Otimização automática de aplicações web usando *templates client-side*

Sara Gonçalves[1], Hugo Lourenço[2], Sérgio Silva[2], e João Costa Seco[1*]

[1] CITI - Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
[2] OutSystems SA

Resumo O crescente poder computacional dos dispositivos móveis e a maior eficiência dos navegadores, fomentam a construção de aplicações web mais rápidas e fluídas, através da troca assíncrona de dados em vez de páginas `HTML` completas. A plataforma *OutSystems* é um ambiente de desenvolvimento usado para a construção rápida e validada de aplicações web, que integra numa só linguagem a construção de interfaces de utilizador, lógica da aplicação e modelo de dados. O modo normal de interação cliente-servidor da plataforma é consistente com o ciclo completo de pedido-resposta, embora seja possível implementar, de forma explícita, aplicações assíncronas.
Neste trabalho apresentamos um modelo de separação, baseado em análise estática do modelo de uma aplicação, entre os dados apresentados nas páginas geradas pela plataforma e o código correspondente à sua estrutura e apresentação. Esta abordagem permite a geração automática e transparente de interfaces de utilizador mais rápidas e fluídas em aplicações *OutSystems*.
O modelo apresentado, em conjunto com a análise estática, permite identificar o subconjunto mínimo dos dados a serem transmitidos entre cliente e servidor para a execução de uma funcionalidade no servidor, e isolar a execução de código no cliente. Como resultado da utilização desta abordagem obtém-se uma diminuição significativa na transmissão de dados.

1 Introdução

A disponibilização de serviços na Internet é tradicionalmente e preferencialmente feita através de páginas `HTML`, mais ou menos dinâmicas, e por pontos de interoperabilidade entre aplicações onde são transmitidos apenas dados. As boas práticas de desacoplamento de software [1,2] recomendam que se separe a apresentação, da lógica da aplicação, tornando os clientes, o mais possível, independentes da aplicação servidora que gere essencialmente dados de forma abstrata. Acrescem outros fatores, como o crescente poder computacional nos dispositivos móveis e a maior eficiência dos navegadores, que fomentam a construção de aplicações web mais rápidas e fluídas, através da troca assíncrona de dados em substituição da

* Trabalho suportado parcialmente por FCT Pest UI527 2011

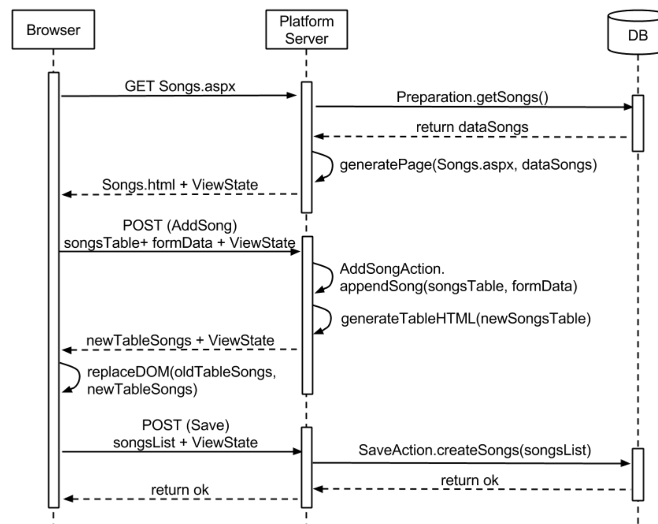

Figura 1: Comunicação cliente/servidor numa aplicação *OutSystems*

transmissão de páginas `HTML` completas, construídas inteiramente no servidor. Algumas aplicações de larga escala como a interface web do *Gmail*, as aplicações *Google Docs*, ou as interfaces para os serviços como o *LinkedIn* ou *Trello*, são exemplos de aplicações com requisitos fortes de usabilidade, manutenção de um contexto complexo e desacoplamento para proporcionar outro tipo de interfaces com o utilizador. Estas aplicações, executam grande parte da sua lógica no cliente, onde geram a interface `HTML` dinamicamente a partir de dados trocados com o servidor de forma assíncrona. Algumas plataformas de desenvolvimento atuais replicam o modelo MVC tradicionalmente usado para o servidor, no cliente. Alguns exemplos a assinalar são as plataformas `AngularJS`, `KnockoutJS` e `EmberJS`. Nestes contextos, as várias páginas `HTML` são obtidas por instanciação dos chamados *templates client-side*. Desta forma, obtém-se uma redução significativa do volume de dados transmitidos na rede, o que proporciona uma experiência de interação mais rápida e fluída.

A plataforma *OutSystems* é um ambiente de desenvolvimento usado para a construção rápida e validada de aplicações web, que integra numa só linguagem a construção de interfaces de utilizador, lógica da aplicação e modelo de dados. A plataforma suporta não só o desenvolvimento de aplicações, mas, talvez mais importante, a sua evolução assistida por mecanismos de validação. No contexto da plataforma, o modelo usual de interação cliente-servidor é baseado em ciclos completos de pedido-resposta, com manutenção do estado[3], embora seja possível implementar de forma explícita aplicações com comunicação assíncrona. Uma aplicação é definida por um conjunto de ecrãs, um conjunto de ações que são executadas no servidor, e um modelo de dados, gerido também pela plataforma.

[3] Mecanismo *View State* da plataforma `.NET`

# **Músicas**	**HTML** (KB)	**JSON** (KB)
50	4,8	0,35
100	6,0	0,55
200	7.6	0,98
300	9,2	1,23
400	10,8	1,50
500	12,3	1,82

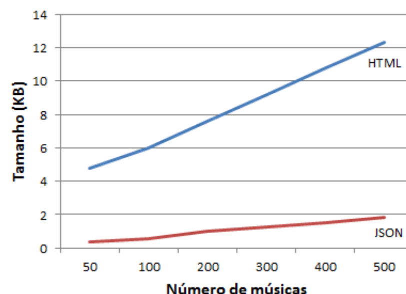

Figura 2: Tamanho relativo de páginas e os respetivos dados.

