INTEGRAÇÃO DE SISTEMAS LEGADOS

Prof. Rodrigo Cantú Polo

CONVERSA INICIAL

Nesta aula, no primeiro tema, serão discutidos os fatores de sucesso de uma migração de dados de sistemas legados. No segundo tema, serão discutidos os riscos de uma migração de dados. O terceiro tema tratará da engenharia de *software* voltada a componentes. No quarto tema, será descrita a arquitetura orientada a serviços, enquanto que o quinto tema tratará da aplicação da arquitetura orientada a serviços a sistemas legados.

CONTEXTUALIZANDO

Na aula passada, foi iniciada a descrição do processo de migração de dados, que finaliza nesta aula. Fatores de sucesso de uma migração de dados foram apresentados, terminando na descrição de riscos a serem observados durante um processo de migração de dados.

Após a descrição dos conceitos de migração de dados, serão descritos conceitos de reaproveitamento de código com a engenharia de *software* voltada a componentes. Esse tipo de engenharia é descrito nesta aula com o objetivo de explicar o conceito de serviços *web*, que é a tecnologia mais utilizada atualmente para integração entre sistemas. Serviços *web*, especialmente os que são implementados com a tecnologia REST, são os mais utilizados também no desenvolvimento de aplicativos para *smartphones* (iOS e Android). Uma breve explicação sobre a aplicação dessa tecnologia na integração de sistemas legados é dada no tema final desta aula.

TEMA 1 – FATORES DE SUCESSO DE UMA MIGRAÇÃO DE DADOS

Apesar dos desafios presentes em um projeto de migração de dados, existem algumas boas práticas que diminuem a probabilidade de falha do projeto. Estas boas práticas são as que se seguem (Mendonça, 2009):

- A migração de dados deve ser vista como um projeto completo. Portanto, faz-se necessário alocar recursos, definir claramente o escopo do projeto, definir um plano de projeto com um cronograma que leva em consideração a possibilidade de ocorrência de problemas inesperados e angariar fundos necessários para a execução do projeto.

- Levar em consideração o tempo necessário para testar o resultado da migração e resolver eventuais problemas ao definir o cronograma do projeto.
- Utilizar ferramentas de *profiling* e auditoria para analisar completamente os dados, refinar o escopo do projeto e escrever as especificações de mapeamento com mais segurança. Entender que informação é importante e como deve ser usada é crítico para o sucesso da empreitada.
- Minimizar a quantidade de dados a serem migrados, quando possível.
- Testar o resultado da migração o quanto antes.
- Enquanto que muitas tarefas podem ser automatizadas, outras devem ser realizadas manualmente. Realizar essas tarefas com atenção ajuda a manter a consistência dos dados.

O maior desafio de um projeto de migração de dados é fazer com que o sistema-alvo entenda o que o sistema-fonte quer dizer. A seguir estão algumas boas práticas e problemas que devem ser evitados em um bom projeto de migração de dados:

- **Mapeamento consistente**: todos os campos de dados que serão migrados do sistema-fonte para o sistema-alvo devem ser definidos e examinados para assegurar consistência com o tamanho dos campos, tipos dos dados, valores de domínio permitidos, regras de sistema, verificações de integridade e qualquer outro problema possível. Um mapa de dados detalhado é crítico para entender para onde a informação está indo; da mesma forma, se existem obstáculos conhecidos ou evitáveis no caminho para o sistema-alvo. Um bom mapa de dados vai detalhar em profundidade a relação entre os campos do sistema-fonte com o sistema-alvo. Preferencialmente, deve incluir:
  - Nomes significativos dos campos origem e destino.
  - Tamanhos e tipos desses campos.
  - Qualquer lógica envolvida no mapeamento como tratamento de *strings* ou validações contrarregras de negócio.
- **Validação da extração**: É sabido que dados nos sistemas fonte geralmente contêm problemas ou escondem erros causados pelas mais variadas razões, desde falhas humanas até regras e validações mal testadas ou definidas em sistemas pouco sofisticados. Regras de validação devem ser utilizadas como primeiro passo para tentar identificar e corrigir esses problemas, estendendo o processo em tantas iterações

quantas forem necessárias. É comum que alguns tipos de erros não apareçam até que outros tipos sejam detectados e corrigidos. Mesmo que o sistema-fonte tenha ignorado discrepâncias, por exemplo, o mesmo cliente possuir endereços de cobrança diferentes armazenados em diferentes arquivos ou tabelas, o sistema-alvo potencialmente deve ter regras de negócio que definam qual endereço será considerado como o que vai ser carregado, eliminando essa inconsistência. Validação e limpeza de dados são essenciais e componentes-chave para um bom plano de migração de dados.

