Sara Alpoim Gonçalves

Licenciada em Engenharia Informática

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

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : João Costa Seco, Professor Auxiliar, FCT-UNL
Hugo Lourenço, Eng. de Software, OutSystems

Júri:

Presidente:	Prof. João Moura Pires
Arguente:	Prof. Francisco da Cunha Martins
Vogal:	Prof. João Costa Seco

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2014

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

Agradecimentos

Em primeiro lugar, quero expressar os meus agradecimentos à Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, que me acolheu durante 5 anos, e que me fez crescer a nível profissional e pessoal. Obrigada também, a todos os colegas de curso que me acompanharam durante o percurso académico, com os quais partilhei um espírito de equipa incrível, que me motivou e ajudou a alcançar esta meta.

Em segundo lugar, quero agradecer aos meus orientadores: João Seco, por me guiar ao longo do ano, pelo apoio, e por todo o conhecimento que me transmitiu; ao Hugo Lourenço, pelas suas observações críticas, e pelas discussões que ajudaram a definir o caminho para o desenvolvimento deste trabalho. Obrigada por me terem proporcionado escolher um tema sobre o qual tive imenso prazer trabalhar, pois uniu todas as áreas e conceitos que me foram atraindo ao longo destes 5 anos. Quero também agradecer à *OutSystems* por financiar uma bolsa ao longo do desenvolvimento desta dissertação.

Aproveito ainda para agradecer às pessoas que também contribuíram para o resultado final desta dissertação: ao Sérgio Silva, segundo co-orientador e padrinho, pelo apoio desde o primeiro dia do meu percurso académico, e por manter a minha motivação sempre em alta; à Lara Luís, madrinha e colega na *OutSystems*, pelas conversas diárias, e por me recordar dos meus objetivos. Agradeço também ao Lúcio Ferrão, Vasco Pessanha, Rodrigo Coutinho, Marco Costa, João Neves, Paulo Ferreira e Rúben Gonçalves, por todas as observações que me permitiram aperfeiçoar esta dissertação.

Como este trabalho foi desenvolvido na *OutSystems*, quero agradecer a todos os elementos da equipa de R&D que me receberam da melhor forma possível, e com os quais aprendi imenso.

Os meus últimos agradecimentos, mas não menos importantes, são aos meus pais, pois graças a eles cheguei até aqui. Aos meus irmãos, por terem contribuído para a pessoa que sou hoje. E por último ao João, pela amizade e amor, e pela paciência de ouvir um relatório diário das minhas preocupações e alegrias que esta dissertação me deu.

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 *OutSystems Platform* 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 modelo normal de interação cliente-servidor da plataforma é coerente 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 sobre a definição 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, a partir do modelo de uma aplicação *OutSystems*.

O modelo apresentado, em conjunto com a análise estática, permite identificar o subconjunto mínimo dos dados a serem transmitidos na rede 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 muito significativa na transmissão de dados, e possivelmente uma redução na carga de processamento no servidor, dado que a geração das páginas Web é delegada no cliente, e este se torna apto para executar código.

Este modelo é definido sobre uma linguagem, inspirada na da plataforma *OutSystems*, a partir da qual é implementado um gerador de código. Neste contexto, uma linguagem de domínio específico cria uma camada de abstração entre a definição do modelo de uma aplicação e o respetivo código gerado, tornando transparente a criação de *templates client-side* e o código executado no cliente e no servidor.

Palavras-chave: Aplicações web, *templates client-side*, análise estática, comunicação assíncrona

Abstract

The growing computational power of mobile devices and greater efficiency of browsers promotes the construction of faster and more fluid web applications, using the asynchronous exchange of data, instead of entire `HTML` pages. The *OutSystems Platform* is a development environment used to quickly build and validate web applications, whose language allows to build user interfaces, business logic, and data model. The typical model of client-server interaction in the platform is consistent with the entire cycle of request-response, although it is possible to implement explicitly asynchronous applications.

We present a model of separation, using static analysis on the definition of an application, between the data presented on pages generated by the platform and the code that corresponds to its structure and presentation. This approach allows the automatic and transparent generation of fluid and faster user interfaces, according to the *OutSystems* application model.

The presented model, together with static analysis, allows us to identify the minimal subset of data to be transmitted on the network for running a feature on the server, and isolate code execution on the client. As a result of using this approach yields a significant reduction in data transmission, and possibly a reduction in the processing time on the server, because the generation of web pages is delegated to the client, who becomes able to execute the application related code.

This model is defined on a language inspired by the *OutSystems Platform*, from which a code generator is implemented. In this context, one domain specific language creates an abstraction layer between the definition of an application model and the corresponding generated code, making transparent the creation of client-side templates and the executed code on the client and server.

Keywords: Web applications, client-side templates, static analysis, asynchronous communication

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens

1

Introdução

Esta tese é desenvolvida no âmbito da equipa de *R&D* da empresa *OutSystems*, cujo principal produto é a *OutSystems Platform* - um ambiente de desenvolvimento que permite construir e manter aplicações web de larga escala, de forma fácil e segura, em que muitos detalhes de desenvolvimento e *deployment* são abstraídos por um ambiente centrado numa linguagem específica para o domínio.

O tema central são portanto as aplicações web, cuja principal característica é a disponibilização de serviços na Internet. Tradicionalmente, estes serviços estão disponíveis para, o chamado utilizador final, através de interfaces homem-máquina constituídas por páginas `HTML`, mais ou menos dinâmicas. O mesmo tipo de serviços pode também ser usado como forma de interoperabilidade entre sistemas, comunicando por formatos convencionados de dados, por exemplo `XML` e `JSON`.

As boas práticas de modularidade e desacoplamento de software [13, 14] recomendam que a construção de aplicações promova uma separação clara entre a apresentação dos dados e a lógica da aplicação (*business logic*). A camada de apresentação corresponde geralmente ao código de definição, ou construção, da interface com o utilizador ou com outros sistemas. As metodologias e ferramentas atuais (normalmente organizadas nas chamadas (*frameworks*) promovem a utilização de *templates* para a especificação da camada de apresentação, seja para definir páginas `HTML`, ou as referidas interfaces de dados. O uso de *templates* aqui referido decorre predominantemente da utilização do padrão arquitetural MVC [8] como base. Terence Parr propôs [11] uma classificação para *templates* com base em algumas das suas propriedades (*e.g.*, parametrização, código externo, etc.), classificando-os quanto ao grau de separação entre interface e lógica da aplicação.

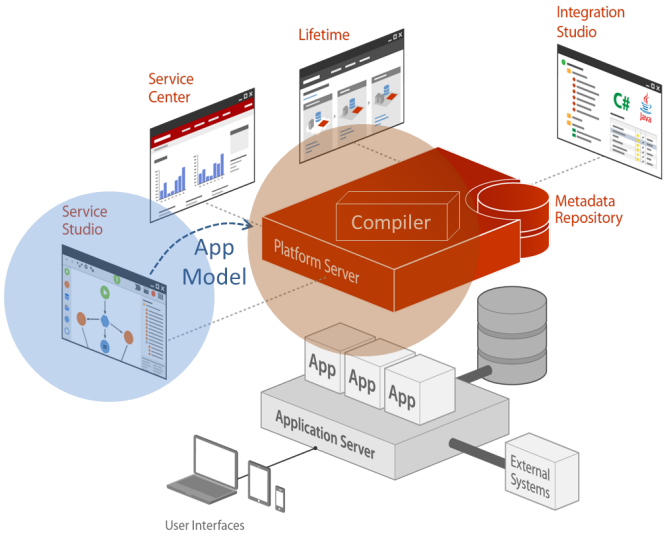


Figura 1.1: Arquitetura dos componentes da *OutSystems Platform*.

A noção de *template* aplicado ao contexto web corresponde à definição da estrutura e conteúdos estáticos de documentos `HTML`, `XML`, ou `JSON`, usados na interface concreta da aplicação e parametrizados de forma a produzir dinamicamente páginas a partir de dados "puros". Nas aplicações web, e concretamente neste trabalho, os *templates* podem ser divididos em:

***Templates* tipo servidor** (*Server-side templates*)

São os *templates* mais comuns nas *frameworks* de desenvolvimento que facilitam a implementação do padrão arquitetural MVC (*Model-View-Controller*). Esse padrão permite manter uma separação clara entre dados, apresentação e lógica da aplicação, a partir das camadas *Model*, *View* e *Controller*, respetivamente. Todas as camadas são definidas no servidor.

A camada *Model* mantém os dados da aplicação numa base de dados, e implementa as respetivas operações de consulta, criação, atualização e remoção. É também ao nível desta camada que são definidas restrições de segurança sobre os dados. A camada *Controller* processa os pedidos desencadeados na interface, e invoca as operações sobre a *Model*, de forma a preparar os dados para a *View*. A camada *View* é composta pelos *templates* tipo servidor, que definem a interface concreta com o cliente. Quando são instanciados com os dados preparados pelo *Controller*, é gerada a página `HTML` completa. A comunicação entre cliente e servidor consiste em: o utilizador submete um pedido através do navegador, com destino ao servidor. Esse pedido pode, por exemplo, corresponder à solicitação de uma página da aplicação que, leva o servidor a executar lógica da aplicação, a preparar os dados e, a construir a página `HTML` completa, que é transmitida na rede para o cliente.

Este comportamento pode influenciar negativamente os tempos de latência, que

variam com o volume do documento `HTML` transmitido, e tempos do desenho das páginas que depende do desempenho do navegador e do dispositivo em que é executado, o que pode ser prejudicial para a fluidez da aplicação.

***Templates* tipo cliente** (*Client-side templates*)
São *templates* usados no dispositivo cliente para gerar a interface concreta apresentada ao utilizador final, instanciando um conjunto de parâmetros a partir de dados fornecidos de forma independente pelo servidor. Atualmente, várias plataformas de desenvolvimento permitem replicar o padrão MVC no cliente, semelhante ao que existe no servidor. Assim, é possível distribuir a execução de lógica da aplicação entre cliente e servidor, e delegar no cliente a geração da interface. Alguns exemplos a assinalar são as plataformas `AngularJS` [20], `KnockoutJS` [28] e `EmberJS` [24]. Nestes contextos, as várias páginas `HTML` são obtidas por instanciação dos chamados *templates* tipo cliente. Sobre essas plataformas é possível construir aplicações mais ricas e fluídas, em comparação com aplicações que delegam a geração dinâmica das páginas web no servidor. O cliente evita esperar que o servidor gere a página e a envie pela rede, visualizando a página mais rápido, nomeadamente se existir um mecanismo de cache no cliente, capaz de detetar que a página não foi alterada desde a última vez que foi visualizada.

Para ilustrar o contraste entre *templates* destas duas categorias, desenvolveu-se um pequeno exemplo, do qual se apresentam alguns detalhes em anexo (Anexo A–página 109). Mostra-se como a mesmo interface pode ser definida utilizando as *frameworks* `ASP.NET` (*template* tipo servidor) e `AngularJS` (*template* tipo cliente).

Com a introdução de outros fatores, como o crescente poder computacional nos dispositivos móveis e a maior eficiência dos navegadores, tem sido fomentada a construção de aplicações web mais rápidas e fluídas. Assim, surgem as aplicações web que adotam a troca assíncrona de dados entre cliente e servidor, usados para instanciar o *template* tipo cliente e, consequentemente criar e atualizar a página `HTML` no navegador.

Algumas aplicações web como o *Gmail*, o *Google Docs*, o *Linkedin* ou o *Trello*, são exemplos com requisitos fortes sobre a experiência do utilizador, um fator a ter em conta ao longo do processo de desenvolvimento. A experiência do utilizador depende da manutenção de contextos complexos e desacoplamento de informação ao nível da interface. É necessário adaptar as interfaces com o utilizador para que a transição entre páginas e a atualização da informação apresentada seja fluída, de forma a que o utilizador não perca o contexto enquanto interage com a aplicação. Estas aplicações optam por executar grande parte da sua lógica no cliente, onde geram a interface `HTML` dinamicamente, a partir de dados trocados com o servidor de forma assíncrona. São também exemplos de aplicações web do tipo "Página Única" (SPA -*Single Page Application*), um tipo de construção que permite obter mais fluidez. A sua execução baseia-se na atualização de apenas algumas zonas da página, reduzindo o tempo de espera, e alternando o contexto de forma mais suave. No navegador, o tempo do carregamento de um documento `HTML` é maior

em relação à substituição de sub-elementos.

Optar por *templates* tipo cliente possibilita também a simplificação do desenvolvimento de aplicações com a interface adaptada aos vários tipos de dispositivo (e.g., tablet, desktop, smartphone). Para isso, são definidos vários *templates* para cada página web, com a sua estrutura adaptada aos diferentes ecrãs dos dispositivos cliente. Quando o cliente solicita uma página da aplicação, envia a informação suficiente que permite a identificação do dispositivo, e assim o servidor, onde são mantidos os *templates*, fornece o mais adequado para a visualização da página. É importante realçar que um *template* tipo cliente é fornecido apenas uma vez, ficando depois em cache no navegador, já que apresenta uma estrutura estática. Independentemente do *template* que é fornecido, o servidor envia os mesmos dados. Se desejarmos otimizar a quantidade de dados transmitidos na rede, é possível definir o conjunto de dados a enviar conforme o dispositivo.

1.1 Problema/Desafio

A *OutSystems Platform*, ou a plataforma *OutSystems*, permite construir de forma rápida e validada aplicações web. Integra uma linguagem visual através da qual é definida a interface de utilizador (ecrãs[1]), a lógica da aplicação (ações[2]) e o modelo de dados. A plataforma suporta não só o desenvolvimento de aplicações, mas, talvez mais importante, a sua evolução ao longo do tempo assistida por mecanismos de validação. A plataforma inclui vários componentes, apresentados na Figura 1.1, destacando-se o *Service Studio*, o ambiente de desenvolvimento, e o *Platform Server*, responsável pela publicação e execução das aplicações. O *OutSystems Compiler* está incluído no *Platform Server*, e gera o código das aplicações sobre as *frameworks* `.NET` ou `JSP`, que no fim são publicadas num servidor.

No contexto da plataforma, o modelo usual de interação cliente-servidor é baseado em ciclos completos de pedido-resposta, ilustrados na Figura 1.2. Embora seja possível implementar aplicações com comunicação assíncrona de forma explícita. 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, tudo representado pelo componente *Application Server* na Figura 1.1.

Nas aplicações geradas na plataforma *OutSystems*, cada ecrã é completamente instanciado no servidor, e todas as ações são também executadas no servidor. O servidor não mantém qualquer estado, como tal é necessário existir uma manutenção do estado com recurso ao *ViewState*[3], mantido entre pedidos da mesma página. 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 1.2.

[1] Um ecrã corresponde a uma página da aplicação, no contexto da *OutSystems*.
[2] Uma ação corresponde a uma sequência de operações, no contexto da *OutSystems*.
[3] Mecanismo *View State* da plataforma `.NET`

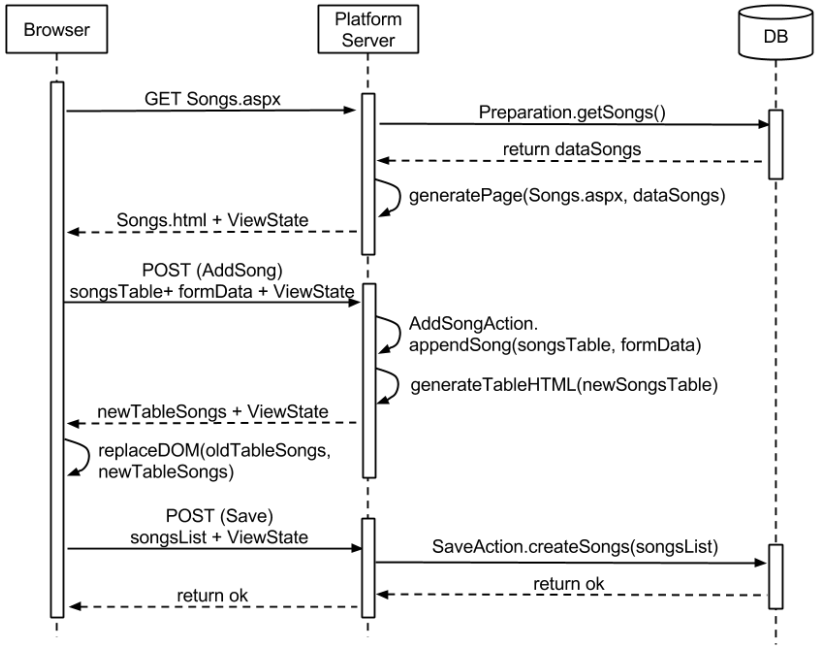


Figura 1.2: Comunicação cliente/servidor numa aplicação desenvolvida na plataforma *OutSystems*.

Embora a passagem de contexto seja transparente para o programador, a fluidez das aplicações geradas pode ser comprometida, com a transmissão constante de um estado em todos os pedidos, tendo ainda em conta que os dados não representam o conjunto mínimo necessário em cada momento.

A comunicação entre cliente e servidor das aplicações geradas pela *OutSystems Platform* tem por base dois tipos de pedidos: normais (pedidos `HTTP - GET` ou `POST`) e `AJAX`. No pedido normal, o cliente solicita uma página, o servidor procede à geração do `HTML` completo, e envia-o para o *browser*. No pedido `AJAX`, o cliente pode submeter pedidos ao servidor, e obtém um fragmento de `HTML`, que é substituído no documento `HTML` existente no cliente, atualizando parcialmente o conteúdo. A interação através de pedidos `AJAX` torna-se mais fluída, pois o tempo de carregamento de um documento `HTML` no *browser* é superior à substituição de sub-elementos no documento já carregado.

Em resposta a qualquer dos dois tipos de pedido, o servidor necessita sempre de processar a totalidade ou fragmentos da página `HTML`, o que constitui uma porção significativa do processamento no servidor, e consome desnecessariamente a largura de banda. Para além disso, em todos os pedidos é enviado o campo *ViewState*, com o estado da página, necessário para a execução de ações no servidor.

Como ponto de partida para a motivação deste trabalho, consideremos o exemplo de uma pequena aplicação web desenvolvida na plataforma *OutSystems*, com um ecrã que lista um conjunto de músicas. No cenário atual quando é registado um pedido do navegador, é gerada uma página `HTML` no servidor através de um *template* instanciado

```
<table class="TableRecords OSFillParent OSAutoMarginTop" cellspacing="0"
        id=" wt13_wtMainContent_wtSongTable">
  ...
  <tbody>
      ...
      <input onclick="OsAjax(arguments[0] || window.event,&#39;
              wtSongTable_ctl07_wt32&#39;,&#39;wtSongTable$ctl07$wt32&#39;
              ,&#39;&#39; ,&#39; __OSVSTATE,&#39; ,&#39;&#39;); return false;"
              type="submit" class="Button ThemeGrid_MarginGutter" value="Remove"
              name="wt36$wtMainContent$wtSongTable$ctl05$wt40"
       id="wt36_wtMainContent_wtSongTable_ctl05_wt40"/>
      ...
      <td class="TableRecords_OddLine" >
          It's the final countdown
      </td>
      <td class="TableRecords_OddLine">
          Europe
      </td>
      ...
  </tbody>
</table>
```

Listagem 1.3: Página HTML gerada pela plataforma *OutSystems*.

```
"songs": [
    "song": {
        "name":"It's the final countdown",
        artist:"Europe", ...
    }, ...
]
```

Listagem 1.4: Dados apresentados na página da Figura 1.3 no formato JSON.

com os dados obtidos por uma consulta à base de dados. Assim, a página HTML contém informação relativa a dados e também à estrutura da página. Na Listagem 1.3 é apresentado um fragmento do HTML gerado, correspondente a uma linha da tabela de músicas. Este cenário é exemplar dos casos em que é notável a repetição de informação na página gerada. As linhas da tabela HTML têm a mesma estrutura, e apenas os dados relativos ao nome da música, artista e álbum representam a informação dinâmica. Colocamos então a hipótese de enviar apenas os dados dinâmicos para o cliente, num formato como por exemplo JSON. Na Listagem 1.4 é apresentado o JSON dos dados apresentados no HTML da Listagem 1.3.

