Muitos dos erros discutidos nos slides anteriores constam nesta lista.

O relatório de 2017 está disponível em formato PDF, e mostra a facilidade, vulnerabilidades e "impacto para o negócio" em cada uma delas.

Além disto, tem tutoriais para várias linguagens sobre como mitigar estes problemas.

A1 :2017 Injection

App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could

References

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection

afecarvalho@ipt.pt

E muito mais...

https://www.owasp.org/index.php/Main_Page