- **Qualidade da transformação**: Dados extraídos do sistema-fonte precisam ser transformados ou traduzidos num formato que o sistema-alvo possa importar e entender. Essas transformações não serão definidas somente no mapeamento dos dados, mas serão executadas sob funções de lógica de negócio que serão essenciais para carregarem estruturas de dados mais complexas. Existem ferramentas de ETL (acrônimo em inglês para “Extração, Transformação e Carga”) no mercado que podem ajudar bastante neste estágio (Mendonça, 2009).

Até agora foram discutidos aspectos técnicos da migração de dados. Entretanto, como acontece nos projetos de TIC, o “como” é tão importante quanto o “o quê”, o “onde” e o “quando”. Quando se trata dos aspectos gerenciais dos projetos de migração de dados, as seguintes preocupações devem ser consideradas (Mendonça, 2009):

- Abordagem *Big Bang* ou *Trickle*: Quando se decide migrar dados de um sistema para outro, faz mais sentido fazê-lo de uma só vez ou mover dados em fases controladas com múltiplas interações? Naturalmente existirão prós e contras em ambas as opções. Considerar qual abordagem será a mais adequada para determinada organização requer avaliar uma variedade de fatores. Alguns exemplos desses fatores são:
  - Qual a quantidade de dados que vai ser migrada?
  - Quanto tempo será necessário, levando em consideração as condições normais de processamento em produção da organização?
  - Quanto tempo a organização suporta parar os sistemas para a migração?
  - A organização tem condições de simular um *big bang* antes da migração definitiva?

– É possível estimar o ROI (acrônimo, em inglês, de “Retorno do Investimento”) de ambas as abordagens?

As respostas dessas questões devem ser encaminhadas antes de um simples objeto ser extraído ou uma simples transformação ser definida. Um projeto de migração de dados não pode prosseguir se estiver pobremente definido em seu escopo, agenda, estimativa de esforço, custos e recursos requeridos.

- ***Rollback***:
  – Quando se carrega dados no sistema-alvo, o que acontece se a migração de dados falhar?
  – A equipe está preparada para utilizar alguma funcionalidade de transação de *rollback* existente ou tem-se capacidade de desenhar e construir uma, caso não exista?
  – Como serão administradas as expectativas dos clientes caso isso aconteça?
  – Há um plano de mitigação dessas questões construído?
  – Essa possibilidade, de retornar, foi discutida com os usuários?

As respostas dessas questões proporcionam uma camada adicional de segurança e contribuem muito em termos de concluir o projeto no prazo, sem alongar o orçamento e gerenciando as expectativas do usuário.

- **Escalabilidade**: Naturalmente, quando alguém começa a falar de migração de dados, com dados críticos indo e vindo, e como isso pode incrementar o negócio da empresa, o assunto *escalabilidade* vem à tona. Como gerente, deve-se começar o projeto somente estando seguro de que se tem infraestrutura para suportar a sobrecarga de processamento provocada pela migração e um eventual crescimento não previsto dele.
- **Replicação**: A questão é a seguinte: o que acontece em caso de um desastre ou uma falha irrecuperável do sistema? Comumente essa preocupação aparecerá em um projeto de migração de dados. Tipicamente nasce da pressão colocada sobre o gerente de fazer uma migração correta e não colocar em risco 100% da produção. Migrar dados para um sistema espelho ao mesmo tempo em que se migra para o sistema-alvo deve ser seriamente considerado para adicionar uma camada a mais de segurança e assegurar que um plano de recuperação de desastre existe (Mendonça, 2009).

Migração de dados é um importante aspecto do esforço da maioria dos desenvolvimentos de *software*, mesmo que essa importância seja frequentemente subestimada ou inadvertidamente minimizada. Parece que, no extraordinário esforço de desenvolver *software*, realmente, a eventual medida de sucesso, de alguma forma, não depende de uma migração de dados precisa. A razão disso é bastante simples: optar por um novo sistema é uma decisão de negócio tendo sua própria prioridade. Entretanto, migrar dados históricos para fazer o novo sistema funcionar parece não ser a prioridade principal. Ao contrário, colocar o novo sistema em produção para suportar os processos críticos da empresa é a prioridade principal. Uma falsa impressão do sucesso de um projeto de migração de dados é quando se busca um simples movimento de dados sem remendos, permanecendo, assim, à sombra da implantação do novo sistema.