A Tabela 1.1 representa a variação do tamanho do HTML gerado pela plataforma *OutSystems* (com e sem compressão), da página com a listagem de x músicas, com o intuito de simular maiores volumes de dados. Na mesma tabela é representada a variação do tamanho do JSON dos dados apresentados na página, conforme o número de músicas mostradas na página.

Músicas	**HTML** (KB)	**HTML.zip** (KB)	**JSON** (KB)	**JSON.zip** (KB)
50	31	6,9	3	0,35
100	49	8,9	6	0,55
200	85	12,8	11	0,98
300	122	16,8	17	1,23
400	158	20,7	22	1,50
500	193	24,6	28	1,82

Tabela 1.1: Variação do tamanho do `HTML` e `JSON` de dados, com e sem compressão, transferidos no carregamento da página de listagem de x músicas, da aplicação Biblioteca Musical desenvolvida na plataforma *OutSystems*.

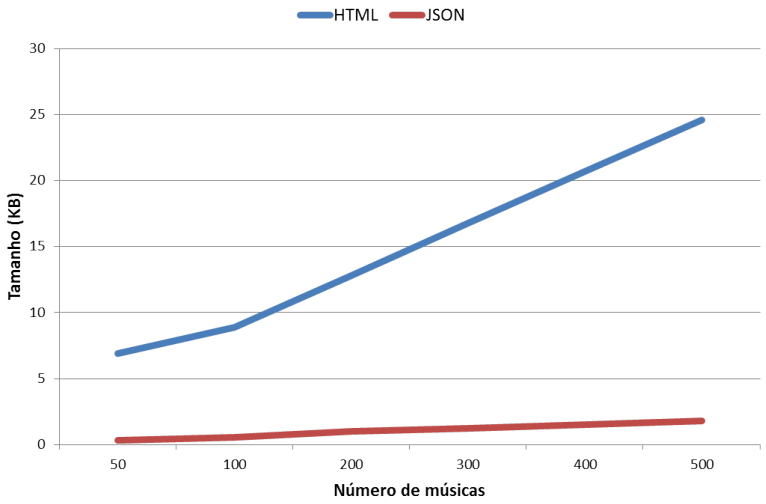


Figura 1.5: Variação dos tamanhos comprimidos do `HTML` e `JSON` apresentados na Tabela 1.1.

O tamanho do conteúdo que é na realidade transmitido na rede corresponde aos valores com compressão. Os valores sem compressão permitem-nos confirmar a redundância existente nas páginas `HTML` geradas pela *OutSystems Platform*, com uma elevada taxa de compressão (Tabela 1.1), como se pode confirmar pelos atributos existentes em todos os seus elementos, inclusive ID's com mais de 30 caracteres, necessários para manter o funcionamento da aplicação. Assim, justifica-se a diferença para o tamanho do `JSON` de dados, que constitui um menor volume de informação transmitida comparativamente à composição `HTML`.

Na Figura 1.5 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.

Note-se que a Figura 1.5 apresenta os volumes de conteúdo transmitidos na rede após

Músicas	**HTML** (KB)	**JSON** (KB)
50	552	32
100	620	64
200	756	128
300	892	191
400	1028	259
500	1164	327

Tabela 1.2: Tamanho do `HTML` e `JSON` de dados, transferidos no carregamento da página das músicas favoritas (variando o número de músicas) do utilizador na aplicação Last FM.

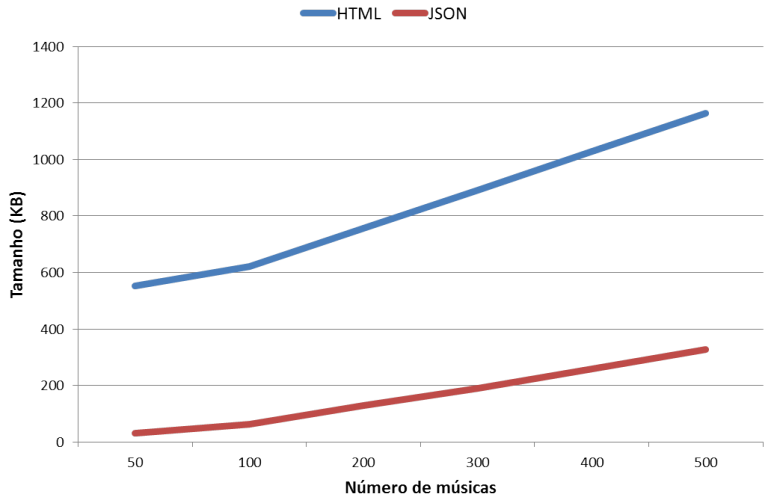


Figura 1.6: Variação dos tamanhos do `HTML` e `JSON` apresentados na Tabela 1.2.

o primeiro pedido da página, não sendo por isso contabilizado o tamanho do *template* tipo cliente, *scripts* necessários e outros ficheiros estáticos, que são fornecidos no primeiro pedido e depois mantidos em cache.

A aplicação LastFM, que permite aos utilizadores organizarem a sua biblioteca musical, foi também submetida a este teste, a fim de estudar os resultados de uma aplicação que não é gerada pela plataforma *OutSystems*. O `JSON` dos dados apresentados pela listagem de músicas foi obtido através da `API REST` disponibilizada pela aplicação LastFM, alterando o parâmetro do número de músicas. Tal como foi apresentado para a aplicação da *OutSystems*, a Tabela 1.2 e a Figura 1.6 representam os valores do tamanho do `HTML` e `JSON` de dados.

Ambos os gráficos apresentam uma redução média de 88% na comparação do tamanho da página `HTML` e o `JSON` dos dados apresentados. Isso comprova a redução do consumo de largura de banda que se obtém numa arquitetura onde o cliente gera as páginas `HTML`.

As aplicações *OutSystems* já são otimizadas a dois níveis: na transmissão de dados, na medida em que o *ViewState* é gerado com base nos dados necessários por todas as ações do ecrã, e na execução de operações das ações. Por exemplo, quando é efetuado um *append* sobre uma lista da página, é enviado apenas o novo elemento em vez de uma lista nova.

Em resumo, o desafio proposto é desenvolver um modelo de interação cliente-servidor que otimize (ainda mais) as aplicações *OutSystems* em dois vetores: fluidez da interação e redução do volume de dados transmitidos. Um requisito importante a ter em conta é que o processo de otimização proposto não pode causar alterações significativas na linguagem corrente e permitir otimizar aplicações já desenvolvidas simplesmente através de um novo *deployment*.

1.2 Solução proposta

A plataforma *OutSystems* utiliza uma representação abstrata unificada das aplicações, resultado de uma linguagem de programação integrada, que é tratada centralmente, e transformada em componentes concretos de tecnologias diversas. Este contexto permite-nos explorar variações da arquitetura das aplicações geradas utilizando o mesmo modelo abstrato, promovendo um maior desacoplamento das aplicações, e introduzindo de forma implícita a comunicação assíncrona entre cliente e servidor. Esta abordagem não causa impacto significativo no processo de desenvolvimento.

A solução proposta foi idealizada a partir da análise da variação dos volumes de conteúdo transmitidos na rede, apresentada na Secção 1.1, quando se comparou uma possível utilização de *templates* tipo cliente, que delega no cliente a geração das páginas concretas `HTML`, e onde o servidor fornece apenas os dados, com a abordagem tradicional.

As alterações necessárias para implementar esta arquitetura devem ser efetuadas ao nível do *OutSystems Compiler*, que gera o código das aplicações. Essas alterações envolvem a geração de *templates* tipo cliente, com base numa *framework* `JavaScript` já existente, que forneça um mecanismo de *templating*. Envolve também a geração dos *scripts* do cliente e do servidor, onde é definida a transmissão de dados nos ciclos de pedido-resposta. Perante um pedido ao servidor, devem ser fornecidos os dados necessários num formato reconhecido pelo cliente (e.g., `JSON`).

O desafio de otimizar as aplicações desenvolvidas atualmente, ao nível da redução dos dados transmitidos, conduz a alguns aspetos de eficiência e segurança que merecem especial atenção. Em particular, o envio dos dados ao cliente, deve ser efetuado de forma a que não sejam transmitidos dados desnecessários, ou que possam comprometer a segurança da aplicação, e cuja omissão permite uma maior eficiência. Para isso, quando se sucede o *deployment* de uma aplicação, durante o processo de compilação, deve ser realizada uma análise dos ecrãs da aplicação, com o intuito de otimizar a transmissão de dados. A análise deve identificar quais os dados estritamente necessários no cliente, com base naqueles que são consumidos pelo *template* tipo cliente, e os que são necessários para

a execução de lógica da aplicação. Sem esta análise, o código é gerado com base no pior caso, ou seja, fornecendo todos os dados ao cliente, incluindo os que não são necessários.

Note-se que o *template* tipo cliente pode conter parâmetros com o cálculo de expressões, que por sua vez usam dados, os quais devem por isso ser transmitidos na rede. Considere-se o exemplo da expressão *email != null*, onde *email* está contido no conjunto de dados necessários e que são transmitidos na rede. No entanto, o *email* é um dado confidencial, logo não é desejável a sua transmissão na rede, pois causa uma quebra de segurança. A análise efetuada aos ecrãs deve, então, identificar todas as expressões que necessitam de ser calculadas, para que o seu cálculo seja efetuado no cliente, e os parâmetros do *template* recebam os valores já calculados. Esta problemática não existe no modelo atual da *OutSystems*, onde todas as expressões que constam nos ecrãs são calculadas no servidor, e os seus valores estão explícitos no `HTML` gerado. É importante manter esta semântica, e não comprometer a segurança da aplicação ao transmitir as subcomponentes das expressões. A separação entre a lógica da aplicação e a interface torna-se também mais clara com a isolação do cálculo de expressões, não sendo incluído no *template*.

O componente *OutSystems Compiler*, incluído na arquitetura da plataforma apresentada na Figura 1.1, é um projeto de grande dimensão e complexidade, pelo que explorar uma solução neste contexto demonstrou-se não ser viável nem produtivo, pois seria necessário resolver muitos detalhes técnicos fora do âmbito deste trabalho. Este trabalho, é exploratório também no sentido de definir uma metodologia de prototipagem, que se pretende servir de base a desenvolvimentos futuros na plataforma *OutSystems*.

O protótipo criado inclui um modelo de um fragmento da linguagem *OutSystems*, que permite a criação de uma solução completa, objetiva, e validável. Esse modelo preserva a separação entre os dados apresentados nas páginas geradas e o código correspondente à sua estrutura e apresentação.

O desenho do protótipo consiste na criação de uma linguagem que permite definir o modelo de uma aplicação. É implementado o respetivo compilador, que gera separadamente: *templates* tipo cliente e *scripts* para o cliente e servidor, onde se executa a lógica da aplicação e se otimiza a transmissão de dados entre o cliente e servidor.

As aplicações produzidas usando o protótipo apresentam dois MVC's, um no cliente e outro no servidor. O primeiro deriva do uso da *framework* `AngularJS` que suporta um mecanismo de *templating* tipo cliente, e cujas aplicações apresentam um padrão arquitetural MVC. Isso permite que as páginas `HTML` sejam construídas no cliente, a partir dos dados transmitidos numa representação abstrata (`JSON`), e atribuir a execução de alguma lógica da aplicação ao cliente. Ao contrário do que acontece atualmente na plataforma *OutSystems*, a linguagem criada no protótipo permite declarar ações executáveis no cliente, evitando pedidos desnecessários. O MVC no servidor destina-se essencialmente a executar lógica da aplicação, e fornecer os dados necessários ao cliente. A preparação desses dados corresponde à camada *View*, a interface para a comunicação com o MVC no cliente.

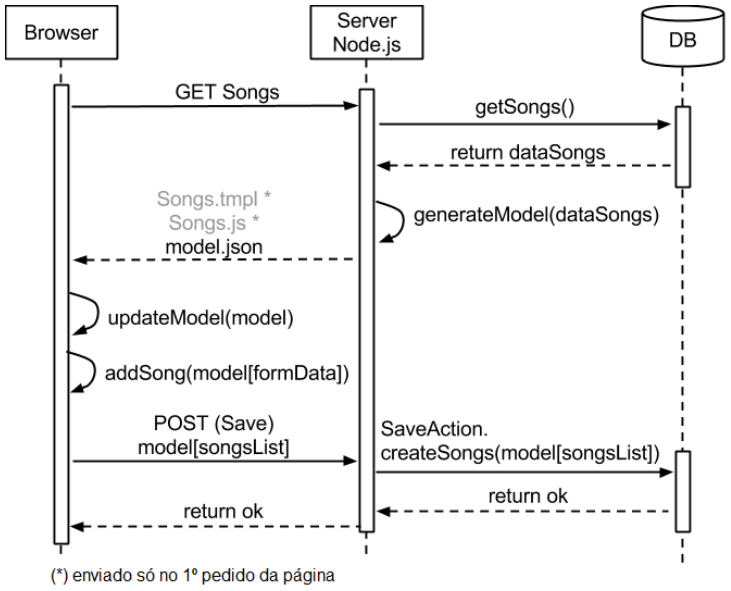


Figura 1.7: Comunicação cliente/servidor numa aplicação desenvolvida sobre o modelo proposto.

Para as aplicações desenvolvidas neste protótipo optou-se por usar um comportamento aproximado às aplicações da *OutSystems*, na medida em que o servidor não mantém qualquer estado da aplicação. Quando uma página é solicitada o servidor fornece um conjunto de dados, que é mantido na camada *Model* no cliente. Nos pedidos seguintes são enviados de novo os dados para o servidor, onde é executada a lógica da aplicação sobre estes, e que são enviados de volta para o cliente. Assim, os dois MVC's não são independentes, pois o estado da camada *Model* de ambos é partilhada através da transmissão dos dados na rede.

A arquitetura instanciada por este modelo é ilustrada na Figura 1.7, com algumas diferenças da Figura 1.2, devido às otimizações implementadas no protótipo, em contraste com o modelo atual da *OutSystems*. Na Figura 1.7 omite-se o envio do *template* tipo cliente e o *script*, efetuado na primeira vez que o cliente solicita uma página. Após esses artefactos serem obtidos, o cliente solicita os dados ao servidor, representados por model.json que contém a estrutura do modelo de dados do ecrã. A ação addSong ilustra a execução de lógica da aplicação no cliente, que neste caso adiciona uma música ao nível da lista na página `HTML`. Caso se deseje adicionar novas músicas à base de dados deve ser invocada a ação Save.

As ações, como a addSong da Figura 1.7, realizam operações de leitura e escrita sobre o modelo de dados mantido no cliente. Esse modelo de dados é consumido pelo *template* tipo cliente, a fim de construir a página `HTML`. Quando existem alterações no modelo de dados, a página `HTML` é automaticamente atualizada através do mecanismo de *templating*. As alterações ao modelo podem ocorrer, não só por ações no cliente, como também por ações no servidor que forneçam, como resposta, um modelo de dados atualizado.

A transmissão do modelo de dados entre cliente e servidor, presente na Figura 1.7, deve ser otimizada pelas razões já referidas, para que não inclua a estrutura completa dos dados do ecrã. A otimização é concretizada pela aplicação de duas análises estáticas do fluxo de dados sobre o modelo de uma aplicação. Essas análises são definidas utilizando `Datalog` [16] no compilador do protótipo apresentado neste trabalho. Assim, durante o processo de geração de código, é identificado o sub conjunto mínimo de dados que deve ser transmitido na invocação de cada ação, tanto do cliente para o servidor, como do servidor para o cliente, a fim de atualizar o modelo de dados equivalente ao estado da aplicação, mantido no cliente após o primeiro pedido.

As duas análises aplicadas sobre o modelo são:

Análise de *liveness*

A análise de *liveness* identifica os dados que são necessários transmitir entre cliente e servidor. No primeiro pedido de uma página devem ser considerados os dados consumidos pelo *template* tipo cliente, e os dados usados para a execução de qualquer ação. Nos pedidos seguintes relacionados com a invocação de uma ação no servidor, devem ser considerados os dados que o servidor necessita para executar essa ação em particular, transmitidos do cliente para o servidor. Deve também ser considerados, quais os dados que foram manipulados e são necessários para a execução de qualquer ação, transmitidos como resposta ao pedido do cliente.

Análise de dependências

A análise de dependências identifica as expressões estritamente necessárias de recalcular no fim da ação que é executada, com base nas variáveis alteradas de que as expressões dependem. A necessidade de calcular expressões advém do envio de dados que possam comprometer a segurança da aplicação, como já foi referido anteriormente. O *template* tipo cliente é gerado para que não contenha o cálculo de quaisquer expressões, em vez disso, contém os parâmetros para os valores das expressões calculadas, que devem ser enviadas do servidor para o cliente após a execução de uma ação. Relativamente às ações executadas no cliente, não está subjacente qualquer transmissão de dados entre cliente e servidor, no entanto a análise anterior considera os dados que são necessários para o cálculo das expressões efetuado no fim das ações no cliente.

Relativamente a ações executadas no cliente, o programador deve estar ciente das suas implicações, em particular quais as expressões que vão ser reavaliadas no fim dessa ação pois podem causar o envio indesejado de dados para o cliente.

Com os resultados de ambas as análises conseguimos minimizar o conjunto de dados transmitido na rede, dependendo da ação que é executada.

1.3 Contribuições

Este trabalho compromete-se em otimizar as aplicações geradas pela plataforma *OutSystems*, com o requisito de não alterar a linguagem sobre a qual são desenvolvidas atualmente. Esse requisito é cumprido, uma vez que existe uma camada de abstração entre o processo de desenvolvimento das aplicações, e o código gerado pelo *OutSystems Compiler* a partir da definição do modelo de uma aplicação.

O desafio deste trabalho apresenta-se em dois vetores: obter mais fluidez na interação das aplicações geradas pela plataforma *OutSystems*; e reduzir o volume de dados transmitidos entre cliente e servidor. Para concretizar a solução para este desafio foi aplicada uma metodologia de prototipagem, que obteve contribuições diretamente relacionadas com os vetores referidos.

O protótipo consiste na criação de uma linguagem, representativa de um fragmento da linguagem *OutSystems* para definir o modelo de uma aplicação. A partir da definição do modelo, o compilador (implementado também neste protótipo) gera aplicações com uma nova arquitetura, em contraste com a arquitetura gerada atualmente para as aplicações desenvolvidas na plataforma *OutSystems*.

A nova arquitetura produzida apresenta um MVC no cliente e outro no servidor. O MVC no cliente permite delegar a geração das páginas `HTML` no cliente, e distribuir a execução de lógica da aplicação entre cliente e servidor. A primeira contribui para a redução do volume de conteúdo transmitido na rede, pois em vez do envio de páginas completas `HTML`, são enviados dados "puros". Assim, é excluída toda a informação estática relativa à estrutura e comportamento da página, que é transmitida atualmente nas aplicações geradas pela plataforma. A segunda, reduz o número de pedidos desnecessários ao servidor, contribuindo assim para a fluidez da aplicação.

A linguagem criada é estendida com a declaração de ações executáveis no cliente ou no servidor. Como no cliente é mantido um modelo de dados, fornecido inicialmente pelo servidor, torna-se possível aplicar operações sobre o modelo. Quando é invocada uma ação do servidor, o modelo é-lhe fornecido, e as operações são executadas sobre esse. Como resposta, o servidor envia ao cliente um modelo atualizado, perante o qual, o mecanismo de *templating* no cliente atualiza automaticamente a interface.