Nas aplicações geradas na plataforma *OutSystems*, cada ecrã é completamente instanciado no servidor, e todas as ações são também executadas no servidor, sendo que o estado da página é mantido explicitamente. Ou seja, são transportados todos os dados necessários na manutenção do estado atual do ecrã corrente, ao longo de sucessivos ciclos completos de pedido-resposta, como é ilustrado na Figura 1. Embora a passagem de contexto seja transparente para o programador, a fluidez padrão das aplicações não é total. A existência de uma representação abstrata única para uma aplicação como ponto central, resultado de uma linguagem de programação integrada, permite-nos explorar variações da arquitetura das aplicações geradas, no sentido de beneficiar de um maior desacoplamento e fornecer aplicações com comunicação assíncrona, sem causar impacto significativo no processo de desenvolvimento.

Como ponto de partida para a motivação deste trabalho, consideremos o exemplo de uma pequena aplicação web desenvolvida na plataforma *OutSystems*, contendo um ecrã que lista informação sobre um conjunto de músicas. Quando é registado um pedido do navegador, é gerada no servidor uma página `HTML`, através de um *template*, instanciado com os dados obtidos por uma consulta à base de dados. A página `HTML` contém informação relativa a dados e também à estrutura da página. Na Figura 2 é possível observar os tamanhos relativos dos dados e da página gerada. Note-se que se apresentam tamanhos comprimidos, uma vez que a maior parte das aplicações de referência o faz por omissão. Este gráfico permite concluir que o volume dos dados é muito menor em relação ao código `HTML` completo. Esta diferença representa um potencial de redução significativa, quer em termos de carga no servidor, aquando a geração das páginas, quer na quantidade de dados transmitidos. A exigência de mais poder computacional ao cliente, para a geração da página, é compensada com maior fluidez na aplicação, pois não existe mudança de contexto de página.

Neste trabalho apresentamos um modelo de separação, baseado em análise estática do modelo de uma aplicação, entre os dados apresentados nas páginas geradas pela plataforma e o código correspondente à sua estrutura e apresentação. Como método de prototipagem da abordagem aqui proposta, apresenta-se uma

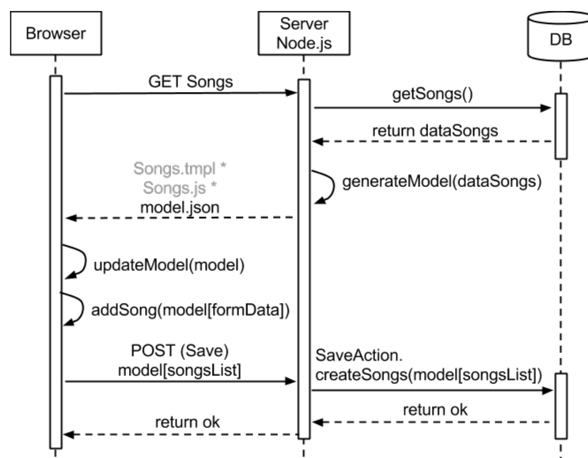

Figura 3: Comunicação cliente/servidor numa aplicação no modelo proposto
(*) enviado só no 1º pedido da página

linguagem de programação, inspirada na linguagem da plataforma *OutSystems*, e destinada à definição do modelo de uma aplicação. A partir deste modelo, define-se um gerador de código que produz aplicações web, onde é separada a produção de páginas `HTML`, da execução da lógica da aplicação, e é isolada e otimizada a transmissão de dados entre cliente e servidor. O código produzido é destinado a uma arquitetura MVC duplo (no cliente e no servidor), onde são usados *templates* no cliente para produzir dinamicamente o código `HTML` da página a partir dos dados transmitidos numa representação abstrata (`JSON`). O código que é gerado para ser executado no servidor aproxima-se a um serviço de dados, contendo parte da lógica da aplicação. Ao contrário do que acontece atualmente na plataforma *OutSystems*, é possível definir nesta linguagem, ações que são incorporadas no código do cliente, evitando assim pedidos desnecessários. A arquitetura aqui proposta é ilustrada na Figura 3.

Com o fim de otimizar a transmissão de dados entre cliente, onde existe um *template*, e o servidor onde são executadas algumas das ações, utiliza-se uma análise estática de fluxo de dados. Durante a fase de geração de código, a análise é aplicada sobre a definição do modelo de uma aplicação, gerando factos `Datalog`, que permite identificar o subconjunto mínimo de dados a transmitir na invocação de cada ação, e a ser atualizado no cliente após a execução no servidor. Para a ação a do ecrã s, composta pela instrução identificada por n, na qual ocorre a afetação $x = y$, são gerados os factos $\mathsf{action}(s, a)$, $\mathsf{node}(a, n)$, $\mathsf{use}(n, y)$ e $\mathsf{def}(n, x)$, onde x e y são variáveis do ecrã s.

A execução de uma ação pode causar a modificação de variáveis, das quais podem depender expressões utilizadas na instanciação do *template*. Como tal, no fim de qualquer ação essas expressões devem ser recalculadas e enviadas para o cliente, de forma a que a página seja atualizada com os novos valores. Uma das

análises aplicadas determina o conjunto de expressões que devem ser recalculadas, e consequentemente enviadas para o cliente. No modelo atual da *OutSystems* todas as expressões que constam dos ecrãs são calculadas no servidor. É importante manter esta semântica e, entre outras coisas, não comprometer a segurança da aplicação ao transmitir as subcomponentes das expressões. Por exemplo, num ecrã em que se testa se um utilizador está autenticado, só se pode enviar para o cliente a informação que decorre do resultado desse teste.