TEMA 2 – RISCOS DE UMA MIGRAÇÃO DE DADOS

Como em todo projeto de TIC, os riscos de um projeto de migração de dados devem ser identificados, mitigados e, se ocorrerem, planos de contingência devem ser colocados em prática. A seguir, conheça alguns riscos mais comumente identificados na maioria dos projetos de migração de dados (Mendonça, 2009):

- Falha ao tratar migração de dados como um projeto em si próprio: migração de dados é um empreendimento complexo que não deve ser considerado meramente um esforço periférico no desenvolvimento do projeto principal. O esforço de migração deve ser tratado como um completo subprojeto com um processo definido, uma cuidadosa estimativa de custo e tempo, e uma série de fases que possam ser rastreadas ou administradas.
- Subestimar o tempo e o custo da migração de dados: é importante realizar uma diligente pesquisa no sistema-fonte a fim de determinar a qualidade da documentação e da fonte dos dados. Se o sistema-fonte não possuir documentação de dados atualizada na forma de modelos de dados e dicionário de dados, a tarefa de determinar a estrutura e o tipo dos dados da fonte de dados desejada e o seu mapeamento para os dados-alvo deverá ser levada em consideração na estimativa de custo e tempo. Se o sistema-fonte é menos exigente nos requerimentos de

qualidade de dados que o sistema-alvo ou se a qualidade dos dados do sistema-fonte tem permitido falhas ao longo do tempo, tem-se que considerar um custo e tempo extras para a tarefa de limpeza dos dados na extração ou após a migração.

- Deficiência no estado final da qualidade dos dados: Se o esforço de migração não especificar formalmente o nível desejado da qualidade dos dados e um conjunto de testes de controle para verificá-los, o domínio-alvo pode ser carregado com dados de qualidade duvidosa. Isso vai impactar negativamente a percepção externa do esforço do desenvolvimento.
- Falha ao obter o suporte organizacional: quando a complexidade e importância da migração dos dados não são adequadamente apreciadas, fica difícil mobilizar o suporte organizacional para essa migração, principalmente em termos financeiros e de recursos. Pode-se até ter mais dificuldades de conseguir um bom nível de suporte organizacional se a equipe que mantém o sistema-fonte se sentir ameaçada pelo novo sistema, ou simplesmente porque o esforço de migração não é a prioridade principal da organização para aquele projeto (Mendonça, 2009).

TEMA 3 – ENGENHARIA DE *SOFTWARE* VOLTADA A COMPONENTES

Atualmente, muitos novos negócios são desenvolvidos com base em sistemas existentes. Porém, quando uma organização não pode utilizar um sistema existente, por não atender à sua demanda, um novo sistema deve ser desenvolvido. Para facilitar essa tarefa, a engenharia de *software* baseada em componentes pode ser utilizada (Sommerville, 2011).

A engenharia de *software* baseada em componentes surgiu no fim da década de 1990, como uma alternativa de engenharia de *software* baseada no reuso de componentes. Componentes são abstrações mais elevadas que objetos, sendo definidos pelas suas interfaces. São, em geral, mais abrangentes do que objetos, em que sua implementação não costuma ser visível para outros componentes. Consiste no processo de definir, implementar e integrar componentes independentes e com baixo acoplamento em sistemas de *software*. Se tornou uma abordagem importante de desenvolvimento de *software* porque sistemas de *software* estão se tornando cada vez maiores e complexos.

Clientes estão exigindo *softwares* que possam ser liberados e implantados cada vez mais rápidos. A única forma de lidar com a complexidade de forma efetiva e de liberar *software* de boa qualidade mais rapidamente é por meio do reuso de módulos de *software*, em vez de novas implementações.