Os dois MVC partilham o mesmo modelo de dados, primeiro, porque o servidor não mantém qualquer estado (estrutura de dados do ecrã), segundo, porque o cliente pode manipular o estado através das ações no cliente. Como tal, o modelo necessita de ser transmitido na rede em todos os ciclos de pedido-resposta, o que não é de todo desejável, pois pretende-se reduzir o volume de conteúdo transmitido na rede.

A transmissão do modelo de dados é otimizada através da aplicação de duas análises estáticas sobre a definição do modelo de uma aplicação. A primeira análise, de *liveness*, permite identificar os dados estritamente necessários a transmitir entre cliente e servidor, dependente do contexto, ou seja a identificação é feita ao nível de cada ação. Tanto o cliente como o servidor, conhecem os dados estritamente necessários a enviar no ciclo

de pedido-resposta, com base na ação invocada. A segunda análise, de dependências, identifica as expressões que devem ser recalculadas com base nos dados manipulados após a execução de uma ação, e dos quais as expressões dependem. Assim, apenas as expressões que devem ser recalculadas são transmitidas na rede.

A informação obtida através das duas análises contribui para se obter um estado diferencial do ecrã transmitido entre cliente e servidor, conforme a ação invocada. Isto substitui o campo *ViewState* transmitido atualmente entre cliente e servidor nas aplicações geradas pela plataforma *OutSystems*, que contém sempre o mesmo conjunto de dados. O *ViewState* inclui os dados necessários por todas as ações.

As contribuições deste trabalho foram publicadas no simpósio de informática INForum 2014, realizado na Universidade do Porto, através do artigo [5], o qual esteve entre os nomeados para melhor artigo.

Em suma, este trabalho contribui com um modelo simplificado das aplicações da *OutSystems*, a partir do qual são produzidas aplicações com dois MVC's, que simplificam a comunicação assíncrona entre cliente e servidor. O compilador implementado gera automaticamente aplicações otimizadas ao nível da transmissão de dados subjacente da comunicação assíncrona. A otimização baseia-se na aplicação de análise estática sobre a definição do modelo de uma aplicação durante o processo de compilação.

1.4 Estrutura do documento

Conhecendo o problema e parte da solução proposta, apresentamos os capítulos em que este documento se estrutura:

Capítulo 2 Pretende contextualizar, com mais detalhe, o leitor sobre a *OutSystems Platform*, tanto a nível do processo de desenvolvimento das aplicações, como da geração de código e execução.

Capítulo 3 Apresenta o trabalho relacionado que é repartido em três secções: a primeira sobre os mecanismos de *templating*, aborda a importância de separação entre lógica aplicacional e a apresentação, características das linguagens de *templating* e o *trade-off* do particionamento de funcionalidade entre cliente e servidor; a segunda apresenta a aplicação do `Datalog` na análise estática de programas; a terceira apresenta o uso de *templates* tipo cliente em aplicações da indústria atual; e a última apresenta um estudo sobre tecnologias de *templating* destinadas ao cliente.

Capítulo 4 Apresenta-se a linguagem criada neste trabalho que permite definir o modelo de uma aplicação.

Capítulo 5 A comunicação entre cliente e servidor das aplicações definidas sobre o modelo apresentado no capítulo 4 envolve a transmissão de dados que, se deseja otimizar. Neste capítulo é apresentada a análise estática que nos permite obter o conjunto mínimo de dados necessários de transmitir na rede.

Capítulo 6 Apresenta a implementação do compilador responsável pela definição da arquitetura das aplicações produzidas no fim.

Capítulo 7 e 8 Apresentam os resultados obtidos com a abordagem apresentada ao longo do documento, as conclusões e o trabalho futuro.

2

A Plataforma OutSystems

A plataforma *OutSystems* permite o desenvolvimento rápido de aplicações web e tem como objetivo aumentar consideravelmente a produtividade das equipas de desenvolvimento de *software*, quando comparada com outras ferramentas mais tradicionais. É formada por vários componentes, destacando-se o *Service Studio*, o ambiente de desenvolvimento das aplicações, e o *Platform Server*, responsável pela compilação das aplicações, publicação e manutenção das versões das aplicações já publicadas.

Neste capítulo contextualizamos o leitor sobre o processo de desenvolvimento de aplicações na plataforma *OutSystems*. A seguir apresentamos o processo de geração de código e o ciclo de vida das aplicações quando executadas.

2.1 Ambiente de desenvolvimento

O *Service Studio* é o ambiente de desenvolvimento que, através de uma linguagem visual, permite aos programadores desenharem uma aplicação desde contendo processos, interface do utilizador, lógica funcional, e dados.

O desenho da interface do utilizador corresponde à definição de ecrãs, que representam as páginas da aplicação. A definição de um ecrã envolve adicionar texto e *widgets*. As últimas podem representar elementos para inserir conteúdo dinâmico nas páginas (e.g., expressões, condições, campos de *input*), ou elementos para estruturar o conteúdo das páginas (e.g., tabelas, formulários, *containers*).

As *widgets* são traduzidas em elementos `HTML` quando a página é gerada para o cliente. Estas *widgets* contêm propriedades, simples ou dinâmicas, que permitem ligar a componente visual aos dados da aplicação. As propriedades simples estão associadas, essencialmente, à interface. No momento em que as *widgets* são inseridas no ecrã, são

inicializadas com referência a valores literais, variáveis, ou até a resultados de consultas à base de dados. As propriedades dinâmicas são afetadas no corpo das ações, ou seja, em tempo de execução. Associada a cada ecrã existe uma ação especial, a ação *Preparation* que obtém todos os dados necessários. Esta ação 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. É possível em cada ecrã definir variáveis locais, parâmetros ou executar consultas ao modelo de dados. Se uma propriedade de uma *widget* for afetada ao longo desta ação, essa afetação não será visível na interface devido ao facto de a *widget* só ser instanciada quando é criada a página, ou seja, a seguir à execução da *Preparation*, o que leva a que a afetação seja esquecida.

Associado a cada ecrã existe um conjunto de ações (de ecrã) que adicionam funcionalidade à aplicação, e são normalmente usadas para definir o comportamento da aplicação em resposta à interação do utilizador. Considere-se o exemplo da aplicação Biblioteca Musical, semelhante à que é implementada na Secção 3.3.3, em particular a página que apresenta a listagem de músicas e um formulário onde o utilizador pode adicionar uma nova música. Após inserir os dados, o utilizador deve premir o botão que submete os dados ao servidor. O botão neste contexto é uma *widget*, à qual é associada uma ação do ecrã. Quando o utilizador prime o botão, é desencadeado um pedido que executa a lógica da ação. Existem três tipos de pedidos:

Normal Consiste na execução da respetiva ação do ecrã, seguindo-se da ação *Preparation* e geração da página que é enviada para o cliente.

AJAX É executada a ação do ecrã, mas a ação *Preparation* não é executada. Apenas algumas partes da página são geradas e enviadas para o cliente.

Navegação Em alternativa aos dois anteriores, o pedido pode desencadear a navegação direta para outro ecrã, não sendo executada qualquer ação especial, somente a ação *Preparation* do ecrã para o qual se navega, e consequentemente a página é igualmente gerada.

Na definição de uma ação é possível criar ou repetir consultas ao modelo de dados, afetar variáveis, inserir condições, ciclos, atualizar *widgets* do ecrã, redirecionar para outro ecrã, ou ainda chamar outras ações relativas à lógica do funcionamento.

Normalmente, uma ação desencadeada por um pedido `AJAX` destina-se à atualização de dados e consequentemente das *widgets* do ecrã que apresentam esses dados.

2.2 Geração

Ao longo do desenvolvimento de uma aplicação, o *Service Studio* permite publicá-la através da funcionalidade *1 Click Publish*, que valida o modelo da aplicação e gera o código

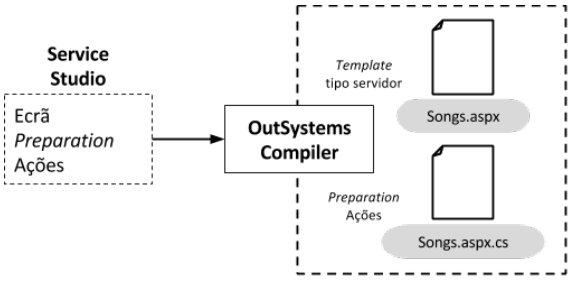


Figura 2.1: Artefactos gerados pelo *OutSystems Compiler*.

de uma aplicação em `ASP.NET` ou `JSP`, modelada no padrão MVC, usando *templates* tipo servidor.

A geração do código é realizada pelo compilador *OutSystems*, um subcomponente do *Platform Server*, que compila o modelo da aplicação desenvolvida. O processo de compilação, ilustrado na Figura 2.1, gera vários ficheiros particulares a um ecrã, entre os quais se encontra o *template* do ecrã, ou os *templates* que definem os blocos partilhados entre ecrãs (e.g., cabeçalho, rodapé, menu). Associado ao *template* está o código que implementa a ação *Preparation* e outras ações do ecrã.

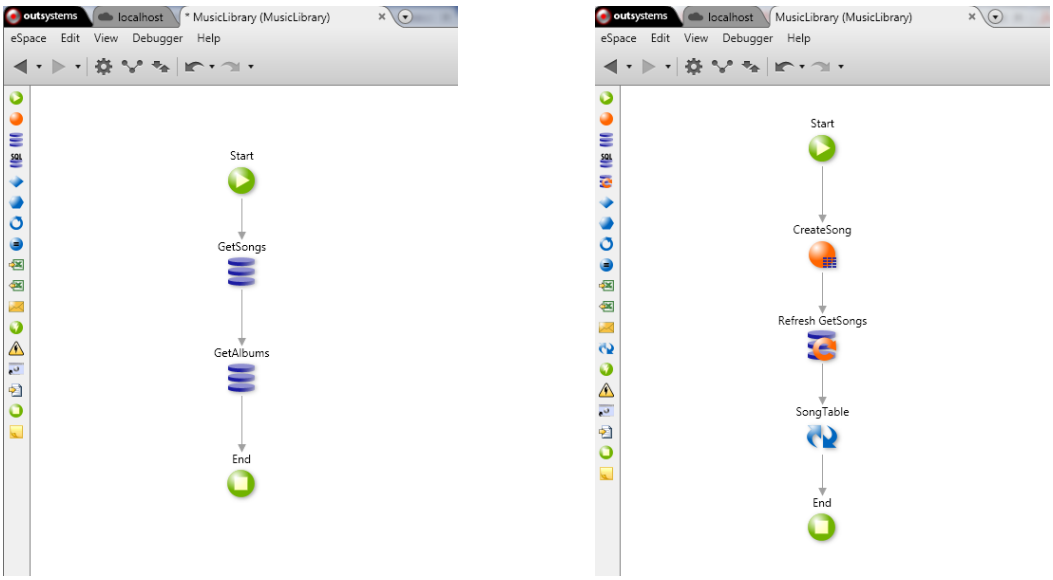
O processo de geração de código é baseado em *visitors*, um padrão de desenho aplicado quando se deseja definir uma operação para uma estrutura de vários tipos de objetos, sem alterar os próprios objetos. O conjunto de elementos de uma aplicação formam uma estrutura complexa de objetos, sobre a qual é aplicada o algoritmo de geração de código, cuja implementação difere entre elementos. No contexto da geração do *template* de um ecrã, os *visitors* permitem delegar no controlador de cada *widget* a geração do respetivo `HTML`.

Na listagem D.1 (em anexo) é apresentada uma simplificação do código gerado relativo à ação *Preparation* ilustrada em 2.2(a), onde se pretende mostrar uma visão geral do que é executado, a fim de obter os dados de uma página. A classe *ScrnSongs* engloba todo o código respetivo ao ecrã de músicas, incluindo outras classes internas que se destinam à execução das consultas das ações do ecrã, como a *FuncssPreparation* que implementa as consultas *getSongs* e *getAlbums* da ação *Preparation*. Na linha 20 da listagem D.1 a consulta à base de dados corresponde à execução do comando que foi definido na string *sql*, e os resultados obtidos são colocados na variável *outParamList*.

2.3 Execução

Na fase de execução de uma aplicação, o ciclo de vida de um ecrã é iniciado na execução da ação *Preparation*, sempre que o cliente solicita a página. Seguidamente, o servidor envia o `HTML` gerado e as ações do ecrã são executadas em sequência dos pedidos desencadeados pela interação do utilizador.

A Figura 2.3 ilustra a comunicação entre o cliente e o servidor, na sequência do pedido



(a) Definição da ação *Preparation* no *Service Studio*.

(b) Definição da ação de adição de uma música no *Service Studio*.

Figura 2.2: Definição de ações no *Service Studio*.

da página de músicas da aplicação Biblioteca Musical (consideremos que esta página contém também uma listagem dos álbuns), e do pedido desencadeado pela adição de uma nova música. Quando a página é solicitada, o servidor executa o código respetivo à ação *Preparation*, onde é efetuada a consulta de dados (*query*) *getSongs* e a *getAlbums*, que através da comunicação com a base de dados obtém os resultados (*dataSongs* e *dataAlbums*). O servidor procede então à geração da página `HTML` com base no *template Songs.aspx* e nos dados obtidos, que inicializam as variáveis usadas nas expressões invocadas pelo *template*. O cliente recebe um documento `HTML` pronto a ser visualizado, e o estado do ecrã representado no campo *ViewState* (ver Secção 2.4), onde estão contidos os valores de variáveis e propriedades de *widgets*. São ainda enviados outros ficheiros, tais como *scripts* que definem o comportamento das *widgets*, `CSS`'s, imagens visualizadas na páginas, que não são referidos porque estão fora do âmbito deste trabalho. Todos os ficheiros são mantidos em cache no cliente, exceto o `HTML`, pois é gerado dinamicamente pelo servidor, e o seu conteúdo pode alterar entre pedidos.

A Figura 2.3 retrata também um pedido `AJAX` que deriva da adição de uma nova música, onde o cliente envia os dados do formulário e o *ViewState*. O servidor executa o código da ação do ecrã relativa à adição de uma música, que consequentemente comunica com a base de dados para adicionar a nova música. De seguida, é necessário atualizar a tabela de músicas na página, para que o cliente apresente a nova música. Por isso a *query getSongs* da *Preparation* é repetida obtendo-se os dados atualizados (esta repetição reflete-se no *Service Studio* indicando a atualização da *query*, visível na Figura 2.2(b)). Após obter o conjunto de dados, é gerado o fragmento `HTML` correspondente à tabela de músicas, que é enviado ao cliente, substituindo o fragmento existente no documento

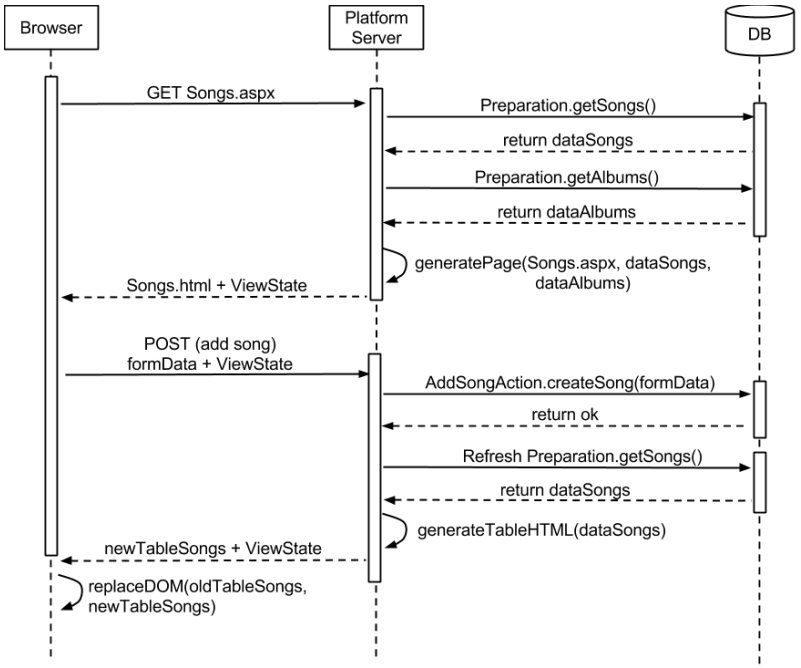


Figura 2.3: Comunicação entre cliente e servidor na arquitetura atual, na sequência do pedido da página de músicas e da adição de uma nova música.

`HTML`.

Sendo um pedido `AJAX`, a ação *Preparation* não é executada, já que a página não é recarregada na sua totalidade, sendo atualizadas apenas algumas partes da página consoante o que o servidor gerou da ação desencadeada pelo pedido.

2.4 *ViewState*

Para introduzir o conceito do *ViewState* existente nas aplicações geradas pela plataforma *OutSystems* considere-se o seguinte exemplo: uma aplicação para registar o tempo da conclusão de um conjunto de tarefas por parte do utilizador com suporte de *multi-tasking*. Portanto, se o utilizador tiver várias páginas abertas no navegador, a contagem do tempo deve ser individual por cada página. Para que o servidor contabilize o tempo da conclusão do conjunto de tarefas, é necessário manter a que hora o utilizador iniciou a sua realização. Por isso, durante o processo de desenvolvimento da aplicação, é criada uma variável e inicializada com a hora do momento em que a ação Preparação é executada, correspondente à hora que o utilizador solicita a página do ecrã da aplicação. Quando o utilizador submeter ao servidor que concluiu todas as tarefas, é necessário efetuar o cálculo do tempo total, que consiste em efetuar a diferença entre a hora inicial que foi guardada na variável do ecrã e a hora atual. É portanto necessário preservar o valor da variável entre os pedidos inicial e final.

Note-se que a plataforma *OutSystems* suporta o conceito de sessão de um utilizador

no servidor, para manter o respetivo estado global da sessão. No entanto, no nosso exemplo pretendemos manter um valor independente por instância da página: abrindo várias *tabs* da mesma página, estas serão associadas à mesma sessão, o que não é o que se pretende.

Apesar de ser possível generalizar o conceito de sessão para poder distinguir diferentes instâncias de uma mesma página, tal não é muito interessante: implica aumentar a quantidade de informação a armazenar no servidor durante um tempo potencialmente longo, uma vez que não é possível antecipar se irá ou não ocorrer um novo pedido para a mesma página.

Surge então o conceito *ViewState*, um mecanismo criado pelo `ASP.NET`, que ajuda o servidor a preservar o estado de uma página. Ao contrário do mecanismo de sessão que mantém os valores no servidor, o *ViewState* é um dicionário (cifrado) que é enviado ao cliente juntamente com o `HTML` da página. Em pedidos consecutivos à mesma página o *ViewState* é enviado como parte desse pedido, permitindo assim ao servidor recuperar o estado.

O facto do servidor não manter qualquer estado é vantajoso, na medida em que se torna mais fácil escalar horizontalmente uma aplicação. Uma aplicação pode ser alocada em vários servidores, e os pedidos do cliente podem ser submetidos a qualquer um. Não existe, por isso, qualquer preocupação sobre prevalência de estado da instância de um ecrã entre servidores, pois o *ViewState* é transmitido em todos os pedidos.

Em *OutSystems*, é importante realçar que a criação do *ViewState* é totalmente transparente no desenvolvimento de uma aplicação no *Service Studio*, uma vez que a sua definição surge na geração do código.

2.5 Atualização da interface

A linguagem visual da *OutSystems* cria uma camada de abstração sobre vários detalhes de desenvolvimento, com o intuito de transformar o processo de desenvolvimento simples e intuitivo para o utilizador. O código das aplicações geradas após o *deployment* é transparente ao longo do desenvolvimento de uma aplicação no *Service Studio*. Ainda assim, existem várias melhorias que podem ser implementadas, algumas das quais detetadas ao longo deste trabalho.