Como resultado deste trabalho obtemos a diminuição do consumo de largura de banda, o desenvolvimento transparente de uma aplicação com uma arquitetura MVC duplo, esperando-se também uma redução da carga no servidor ao gerar `HTML`. Uma vez que o modelo da linguagem proposto foi inspirado na linguagem *OutSystems*, a contribuição principal deste trabalho é a proposta de um novo modelo para simplificar a programação de aplicações assíncronas.

Nas secções seguintes apresenta-se uma breve contextualização sobre o que é a plataforma *OutSystems* (Secção 2), seguindo-se a apresentação do modelo proposto (Secção 3), as estruturas de dados subjacentes (Secção 3.1), o método de análise estática aplicada neste trabalho (Secção 4), descreve-se a sua implementação e método de validação (Secção 5). No final identificam-se alguns aspetos que se esperam ser os próximos passos.

2 Plataforma OutSystems

A plataforma *OutSystems* permite o desenvolvimento rápido de aplicações web. É formada por vários componentes, com destaque o ambiente de desenvolvimento *Service Studio* onde são construídas as aplicações utilizando uma linguagem visual, e o *Platform Server*, responsável pela compilação das aplicações, publicação e execução das versões das aplicações já publicadas. Numa breve contextualização da plataforma, descrevem-se de seguida alguns dos aspetos essenciais, relacionados com a construção e implementação dos ecrãs das aplicações.

O elemento mais visível de toda a plataforma é o *Service Studio*, que integra o desenvolvimento das camadas de processos, ecrãs (interface do utilizador), lógica funcional e dados. O desenho da interface corresponde à criação de ecrãs, representantes das páginas da aplicação. Um ecrã é construído por composição de componentes pré-construídos, chamados *widgets*, entre os quais estão literais e expressões, tabelas, blocos condicionais, e outros. A sua conjugação estrutura o conteúdo dinâmico de uma página.

Associada a cada ecrã existe uma ação especial, chamada de *Preparation*, onde os dados associados aos vários *widgets* são obtidos. É possível em cada ecrã definir variáveis locais, parâmetros ou levar a cabo a execução de consultas ao modelo de dados. A ação *Preparation* é executada antes da instanciação da página, sempre que esta é solicitada pelo cliente, ou entre pedidos que forcem a atualização de toda a página.

Também associado a cada ecrã, existe um conjunto de ações (do ecrã) que adicionam funcionalidade à aplicação, e são normalmente usadas para imple-

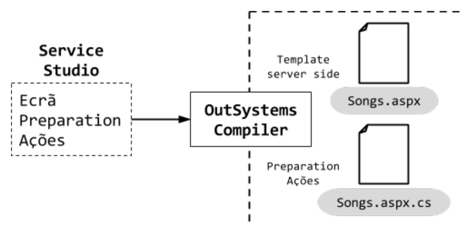

Figura 4: Artefactos gerados pelo *OutSystems Compiler*.

mentar a interação com o utilizador. Ao invocar uma ação no ecrã pode ser desencadeado um de 3 tipos de pedidos ao servidor:

Normal. O servidor executa a ação, voltando ao mesmo ecrã, executando novamente a ação *Preparation* e gerando uma página `HTML` nova. O campo *ViewState* mantém o estado da página entre pedidos (ver definição abaixo).

AJAX. É feito um pedido assíncrono ao servidor, que executa a ação, a resposta contém fragmentos de página para actualizar o contexto do cliente. O campo *ViewState* é também utilizado nestes casos.

Navegação. Trata-se de um pedido que permite ao cliente navegar para outro ecrã, inicia-se um novo ciclo começando pela ação *Preparation* do novo ecrã.

A geração do código é realizada pelo *OutSystems Compiler*, um componente do *Platform Server*, que compila o modelo da aplicação desenvolvida. Após submetido o modelo da aplicação ao processo de compilação, são gerados vários ficheiros particulares a um ecrã. Entre os quais o *template server-side* do mesmo, e o código executado no servidor, que incorpora a ação *Preparation*, e as ações do ecrã, como é apresentado na Figura 4.

A Figura 1 ilustra a comunicação entre cliente e servidor num ecrã gerado pela plataforma *OutSystems*, neste caso para a aplicação das músicas que suporta uma listagem de músicas com a adição de novos elementos. No primeiro pedido é executada a ação *Preparation*, que obtém os dados relativos ao conjunto de músicas, com base nos quais é gerada a página, sucedendo-se o envio do `HTML` para o cliente. O segundo pedido, explicitamente programado para ser assíncrono no mesmo ecrã, chega ao servidor onde é executada a ação (*append* à tabela). Essa ação gera o fragmento da página correspondente à listagem de músicas, que é então atualizado no cliente. A programação deste segundo pedido é feita com recurso a construções específicas da linguagem para o efeito. No último pedido é executada a ação *Save*, que cria na base de dados cada música da tabela.

ViewState É um mecanismo criado pelo `ASP.NET`, para suportar a manutenção do estado de uma página entre pedidos. Ao contrário do mecanismo de sessão que mantém os valores no servidor, o *ViewState* é um dicionário enviado ao cliente juntamente com o código `HTML` da página. Em pedidos consecutivos à mesma página o *ViewState* é enviado como parte desse pedido, permitindo assim ao

```
app SplitTheBill {
    dataModel{ entity Friend (Name) records=[ {Name:"Sara"}, {Name:"Hugo"} ]}
    function CountUncheck(list){ ... }
    screen Bill { ... }
}
```

Figura 5: Definição de uma aplicação.

servidor recuperar o estado do ecrã. É importante realçar que a utilização do *ViewState* é totalmente transparente para o programador no desenvolvimento de uma aplicação na plataforma. A Figura 1 ilustra o envio do campo *ViewState* em todos os ciclos de pedido-resposta, exceto no primeiro pedido do cliente para o servidor, uma vez que ainda não existe qualquer estado.