Os fundamentos da engenharia de *software* baseada em componentes são:

- Componentes independentes, que são definidos por completo pelas suas interfaces. Deve haver uma separação clara entre a interface do componente e a sua implementação. Isso significa que a implementação de um componente pode ser substituída por outra, sem que o sistema que o utiliza seja afetado.
- Padrões de componentes que facilitam a integração de componentes. Esses padrões estão embutidos no modelo do componente. Eles definem, ao menos, como as interfaces do componente devem ser especificadas e como componentes se comunicam. Alguns modelos vão além, e especificam interfaces que devem ser implementadas por todos os componentes que suportam a especificação. Se os componentes seguem os padrões, então sua operação independe de linguagens de programação. Componentes desenvolvidos em linguagens diferentes podem ser integrados ao mesmo sistema.
- *Middleware*, que disponibiliza suporte para integração de componentes. Para fazer com que componentes independentes e distribuídos funcionem de forma integrada, é necessário o suporte de *middleware*, que suporta a comunicação entre componentes. *Middleware* para componentes suporta questões de baixo nível de forma eficiente, permitindo que desenvolvedores possam manter o foco em problemas relacionados à aplicação em desenvolvimento. Além disso, *middleware* para componentes pode disponibilizar suporte para alocação de recursos, gerenciamento de transações, segurança e concorrência.

O desenvolvimento voltado para componentes abrange boas práticas de engenharia de *software*. A arquitetura de sistemas que utiliza componentes possui uma qualidade melhor, mesmo que os componentes sejam desenvolvidos pela própria organização. Esse tipo de desenvolvimento possui bons princípios, que suportam a construção de *software* de fácil compreensão e manutenção:

- Componentes são independentes e não interferem no funcionamento de outros componentes. A sua implementação pode ser modificada sem afetar o resto do sistema.
- Componentes se comunicam por interfaces bem definidas. Se essas interfaces têm manutenção, um componente pode ser substituído por outro, que pode disponibilizar funcionalidades adicionais, ou melhores.
- As infraestruturas de componentes oferecem uma variedade de serviços padronizados, que podem ser utilizados em sistemas de aplicação. Isso reduz a quantidade de código novo para ser desenvolvido.

A motivação inicial da engenharia de *software* voltada a componentes foi a necessidade de suportar o reuso e a engenharia de *software* distribuída. Um componente era visto como um elemento de um sistema de software que poderia ser acessado utilizando um mecanismo de RPC (Remote Procedure Call) por outros componentes em execução em computadores separados. Cada sistema que reutilizava um componente tinha que incorporar a sua própria cópia do componente. Esta ideia de componentes estendia o conceito de objetos distribuídos, implementado por modelos como a especificação CORBA. Vários protocolos e padrões foram desenvolvidos para suportar esta visão de componentes, assim como Enterprise Java Bens (EJB) e os padrões COM e .NET da Microsoft.

Na prática, esses vários padrões impediram a popularização da engenharia de *software* voltada a componentes. Era impossível que componentes desenvolvidos por diferentes abordagens pudessem ser integrados. Componentes desenvolvidos por diferentes plataformas, como .NET e J2EE, são incompatíveis. Além disso, os padrões e protocolos propostos eram complexos e difíceis de compreender.

Como resposta a esses problemas, a noção de componente como um serviço foi desenvolvida, e padrões foram propostos para suportar a engenharia de *software* orientada a serviços. A principal diferença entre um componente utilizado como um serviço e a noção original de um componente é a de que serviços são entidades externas aos programas que os utilizam. Em um sistema orientado a serviços, os serviços a serem utilizados são referenciados em vez de serem copiados no computador cliente.

A engenharia de *software* orientada a serviços é um tipo de engenharia de *software* voltada a componentes. Essa metodologia utiliza uma noção mais simples de componentes, sendo direcionada por padrões. Em situações nas

quais a reutilização de sistemas existentes é impraticável, a engenharia de *software* orientada a serviços está se tornando a abordagem dominante em sistemas de negócio.

TEMA 4 – ARQUITETURA ORIENTADA A SERVIÇOS

A arquitetura orientada a serviços (SOA – Service Oriented Architecture) promove a transição de um sistema fechado, sem interoperabilidade, em um sistema orientado a serviços. A sua vantagem principal consiste em como a arquitetura de infraestrutura é feita quando serviços, ao invés de aplicações inteiras, são construídos. Serviços são pequenas unidades de *software* que disponibilizam uma funcionalidade específica a qual pode ser reutilizada em várias aplicações. SOA aplica um princípio de arquitetura de baixo acoplamento, que faz com que cada serviço seja uma entidade isolada com dependências limitadas com outros recursos compartilhados como bancos de dados, aplicações legadas ou APIs (Sommerville, 2011).