Como a linguagem apresentada neste trabalho é inspirada na linguagem *OutSystems*, é necessário explicar o comportamento da operação AjaxRefresh usada para especificar que elementos da interface devem ser atualizados ao longo, ou no fim, de uma ação do ecrã.

A operação AjaxRefresh, invocada numa ação associada a um ecrã, consiste em atualizar uma *widget* desse ecrã com novos dados alterados ao longo da execução dessa ação. No entanto, após vários testes identificou-se que o comportamento desta operação não é uniforme para todas as *widgets*.

Normalmente, o comportamento da operação AjaxRefresh consiste em duas fases: instanciar a *widget* com os dados atualizados, e gerar o respetivo `HTML` para substituir na página `HTML` no cliente. Esta operação é usada para implementar pedidos assíncronos num ecrã e permite atualizar zonas da página, evitando o seu total carregamento.

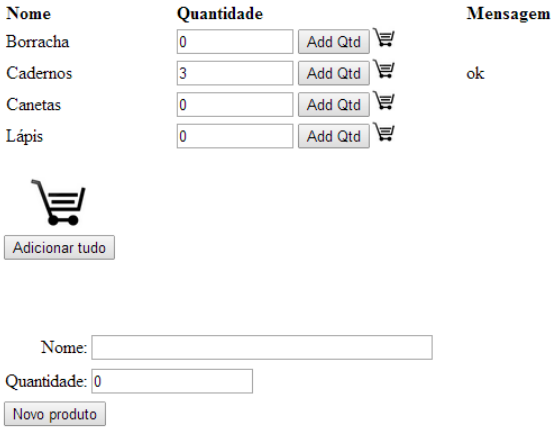


Figura 2.4: Aplicação com listagem de produtos.

Considere-se a aplicação apresentada na Figura 2.4, composta por um ecrã que lista um conjunto de produtos, e um formulário que permite adicionar produtos. Para cada linha existe um *input* onde podemos especificar a quantidade desse produto, um botão para adicionar o pedido ao carrinho e, no fim da lista, um botão para adicionar todos os pedidos de uma só vez. Interagir com os botões do ecrã tem como *feedback* uma mensagem de sucesso, ou erro, conforme a quantidade de um produto é superior a zero ou não, respetivamente.

Quando premimos o botão, presente em cada linha da lista, é executada a sua ação, que verifica a quantidade inserida no *input*, e afeta a mensagem de *feedback* da linha corrente conforme o resultado do teste. No fim da ação desejamos que a interface seja atualizada com a mensagem de *feedback*. Por isso, ainda na ação, é invocada a operação AjaxRefresh da *widget* correspondente à lista, passando como parâmetro o número da linha corrente. Nesta situação o comportamento é o esperado.

Relativamente à ação atribuída ao botão presente no fim da lista, itera sobre a lista, verifica todos os produtos, e afeta a mensagem de cada linha. Desejamos que a interface seja atualizada com uma mensagem por cada linha. Logo invocamos a operação AjaxRefresh sobre cada linha da lista. Mas o comportamento não é o esperado neste cenário. Quando a lista é iterada o índice da linha corrente é atualizado, o que também acontece quando é desencadeado um evento numa linha da lista. Note-se que neste exemplo aplica-se o primeiro caso, e a partir do índice da linha corrente queremos aceder às *widgets* filhas nessa linha, nomeadamente os *inputs* com os novos dados, e às expressões que representam a mensagem de *feedback*. Contudo, no contexto de uma iteração, apenas é possível aceder aos dados consumidos pela lista, e nada mais, o que nos impossibilita de atualizar a interface devidamente.

Para além da problemática da noção de corrente numa *widget* com uma lista, existem outros comportamentos na operação AjaxRefresh cuja semântica é diferente.

Considere-se uma *widget* com uma propriedade dinâmica do tipo lista, e uma propriedade simples do mesmo tipo. Um dos pormenores do ambiente de execução das aplicações *OutSystems* é que uma *widget* que está ligada a um conjunto de dados tem também uma propriedade dinâmica que consiste numa cópia desses dados. A lista dinâmica para além de conter uma cópia, contém também as *widgets* filhas instanciadas em cada linha da *widget*. No corpo de uma ação pode ser manipulada a lista dinâmica, quando por exemplo se deseja fazer um *append* ao conjunto de dados apresentado interface. Mas, as propriedades das *widgets* filhas só podem ser manipuladas no contexto de uma ação ser desencadeada numa linha da lista. O que se sucede é que ao efetuar a operação AjaxRefresh sobre essa *widget* qualquer alteração sobre a lista dinâmica perde-se, pois neste contexto particular a operação em questão efetua uma nova cópia da lista de registos que é consumida. Em suma, qualquer alteração efetuada sobre a lista dinâmica perde-se, nomeadamente as *widgets* filhas que são instanciadas ao nível das linhas da lista, e os próprios dados apresentados.

Se a lista for manipulada ao nível dos dados que consome, mas apenas numa linha, deve-se usar a operação AjaxRefresh sobre a linha. Caso seja sobre toda a lista, as alterações devem ser efetuadas sobre a lista original de registos, e no fim invocar o AjaxRefresh sobre a lista, que vai efetuar uma cópia da nova lista de registos. No entanto, caso as alterações sejam efetuadas sobre propriedades dinâmicas das *widgets* filhas, essas alterações só permanecem caso sejam efetuadas sobre a linha corrente, no contexto de uma ação desencadeada numa linha da lista.

Com o protótipo desenvolvido neste trabalho pretende-se simplificar a linguagem *OutSystems*, e uniformizar ou até mesmo remover a operação `AjaxRefresh` no desenvolvimento das aplicações.

3

Trabalho relacionado

Neste capítulo é apresentada uma análise às linguagens de *templating*, que permitem a separação entre a apresentação e lógica aplicacional. Relacionado com os mesmos mecanismos, apresenta-se uma análise de custo e benefício sobre o particionamento da lógica aplicacional entre cliente e servidor.

Apresentam-se as vantagens da linguagem `Datalog` [16] aplicada à implementação de análise estática sobre programas. Essa linguagem foi usada neste trabalho para aplicar a análise estática sobre a definição do modelo de uma aplicação, a fim de minimizar o conjunto de dados transmitido entre cliente e servidor, em alternativa aos algoritmos de ponto fixo.

Como suporte à otimização das aplicações geradas pela *OutSystems Platform*, são analisados os casos na indústria que adotaram os *templates* tipo cliente, observando as suas motivações.

Por fim, apresenta-se uma comparação entre algumas tecnologias com mecanismos *templating* tipo cliente, com base numa aplicação exemplo.

3.1 *Templating*

Os *templates* aplicados ao contexto web são documentos que definem a interface com o utilizador através da declaração da estrutura estática das páginas `HTML`. Usam uma linguagem para a especificação parametrizada das páginas, independente do conteúdo concreto e da lógica aplicacional que produz esse conteúdo.

Parr formaliza, em [11], a separação entre a lógica aplicacional e a apresentação, através do estudo de mecanismos de *templating*. Para classificar e categorizar os *templates*, apresenta um conjunto de 5 regras para garantir a separação entre lógica da aplicação e

```
<table>
<% for ( int i = 0; i < n; i++ ) { %>
   <tr> <td> <%= i+1 %> </td> </tr>
<% } %>
</table>
```

Listagem 3.1: *Template* não restrito, definido em `JSP`, que gera uma tabela com *n* linhas, através de código `Java`.

interface, e como são aplicadas pelos *templates*. Parr identifica um problema: a maioria das *frameworks* que usam mecanismos de *templating*, como o `ASP.NET`, `JSP` ou `Ruby on Rails`, permitem inserção de código executável nos *templates*. Um *template* deve apenas representar a estrutura de uma página com base num conjunto de dados, sem envolver a execução de lógica da aplicação, mantendo assim a separação entre essa e a camada de apresentação. O isolamento dos dados é importante quando se deseja manter a coerência do modelo, e não comprometer a segurança da aplicação, como se explica ainda neste capítulo.

3.1.1 Classificação de *templates*

Parr define os *templates* como documentos que incluem ações interpretadas pelo mecanismo que gera a página `HTML`, e classifica-os como restritos e não restritos.

Templates não restritos

Um *template* não restrito baseia-se numa parametrização do conteúdo a visualizar e na inserção de código. Os *templates* definidos sobre a plataforma `JSP` são um exemplo de *templates* não restritos, onde é possível embeber código, como é exemplificado na Listagem 3.1.

Consequentemente, neste tipo de *templates* podem ser invocadas funções com efeitos secundários nos dados da aplicação, quebrando a separação entre a apresentação e lógica aplicacional.

Templates restritos

A definição de um *template* restrito consiste na parametrização de um conjunto de dados, fornecidos pela camada *Model* da aplicação web que tipicamente instancia o padrão MVC. A camada *Model* destina-se às operações sobre os dados, os quais são usados na instanciação do *template*, evitando a manipulação de dados na camada de apresentação. O *template* é instanciado por dados que podem ser individuais (e.g. 2), coletivos (e.g. [A,B,C,D]) ou até mesmo agregados (e.g. [nome=Sara, nac=Portugal]).

Os *templates* restritos são divididos em 3 classes:

```
<tr ng−repeat="song in songs">
  <td> {{song.songName}} </td>
</tr>
```

Listagem 3.2: *Template* restrito regular, que itera sobre conjunto de dados *songs* referindo a propriedade *songName* como o conteúdo a inserir, definido com a anotação do `AngularJS`.

```
<input type="checkbox" ng−model="checked" >
<p ng−if="checked" > Checked </p>
<p ng−if="!checked" > Unchecked </p>
```

Listagem 3.3: *Template* restrito e dependente do contexto, que altera o texto consoante o valor da *checkbox*, definido com a anotação do `AngularJS`.

1. **Regular -** Os *templates* regulares podem iterar sobre dados coletivos para gerar listas de conteúdo, podendo referir propriedades do *item* na iteração corrente, como é exemplificado na Listagem 3.2.

2. **Independente do contexto -** Estes *templates* acrescentam aos *templates* regulares a possibilidade de inserir outros sub-*templates*. Possibilitando a definição de uma estrutura da página em árvore.

3. **Dependente do contexto -** Adiciona à classe anterior a inserção de sub-*templates* dependendo do contexto. A Listagem 3.3 exemplifica um *template* desta classe. Através do parâmetro *checked*, que representa o valor booleano da *checkbox*, condiciona-se a visualização do texto *Checked* ou *Unchecked*.

3.1.2 Separação Lógica-Apresentação

Após a definição de *templates* restritos, Parr apresenta o conjunto de regras que garantem uma separação estrita entre a definição da interface (constituída pelos *templates*) e a lógica aplicacional.

1. O *template* não deve conter a invocação de funções que causem efeitos secundários nos dados da aplicação. Ou seja, apenas deve receber o conjunto de dados para instanciar a sua parametrização.

2. O *template* não deve conter o cálculo de expressões que dependem de dados fixos. Considere-se a expressão "*preço*0.90*", onde *preço* é um parâmetro do *template* e *0.90* é um dado fixo, simulando o preço de um produto em saldos. Se os saldos eventualmente terminarem e tivermos que alterar a aplicação, não queremos fazê-lo na camada de apresentação mas sim na de lógica. Por isso para existir independência entre *template* e lógica, o *template* deve apenas apresentar o valor final do cálculo

substituindo a expressão pelo parâmetro *preçoSaldo*, cujo valor é calculado no modelo.

3. De forma semelhante à regra anterior, o *template* não deve também conter expressões condicionais dependentes de dados fixos. Decisões como "*taxaAlcool<0.5*", para condicionar a visualização de informação que indica se a pessoa está apta a conduzir com base na sua taxa de alcoolemia. Tal como no exemplo anterior existe uma dependência da lógica aplicacional, pois o valor fixado como *0.5* pode alterar, o que mais uma vez não deve representar uma alteração ao *template*, mas sim à camada de lógica. A forma correta para permitir o *template* de condicionar a visualização de conteúdo é substituir a expressão por "*taxaAlcoolOk*", que representa o valor booleano da expressão condicional calculada no modelo.

4. O *template* não deve fazer suposições do tipo dos dados. Isto quer dizer que o *template* não deve usar os dados assumindo que os seus tipos não alteram. Observe-se a indexação de *arrays*, como por exemplo *nome[id]* em que *id* se assume como um inteiro não é possível. O *template* não deve, por isso, invocar métodos ou estruturas que recebam argumentos, pois na invocação é assumido um tipo do argumento. A solução passa por uma boa estruturação dos dados como objetos no modelo, a fim de serem corretamente parametrizados no *template*.

5. Os dados fornecidos ao *template* não devem conter informações sobre como apresentar o conteúdo. Ou seja, o *template* não deve obter dados como por exemplo a cor a aplicar ao fundo da página.

As regras 2 e 3, relativas ao cálculo de expressões, inclusive condicionais, são particularmente importantes quando se usam *templates* tipo cliente, a fim de não comprometer a segurança da aplicação. Considere-se o seguinte exemplo: um fórum onde os utilizadores escrevem as suas mensagens publicamente. Em cada mensagem existe um botão que permite o envio de uma mensagem privada, diretamente enviada para o email do utilizador que a escreveu (confidencial entre utilizadores). O *template* correspondente à página Web que contém a lista de mensagens de um dado tópico do fórum, deve conter uma expressão que condicione a visualização do botão para o envio da mensagem, dependente do facto do utilizador ter ou não o e-mail definido, i.e., *email != null*. A transição deste *template* para o cliente, leva a que o envio de dados contenha os e-mails para calcular o valor booleano da expressão. Este cenário quebra a confidencialidade dos e-mails dos utilizadores, uma vez que são transmitidos na rede, comprometendo a segurança da aplicação. Como tal, as expressões devem ser maioritariamente calculadas no servidor, enviando somente o valor final que é usado no cliente. Podem ainda assim existir expressões a ser calculadas no lado do cliente, que expressem o comportamento da interface, por exemplo, o número de *items* a visualizar numa lista.

Relativamente ao fluxo dos dados entre os *templates* e o modelo que implementa a lógica aplicacional através de operações sobre os dados, Parr apresenta duas estratégias:

```
1 <p> Names: $model.getNamesOfSongs() </p>
2 <p> There are $model.getNumberOfSongs() </p>
```

Listagem 3.4: *Template* com uma estratégia *pull*, que apresenta os nomes de todas as músicas e o seu número total.

Pull Nesta estratégia define que os dados usados no *template* são obtidos explicitamente invocando as funções do modelo, como é exemplificado na Listagem 3.4. Ou seja, todos os dados que são usados pelo *template* são obtidos no momento da página ser gerada, através da invocação das funções do modelo.

Push Nesta estratégia todos os dados são pré-calculados no modelo, e no momento de geração da página os dados estão disponíveis para o uso no *template*.

A estratégia de *pull* pode prejudicar o bom funcionamento da aplicação, mesmo que as funções invocadas não manipulem os dados. Considere-se o exemplo da Listagem 3.4, onde são invocadas as funções *getNamesOfSongs*, que devolve uma lista dos nomes das músicas, e *getNumberOfSongs*, que devolve o número de músicas. Se existir uma alteração a nível do *template* onde se pretende que o número de músicas seja apresentado antes da lista dos nomes, a ordem de invocação das funções será diferente. O que acontece é que a implementação da função que devolve o número de músicas, efetua o cálculo com base no tamanho da lista instanciada pela função *getNamesOfSongs*. Ou seja, uma vez que a lista ainda não foi calculada, é apresentado um número inválido, ou no pior caso existe uma exceção. A solução neste caso passaria por corrigir a ordem da invocação das funções, ou calcular o número de músicas sem depender do resultado de outras funções.

Portanto, a ordem da invocação das funções numa estratégia *pull* pode causar resultados diferentes na visualização de uma página. Um problema que é contornado pela estratégia de *push*, que calcula os dados antes do processo de geração da página, e assim a sua apresentação não depende da lógica de implementação das funções que os fornecem.

Perante o conjunto de regras e a classificação dos *templates*, Parr demonstra que é possível obter um mecanismo de *templating* obedecendo a uma separação estrita entre lógica aplicacional e a apresentação, excluindo a 5ª regra, pois é impossível determinar o significado dos dados, ou seja, se revelam alguma informação sobre a formatação da página web. Para provar isso, a abordagem apresentada por Parr em [11] consiste em, passo a passo, adicionar propriedades necessárias ao *template*:

1. Inicia-se com um *template* restrito regular, no qual é possível iterar um conjunto de dados, sem existir qualquer cálculo, ou condicionamento do conteúdo visualizado.

2. Para se reutilizar componentes do *template* (zonas da página Web, e.g., elementos de navegação), é necessária a noção de sub-*templates*, que nos permite partilhar uma estrutura entre as páginas web, através da especificação de inclusão de um *template*.

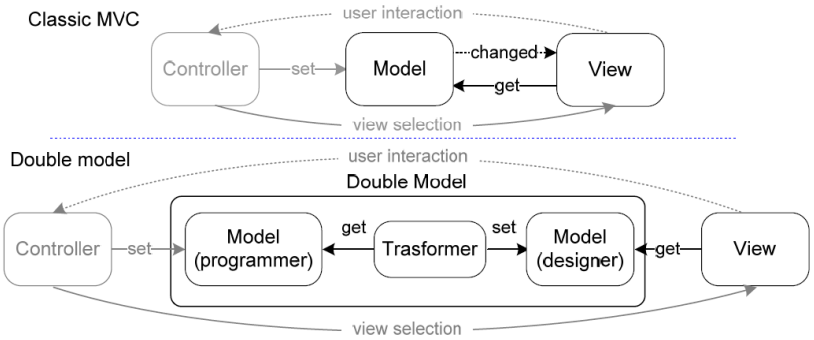


Figura 3.5: Comparação entre arquitetura do padrão MVC com o padrão de dois modelos (in [3]).

Este tipo de *templates* são restritos e independentes de contexto, com uma estrutura em árvore dos *templates*, mantendo a separação formalizada por Parr.

3. Falta, agora, o *template* ser capaz de condicionar o conteúdo visualizado, através de expressões que respeitam a regra 3. Assim, alcança-se a classe de *template* restrito e dependente do contexto, sem quebrar a separação entre a camada da apresentação e da lógica.

No contexto deste trabalho, é necessário reutilizar um mecanismo de *templating* para a implementação de uma arquitetura baseada em *templates* tipo cliente. Na secção 3.3 é apresentado o estudo de duas plataformas, `AngularJS` e `KnockoutJS`, que suportam esse mecanismo. Segundo a classificação definida por Parr, os *templates* definidos nestas plataformas são restritos e dependentes do contexto, o que é importante para a solução do problema apresentado neste documento, visto que as propriedades de um *template* restrito e dependente do contexto possibilitam reutilizar código, e representar toda a interface definida num ecrã no *Service Studio*.

Uma vez que a linguagem de *templating* destas plataformas se baseia em anotações no `HTML`, esta classificação só é viável se não considerarmos a inserção de código `JavaScript` no `HTML`.

O trabalho desenvolvido por Parr foi a base de outros trabalhos de investigação sobre os mecanismos de *templating*, com foco na separação entre a lógica aplicacional e a apresentação, como por exemplo, [3, 17] que passamos a descrever.

Garcia *et al*, introduziram uma abordagem, ilustrada na Figura 3.5, com o objetivo de proteger a aplicação de alterações indesejadas na visualização da página web e no servidor [3] . São criados dois modelos: um no cliente, destinado exclusivamente à apresentação da página, e outro no servidor, correspondente ao modelo existente no padrão MVC, gerindo os dados que persistem na base de dados. Assim, adquire-se uma separação que permite executar a aplicação web com dados de teste, e visualizar a aparência das páginas, independentemente da comunicação com o servidor.

```
<h1> { $artistName} </h1>
<table>
  <tr> <th>Song Name</th> </tr>
  {foreach item=song from=$songs}
      <tr> <td>{ $song}</td> </tr>
  {/foreach}
</table>
```

Listagem 3.6: Fragmento de *template* (usando a linguagem de *templating* da plataforma `Smarty`) com os parâmetros *songs* e *artistName*.

```
<script>
  fill_template("songs.tpl",
  [["artistName","Artist_A"],
  ["songs",["Song_A","Song_B","Song_C","Song_D"]]);
</script>
```

Listagem 3.7: Exemplo de *script* para substituir os parâmetros do *template*.