3 Modelo proposto

Para definir formalmente o método de análise estática e como método de prototipagem, propõe-se uma linguagem modelo, inspirada na linguagem da plataforma *OutSystems* e na representação abstrata de uma aplicação. Esta linguagem permite a definição do modelo de uma aplicação completa, com todos os seus componentes e permite explorar novas arquiteturas na geração das aplicações. A sintaxe da linguagem é definida por um conjunto de regras[4] , como por exemplo:

```
App: 'app' name = ID '{' d = DataModel, f += Function*, s += Screen* '}'
```

O modelo de uma aplicação engloba a definição de uma base de dados, funções globais, e o conjunto de ecrãs, como é ilustrado no fragmento de forma resumida no exemplo da Figura 5. Nesta segunda aplicação de exemplo, um utilizador pode calcular como vai ser dividido o custo total de um jantar por um conjunto de amigos. A Figura 6 apresenta a definição do ecrã Bill, omitido na Figura 5, e que implementa a interface principal da aplicação SplitTheBill.

O componente dataModel, apresentado na Figura 5, do modelo da aplicação, define o modelo de dados da aplicação. Introduz as entidades que podem ser usadas para consulta de dados nas ações da aplicação. Apenas por questões de simplificação, no modelo proposto as entidades são representadas em memória no servidor. É também possível declarar funções globais à aplicação, (function F(){...}), que podem ser usadas, por exemplo, na inicialização de variáveis ou propriedades de *widgets*. Note-se na Figura 6, onde é invocada a função CountUncheck na inicialização da propriedade value do *widget* Expression.

A definição de ecrãs pode envolver a declaração de variáveis locais, instanciação de *widgets*, ações locais ao ecrã, e a ação especial de preparação (chamada *Preparation*). Por exemplo, na Figura 6 declaram-se algumas variáveis locais (queryFriends, newFriend, total) que são visíveis nas ações e nas expressões de inicialização dos *widgets*. Note-se que a linguagem de tipos utilizada nesta linguagem

[4] Regras definidas em `Xtext`

```
screen Bill {
  variables { Query queryFriends,  Record newFriend, Integer total default:0 }
  widgets {
    <Input variable=total/>
    <Expression name=costPPerson value=(total / friendsList.list.Length) />
    <Expression name=friendsToPay value=CountUncheck(friendsList.list) />
    <ListRecords name=friendsList src=queryFriends>
      <item>
        <Input variable=current.Friend.Check type="checkbox" />
        <Expression value=current.Friend.Name />
      </item>
    </ListRecords>
    <Input name=inputFriend variable=newFriend.Friend.Name/>
    <Button onClick=AddFriend label="Add" />
    <Button onClick=Save label="Save" />
    <Button onClick=Reset label="Reset" />
  }
  preparation { queryFriends = getRecords(Friend) }
  actions { ... }
}
```

Figura 6: Definição de um ecrã para a aplicação Split the Bill.

modelo segue o padrão esperado, e por razões de espaço não é aqui descrita. São utilizados no mesmo exemplo *widgets*, cuja definição não é aqui incluída, para incluir expressões simples no ecrã (Expression) e para conter uma lista de registos (ListRecords). Note-se ainda um espaço especial para definir a ação de preparação do ecrã (preparation), que é executada no início do pedido, e apenas uma vez para o mesmo ecrã, ao contrário da plataforma *OutSystems* onde é executada sempre que há um pedido dentro do ecrã.

As ações do ecrã, abreviadas na Figura 6, e definidas na Figura 7, podem ser anotadas para serem executadas no cliente (client) ou no servidor (server). A hipótese desta separação ser inferida, com base nas operações que utiliza, não faz parte do âmbito deste trabalho, mas é uma possível extensão. Cada ação é definida por uma linguagem imperativa com as operações básicas esperadas que incluem: afetação de variáveis ou propriedades dinâmicas de *widgets*; operações básicas sobre listas; iteração (forEach); expressões condicionais (if else); e operações sobre as entidades do modelo de dados (create, getRecords, etc.).

Ao contrário da plataforma *OutSystems*, que disponibiliza um conjunto pré-definido de *widgets*, a linguagem aqui proposta permite definir a utilização de novos *widgets*, que é possível instanciar na definição dos ecrãs de uma aplicação.

A definição de um *widget*, como é exemplificado na Figura 8, envolve a declaração de propriedades simples (designtimeProperties), e propriedades dinâmicas (runtimeProperties), e a definição do código `HTML` para a apresentação do conteúdo. As propriedades declaradas podem ser referenciadas no código `HTML`, e a sua inicialização estabelece os dados que são consumidos pelo *widget* quando é desenhado. Por outro lado, as propriedades dinâmicas podem ser diretamente manipuladas pelas ações do ecrã. Por exemplo, o *widget* Expression da Figura 8 exemplifica a parametrização do código `HTML` com a propriedade value, que é instanciado no ecrã através de atributos (Figura 6). É usada a notação `AngularJS`, que é também a plataforma `JavaScript` usada como suporte à implementação.

```
actions{
  client action AddFriend{ append(friendsList.list, newFriend) }
  server action Reset{ resetRecords(Friend) }
  server action Save{ forEach (friendsList.list) { create(Friend, friendsList.list.current.Friend) } }
}
```

Figura 7: Definição das ações do ecrã.

```
widget Expression {
  designtimeProperties { Exp value }
  html { <div>{{ value }}</div> }
}
widget ListRecords {
  designtimeProperties { List src }
  runtimeProperties { List list }
  refreshDataSource { list = src }
  placeholders { item }
  html { <ul><li ng−repeat=list>item</li></ul> }
}
```

Figura 8: Declaração da *widget* Expression e ListRecords.