Sua implementação é feita por serviços *web*. Serviços *web* evoluíram de aplicações *web*. De fato, são uma simplificação de aplicações *web*: em vez de suportar respostas a requisições em que os dados são referentes à interface de usuário e aos dados, os serviços *web* servem apenas aos dados, em que a aplicação cliente deve se encarregar de apresentar os dados para o usuário.

A principal vantagem de SOA/serviços *web* é a de que o mesmo serviço pode ser utilizado por vários clientes. Os dados que foram originalmente feitos para que uma aplicação *web* pudesse utilizar podem ser utilizados, sem modificações, por outro tipo de cliente. Exemplos incluem aplicações *desktop*, que obtêm seus dados de um servidor sem precisar enviar consultas específicas de banco de dados (SQL) para o banco de dados.

O acesso a um sistema legado pode ser disponibilizado como um serviço *web*, e responder diretamente a requisições pelo protocolo HTTP, ou ser executado sob um *proxy* que traduz as requisições para a linguagem do sistema legado. As mensagens HTTP trocadas entre o serviço e os clientes são em texto simples, que qualquer sistema ou linguagem de programação pode produzir.

4.1 Tecnologias

Ao planejar um serviço *web*, deve-se definir um conjunto de regras utilizadas para a troca de informações. As principais tecnologias para isso são SOAP e REST.

SOAP (Simple Object Access Protocol) é uma tecnologia mais antiga, que suporta vários protocolos de transporte da internet, como HTTP, SMTP, entre outros. Os dados são transmitidos no padrão XML, o que pode causar problemas de desempenho, caso a quantidade de informações a serem transmitidas seja grande.

REST (Representational State Transfer) é uma tecnologia mais recente. Utiliza apenas o protocolo HTTP para transporte, mas suporta vários formatos diferentes para dados (XML, JSON, entre outros), no qual o padrão mais utilizado é o JSON (Javascript Object Notation). REST é uma alternativa com melhor desempenho do que SOAP pelo fato de não exigir uma implementação rígida, ter um grau de flexibilidade mais alto e por ser menos dependente de documentação (Serrano; Hernantes; Gallardo, 2015).

A vantagem principal dos serviços REST é a de que são mais simples e mais eficientes do que “grandes” serviços *web*. A quantidade de processamento é reduzida, o que é muito importante para serviços *web* simples que implementam funcionalidades simples.

SOAP e REST não são incompatíveis, e é possível que provedores de serviços *web* possam oferecer interfaces SOAP e REST. Grandes provedores como a Amazon oferecem as duas interfaces, e a experiência desses provedores é a de que REST é a opção preferida dos desenvolvedores. O quadro 1 descreve as diferenças entre as duas tecnologias (Sommerville, 2011).

Quadro 1 – Diferenças entre SOAP e REST

SOAP	REST
Protocolo baseado em XML.	Protocolo independente de conteúdo (XML, JSON etc.).
Utiliza WDSL para comunicação entre produtor e consumidor.	Utiliza XML ou JSON para enviar e receber dados.
Invoca serviços pela chamada de métodos RPC.	Simplesmente invoca serviços por caminhos de URL.
Não retorna resultados de forma inteligível para humanos.	Resultado é legível, que é apenas texto XML ou JSON.

(continua)

(Quadro 3 – Conclusão)

SOAP	REST
Transferência é feita sobre HTTP. Também utiliza outros protocolos, como SMTP, FTP etc.	Transferência ocorre apenas sobre HTTP.
SOAP pode ser chamado por Javascript, mas é difícil de implementar.	Fácil de realizar chamadas por Javascript.
Desempenho não é bom, comparado com REST.	Desempenho é bem melhor comparado com SOAP.

Fonte: Paul, 2015.