Existindo dois modelos, é necessária uma integração para que as páginas da aplicação apresentem informação coerente com a atual. Esta integração consiste no fornecimento de dados pelo modelo da aplicação, que são reestruturados de acordo com o modelo de apresentação e, através da estratégia de *push* apresentada por Parr em [11], são inseridos no *template*, ao invés de o *template* obter os dados do modelo. Desta feita, o número de pedidos realizados pelo cliente ao servidor pode ser reduzido, se o modelo da aplicação fornecer mais dados do que o necessário para a visualização da página, usando-os para a resposta a certas interações do utilizador. O objetivo desta abordagem é, então, atribuir à página no cliente, um *script* que obtenha os dados do seu modelo.

Tatsubori *et al* [17] apresentam também um sistema de *templating* com o intuito de resolver os problemas de redução do consumo de largura de banda e carga do servidor, visando a separação entre lógica funcional e apresentação. Adicionalmente, mostram a preocupação em delegar no cliente todo o processo de geração das páginas `HTML` não dependendo do servidor.

O uso de um mecanismo de *templating* envolve essencialmente três passos: a especificação do *template* a usar; a associação entre dados e os parâmetros a fim de gerar o conteúdo correto; a substituição os parâmetros por conteúdo no *template* produzindo `HTML`. Tendo isto em consideração, Tatsubori cria um mecanismo que fornece ao cliente o *template* (e.g., 3.6), e um *script* com os dados do *template* (e.g., 3.7), encarregando-o de gerar a página. Subjacente a esse *script* é enviado outro ficheiro `JavaScript`, do qual é invocada a função *fill_template*, usada no exemplo 3.7, que gera o `HTML` completo com base no *template* e o mapeamento dos dados e parâmetros do *template*, indicados nos seus argumentos.

Conclusão

O compilador implementado no protótipo desenvolvido neste trabalho, apresentado no capítulo 6, restringe o tipo de expressões parametrizadas. Portanto os *templates* tipo cliente gerados a partir do modelo de uma aplicação são restritos. Contudo, se o mesmo *template* fosse criado manualmente, o *developer* teria toda a liberdade para inserir código no documento, já que estamos perante um documento `HTML` parametrizado segundo a notação da plataforma `AngularJS`. O código com potencial para ser inserido no *template* é o cálculo de expressões com valores apresentados na interface. Parte dos dados enviados do servidor para o cliente, derivam do cálculo de expressões já efetuado no servidor, ou seja, antes do *template* ser instanciado. Existe uma estrutura intermédia (estrutura de suporte), que se pode considerar como o modelo de dados do ecrã, onde todos os dados representam os valores finais parametrizados no *template*. Considere-se um ecrã com conteúdo condicional, a avaliação da condição deve ser efetuada no *template* de forma a condicionar que conteúdo será apresentado. A condição é expressa por um valor booleano que corresponde à execução do teste implementado no servidor, e é representada por um elemento da estrutura de suporte enviada para o cliente e consumida pelo *template*.

O uso da plataforma `AngularJS` permite declarar a iteração de conjuntos de dados na declaração do *template*, assim como condicionar a apresentação do seu conteúdo. Como tal, os *templates* restritos gerados na nossa solução classificam-se como regulares dependentes do contexto.

A solução implementada neste trabalho mantém a separação entre a lógica da aplicação e a apresentação da interface, defendida por Parr [11]. A arquitetura das aplicações *OutSystems* consiste em delegar a execução de toda a lógica no servidor, tal como a geração da página `HTML`. Todos os dados inseridos na página são previamente calculados, por isso nunca são enviados dados extra para o cliente para serem usados no cálculo de expressões. A separação entre apresentação e a camada de lógica foi importante para manter este comportamento na solução deste trabalho. Assim, o cálculo de expressões é considerado como parte da camada de lógica, implementada no servidor, e os valores calculados são aqueles enviados para o cliente para instanciar o *template* tipo cliente e gerar a página `HTML`. Não enviar dados extra para o cliente, que não pertencem ao conjunto de dados estritamente necessário, contribui também para as boas práticas de segurança da aplicação, pois evita que o cliente aceda a dados a que não devia ter acesso.

A solução implementada cumpre as regras 2 e 3 do conjunto de 5 regras apresentadas por Parr, as quais tiveram maior prioridade ao longo do desenvolvimento deste trabalho, cruciais para garantir a segurança das aplicações geradas. Relativamente à primeira regra, é cumprida parcialmente. Neste trabalho, o *template* de um ecrã pode conter referências a variáveis locais, ou propriedades simples e dinâmicas de *widgets*. As variáveis são possíveis de alterar através de elementos *input*, e consequentemente as expressões (propriedades simples de *widgets*) apresentadas no ecrã, que dependem de uma variável alterada num *input*, devem ser atualizadas. Na implementação deste trabalho, quando é

alterado um *input*, é invocada uma função através do *template* que atualiza as expressões necessárias. No entanto, as expressões não são de facto dados da aplicação, o que nos leva a afirmar que a primeira regra não é cumprida na íntegra.

Neste trabalho é usada a estratégia *push* no código gerado a partir do modelo de uma aplicação, ou seja, o *template* tipo cliente consome os dados já preparados na camada a lógica da aplicação implementada no servidor.

A solução proposta aproxima-se um pouco do trabalho que apresenta uma arquitetura com dois modelos de dados. Na nossa solução existe um modelo de dados mantido no cliente, correspondente à estrutura de suporte que contém todos os dados necessários para o cliente. Esse modelo é partilhado com o servidor, embora sejam duas instâncias diferentes. Quando se envia o modelo para o servidor, ou vice versa, a estrutura pode não estar completa, pois nem todos os dados são necessários para executar uma determinada ação. Poupa-se assim no envio de conteúdo desnecessário, reduzindo o consumo de largura de banda, um dos objetivos deste trabalho.

3.1.3 Verificação de linguagens de *templating*

O uso de *templates* não se destina apenas à definição da estrutura de páginas web: podem também ser usados para a geração de artefactos de software, e.g., o código gerado nas aplicações *OutSystems* que implementa as ações de um ecrã. Para esse uso em particular, é importante certificar que os artefactos gerados pelos *templates* não contêm erros. Esta certificação não é possível através de *templates* que geralmente se baseiam numa parametrização do conteúdo, não tendo qualquer relação com a linguagem do código a ser gerado e, por isso, não é possível detetar erros sintáticos ou semânticos da linguagem em questão. A alternativa é esperar que o código seja gerado, e quando executado são detetados erros que podiam ser evitados se fossem detetados antes da geração do código.

Heidenreich *et al*, em [6], apresentam como estender uma linguagem existente com conceitos genéricos de *templating* (parametrização, iteradores, condições), a fim de obter uma linguagem de *templating* onde os erros do código a gerar sejam verificados antes da sua geração.

Na *OutSystems Platform*, a geração de código da aplicação é feita através de *templates*, mas, atualmente, já é efetuado um processo de verificação da aplicação desenvolvida no *Service Studio* antes de se proceder à geração do seu código. Por este motivo, assumindo que a geração de código através do modelo da aplicação é correta, então os *templates* finais representantes das páginas, tal como o código do *backend*, podem assumir-se sem erros.

3.1.4 Particionamento Cliente-Servidor

O particionamento do cliente e servidor corresponde à compensação da execução da lógica da aplicação ser efetuada no cliente ou servidor, com uma variação entre *thin client* e *fat client*, conceitos que são abordados por Leff e Rayfield em [8].

Este tópico afasta-se um pouco do contexto deste trabalho, no entanto, ao explorar uma arquitetura em que o cliente gera as páginas `HTML`, pondera-se a possibilidade de esse executar mais funcionalidade da aplicação.

Yang *et al*, em [19], apresentam a construção de uma plataforma que procede ao particionamento automático de aplicações direcionadas a dados, com o objetivo de otimizar o tempo médio de resposta ao cliente. Levanta-se então a questão de que lógica pode ser movida para o cliente, evitando pedidos desnecessários ao servidor. Um exemplo prático desta situação é uma aplicação que permite ao utilizador ordenar uma tabela: esta ordenação tanto pode ser efetuada no servidor através de uma consulta SQL, como no cliente através da execução de `JavaScript`, com algumas condicionantes. Se a listagem da tabela envolver paginação, é necessário que o cliente já tenha obtido todos os dados para apresentar a ordenação correta, ou então o servidor deve disponibilizar um recurso que permita o cliente obter os dados de cada página. A paginação em listas é outro exemplo de uma operação cuja implementação pode ser particionada entre cliente e servidor. Existe um custo e benefício entre implementar a paginação totalmente no cliente, ou implementar parcialmente no cliente e servidor. O primeiro caso implica obter todos os dados no primeiro pedido da página, o que pode não ser muito eficiente se o volume de dados for grande, e se o utilizador visitar apenas as primeiras páginas. No segundo caso, considere-se a plataforma `Ruby on Rails` que disponibiliza um grande conjunto de bibliotecas (*gems*). A partir destas podemos adicionar funcionalidade à aplicação, inclusive através da execução de `JavaScript` no cliente, nomeadamente a *gem AJAX_pagination*. Podemos implementar a paginação no cliente através de pedidos `AJAX`, aos quais o servidor responde com os dados de página x. Assim, tenta-se obter um equilíbrio da carga de processamento no cliente e no servidor.

O conceito de *fat client* surge com esta transição da funcionalidade da aplicação para o cliente, tornando este mais independente do servidor. Consequentemente, estas aplicações podem tornar-se mais ricas e fluídas, aproveitando o poder computacional dos dispositivos cliente de hoje. Contudo, estão em desvantagem os dispositivos mais antigos, com menor poder computacional, onde as vantagens desta abordagem não são tão claras, por isso é necessário encontrar o equilibro, ou desenvolver aplicações capazes de adaptar o seu comportamento com base no dispositivo em questão.

Em conjunto com a transição de funcionalidade para o cliente, existe a transição da geração de páginas, tornando a aplicação mais fluída, pois o tempo de espera para visualizar a página é menor, mais ainda com o mecanismo de *caching* para os *templates*. Para além de reduzir o tempo de espera, torna-se possível simular aplicações nativas, caracterizadas por permanecerem no dispositivo e, consequentemente darem um *feedback* mais rápido às interações do utilizador. As aplicações de página única, onde apenas certos componentes `HTML` são atualizados na navegação entre páginas, aproximam-se desse comportamento. As aplicações que normalmente são executadas em vários tipos de dispositivos (e.g., *tablet*, *desktop*, *smartphone*), podem oferecer uma interface Web ajustada a

cada tipo. Para tal, como a geração das páginas é efetuada no cliente, podem ser definidos vários *templates* para a mesma página, e consoante o dispositivo cliente o servidor fornece o *template* que oferece uma estrutura adequada ao dispositivo. Este trabalho pretende atingir um resultado final que facilite as alterações à linguagem atual que define as aplicações desenvolvidas, para o desenvolvimento deste tipo de aplicações.

A fim de mostrar as vantagens do movimento de lógica para o cliente, Yang *et al* [19] usou uma aplicação de um sistema de gestão de cursos e uma aplicação de uma loja *online* de livros, em que ambas envolvem várias operações e grande interação com o utilizador. O objetivo presente é calcular o tempo médio de resposta e a quantidade média de dados transferidos por operação. Para tal, registaram as operações ao longo de determinado tempo, para no fim ser realizada uma comparação entre duas implementações das aplicações, uma implementação tradicional e outra com o mecanismo automático de particionamento. Foram observados ganhos com a segunda implementação tanto a nível de tempo como dados transferidos, contribuindo para uma melhor experiência do utilizador.

Discussão

Ao nível do trabalho apresentado neste documento, o particionamento da lógica da aplicação entre o cliente e servidor é efetuado explicitamente. A linguagem proposta na secção 4, permite declarar que uma ação é executada no servidor ou cliente, mas tendo em atenção que nem todas as ações devem ser executadas no cliente. Por exemplo, operações sobre a base de dados podem ser executadas no servidor.

Como trabalho futuro coloca-se a hipótese deste particionamento ser automatizado através de análise estática do modelo da aplicação, observando o tipo de operações contidas em cada ação.

3.2 *Templates* tipo cliente na indústria

Os dispositivos mobile (e.g., *smartphones* e *tablets*) começam a representar uma grande percentagem do ambiente de utilização de aplicações Web. Por isso existe a necessidade das aplicações serem mais fluídas, já que o utilizador tem expectativas altas sobre uma interação rápida com as aplicações Web no seu dispositivo móvel. A indústria acompanha este facto adaptando as aplicações Web a fim de serem usadas num contexto móvel.

A indústria tem começado a adotar a arquitetura com geração das suas páginas do lado do cliente, obtendo aplicações Web mais fluídas e com um *feedback* mais rápido. Alguma funcionalidade lógica das aplicações tem também transitado para o lado do cliente, aproveitando o grande poder computacional que os dispositivos oferecem atualmente. Esta transição é visível no aumento da quantidade de `JavaScript` que é transferida para o cliente nos últimos tempos, visível na Figura 3.8.

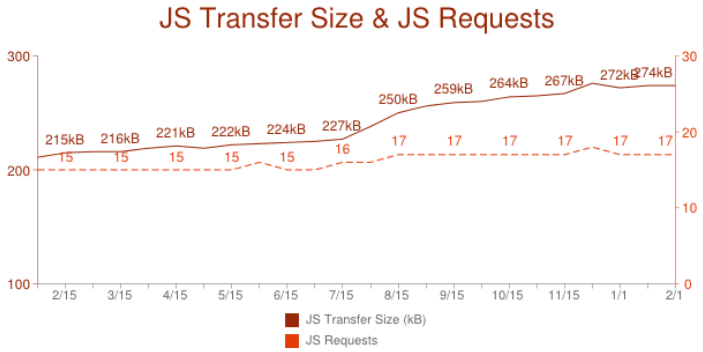


Figura 3.8: *JS Transfer Size*: tamanho médio de transferência de todas as respostas `JavaScript` para uma aplicação. *JS Requests*: número médio de pedidos `JavaScript` para uma aplicação. O período apresentado corresponde desde de Fevereiro-2013 a Fevereiro-2014. (in [26])

Entre as aplicações mais populares que geram as suas páginas no cliente, encontramos o *Gmail*, *Trello*, *Facebook* e *LinkedIn*. Todas com uma característica em comum, a interação frequente com o utilizador, novas mensagens, novas atualizações de conteúdo, etc. Estas aplicações têm o objetivo de proporcionar uma boa experiência de navegação ao utilizador, retribuindo com um *feedback* rápido às suas interações.

A geração de páginas no cliente é vantajosa, pois permite efetuar cache dos *templates* e mantém a separação clara entre apresentação, lógica e dados. Consequentemente, facilita a adaptação da aplicação a novos tipos de dispositivos, proporcionando visualizações das páginas adequadas ao dispositivo, através de múltiplos *templates* para uma página. Com a estrutura da página em cache, o cliente acaba por obter apenas dados `JSON`, consumindo assim menos largura de banda, e obtendo uma aplicação mais fluída.

A aplicação *Trello* destina-se à gestão colaborativa de projetos, envolvendo grande interação dos utilizadores, e por isso um *feedback* rápido é importante. A abordagem para a geração das páginas no cliente é apresentada por *Kiefer* em [27]. Para o suporte do padrão MVC no cliente foi usado o `Backbone.js` [22] que se encarrega de gerar a página a partir dos dados recebidos pelo servidor e *templates* definidos em `Mustache` [30].

A plataforma `Backbone.js` é responsável por detetar alterações aos dados para conseguir gerar de novo os blocos `HTML` que os apresentam, e por responder aos eventos da página `HTML` informando o servidor. Foi escolhida pela sua simplicidade, legibilidade do código e fácil manutenção.

A plataforma `Mustache` especifica os seus *templates* sem qualquer lógica, como iteradores e condições, substituindo os parâmetros por dados. Esta opção justifica-se por não envolver muito código nos *templates*, mantendo-os simples, e facilmente reutilizáveis sem alterar lógica de funcionamento.

Relativamente à aplicação *LinkedIn*, uma rede social profissional e empresarial, também recentemente decidiu transitar a geração das suas páginas para o cliente. *Basavaraj* apresenta, em [23], a análise efetuada aos sistemas de *templating* destinados ao cliente,

onde alguns foram excluídos à partida devido à sua pequena comunidade, falta de documentação, ou dificuldade de testar os resultados finais. As tecnologias restantes foram divididas em dois grupos, as que embebem `JavaScript` nos seus *templates*, e as que oferecem *templates* sem qualquer lógica, onde se insere o `Mustache`.

A partir de um conjunto de critérios, nomeadamente a reutilização de código, suporte para várias linguagens, tempo de aprendizagem, comunidade de desenvolvimento, testes, maturidade, documentação, entre outros, foram avaliados vários sistemas *templating*.

A equipa do *LinkedIn* desenvolveu a página de perfil de um utilizador usando todas as tecnologias [23], com o servidor a fornecer os dados `JSON`, a fim de adquirir alguma experiência para a avaliação dos critérios em cada uma. Através de uma pontuação foram obtidos 4 finalistas, de onde foi escolhido a plataforma *Dust.js*, destacando-se por cumprir a maioria dos critérios, mantendo o seu código claro e rápido. Com o intuito de continuar a evoluir os *templates* do lado do cliente a equipa do *LinkedIn* tem feito alterações ao `Dust.js`, visto ser *open source*.

A plataforma `AngularJS` não pertence ao conjunto de sistemas de *templating* analisado pelo *LinkedIn*, o que se pode justificar por na altura não ter atingindo ainda a maturidade necessária. Contudo, em [25] encontramos várias aplicações que usam esta plataforma para a implementação do padrão MVC do lado do cliente, com *templates* definidos através de anotações no documento `HTML`.

Contrariamente a estas aplicações, o *Twitter* não viu vantagens em adotar uma arquitetura com geração de páginas no cliente [33]. Após a comparação entre geração de páginas no cliente e servidor, concluiu-se que com a primeira o tempo até visualizar a página era 5 vezes superior comparativamente no servidor. No entanto, as razões para esta quebra de desempenho ficam por explicar, sendo indicado que para gerar as páginas no cliente era necessária a execução de uma grande quantidade de `JavaScript`, e consequentemente a página só seria visualizada após a sua execução, conduzindo assim a um maior tempo de espera.

A transição de geração das páginas para o cliente é um padrão recente, e as tecnologias que facilitam a sua implementação surgiram maioritariamente no ano de 2010. Em 2006 foi publicada uma patente [7] que apresenta um sistema para gerar páginas `HTML` no cliente. O objetivo desta arquitetura consiste na diminuição de dados transmitidos na rede, para isso a solução patenteada envia ao cliente somente os dados necessários para gerar a página.

3.3 Tecnologias de *templating*

No contexto deste trabalho, onde se pretende gerar as páginas `HTML` no cliente, é necessário reutilizar um mecanismo de *templating* para correr no *browser*.

Atualmente, existem várias plataforma em `JavaScript` [31] que permitem a implementação do padrão MVC com *templates* tipo cliente, e consequentemente a geração das páginas. Em [32] diversos *developers* classificam e avaliam algumas destas plataformas,

considerando a sua contribuição para a Web, e o quanto preparadas estão para uma aplicação no mundo real.