O *widget* `ListRecords` é mais sofisticado, permitindo iterar uma lista de registos, instanciando *widgets* filhos para cada registo. A sua propriedade `src` é a lista que fornece o conjunto de registos, e a sua inicialização é apresentada na Figura 6. A propriedade dinâmica *list* permite ao *widget* manter uma cópia da lista original de registos, podendo assim ser manipulada de forma independente (e.g., podemos adicionar novos elementos à lista da *widget* sem afetar a lista original). O elemento refreshDataSource é constituído por uma lista de atribuições, permitindo ao *widget* inicializar as suas propriedades dinâmicas.

Através da utilização direta da plataforma `AngularJS` no cliente, é possível manter um padrão MVC no cliente, e tirar partido da sua linguagem de *templating*. A Figura 8 que exemplifica o uso da ligação ao modelo {{ value }}, e a iteração de uma lista, através do atributo ng−repeat=list num elemento `HTML`.

3.1 Estrutura do modelo de dados

Esta secção apresenta a proposta da estrutura de dados que é mantida no cliente, servindo de base à instanciação do *template*, e transmitida nos pedidos assíncronos de cada ecrã. Esta estrutura substitui o campo *ViewState* usado atualmente nas aplicações *OutSystems*, para preservar o estado de um ecrã, constituído pelas variáveis locais e propriedades dinâmicas dos *widgets*. Note-se ainda que quando existem pedidos dentro de um ecrã, este é recalculado e recarregado no navegador. Neste processo o ecrã pode mudar em virtude da reavaliação das expressões que o compõem. Ao utilizar comunicação assíncrona, os valores das expressões que instanciam o *template* no cliente podem ser alterados em cada pedido. No modelo que aqui se apresenta, estas expressões são calculadas no ser-

```
{ newFriend: { Friend: { Name: "" } },
  queryFriends: [ { Friend: { Name: "Sara" }}, ... ],
  RuntimeProps: { inputFriend: { valid: true, validationMessage: "" } } },
  EvalData: { friendsToPay: { value: 1 }, costPPerson: { value: "$12.5" }, ... },
  friendsList_list: [{Item: {Friend: { Name: "Sara"}}, RuntimeProps: {friendCheck: {...}}, EvalData: {}},...]}
```

Figura 9: Exemplo da estrutura

vidor e mantidos os seus valores na estrutura de dados de suporte. Logo, devem ser também retransmitidos quando modificadas.

O formato da estrutura apresentada é reconhecido pelo código presente no cliente e servidor, permitindo a manipulação por qualquer ação. Para que se mantenha atualizada, é necessário que as modificações sobre a estrutura sejam transmitidas na sequência de qualquer pedido/resposta.

A estrutura do modelo de dados proposta consiste num objeto, exemplificado na Figura 9, composto pelos seguintes elementos: o conjunto de variáveis locais do ecrã (newFriend, queryFriends); conjunto das propriedades dinâmicas dos *widgets* (RuntimeProps); conjunto das expressões pré-calculadas dos *widgets* e utilizadas para instanciar o *template* (EvalData); e um conjunto especial destinado aos widgets com propriedades dinâmicas de tipo lista, que são compostas pelos *widgets* filhos instanciados ao longo de uma iteração (friendsList_list). Uma expressão pré-calculada é uma propriedade simples do *widget*, referida no *template*, não literal, e dependente de variáveis ou propriedades dinâmicas.

A diminuição do volume de dados transmitidos é obtida pelo seccionamento desta estrutura, transmitindo apenas o subconjunto que estaticamente se determine poder ser modificado. De seguida apresenta-se uma análise estática, aplicada ao modelo de uma aplicação, que permite esta optimização.

4 Análise Estática

Descreve-se agora a análise estática aplicada ao modelo de uma aplicação, no sentido de minimizar os dados transmitidos na rede entre cliente e servidor, nas invocações das várias ações do ecrã, e ao mesmo tempo garantir que todas as expressões pré-calculadas, que são afetadas, são recalculadas no servidor e atualizadas no cliente.

A técnica de análise estática apresentada neste trabalho é uma análise de fluxo de dados aplicada ao grafo de controlo do modelo de um ecrã. Como exemplo, na Figura 13 apresenta-se o grafo do ecrã Bill, definido na Figura 6. Note-se neste exemplo que as ações do tipo servidor (Reset e Save) comunicam com o modelo no cliente, o qual fornece a estrutura de dados ligada ao *template* que mantém a interface atualizada. O cliente comunica com o servidor apenas para invocar ações do ecrã do tipo servidor. A análise estática divide-se em duas partes: uma análise de *liveness*, que nos permite determinar quais as variáveis (ou propriedades dinâmicas de *widgets*) necessárias para executar uma determinada

ação; e uma análise de dependências, que nos dá informação sobre as expressões que devem ser recalculadas no final da execução de uma ação (no cliente ou servidor), com base nas variáveis modificadas. Ambas as análises são combinadas de forma a que o cliente e servidor troquem o conjunto de dados estritamente necessário, tanto para a execução das ações como para a atualização da página.

O *template* gerado a partir da definição do ecrã da Figura 6 contém referências para expressões pré-calculadas, por exemplo a divisão da variável value com o tamanho da propriedade list. A modificação da última pela ação AddFriend faz com que a página não apresente conteúdo atualizado, nomeadamente o resultado da divisão. Como tal, o cliente deve recalcular a expressão que efetua a divisão.