TEMA 5 – SERVIÇOS PARA SISTEMAS LEGADOS

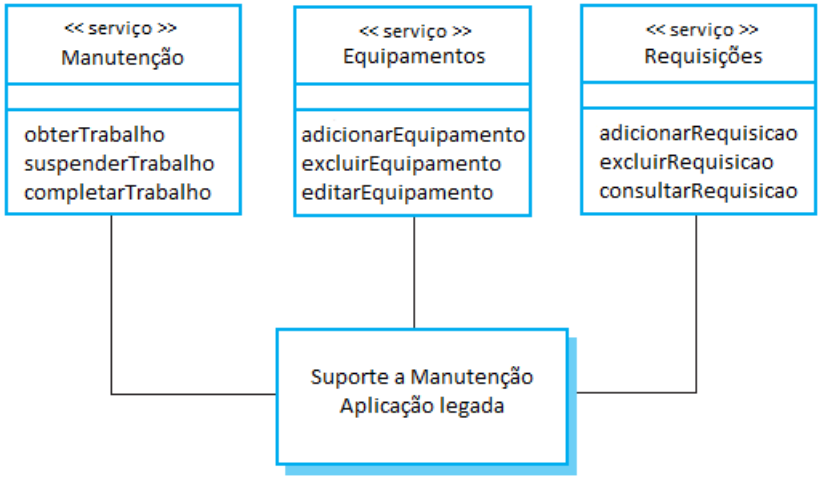
Sistemas legados são sistemas antigos em utilização por uma organização. Pode não ser vantajoso reescrever ou substituir esses sistemas, e uma alternativa de integrá-los a sistemas mais modernos pode ser considerada. Esses sistemas podem ser acessados pela internet, ou por uma intranet, e podem ser integrados com outros aplicativos.

Como exemplo, imagine que uma grande companhia mantém um inventário dos seus equipamentos e um banco de dados que guarda um registro de manutenção e reparos de equipamentos. Esse sistema mantém um controle de quais requisições de manutenção foram feitas para diferentes partes do equipamento, de qual manutenção regular está agendada, quando a última manutenção foi feita, quanto tempo foi gasto em manutenções etc. Esse sistema legado foi originalmente utilizado para gerar listas diárias de tarefas para equipes de manutenção, mas, com o passar do tempo, novas funcionalidades foram adicionadas ao sistema. Essas novas funcionalidades disponibilizam dados sobre quanto foi gasto em manutenção de cada parte dos equipamentos, assim como informações para ajudar na previsão de custos de trabalho a ser conduzido por equipes externas de manutenção. O sistema é executado como um sistema cliente servidor, com um cliente em execução em um PC.

A empresa agora deseja disponibilizar acesso em tempo real para esse sistema por meio de *tablets* Android, utilizados pela equipe de manutenção. Eles atualizarão o sistema diretamente com o tempo e recursos gastos em manutenção, e vão consultar o sistema para verificar o seu próximo trabalho de manutenção. Adicionalmente, a equipe de *call center* precisará acessar o sistema para registrar requisições de manutenção, assim como verificar o seu progresso.

É praticamente impossível de evoluir o sistema para suportar esses requisitos. Por esse motivo, a companhia decide adquirir novas aplicações para as equipes de manutenção e *call center,* as quais se baseiam no sistema legado, que deve ser utilizado como base para implementar vários serviços *web*. Esse cenário pode ser visto na figura 1.

Figura 1 – Serviços disponibilizando acesso a um sistema legado



Fonte: Sommerville, 2011.

Novas aplicações trocam mensagens com esses serviços *web* para acessar funcionalidades do sistema legado. Alguns dos serviços são os seguintes:

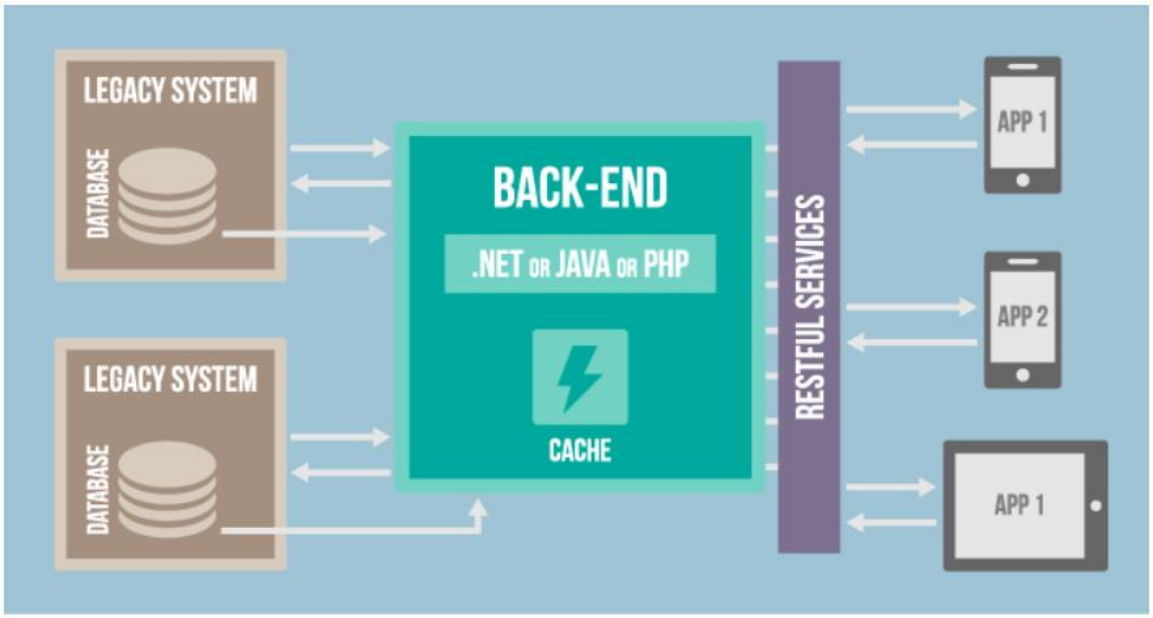
- **Serviço de manutenção**: inclui operações para obter uma tarefa de manutenção de acordo com o seu número de identificação, prioridade e localização geográfica, assim como realizar a transferência de detalhes do serviço de manutenção que foi realizado. O serviço também provê operações que permitem que um trabalho de manutenção que tenha começado, mas está incompleto, possa ser reiniciado.
- **Serviço de equipamentos**: inclui operações para adicionar e excluir novos equipamentos, além de modificar a informação associada com o equipamento no banco de dados.
- **Serviço de requisições**: tem operações para adicionar um novo pedido por serviços, excluir requisições de manutenção e consultar o estado de requisições pendentes.

Repare que o sistema legado existente não é representado como um serviço único. Em vez disso, os serviços que são desenvolvidos para acessar o

sistema legado suportam apenas uma área de funcionalidade do sistema legado. Isso reduz a complexidade desses serviços, facilitando a compreensão e a reutilização deles em outras aplicações.

Uma forma de implementá-los pode ser vista na figura 3. Um servidor intermediário poderia possibilitar a comunicação entre o sistema legado e as aplicações clientes de serviços *web*, ilustrados na figura 2 como dispositivos móveis. A vantagem desse tipo de abordagem seria a ausência de alterações no sistema legado para disponibilizar o acesso de novos tipos de clientes ao sistema.

Figura 2 – Modelo de interoperabilidade entre sistemas legados e dispositivos móveis



Fonte: Dzimchuk, 2015.

FINALIZANDO

Nesta aula, foi finalizada a descrição dos conceitos de migração de dados, falando sobre fatores de sucesso relacionados e sobre seus riscos. Depois, foi apresentada a engenharia de *software* voltada a componentes. Em seguida, a arquitetura orientada a serviços foi discutida, finalizando com aplicações de serviços *web* na integração de sistemas legados com aplicações externas.

REFERÊNCIAS

DZIMCHUK, A. **Integrating mobile enterprise applications with a Legacy System**: a “Can-do” attitude. Disponível em: <https://www.scnsoft.com/blog/integrating-mobile-enterprise-applications-with-a-legacy-system--a-cando-attitude>. Acesso em: 24 out. 2017.

MENDOÇA, M. H. R. de. **Metodologia de migração de dados em um contexto de migração de sistemas legados**. 2009, 151 f. Dissertação (Mestrado em Ciências da Computação) – Universidade Federal de Pernambuco, Recife, 2009. Disponível em: <http://repositorio.ufpe.br/bitstream/handle/123456789/1934/arquivo1908_1.pdf?sequence=1&isAllowed=y>. Acesso em: 24 out. 2017.

PAUL, J. **Difference between SOAP and RESTful Web Service in Java**. 2015. Disponível em: <http://javarevisited.blogspot.com.br/2015/08/difference-between-soap-and-restfull-webservice-java.html>. Acesso em: 24 out. 2017.

SERRANO, N; HERNANTES, J; GALLARDO, G. Service-Oriented Architecture and Legacy Systems. **InfoQ**, feb, 2015. Disponível em: <https://www.infoq.com/articles/service-oriented-architecture-and-legacy-systems>. Acesso em: 24 out. 2017.

SOMMERVILLE, I. **Engenharia de *software***. 9. ed. Addison-Wesley, 2011.