Destas plataformas foram selecionadas duas, a fim de efetuar uma análise semelhante à que o *LinkedIn* realizou, mas mais reduzida, pois o tempo para a primeira fase deste trabalho não seria suficiente para desenvolver uma aplicação em todas as plataformas disponíveis. A primeira escolha foi a plataforma `AngularJS` [20], escolhida pela sua popularidade, por ser desenvolvida pela Google, pela sua grande comunidade no *GitHub* (com cerca de 600 colaboradores), e pela simplicidade da linguagem de *templating*. A segunda escolha foi a plataforma `KnockoutJS` [28], escolhida por apresentar uma abordagem diferente, oferecendo uma arquitetura MVVM (*Model View ViewModel*) alternativa à convencional MVC usada pelo `AngularJS`.

Desenvolveu-se uma aplicação (Biblioteca Musical) sobre as duas plataformas com o intuito de comparar o seu processo de desenvolvimento, como também observar os tempos de espera até visualizar uma página e o conteúdo que é enviado na rede.

3.3.1 AngularJS

O `AngularJS` usa o padrão MVC na implementação de uma aplicação, em que o componente *Model* corresponde ao objeto *$scope*, que através das suas propriedades estrutura os dados da aplicação. Essas propriedades são acedidas no *template* de forma a serem substituídas pelos dados. A *View* corresponde à definição dos *templates*, a partir dos quais são geradas as páginas `HTML`. O *Controller* é definido a partir do uso da diretiva `ngController` (explicada em anexo) e permite adicionar lógica funcional à aplicação.

No anexo C.1 é apresentada a plataforma em detalhe.

3.3.2 KnockoutJS

O `KnockoutJS`, ao contrário do `AngularJS`, implementa o padrão MVVM (*Model View ViewModel*) nas suas aplicações. A diferença para o MVC relaciona-se com a substituição do controlador pelo componente *ViewModel*, onde são declarados os objetos do *Model*, e especificado o seu comportamento na *View*, não existindo assim uma separação clara entre apresentação e dados.

No anexo C.2 é apresentada a plataforma em detalhe.

3.3.3 Aplicação Biblioteca Musical

A fim de testar as duas plataformas referidas, foram desenvolvidas duas aplicações (iguais), cada uma usando a respetiva plataforma, mas executadas através do mesmo servidor Web, e partilhando a mesma base de dados. Tornando-se assim clara a separação entre apresentação e dados provenientes do servidor.

A aplicação biblioteca musical consiste em 3 páginas: músicas, álbuns, e artistas. Todas as páginas contêm uma tabela, destinada à apresentação de dados. A página de

```
<tbody>
  <tr ng-repeat="song in songs">
    <td>{{song.songName}}</td>
    <td>{{song.artistName}}</td>
    <td>{{song.albumName}}</td>
  </tr>
</tbody>
```

Listagem 3.9: Fragmento de *template* em `AngularJS` que gera tabela com nomes das músicas, artista e álbum.

```
<tbody data-bind="foreach: songs">
  <tr>
    <td data-bind="text: songName"></td>
    <td data-bind="text: artistName"></td>
    <td data-bind="text: albumName"></td>
  </tr>
</tbody>
```

Listagem 3.10: Fragmento de *template* em `KnockoutJS` que gera tabela com nomes das músicas, artista e álbum.

músicas permite também a inserção de uma música na base de dados, a fim de testar a atualização da interface após a adição.

Para o desenvolvimento desta aplicação foi implementado um servidor Web usando a plataforma `node.js`, com a configuração dos pedidos invocados pelo cliente, diferenciando-os entre leitura, escrita ou modificação dos dados da aplicação. Os dados são obtidos de uma base de dados definida através do `MongoDB`, que disponibiliza um serviço de acesso através de uma `API REST`.

A aplicação contém uma diretoria *data* onde o `MongoDB` aloca a base de dados. A comunicação entre o `node.js` e o serviço criado pelo `MongoDB` é efetuada através do ficheiro *app.js*, que define o comportamento da aplicação no servidor, ou seja, onde são definidos os caminhos, que estabelecem o acesso do cliente aos dados.

A aplicação foi desenvolvida seguindo um modelo de página única. Em contraste com um modelo de várias páginas independentes, este permite um *feedback* mais rápido na interação do utilizador com a aplicação, tornando a interação mais fluída, já que apenas alguns componentes da página são alterados, ao invés de alternar entre várias páginas, necessitando de um carregamento completo de cada página.

As páginas da aplicação derivam de um documento `HTML` principal, usando os *templates* representantes de cada página, e um ficheiro `JavaScript` que especifica o comportamento da aplicação no cliente, ou seja, o que executa os pedidos ao serviço de `API REST`, e que define o *ViewModel* ou o *Controller* em `KnockoutJS` e `AngularJS` respetivamente. O código `HTML` principal indica o elemento `HTML` onde será embutido o *template* respetivo a uma página.

As listagens 3.9 e 3.10 representam fragmentos dos *templates* em `AngularJS` e `KnockoutJS`, que geram uma tabela das músicas usando, para isso, um operador iterativo sobre o conjunto de dados *songs*. A geração dos elementos `HTML` consoante a iteração difere entre as duas plataformas: em `AngularJS` cada iteração produz o próprio elemento onde é definido o atributo respetivo ao operador, enquanto em `KnockoutJS` o conteúdo do elemento onde é definido o operador é o que é gerado em cada iteração.

No fim do desenvolvimento de ambas as aplicações foi possível observar que o código resultante em `AngularJS` é mais claro e simples, em particular, os seus *templates* são mais legíveis comparativamente aos do `KnockoutJS`, onde é sempre usado o atributo *data-bind*. Relativamente a código `JavaScript`, em `Knockout` foram necessárias mais 100 linhas em relação ao `AngularJS`. A parametrização dos dados no *template* é também mais simples de efetuar no `AngularJS`, visto não ser necessário declarar em `JavaScript` os objetos ou variáveis representantes dos dados a substituir no *template*, como se sucede com o `KnockoutJS`. A plataforma `AngularJS` é também capaz de abstrair os serviços de navegação entre páginas, e permite o uso de *templates* externos, ao contrário do `Knockout` que necessita de plataformas externas (`sammy.js` e `Knockout External Template Engine`).

Por outro lado, a separação entre apresentação e dados torna-se menos clara no `KnockoutJS`, devido ao componente *ViewModel*, uma vez que a definição do *template* depende do que é declarado no código `JavaScript`.

Relativamente ao processo de aprendizagem, a documentação de ambas as plataformas está bem conseguida, com a experiência adquirida observou-se o desenvolvimento de uma página funcional em `AngularJS` mais rápido do que em `KnockoutJS`. O facto do `AngularJS` ter sido a primeira plataforma a ser testada, pode ter influenciado, no entanto o `KnockoutJS` revela mais pormenores de desenvolvimento a ter em conta.

Com estas considerações, optámos por utilizar o `AngularJS` para o desenvolvimento de uma aplicação Web.

3.4 Análise estática

A linguagem criada neste trabalho permite definir o modelo de uma aplicação sobre o qual é aplicada a análise estática, no sentido de minimizar os dados transmitidos na rede entre cliente e servidor.

A análise estática do fluxo de dados em programas é normalmente implementada através de algoritmos de ponto fixo [15]. Um programa é representado por um grafo do fluxo de controlo, onde cada nó representa uma instrução. As instruções do programa podem ser categorizadas entre: declarações de variáveis, afetações de variáveis, condições, *outputs* de informação, instruções finais e, por fim, outras instruções que não coincidam com as categorias anteriores.

Schwartz apresenta, em [15], esta categorização como auxílio à aplicação de um conjunto de regras que definem a análise de *Liveness*. Esta análise identifica uma variável

como *live*, se for lida no nó corrente ou em nós da execução futura, a não ser que o seu valor seja afetado no nó corrente.

Considere-se que v denota um nó do grafo de fluxo de controlo, $exit$ representa o último nó, $[[v]]$ denota o conjunto de variáveis *live* antes do nó v, $succ(v)$ o conjunto de nós sucessores do nó v, id a variável declarada ou afetada, E a expressão que pode ser devolvida ou cujo valor afeta uma variável, e por fim $vars(E)$ o conjunto de variáveis ocorrentes na expressão E. É apresentado o seguinte conjunto de regras que definem a análise de *Liveness*:

Usando a definição auxiliar: $JOIN(v) = \bigcup_{w \in succ(v)} [[w]]$

Para o nó de saída, a regra é: $[[exit]] = \{\}$ (1)

Para condições e outputs, a regra é: $[[v]] = JOIN(v) \cup vars(E)$ (2)

Para afetações de variáveis, a regra é: $[[v]] = JOIN(v) \setminus id \cup vars(E)$ (3)

Para declaração de variáveis, a regra é: $[[v]] = JOIN(v) \setminus \{id_1, \ldots, id_n\}$ (4)

Para todos os outros, a regra é: $[[v]] = JOIN(v)$ (5)

Considere-se o programa exemplo 3.11, usado por Schwartz em [15], onde existem três variáveis, em que a variável x assume o valor de input pelo utilizador, durante um ciclo sofrem afetações, e no fim é feito o output da variável x. O resultado intermédio de aplicar o conjunto de regras, e o final onde se obtém as variáveis vivas em cada nó do programa, é apresentado na Listagem 3.12. Cada conjunto representa o conjunto de variáveis $live$ em cada instrução do programa.

```
1  var x,y,z;
2  x = input;
3  while (x>1) {
4     y = x/2;
5     if (y>3)
6        x = x−y;
7     z = x−4;
8     if (z>0)
9        x = x/2;
10    z = z−1;
11 }
12 output x;
```

Listagem 3.11: Programa exemplo. (in [15])

```
1  [[entry]] = {}
2  [[var x,y,z;]] = {}
3  [[x=input]] = {}
4  [[x>1]] = {x}
5  [[y=x/2]] = {x}
6  [[y>3]] = {x, y}
7  [[x=x−y]] = {x, y}
8  [[z=x−4]] = {x}
9  [[z>0]] = {x, z}
10 [[x=x/2]] = {x, z}
11 [[z=z−1]] = {x, z}
12 [[output x]] = {x}
13 [[exit ]] = {}
```

Listagem 3.12: Resultados da análise de *liveness* aplicada ao programa 3.11.

Pfenning [12] apresenta uma alternativa a estes algoritmos, expressando a análise por regras de lógica. A análise de *liveness* usada neste trabalho é definida pelas regras ilustradas na Figura 3.13. Os resultados da análise são inferidos pelas interrogações efetuadas

à base de conhecimento, que é composta por factos, a partir dos quais são declaradas relações entre dados de um programa, e por regras, que permitem inferir novos factos.

Os seguintes factos são extraídos de um programa, e usados para inferir os factos do predicado $live$ definido pelas regras (1) e (2) da Figura 3.13.

$\mathsf{use}(l, x)$	A instrução l usa a variável x.
$\mathsf{def}(l, x)$	A instrução l escreve a variável x.
$\mathsf{succ}(l, l')$	A instrução l' é sucessora da instrução l.

O predicado $succ$ cria um conjunto de relações entre os identificadores das instruções, aproximando-se a um grafo do fluxo de controlo onde cada nó corresponde a uma instrução identificada por l.

$$\frac{\mathsf{use}(l, x)}{\mathsf{live}(l, x)}\ (1) \qquad \frac{\begin{array}{c}\mathsf{live}(l', u)\\ \mathsf{succ}(l, l')\\ \neg\mathsf{def}(l, u)\end{array}}{\mathsf{live}(l, u)}\ (2)$$

Figura 3.13: Regras da análise de *liveness*. (in [12])

Smaragdakis recorre também a regras de lógica para implementar a análise oferecida pela plataforma `Doop` [16]. Para exprimir as regras usa a linguagem de programação lógica `Datalog`, que permite interrogar uma base de conhecimento de forma determinista. A complexidade de inferir resultados a partir de um programa `Datalog` é a mesma dos algoritmos de ponto fixo, implementados usualmente em compiladores. Um programa `Datalog` é composto por factos, regras, e interrogações que permitem inferir relações entre dados. Se expressarmos as regras de lógica apresentadas por Pfenning em `Datalog`, tal como Smaragdakis procede:

```
live(?l, ?x) :- use(?l, ?x).
live(?l, ?u) :- not def(?l,?u), succ(?l, ?l2), live(?l2,?u).
```

Assumindo que as instruções do programa da Listagem 3.11 são identificadas por $L1$, $L2$, $L3$ e assim sucessivamente, com correspondência à numeração das linhas do código, os factos extraídos do programa correspondem a:

```
succ('L1', 'L2').        use('L6', 'y').
succ('L2', 'L3').        use('L6', 'x').
...                      def('L7', 'z').
def('L2', 'x').          use('L7', 'x').
use('L2', 'input').      use('L8', 'z').
use('L3', 'x').          use('L9', 'x').
def('L4', 'y').          def('L9', 'x').
use('L4', 'x').          use('L10', 'z').
use('L5', 'y').          def('L10', 'z').
def('L6', 'x').          use('L12', 'x').
```

Se forem efetuadas interrogações ?-live(L, ?v), onde L representa o identificador de uma instrução, obtemos os mesmos resultados apresentados na Listagem 3.12. Por exemplo, a interrogação ?-live('L6', ?v), onde L6 corresponde à linha 6 do exemplo da Listagem 3.11. Obtemos assim exatamente os mesmos resultados que o algoritmo de ponto fixo.

A análise estática é aplicada sobre programas a fim de determinar possíveis comportamentos do código a executar através da análise da sua estrutura estaticamente. Os resultados inferidos pela análise aplicada sobre o programa exemplo (Listagem 3.11), permitem deduzir que as variáveis y e z nunca estão "*live*"em simultâneo, e o novo valor atribuído a z na linha 10 nunca é usado. Por isso, a partir dos resultados inferidos é possível reformular o código, obtendo uma otimização, neste caso das variáveis envolvidas.

Smaragdakis apresenta a plataforma `Doop` [16], uma plataforma versátil para análise de programas, e que usa `Datalog` para exprimir a análise. O autor apresenta o `Datalog` como a escolha à medida para exprimir a análise de um programa, permitindo que toda a implementação seja efetuada de forma declarativa e, consequentemente, mais legível para o programador.

O autor realça que a recursividade mútua é um dos fatores para a complexidade dos algoritmos de análise estática, algo que é possível definir em `Datalog` através de relações recursivas. A recursividade mútua está presente, por exemplo, na regra (2) da Figura 3.13, em que no corpo da regra estão apenas os factos já inferidos mas com o mesmo predicado da regra.

Conclusão

Na plataforma *OutSystems* já é aplicada análise estática a fim de otimizar as aplicações geradas. O fluxo da sequência de operações de cada ação e as *widgets* instanciadas num ecrã, são submetidos a uma análise de forma a filtrar os dados necessários para o ciclo de vida de uma aplicação. Também as consultas efetuadas à base de dados são submetidas a esta análise, a fim de filtrar os dados necessários de uma entidade usados ao longo da execução da aplicação.

No contexto deste trabalho, a análise estática pretende minimizar os dados transmitidos na rede entre cliente e servidor. Esta otimização consiste em inferir o conjunto mínimo de dados com base no que o servidor e cliente necessitam ao longo do ciclo de vida da aplicação. No capítulo 5 são apresentadas as duas análises (análise de dependências e de *liveness*) implementadas neste trabalho, feitas sobre a definição do modelo da aplicação, durante o processo de geração de uma aplicação.

A análise de dependências obtém o subconjunto mínimo das expressões que devem ser recalculadas no final de uma ação, com base na dependência das variáveis ou propriedades dinâmicas escritas. Isto permite efetuar o cálculo das expressões estritamente necessárias no fim de uma ação, e depois enviadas para o cliente, evitando o envio desnecessários de dados.

A análise de *liveness* determina 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. Aproxima-se da análise de *liveness* apresentada por Schwartz, que define se uma variável está "*live*"como aquela que é lida no nó corrente ou em nós da execução futura, a não ser que o seu valor seja afetado no nó corrente. Os resultados obtidos por esta análise permite-nos minimizar o conjunto de dados transmitidos entre cliente e servidor, filtrando apenas o conjunto de dados estritamente necessários, que corresponde ao modelo diferencial dos dados do ecrã com base na ação invocada. No contexto deste trabalho os nós equivalem às instruções de uma ação, e uma ação corresponde a um grafo, sobre o qual é aplicada a análise. Logo, ao nível da aplicação existem vários grafos, necessários para se conseguir determinar os resultados ao nível de uma ação, e minimizar os conjuntos de dados transmitidos de forma individual à ação que será executada.

3.5 Sumário

Este trabalho define uma arquitetura de MVC duplo, uma extensão do padrão MVC usualmente presente no servidor. Esta abordagem é inspirada em alguns avanços tecnológicos ao nível de plataformas de programação, mas também na arquitetura proposta por Garcia *et al* [4], que introduz uma arquitetura com dois modelos: um no cliente destinado exclusivamente à apresentação da página; e outro no servidor gerindo os dados que persistem na base de dados. O objetivo desta abordagem é proteger a aplicação de alterações indesejadas na visualização da página Web e no servidor. Neste trabalho o objetivo é minimizar o volume de dados transmitido.

Com o intuito de manter uma separação entre a lógica aplicacional e apresentação, Parr [11] apresenta uma formalização desta separação através do estudo de mecanismos de *templating*, introduzindo um conjunto de regras para a criação de um *template*, 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 o cálculo de expressões não é definido no *template*. Através da aplicação da análise proposta a uma linguagem que integra toda a aplicação, o trabalho aqui apresentado permite a separação automática e otimizada, e não apenas um conjunto de boas práticas e recomendações de desenvolvimento.

Leff *et al* [9] 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. Na linguagem proposta neste trabalho, o programador identifica quais as ações a executar no cliente ou servidor, o que pode causar decisões pouco adequadas durante a fase de desenvolvimento. Em [9] é proposta uma plataforma que define um particionamento automático a fim de maximizar o desempenho da aplicação. Yang *et al* também apresentam uma plataforma com este objetivo [19]. Embora este tipo de separação esteja fora do âmbito da abordagem apresentada neste documento, está relacionada com algum trabalho ainda a

desenvolver.

A separação automática entre lógica e apresentação é essencial em diversos aspetos. Neste trabalho é essencial pois a partir da definição do modelo de uma aplicação, o gerador de código cria uma arquitetura transparente para o programador ao longo da fase de desenvolvimento, que deve separar os componentes de lógica e apresentação. No caso de [1] é essencial para garantir propriedades na modificação da camada de apresentação, sem interrupção do serviço fornecido pela aplicação.

Neste trabalho optámos pela definição da análise estática usando a linguagem `Datalog` [16] e as ferramentas associadas, em vez dos tradicionais algoritmos de ponto fixo [10], que iam implicar um processo de implementação moroso e pouco ágil. Existem várias bibliotecas de fácil usabilidade, que permitem interpretar um programa `Datalog`, e não perder eficiência na sua execução. Também se torna mais fácil definir as várias propriedades a analisar e as suas regras de propagação.

4

O Modelo

A linguagem visual da plataforma *OutSystems* permite definir o modelo de uma aplicação com abstração entre o processo de desenvolvimento e o código gerado pelo compilador. Essa abstração permite que a arquitetura das aplicações geradas seja modificada, introduzindo otimizações com recurso a novos artefactos ou modificando os existentes, sem que para isso seja necessário alterar a linguagem fonte. Este trabalho explora uma novo tipo de arquitetura para as aplicações geradas, com o objetivo de otimizar (ainda mais) as aplicações da plataforma *OutSystems*.

O compilador da plataforma *OutSystems*, responsável por gerar o código das aplicações, abstrai, através da linguagem, vários detalhes de desenvolvimento e *deployment*, pelo que é um projeto com grande dimensão e complexidade. Para alterar a arquitetura das aplicações geradas atualmente, foi necessário adaptar o compilador, a fim de produzir os artefactos que compõem as aplicações segundo esta nova arquitetura. Neste contexto, surgiram várias dificuldades técnicas que nos conduziram à implementação de um protótipo (modelo) fora da plataforma *OutSystems*.