Neste trabalho opta-se pela definição da análise estática usando a linguagem `Datalog` [3] e as ferramentas associadas, em vez dos tradicionais algoritmos de ponto fixo [4], o que permite a utilização de bibliotecas com mais facilidade e não perder eficiência na sua execução. Também se torna mais fácil definir as várias propriedades a analisar e as regras de propagação das mesmas.

Apresentamos agora duas análises: a primeira baseada em dependências; a segunda uma análise de *liveness*. A ordem pela qual são apresentadas deve-se ao facto da segunda depender da primeira. Para definir formalmente a análise estática proposta introduzem-se as seguintes notações. Define-se o conjunto dos nós das ações de uma aplicação, com $m, n \in \mathcal{N}$; o conjunto das variáveis locais dos ecrãs (assumindo que não há colisões), com $v \in \mathcal{V}$; o conjunto dos nomes das ações de uma aplicação, com $a \in \mathcal{A}$; o conjunto dos identificadores (com o formato nomeWidget.nomePropriedade) das expressões pré-calculadas, referenciadas no *template*, com $f, g \in \mathcal{E}$; o conjunto dos identificadores de ecrãs com $s \in \mathcal{S}$.

Análise de Dependências

A primeira análise proposta destina-se a obter o subconjunto mínimo das expressões que devem ser recalculadas no final de uma ação, com base na dependência das variáveis modificadas. Introduzimos os seguintes factos extraídos diretamente do modelo.

- $\mathsf{def}(n, v)$ A variável v do modelo é escrita no nó n.
- $\mathsf{node}(a, n)$ O nó n pertence à ação a.
- $\mathsf{need}(f, v)$ A expressão f depende da variável v
- $\mathsf{need}(f, g)$ A expressão f depende da expressão g

Com base nestes factos definimos um conjunto de regras, Figura 10, para exprimir as propriedades pretendidas na análise de dependências entre as ações de um ecrã e as expressões pré-calculadas. São elas,

- $\mathsf{mod}(a, v)$ A ação a modifica a variável v.
- $\mathsf{needEval}(a, f)$ Após a execução da ação a, a expressão identificada por f deve ser recalculada.

Na regra (1) define-se que as variáveis modificadas numa ação são aquelas que são redefinidas num dos seus nós; na regra (2) define-se que uma expressão pré-calculada (f) depende das variáveis (v) que são usadas na sua definição ($\mathsf{need}(f, v)$); a regra (3) determina a transitividade da relação de dependência.

$$\frac{\mathsf{def}(n,v)}{\mathsf{mod}(a,v)}\ (1) \qquad \frac{\mathsf{mod}(a,v) \quad \mathsf{need}(f,v)}{\mathsf{needEval}(a,f)}\ (2) \qquad \frac{\mathsf{needEval}(a,g) \quad \mathsf{need}(f,g)}{\mathsf{needEval}(a,f)}\ (3)$$

Figura 10: Análise de dependências.

$$\frac{\mathsf{use}(n,v)}{\mathsf{live}(n,v)}\ (4) \qquad \frac{\mathsf{live}(m,v) \quad \mathsf{succ}(n,m) \quad \neg\mathsf{def}(n,v)}{\mathsf{live}(n,v)}\ (5) \qquad \frac{\mathsf{live}(n,v) \quad \mathsf{entry}(a,n) \quad \mathsf{action}(s,a)}{\mathsf{live}(s,v)}\ (6) \qquad \frac{\mathsf{need}(f,v) \quad \mathsf{needEval}(a,f) \quad \mathsf{node}(a,n) \quad \neg\mathsf{mod}(a,v) \quad \neg\mathsf{exp}(v)}{\mathsf{use}(n,v)}\ (7)$$

Figura 11: Análise de *liveness*.

Liveness

A segunda análise permite determinar o conjunto de variáveis necessárias para a execução de uma ação, e por isso têm que ser transmitidas para o servidor, e o conjunto de variáveis necessárias na atualização da interface após a execução de uma ação. Aos factos anteriores, extraídos do modelo, acrescentam-se:

- $\mathsf{use}(n,v)$ A variável v é lida no nó n, ou ocorre no cálculo de expressões definidas no *template* (nó especial).
- $\mathsf{action}(s,a)$ A ação a pertence ao ecrã s.
- $\mathsf{succ}(n,m)$ O nó n tem como sucessor o nó m na definição de uma ação.
- $\mathsf{entry}(a,n)$ O ponto de entrada da ação a é o nó n, correspondente à primeira instrução da ação.
- $\mathsf{exp}(f)$ A expressão identificada como f, é pré-avaliada e referenciada no template. Não é uma variável dinâmica, embora a sua avaliação dependa de outras variáveis.
- $\mathsf{live}(n|s,v)$ A variável v é necessária à entrada do nó n, ou no ecrã s.

As regras definidas na Figura 11 permitem determinar o subconjunto de dados a ser transmitido do servidor para o cliente, e vice-versa. O cliente deve enviar para o servidor o conjunto de dados determinado pelas regras (4) e (5) no nó de entrada de cada ação invocada. Como resposta, o servidor deve enviar o conjunto de dados determinado pela regra (6), a união de todas as variáveis necessárias nos pontos de entrada das ações do ecrã s. A análise *liveness* aqui apresentada tem como ponto de partida o nó de entrada do grafo (identificado pelo facto entry para cada ação), com caminho pelo fluxo de controlo.

A regra (7) é um caso especial do facto use, determinando que uma variável v ocorre no nó n se a mesma não for modificada ao longo da ação a, e for necessária para a computação de uma expressão no final da ação a. A negação do facto exp pretende ignorar as dependências entre expressões pré-calculadas, uma vez que a transmissão dessas é determinada pela primeira técnica apresentada.

$$\frac{\mathsf{entry}(a,n) \quad \mathsf{live}(n,v)}{\mathsf{CtoS}(a,v)} \ (8) \qquad \frac{\mathsf{action}(s,a) \quad \mathsf{live}(s,v) \quad \mathsf{mod}(a,v)}{\mathsf{StoC}(a,v)} \ (9)$$

Figura 12: Resultados finais.

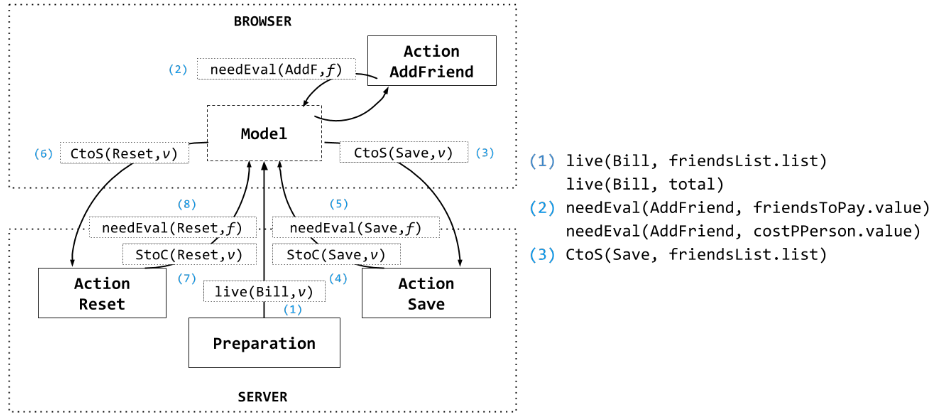

Figura 13: Comunicação entre os componentes da aplicação, com transmissão dos dados inferidos pela análise estática.

Aplicação da análise

Quando o cliente invoca uma ação do servidor, é enviado o conjunto de dados determinado pela regra (8) apresentada na Figura 12, que obtém o conjunto de variáveis necessárias para a execução da ação, considerando os resultados da análise *liveness* no nó de entrada (n) da ação. Como resposta, o servidor envia o conjunto de variáveis para as quais o predicado $\mathsf{live}(s,v)$ é válido. O cliente não necessita de receber dados que não foram modificados pelo servidor, a regra (9) interseta os conjuntos determinados pelas regras (6) e (1).

A Figura 13 para além de apresentar o grafo de controlo do ecrã Bill identifica os conjuntos de dados determinados pela análise estática que são transmitidos entre cliente e servidor, ou que são recalculados de forma a atualizar a página. Os resultados da análise são apresentados à direita na Figura 13, a partir dos quais é gerada a estrutura apresentada na secção 3.1.

Durante a geração de código, estes resultados são usados da maneira que se descreve a seguir. Para cada ação do tipo servidor, são determinados, o conjunto de variáveis locais alteradas pela ação (predicado $\mathsf{StoC}(A,v)$), e o conjunto das expressões que necessitam de ser recalculadas (necessárias no cliente) (predicado $\mathsf{needEval}(A,e)$).

Na aplicação exemplo SplitTheBill, o conjunto de dados recebido pelo cliente no primeiro pedido ao ecrã Bill é composto pelas variáveis/propriedades

friendsList.list e total. A ação Save usa a variável friendsList.list na iteração. A ação Reset, não transmite quaisquer dados pois opera apenas sobre a BD.

Para gerar o código cliente correspondente ao *template* e à mecanização dos pedidos ao servidor é necessário ter em conta a informação sobre que variáveis são necessárias à execução de cada ação, pelo predicado $\mathsf{CtoS}(A, v)$, e reavaliação de expressões no servidor, pelo predicado $\mathsf{needEval}(A, e)$. A invocação da ação AddFriend implica recalcular as expressões friendsToPay.value e costPPerson.value.

5 Implementação e resultados

O protótipo da linguagem aqui apresentada foi realizado utilizando ferramentas de definição de linguagens de programação textuais. Neste caso utilizou-se a ferramenta `Xtext` [5], integrada no ambiente de desenvolvimento `Eclipse`. Definiu-se a linguagem, e a representação abstrata de uma aplicação, tendo como inspiração a representação usada pela plataforma *OutSystems*. Implementou-se um gerador de código que produz para cada aplicação modelada, um padrão arquitetural MVC duplo, e o respetivo código de comunicação. O alvo do gerador de código é a plataforma `Node.js` onde se implementa um servidor com comunicação assíncrona, e `AngularJS` [6] para suportar uma interface de utilizador que é automaticamente atualizada. No sentido de simplificar a implementação do protótipo optou-se por manter em memória no servidor os dados persistentes das aplicações. A este conjunto de ferramentas, foi adicionado o interpretador de `Datalog IRIS` [7], para implementar a análise estática definida na secção 4. A partir do modelo é produzido o conjunto de factos que define o grafo de controlo da aplicação, e os restantes factos não inferidos pela análise como o def e use, o que posteriormente, permite interrogações acerca dos predicados needEval e live.

Em conclusão, o protótipo implementado tem como input um modelo de uma aplicação, e gera uma aplicação `Node.js` e `AngularJS`, que se encontra otimizada no que diz respeito aos dados transmitidos, e ao cálculo de expressões. A Figura 14 apresenta um conjunto de operações que surgem da interação de um utilizador com a aplicação usada como exemplo no início do artigo (aplicação de músicas), estendida com ações assíncronas, nomeadamente a adição de uma nova música, remoção e edição, a informação das músicas é agora apresentada em *inputs*, permitindo a sua edição. Neste exemplo a operação *GetData* representa o primeiro pedido ao ecrã que apresenta a listagem de 500 músicas. Note-se que os valores são ligeiramente diferentes, relativamente aos valores na Figura 2, devido à mudança da página. Observa-se uma redução significativa no volume do conteúdo transmitido na resposta ao cliente, na comparação entre a mesma aplicação gerada em *OutSystems* e no protótipo apresentado neste trabalho.