Neste capítulo apresentamos a linguagem que faz parte do desenho desse protótipo, representativa de um fragmento da linguagem *OutSystems* e inspirada na representação abstrata de uma aplicação. Na última secção deste capítulo, é apresentada a definição da estrutura usada para manter os dados representativos do estado de um ecrã.

4.1 Linguagem

Nesta secção apresentamos a linguagem para definir o modelo de uma aplicação, com abstração de detalhes do comportamento da página e da representação da hierarquia das *widgets*, a fim de simplificar o modelo e não cobrir pormenores fora do âmbito deste

$$
\begin{array}{lcll}
M & ::= & (\overline{W}, \overline{E}, \overline{F}, \overline{S}) & \text{(Modelo)} \\
W & ::= & \text{widget } w\ (\overline{p_d}, \overline{p_r}, [init], \overline{p_h}, \overline{html}) & \text{(Declaração de widget)} \\
E & ::= & \text{entity } e\ (\ \overline{e_{attr}}, \overline{record}) & \text{(Entidade)} \\
F & ::= & \text{function } f\ (\ \overline{v}, fbody, \text{return } exp) & \text{(Função)} \\
S & ::= & \text{screen } s\ \{\ \overline{v}, \overline{W_{inst}}, fbody, \overline{A}\} & \text{(Ecrã)} \\
Winst & ::= & w\ z(\overline{p_d = exp}, \overline{p_h(Winst)}) & \text{(Instância de widget)} \\
A & ::= & \text{action } a(fbody) & \text{(Ação)}
\end{array}
$$

Figura 4.1: Sintaxe das definições dos elementos do modelo.

trabalho.

As Figuras 4.1 e 4.2 apresentam as regras que especificam a sintaxe abstrata, com o intuito de simplificar a visualização da relação entre elementos de uma aplicação. A especificação da sintaxe tem como base a seguinte representação:

w	Identificador de uma *widget* declarada.
p_d	Identificador de uma propriedade simples da *widget*.
p_r	Identificador de uma propriedade dinâmica da *widget*.
p_h	Identificador de um *placeholder* da *widget*.
e	Identificador de uma entidade.
f	Identificador de uma função.
s	Identificador de um ecrã.
z	Identificador de uma *widget* instanciada.
a	Identificador de uma ação do ecrã.
v	Identificador de uma variável.

A seguir são apresentados os elementos de maior relevância da linguagem, com base na sintaxe abstrata. De forma a contextualizar sobre o uso da linguagem na prática, ou seja, aplicada ao desenvolvimento de uma aplicação, é apresentado um resumo de cada elemento, relativamente à sua sintaxe concreta no protótipo. O `Xtext` [34] foi a ferramenta usada para criar a linguagem no protótipo deste trabalho, através de um conjunto de regras, como se apresenta na Listagem 4.3.

4.1.1 Modelo da aplicação

O modelo de uma aplicação M é composto por *widgets* W, entidades que definem o modelo de dados E, funções globais da aplicação F e ecrãs S. Ao contrário da plataforma *OutSystems*, que disponibiliza um conjunto pré-definido de *widgets*, aqui permite-se definir a utilização e representação de novas *widgets*, a instanciar na definição dos ecrãs da aplicação.

$init$	::=	$p_r = p_d$	(DataSource)
x	::=		(Nomes)
		a	(Ação)
	|	v	(Variável)
	|	z	(Instância de uma *widget*)
	|	e	(Entidade)
	|	$p_{d,r}$	(Propriedade)
	|	e_{attr}	(Atributo de entidade)
id	::=	x | $x.id$ | id.current | current	(Identificadores)
$record$	::=	$\{\ \overline{e_{attr} : k}\ \}$	(Registo da entidade)
$fbody$	::=	$(\overline{v}, \overline{st})$	(Corpo da função ou ação)
st	::=		(Nó de ação ou função)
		$x = exp$	
	|	$listop\ x\ exp$	
	|	foreach(id) $\overline{st}$ end	
	|	if exp $\overline{st}$ else $\overline{st}$	
	|	$dbop\ e\ [exp]$	
	|	refreshQuery id	
	|	refreshDataSource z	
$listop$	::=	append | remove	(Operações sobre listas)
$dbop$	::=	create | updateRecord	(Operações sobre entidades)
	|	resetRecords | removeRecord	
$html$	::=	$\overline{html_{attr} = p_{d,r}}$ $\overline{ng_{attr} = p_{d,r}}$ $\overline{html}$	(Definição do HTML)
	|	String	
	|	$\{\{p_{d,r}\}\}$	
	|	p_h	
ng_{attr}	::=	ng-repeat | ng-model | ng-change	(Atributos do AngularJS)
	|	ng-click | ng-if	
exp	::=		(Expressões)
		id	
	|	k	
	|	getRecords e	
	|	$exp\ \overline{op\ exp}$	
	|	$f(\ \overline{exp}\ )$	
	|	length id	
	|	currentRow id	
k	::=	String | true | false | n | $\overline{k}$	(Valores)
op	::=	$+$ | $-$ | $*$ | $/$	(Operadores binários)
	|	$>$ | $<$ | $>=$ | $<=$ | $==$ | $!=$	

Figura 4.2: Sintaxe das folhas do modelo.

```
App:
  imports += Import*
  'app' name = ID '{'
    datamodel = DataModel
    functions += Function*
    screens += Screen*
  '}'
```

Listagem 4.3: Regra em `Xtext` que define a sintaxe da criação de uma aplicação.

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

Listagem 4.4: Exemplo da definição de uma aplicação.

Modelo da aplicação em Xtext

A Listagem 4.4 ilustra, de forma resumida, a definição de uma aplicação exemplo, que permite ao utilizador dividir o custo total de um jantar por um conjunto de amigos. De forma a isolar a declaração de *widgets* dos restantes elementos da aplicação, é declarada a importação do ficheiro onde estão definidas as *widgets*.

No elemento dataModel é declarado um conjunto de entidades onde, para cada entidade, é definido um conjunto de registos especificando os valores dos atributos da entidade. Neste exemplo, declara-se a entidade Friend com o atributo Name.

As funções da aplicação são definidas em elementos function, cuja invocação corresponde a uma expressão exp.

Os elementos screen definem os ecrãs da aplicação, onde são instanciadas as *widgets*.

A definição da aplicação é verificada pelo `Xtext` com base na regra da Listagem 4.3.

4.1.2 Widgets

Uma *widget* W é composta por propriedades simples p_d, propriedades dinâmicas p_r, *placeholders* p_h, e pela definição do código $html$ que, especifica a apresentação do conteúdo consumido pela *widget*.

As propriedades de uma *widget* representam os dados que esta consome. Existe, ainda assim, uma diferença entre as propriedades dinâmicas e simples. As primeiras, são as únicas que podem ser manipuladas na sequência de operações de uma ação. As segundas, estão essencialmente associadas à interface, sendo por isso referenciadas na definição do $html$.

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

Listagem 4.5: Declaração das *widgets* Expression e ListRecords.

```
widget Link {
  designtimeProperties {
    Text linkValue
    Text linkDest
  }
  html { <a href=linkDest>{{linkValue}}</a> }
}
```

Listagem 4.6: Declaração da *widget* Link.

O elemento $html$ pode representar texto, um parâmetro com referência a uma propriedade da *widget* $p_{d,r}$ (anotado como $\{\{p_{d,r}\}\}$), referência a um *placeholder* p_h, um conjunto de afetações a atributos do elemento `HTML` $html_{attr}$, conjunto de atributos representativos de diretivas do `AngularJS` ng_{attr}. O corpo do elemento $html$ contém um conjunto de sub elementos $html$. A notação do `AngularJS` presente no $html$ é usada na geração do *template* tipo cliente a partir do qual é produzida a interface do ecrã, uma vez que é a plataforma `JavaScript` usada como suporte à implementação.

A afetação de outros atributos adicionais $html_{attr}$, permitem definir algum comportamento ou estilo do elemento `HTML`. Os atributos são inicializados com referências a propriedades da *widget* ou valores literais. Por exemplo, na Listagem E.1 no anexo E, a *widget* Input inicializa os atributos ng-model, type, class e placeholder.

Através de atributos adicionais no código `HTML` pode-se simular a navegação entre ecrãs. Por exemplo, a *widget* Link, apresentada na Listagem 4.6, contém duas propriedades simples: linkValue e linkDest. As mesmas propriedades são referenciadas no $html$ da widget como um parâmetro e como inicialização de um atributo $html_{attr}$. Na instanciação da *widget* inicializa-se o linkDest com o nome do ficheiro `HTML` de outro ecrã, a fim de redirecionar o utilizador para outra página da aplicação quando clica no link. O fragmento do *template* gerado que representa esta *widget* é traduzido no seguinte elemento `HTML`: < a href="screen_name.html"> screen_name .

O modelo proposto tem também como objetivo simplificar o desenvolvimento de

aplicações com comunicação assíncrona. O desenvolvimento dessas aplicações na plataforma *OutSystems* implica invocar a operação AjaxRefresh para atualizar uma *widget* específica do ecrã na sequência da alteração de dados da aplicação. Neste modelo, essa operação torna-se obsoleta, pois o mecanismo da *framework* `AngularJS` atualiza automaticamente a interface quando o modelo de dados, consumido pelo *template* tipo cliente, é alterado. Ainda assim, a operação `AjaxRefresh` no contexto *OutSystems*, não apresenta um comportamento uniforme, em particular sobre *widgets* com listas dinâmicas. O comportamento dessa operação é apresentado na Secção 2.

No âmbito da plataforma *OutSystems*, uma *widget* equivalente a uma tabela de dados é composta por uma propriedade simples, inicializada com os resultados provenientes de uma consulta à base de dados. Para além dessa propriedade, contém também uma propriedade dinâmica, que em tempo de execução mantém uma cópia dos dados originais da propriedade simples. É essa cópia que a interface do utilizador apresenta. Quando é invocada a operação `AjaxRefresh` sobre a *widget*, é efetuada uma nova cópia dos dados da propriedade simples, ou seja, a lista dinâmica não mantém uma relação direta com os dados da aplicação. Para que seja possível obter um processo de desenvolvimento sobre o modelo proposto, próximo do que existe na plataforma *OutSystems*, foi necessário adicionar semântica extra.

Essa semântica extra diz respeito ao elemento *init* que compõe uma *widget*. Nesse elemento é possível inicializar uma propriedade dinâmica com uma cópia dos dados de uma propriedade simples. Assim, permite-se que a *widget* mantenha uma cópia do conjunto de dados, obtido por uma consulta à base de dados, que deu origem à inicialização da propriedade simples na instanciação da *widget*. Neste caso, o código *html* da *widget* referencia a iteração da lista dinâmica, como se pode observar na Listagem 4.5 (i.e., ng-repeat=list). Quando os dados da propriedade simples são alterados, a lista dinâmica mantém uma cópia antiga, por isso, a operação refreshDataSource criada no modelo proposto é usada para efetuar uma nova cópia. Isso provoca uma atualização automática da interface, pois é detetada a alteração dos dados da lista dinâmica.

Widgets em Xtext

A Listagem 4.5 apresenta a declaração das *widgets* Expression e ListRecords. As propriedades simples são declaradas no elemento designtimeProperties, e as propriedades dinâmicas no elemento runtimeProperties. A inicialização de uma propriedade dinâmica é feita no elemento dataSource.

O `HTML` que representa a estrutura do conteúdo apresentado pela *widget* é definido no elemento html, e a notação usada tem o propósito de se aproximar de código `HTML`. No interior da declaração de um elemento `HTML` são definidos os seus atributos `AngularJS` ou quaisquer outros, com referências a propriedades da *widget*, ou valores literais. No seu corpo declaram-se outros elementos `HTML`, e referenciam-se *placeholders*, declarados no elemento placeholders. É possível, em alternativa a definir elementos `HTML`, usar apenas

texto ou um parâmetro para uma propriedade.

4.1.3 Propriedades simples e dinâmicas

As propriedades dinâmicas p_r diferem das propriedades simples p_d, na medida em que são as únicas que podem ser manipuladas na sequência de operações de uma ação. As propriedades simples estão essencialmente associadas à interface, através das referências que contêm no $html$, embora seja possível referenciar também propriedades dinâmicas.

A união das propriedades referenciadas no código `HTML` das *widgets* instanciadas num ecrã, constitui os dados que a página consome.

As propriedades simples são inicializadas ao instanciar a respetiva *widget* W_{inst}, enquanto que as dinâmicas só são afetadas nas ações do ecrã. Os valores que as inicializam podem ser atómicos (literais, referências a variáveis do ecrã ou propriedades de outras *widgets*), expressões aritméticas compostas por esses valores, ou chamadas a funções da aplicação. Relativamente aos dois últimos, atribuímos o nome de expressões pré-calculadas. Essas expressões não são incluídas explicitamente no *template* gerado, como tal, o servidor deve fornecer todos os valores já calculados. Neste modelo, como também o cliente pode executar ações, o cálculo de expressões pode ser efetuado no cliente.

Propriedades em Xtext

A *widget* Expression, na Listagem 4.5, ilustra a parametrização do código `HTML` com a propriedade value, enquanto que a *widget* ListRecords, mais sofisticada, itera uma lista de registos instanciando *widgets* filhas para cada um. A propriedade src é a lista que fornece o conjunto de registos, e a sua inicialização é apresentada na Listagem 4.10, onde é instanciada a *widget* friendsList. A propriedade dinâmica list (inicializada no elemento dataSource) mantém uma cópia da lista original de registos, podendo ser manipulada de forma independente (e.g., pode-se adicionar novos elementos à lista da *widget* sem afetar a lista original).

A ação Reset, apresentada na Listagem 4.11, remove todos os registos da base de dados, associados à entidade Friend. A atualização da variável com o conjunto de registos (queryFriends) é efetuada através da operação refreshQuery, que repete a consulta à base de dados. Com a remoção dos dados, a lista de registos apresentada no ecrã deixa de estar consistente, por isso, é necessário atualizar a *widget*. Neste cenário, deve ser invocada a operação refreshDataSource, para atualizar a lista dinâmica com uma nova cópia dos dados (queryFriends), referenciados pela propriedade simples da *widget*. No fim o ecrã é atualizado com uma lista vazia.

4.1.4 Parâmetros da *widget*

Quando uma *widget* é definida, é declarado um conjunto de parâmetros (*placeholders*) que permitem à *widget* estruturar as suas *widgets* filhas. Por exemplo, para estruturar

```
1  widget If {
2    designtimeProperties {
3      Boolean conditionValue
4    }
5    placeholders {
6      trueBranch,
7      falseBranch
8    }
9    html {
10     <span>
11       <span ng-if = conditionValue>
12         trueBranch
13       </span>
14       <span ng-if = !conditionValue>
15         falseBranch
16       </span>
17     </span>
18   }
19 }
```

Listagem 4.7: Declaração da *widget* If.

conteúdo condicional dependendo do valor de uma expressão booleana, é necessário conter dois *placeholders*, um caso o valor seja verdadeiro, outro caso o valor seja falso.

Um *placeholder* p_h é identificado por um nome, o qual é referenciado no código `HTML`, no sentido de criar divisórias na apresentação do conteúdo. Quando uma *widget* é instanciada, são instanciados os seus *placeholders*, cujo corpo consiste na instanciação de outras *widgets*.

Quando o modelo da aplicação é submetido ao compilador, é gerado um *template* tipo cliente, que itera as *widgets* do ecrã, nomeadamente o `HTML` de cada uma. Os *placeholders* referenciados no código `HTML` são substituídos pelas *widgets* definidas no seu corpo, quando instanciado.

Placeholders em Xtext

A *widget* ListRecords apresentada na Listagem 4.5, é composta pelo *placeholder* item, declarado no elemento placeholders. Observe-se, na Listagem 4.10 (linhas 17-20), a inicialização do conteúdo do *placeholder* item com a instanciação de outras *widgets*.

Relativamente à *widget* If, apresentada na Listagem 4.7, os *placeholders* permitem definir conjuntos distintos de *widgets*, cuja apresentação depende do valor da propriedade condicional conditionValue. Na Listagem 4.8 os *placeholders* são instanciados para apresentar condicionalmente um conjunto de *widgets* filhas.

```
<If name=ifCount conditionValue = (friendsList.list.Length > 0)>
  <trueBranch>
    <Expression name=costPPerson value=(total / friendsList.list.Length) />
    <Expression name=friendsToPay value=CountUncheck(friendsList.list) />
  </trueBranch>
  <falseBranch/>
</If>
```

Listagem 4.8: Instanciação da *widget* If.

```
function CountUncheck(List list){
    Integer res = 0
    forEach(list){
        if(list.current.Friend.Check){ }
        else res = (res + 1)
    }
    return res
  }
```

Listagem 4.9: Definição de uma função.

4.1.5 Entidades

O conjunto de entidades E corresponde à base de dados da aplicação. Com o intuito de na primeira execução da aplicação existir já um conjunto pré-definido de dados, na declaração de uma entidade é possível inicializar um conjunto de registos ($records$). Os registos consistem na atribuição de valores aos atributos da entidade (e_{attr}).

Este modelo de dados é uma aproximação à camada de dados oferecida pela plataforma, sendo que neste protótipo por questões de simplificação as entidades são representadas por objetos em memória no servidor.

Entidades em Xtext

O componente dataModel, apresentado na Listagem 4.4, define a base de dados da aplicação, através da declaração de entidades (E) identificadas por e e compostas por uma lista de atributos (e_{attr}). São assim introduzidas as entidades usadas para consulta de dados nas ações da aplicação.

4.1.6 Funções

A declaração de uma função global F, consiste na sua identificação por f, no conjunto de parâmetros, representados por variáveis, e pelo seu corpo $fbody$. O corpo de uma função consiste na declaração de variáveis locais v, no conjunto de instruções st, e no retorno da função.

As funções podem ser invocadas no corpo de uma ação ou função. A sua invocação pode inicializar variáveis do ecrã ou propriedades de *widgets*.

Funções em Xtext

Na Listagem 4.9 é apresentada a declaração da função CountUncheck, que itera uma lista de registos com o atributo booleano Check, contabilizando o número de ocorrências do valor a falso. É composta pelo parâmetro list, a variável local res, uma sequência de operações definidas na regra st, definida na Figura 4.2, e, por fim, o retorno uma expressão (exp), que neste caso corresponde a uma referência para a variável local.

Na Listagem 4.10 é invocada a função CountUncheck (na linha 12) como valor da propriedade value da *widget* Expression.

Embora seja ocultado na sintaxe abstrata, as funções podem ser anotadas como sendo executáveis apenas no servidor. Por exemplo, se o corpo da função contiver operações sobre entidades, cujos registos são mantidos no servidor, então não é possível executá-la no cliente. Por omissão, na ausência de anotações assume-se que a função é executada tanto no servidor como no cliente. A declaração de ações como sendo executáveis no cliente é uma extensão à linguagem *OutSystems*, que nos vai permitir explorar a execução de lógica da aplicação no navegador.

4.1.7 Ecrãs

A definição de um ecrã S consiste na declaração de variáveis locais v, na instanciação de *widgets* W_{inst}, ações do ecrã A, e ainda no corpo $fbody$ da ação especial *Preparation*.

As variáveis de um ecrã podem ser usadas na inicialização de propriedades simples das *widgets*, e manipuladas ou lidas ao longo das ações. Os seus valores podem variar entre a referência para outra variável e valores literais, inclusive valores compostos, nos quais os campos são acessíveis através de uma composição de identificadores (id).

A ação *Preparation* é executada uma única vez no início do pedido, ao contrário da plataforma *OutSystems* onde é executada sempre que há um pedido síncrono dentro do ecrã. Tem como objetivo preparar os dados que serão apresentados ou usados na execução de ações, ao longo do ciclo de vida do ecrã.