[5] http://www.eclipse.org/Xtext, acedido em 21-07-2014
[6] http://www.angularjs.org, acedido em 21-07-2014
[7] http://www.iris-reasoner.org, acedido em 21-07-2014

Operações	**HTML** (KB)	**Modelo** (KB)
GetData	49,4	2,06
Add	56,6	2,15
Remove	96,4	2,08
Update	1,0	0

Figura 14: Tamanhos relativos entre HTML e estrutura do modelo de dados.

6 Trabalho relacionado

Neste trabalho utiliza-se uma arquitetura de MVC duplo, uma extensão do padrão MVC usualmente presente no servidor. Esta abordagem inspira-se em algumas plataformas de programação atuais, mas também na arquitetura proposta por *Garcia et al* [5], que introduz uma arquitetura com dois modelos, um no cliente destinado exclusivamente à apresentação da página, e outro no servidor para gerir os dados que persistem na base de dados. O objetivo aqui é manter a consistência entre os dados visualizados na página web e os dados no servidor. O nosso objetivo é minimizar o volume de dados transmitido.

No sentido de manter uma separação entre a lógica aplicacional e apresentação, *Parr* [6] apresenta uma formalização desta separação através do estudo de mecanismos de *templating*, introduzindo um conjunto de regras para a criação de um *template*, e garantindo que essa separação não é quebrada. Uma das técnicas de análise estática apresentada neste trabalho vai de encontro a essas regras, na medida em que a computação de expressões não é definida no *template*. Através da aplicação da análise proposta, a uma linguagem que integra toda a aplicação, obtemos a separação automática e otimizada, e não apenas um conjunto de boas práticas e recomendações de desenvolvimento.

Leff et al [7] esclarecem que usualmente a camada de lógica aplicacional e modelo de dados de uma arquitetura MVC é distribuída entre o cliente e servidor, ficando a cargo do programador tomar a decisão do que deve ser incorporado no cliente. Tal como na linguagem proposta em que o programador identifica quais as ações a executar no cliente ou servidor. Estas decisões, durante a fase de desenvolvimento, podem não ser as mais adequadas. Como tal propõem uma plataforma que define um particionamento automático a fim de maximizar a performance da aplicação. *Yang et al* também apresentam uma plataforma neste sentido [8]. Embora este tipo de separação esteja fora do âmbito da abordagem aqui apresentada, está relacionada com algum trabalho ainda a desenvolver.

A separação automática entre lógica e apresentação é essencial em diversos aspetos. No caso presente é essencial para a geração de código usando uma arquitetura diferente, no caso da plataforma `TouchDevelop` [9] é essencial para garantir propriedades na modificação da camada de apresentação, sem interrupção do serviço fornecido pela aplicação.

7 Conclusões e trabalho futuro

Este trabalho apresenta uma abordagem inovadora que permite analisar o modelo de uma aplicação web, no contexto da plataforma *OutSystems*, gerando automaticamente aplicações otimizadas, baseadas em comunicação assíncrona dos dados, e onde a geração de páginas `HTML` é efetuada no cliente. A abordagem baseia-se numa linguagem modelo e uma análise de fluxo de dados que permite minimizar a transmissão de dados e garantir que a informação, necessária para atualizar as interfaces, é transmitida do cliente para o servidor.

Os resultados obtidos mostram uma redução drástica do conteúdo transmitido na rede, redução da carga no servidor, menor ocorrência de comunicações entre cliente e servidor, e consequentemente aplicações mais rápidas e fluídas.

As direções de trabalho futuro apontam para a extensão da linguagem para que seja possível anotar os dados que não devem ser transmitidos na rede por razões de segurança, e determinar automaticamente sobre que parte da lógica aplicacional deve ser incorporada no cliente ou servidor.

Referências

1. Pree, W.: Design Patterns for Object-oriented Software Development. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
2. Pressman, R.S.: Software Engineering: A Practitioner's Approach. 5th edn. McGraw-Hill Higher Education (2001)
3. Smaragdakis, Y., Bravenboer, M.: Using Datalog for Fast and Easy Program Analysis. In: Proceedings of the First International Conference on Datalog Reloaded. Datalog'10, Berlin, Heidelberg, Springer-Verlag (2011) 245–251
4. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
5. García, F.J., Castanedo, R.I., Fuente, A.A.J.: A Double-model Approach to Achieve Effective Model-view Separation in Template Based Web Applications. In: Proceedings of the 7th International Conference on Web Engineering. ICWE'07, Berlin, Heidelberg, Springer-Verlag (2007) 442–456
6. Parr, T.J.: Enforcing Strict Model-view Separation in Template Engines. In: Proceedings of the 13th International Conference on World Wide Web. WWW '04, New York, NY, USA, ACM (2004) 224–233
7. Leff, A., Rayfield, J.T.: Web-Application Development Using the Model/View/Controller Design Pattern. In: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing. EDOC '01, Washington, DC, USA, IEEE Computer Society (2001) 118–
8. Yang, F., Gupta, N., Gerner, N., Qi, X., Demers, A., Gehrke, J., Shanmugasundaram, J.: A Unified Platform for Data Driven Web Applications with Automatic Client-server Partitioning. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, New York, NY, USA, ACM (2007) 341–350
9. Burckhardt, S., Fahndrich, M., de Halleux, P., McDirmid, S., Moskal, M., Tillmann, N., Kato, J.: It's Alive! Continuous Feedback in UI Programming. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13, New York, NY, USA, ACM (2013) 95–104