Ecrãs em Xtext

A Listagem 4.10 apresenta a definição do ecrã Bill, omitido na Figura 4.4, e que implementa a interface da aplicação SplitTheBill. Neste exemplo são declaradas variáveis locais (queryFriends, newFriend, total). A Listagem E.1 inclui a definição de algumas *widgets* instanciadas neste ecrã.

A ação *Preparation* neste caso executa uma instrução que corresponde a uma afetação de uma variável local, mas também pode ser usada para inicializar propriedades dinâmicas das *widgets* do ecrã.

```
screen Bill {
  variables {
    Query queryFriends
    Record<Friend> newFriend
    Integer total = 0
  }
  widgets {
    <Input variable=total/>
    <If name=ifCount conditionValue = (friendsList.list.Length > 0)>
        <trueBranch>
            <Expression name=costPPerson value=(total / friendsList.list.Length) />
            <Expression name=friendsToPay value=CountUncheck(friendsList.list) />
        </trueBranch>
        <falseBranch/>
    </If>
    <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 { ... }
}
```

Listagem 4.10: Definição de um ecrã para a aplicação Split the Bill.

4.1.8 Identificadores

Os identificadores id são referências para nomes de elementos (x), incluindo variáveis, ações, *widgets* instanciadas, entidades, propriedades de *widgets*, e atributos de entidades. As variáveis podem ser criadas em diversos contextos, como ecrã, corpo das ações ou funções, e parâmetros de uma função.

Os identificadores podem ser nomes únicos ou uma composição de nomes. Consideremos uma variável v do tipo Record<Friend>, ou seja, um registo da entidade Friend, que contém o atributo Name. Para acedermos ao atributo Name usamos a seguinte expressão: v.Friend.Name. Caso a variável v seja do tipo List, composta por registos da entidade Friend, é possível aceder aos valores de cada elemento quando a variável é iterada, através da expressão v.current.Friend.Name.

A palavra current é reservada da linguagem e pode ser encontrada em três situações:

1. Quando uma lista é iterada através da instrução forEach, a palavra reservada current contém o valor do índice j da iteração, e é usada no contexto do corpo de uma função ou ação. Possibilita assim, aceder ao elemento j de uma propriedade dinâmica, por exemplo friendsList.list.current.

```
actions{
  client action AddFriend(){
    if( (newFriend.Friend.Name.Length > 0) ){
      append(friendsList.list, newFriend)
    } else{ }
  }
  server action Reset(){
    resetRecords(Friend)
    refreshQuery(queryFriends)
    refreshDataSource(friendsList)
  }
  server action Save(){
    forEach (friendsList.list) {
      create(Friend, friendsList.list.current.Friend)
    }
  }
}
```

Listagem 4.11: Definição das ações do ecrã.

2. Na definição do *html* de uma *widget*, o atributo `AngularJS` ng-repeat itera uma lista, e no seu conteúdo são referenciados *placeholders* p_h. O conteúdo dos *placehoders* corresponde à instanciação de outras *widgets*, notando-se que são geradas repetidamente ao longo da iteração da lista. Neste contexto, a palavra current é usada sem qualquer prefixo, restringindo-se ao acesso do elemento corrente da iteração, por exemplo current.Friend.Name.

3. Quando é desencadeado um evento ao nível de uma linha de uma lista gerada através da iteração pela diretiva ng-repeat, existe um novo contexto para current, numa ação ou função, fora do corpo da operação foreach. O valor de current corresponde ao índice da linha em que ocorreu o evento, e é usado para aceder ao elemento correspondente ao índice, por exemplo friendsList.list.current. Difere para o primeiro caso que é independente de eventos, e representa o índice da iteração.

Quando ocorre um evento sobre a linha de uma lista, existe também a palavra reservada currentRowNumber, usada em expressões como por exemplo friendsList.list.currentRowNumber. É usada para obter o índice da linha onde ocorreu o evento, sendo friendsList.list a respetiva lista.

Em suma, currentRowNumber é o índice da lista onde ocorreu um evento, e current, embora esteja associada a vários contextos, permite aceder a sub-elementos através de uma composição de identificadores, nomeadamente a um elemento da lista.

4.1.9 Ações

Uma ação A é definida por uma linguagem imperativa com as operações básicas esperadas, definidas pela regra *st*, na Figura 4.2, que incluem: afetação de variáveis ou propriedades dinâmicas de *widgets*; operações básicas sobre listas; iteração (forEach); expressões condicionais (if else); operações sobre as entidades do modelo de dados (create, getRecords, etc.); e as operações especiais de refreshQuery e refreshDataSource. As duas últimas operações resumem-se a uma abstração de uma série de atribuições frequentes na lógica de uma aplicação: a primeira, para a atualização de uma variável que contém os registos de uma entidade, e a segunda para a atualização de uma *widget*, efetuando de novo a atribuição do elemento $init$.

Ações em Xtext

As ações do ecrã, abreviadas na Listagem 4.10, e definidas na Listagem 4.11, 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.

A flexibilidade para uma ação ser executada no servidor ou cliente, obriga a restrição do tipo de operações que é possível executar caso pertença ao cliente. No âmbito desta linguagem as operações efetuadas sobre entidades devem ocorrer somente no servidor. Consequentemente, as funções globais também podem ser anotadas como server, quando a sua implementação pertence somente ao servidor. Não existindo qualquer validação sobre a definição de um modelo, caso exista uma falha entre a anotação e o conjunto de operações permitidas, a execução da aplicação apresenta erros.

4.1.10 Widgets instanciadas

Instanciar uma *widget* W_{inst} requer indicar o identificador da *widget* que se pretende usar e definir um identificador para a instância, para que ao longo do ecrã e ações seja possível referenciar as propriedades dinâmicas dessa instância. É no momento de instanciação que são inicializadas as propriedades simples. A *widget* contém ainda a instanciação dos seus *placeholders*, permitindo instanciar *widgets* filhas.

Através da composição de identificadores apresentada anteriormente, podemos navegar entre as *widgets* de um ecrã, *widgets* filhas, propriedades de *widgets* e propriedades das respetivas *widgets* filhas, e caso os seus valores sejam compostos, como por exemplo registos de uma entidade, é possível navegar entre os atributos da entidade.

Widgets instanciadas em Xtext

As *widgets* definidas nos ficheiros importados pela aplicação, estão disponíveis para instanciar nos ecrãs.

A notação usada para instanciar as *widgets* aproxima-se de código `HTML`, onde cada elemento é uma widget, e a identificação da instância é efetuada no atributo name.

Considere-se a *widget* friendsList instanciada no ecrã da Listagem 4.10 (linha 16). É possível observar que na ação Save, onde se itera a sua propriedade dinâmica acedida através da expressão friendsList.list, é também possível aceder ao valor do elemento corrente da iteração através da expressão friendsList.list.current.Friend.

A instanciação de uma *widget* consiste em inicializar as propriedades simples, como é apresentado na Listagem 4.10, entre as *widgets* Expression com a sua propriedade value. Para além disso é possível definir as suas *widgets* filhas quando se define o conteúdo dos *placeholders*.

4.1.11 Observações

Relativamente à tipificação da linguagem, visível na criação de variáveis do ecrã na Listagem 4.10, esta não é usada para fins de validação. Existe apenas como auxílio da geração de código e, por essa razão, também não é representada na sintaxe da linguagem. Note-se portanto que, ao invocar uma operação sobre uma lista, não é verificado se a referência do identificador *id* corresponde a um valor do tipo List.

O gerador de código itera a definição do modelo de uma aplicação, gerando a arquitetura desejada, onde se garante a transmissão dos dados necessários entre cliente e servidor. É ao longo do processo de geração de código que é aplicada a análise estática. A implementação é apresentada na secção 6.

Através da utilização direta da plataforma `AngularJS` no cliente, mantém-se um padrão MVC no cliente, e tira-se partido da linguagem de *templating*. A Listagem 4.5 exemplifica o uso da ligação ao modelo {{ value }}, e a iteração de uma lista, através do atributo ng−repeat=list num elemento `HTML` indicando uma lista como argumento.

Neste protótipo, o modelo que define uma aplicação não é validado, logo assume-se que o modelo é definido corretamente pelo programador.

4.2 Suporte ao estado de um ecrã

Atualmente, as aplicações *OutSystems* preservam o estado de um ecrã, ao longo da comunicação entre cliente e servidor, constituído pelas variáveis locais e propriedades dinâmicas das *widgets*, através da transmissão do campo *ViewState*. Uma vez que o servidor não mantém sessões das instâncias das páginas, é necessário que o cliente mantenha o estado atualizado, consistente com as ações executadas. Como a linguagem proposta neste documento permite tanto ao cliente como ao servidor manipular dados através de ações, torna-se mais importante e desafiante manter a coerência do estado da aplicação.

Esta secção apresenta a proposta da estrutura de dados que é fornecida inicialmente pelo servidor e mantida no cliente, servindo de base à instanciação do *template*, e transmitida nos pedidos assíncronos de cada ecrã, substituindo assim o campo *ViewState* usado

atualmente na plataforma *OutSystems*. Note-se ainda que nos pedidos de um ecrã, o *ViewState* é recalculado e recarregado no navegador.

Como já foi referido anteriormente, as propriedades de uma *widget* podem ser inicializadas com expressões que devem ser calculadas fora do *template*, a fim da interface apresentar os valores calculados. Neste processo o ecrã (interface) pode mudar em virtude da reavaliação das expressões que o compõem. A reavaliação de expressões é necessária pois os seus subcomponentes podem ser dados da aplicação, que por sua vez podem ser manipulados por ações do ecrã. 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 servidor e sendo que os seus valores são mantidos na estrutura de dados de suporte, devem ser também retransmitidos quando modificadas, para atualizar a interface. Relativamente a ações no cliente, essas também podem manipular dados, logo as expressões devem ser reavaliadas no cliente, mas nunca é necessário retransmitir as expressões calculadas para o servidor.

O formato da estrutura apresentada é reconhecido pelo código presente no cliente e no servidor, permitindo a sua 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 Listagem 4.12, composto pelos seguintes elementos: o conjunto de variáveis locais do ecrã (newFriend, queryFriends); conjunto das propriedades dinâmicas das *widgets* (RuntimeProps); conjunto das expressões pré-calculadas das *widgets* e utilizadas para instanciar o *template* (EvalData); e um conjunto especial para as *widgets* com propriedades dinâmicas com tipo lista, que são compostas pelas *widgets* filhas instanciadas ao longo de uma iteração (friendsList_list).

Para além das variáveis locais do ecrã, existem as variáveis declaradas no corpo de funções e ações, que não são mantidas na estrutura, já que não existe qualquer estado a manter entre os pedidos. No entanto, se as variáveis das ações executadas diretamente a partir da interação com o ecrã fossem parâmetros de entrada, seria necessário manter o seu valor.

A necessidade de preservar os valores das propriedades dinâmicas, advém da sua manipulação entre as ações. Uma expressão pré-calculada é uma propriedade simples da *widget*, referida no *template*, cujo valor não é atómico, e dependente de variáveis ou propriedades dinâmicas, manipuláveis em ações. Acaba por corresponder ao conjunto exp excluindo os elementos x e k, uma vez que os valores literais (k) são estáticos no *template* HTML gerado, e os identificadores (x) são referências para propriedades ou variáveis na estrutura de suporte.

A expressão currentRowNumber, apresentada anteriormente, embora não seja uma propriedade ou variável é enviada juntamente com o modelo. Quando uma ação é desencadeada na linha de uma lista e é executada no servidor, este deve conhecer o índice em

```
model: { newFriend: { Friend: { Name: "" } },
  queryFriends: [ { Friend: { Name: "Sara" }}, ... ],
  RuntimeProps: { inputFriend: { valid: true, validationMessage: "" } } },
  EvalData: { ifCount: { conditionValue: true }, result: { value: "$12.5" }, ... },
  friendsList_list: [
    { Item: {Friend: { Name: "Sara"} },
      RuntimeProps: { friendCheck: {...} }
      EvalData: { } }, ... ] }
```

Listagem 4.12: Exemplo da estrutura de suporte.

questão.

A diminuição do volume de dados transmitidos é obtida pelo seccionamento desta estrutura. Para isso, é necessária uma análise estática que determine o conjunto mínimo de dados estritamente necessários. Na secção seguinte apresenta-se uma análise estática, aplicada ao modelo de uma aplicação, que permite esta otimização.

Árvore de widgets

A estrutura aqui proposta derivou de uma abordagem analisada previamente, com uma definição mais direta da estrutura a ser trocada entre cliente e servidor, composta pelas variáveis locais e a árvore de *widgets* do ecrã, onde cada *widget* seria composta pelas suas propriedades e *widgets* filhas.

Considere-se um ecrã composto por uma ação que altera o valor de uma variável. Essa variável é referenciada por uma propriedade simples de uma *widget*. No fim da execução da ação é necessário reavaliar a propriedade da *widget*, mas o ecrã pode conter muitas mais *widgets* com propriedades dependentes dessa variável. A solução passa portanto por reconstruir a árvore de *widgets*. Outro cenário é quando a mesma ação manipula uma propriedade dinâmica de outra *widget*, logo o novo valor perde-se quando toda a árvore é reconstruída de raiz.

Este comportamento permitiu detetar os potenciais problemas no suporte do estado de um ecrã, acabando por nos conduzir à estrutura aqui proposta.

4.3 Sumário

A linguagem apresentada permite abstrair vários detalhes da linguagem da plataforma *OutSystems* e, ainda assim, definir o modelo de uma aplicação incluindo as camadas de modelo de dados, lógica, e interface.

Este modelo permite-nos definir uma solução de forma mais objetiva e ágil, explorando através da definição de uma aplicação a respetiva geração de código.

Com o suporte ao estado de um ecrã, através da estrutura apresentada, em conjunto com a análise estática apresentada na secção 5, é garantida a transmissão apenas dos

dados necessários, para atualizar as interfaces e executar as ações. A criação desta linguagem permitiu obter um modelo simplificado das aplicações *OutSystems*, simplificar alguns detalhes de desenvolvimento e, explorar novas funcionalidades.

Com este modelo obtém-se a atualização automática da interface perante alterações de dados da aplicação, o que é possível pois a aplicação gerada produz um MVC no cliente com base na *framework* `AngularJS`. Como tal, a operação `AjaxRefresh` da linguagem *OutSystems* usada para atualizar elementos específicos da página, tendo em conta os dados alterados, torna-se obsoleta.

Ao contrário da linguagem *OutSystems*, é possível iterar uma lista dinâmica e aceder às *widgets* filhas do índice corrente, sem que exista um evento desencadeado no contexto de uma linha da lista[1]. Sobre este modelo torna-se também possível definir aplicações com a sua lógica distribuída entre cliente e servidor, através da declaração de ações executáveis no cliente ou no servidor. Um comportamento que não é possível expressar sobre a plataforma *OutSystems*.

[1] Ver secção da *OutSystems* sobre o comportamento atual do acesso ao current de uma lista dinâmica

5

Análise Estática

Descreve-se agora a análise estática aplicada ao modelo de uma aplicação, definida sobre a linguagem modelo proposta, no sentido de minimizar os dados transmitidos na rede entre cliente e servidor. Pretende-se também garantir que todas as expressões pré-calculadas, afetadas pela execução de uma ação, são recalculadas no servidor ou cliente, e atualizadas na página no cliente.

A técnica de análise estática apresentada neste trabalho baseia-se no grafo do fluxo de controlo (CFG) do modelo de um ecrã. No âmbito da linguagem modelo proposta, é gerado um grafo ao nível de cada ecrã da aplicação. O grafo de um ecrã é dividido em subconjuntos de nós, que correspondem às ações do ecrã e ao modelo de dados consumido pelo ecrã. A ação é composta por várias instruções, cada uma representada por um nó do grafo. Cada *widget* instanciada num ecrã, também é representada por um nó, que contém o conjunto de dados consumido pela *widget*.

O grafo do subconjunto de nós de uma ação é direcionado, ou seja, um nó representativo da instrução n que seja ligado a outro nó, representativo da instrução m, indica que m é sucessora de n. Na Figura 5.1 é ilustrado o CFG respetivo à ação AddFriend. Por outro lado, o grafo do modelo de dados do ecrã não apresenta qualquer ligação entre os nós.

O ecrã Bill, definido na Listagem 4.10, origina o grafo representado na Figura 5.1. Cada retângulo representa um sub conjunto de nós do grafo, identificado pelos nomes das ações e pelo modelo de dados consumido pelas *widgets* do ecrã, Model.

O *template* tipo cliente é gerado com base no código `HTML` de cada *widget*, no qual são referenciadas propriedades, que coincidem com os dados consumidos pelo ecrã. São aplicadas otimizações ao longo do processo de geração, relativamente às referências que são colocadas no *template*, que incluem as propriedades simples das *widgets*. Por exemplo,

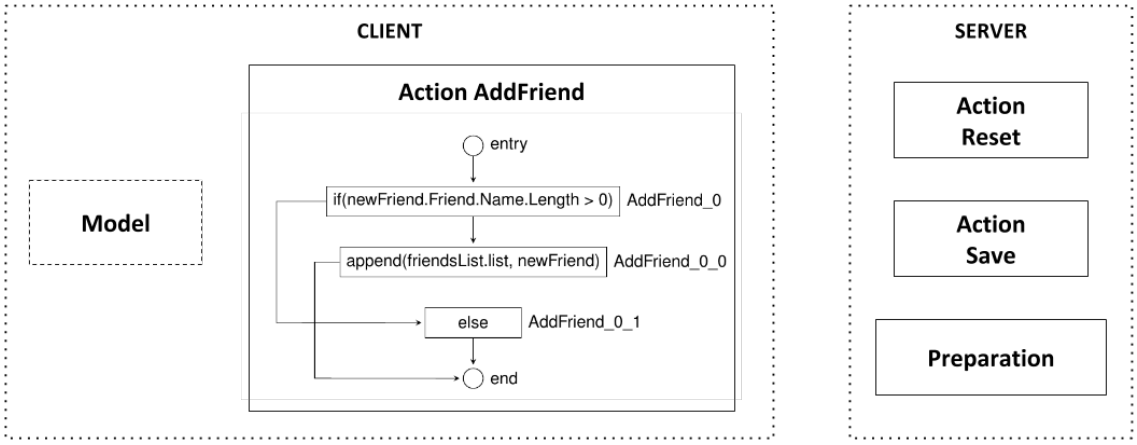


Figura 5.1: Grafo do fluxo de controlo dos componentes da aplicação, com o sub grafo da ação AddFriend expandido.

se uma propriedade simples for inicializada com um valor literal, e tendo em conta que uma propriedade simples não é alterável, então o valor é expressado diretamente no *template*.

Não existe uma ligação entre os sub conjuntos do grafo, apenas a transmissão de um conjunto de dados, que é inferido através da análise. Neste exemplo as ações do tipo servidor (Reset e Save) comunicam com o Model no cliente, que fornece a estrutura de dados associada ao *template* do ecrã mantendo a interface atualizada. O cliente comunica com o servidor para invocar ações do ecrã executáveis no servidor.

Neste trabalho opta-se pela definição da análise estática usando a linguagem `Datalog` [16] e as ferramentas associadas, em vez dos tradicionais algoritmos de ponto fixo [10], o que permite a utilização de bibliotecas com mais facilidade não perdendo eficiência na sua execução. Também se torna mais fácil definir as várias propriedades a analisar e as regras de propagação das mesmas. A integração desta análise no compilador da linguagem, criada neste trabalho, é apresentada na secção 6.

A análise estática divide-se essencialmente em duas partes:

Análise de *liveness*: Determina quais as variáveis e propriedades dinâmicas de *widgets* que são necessárias para executar uma determinada ação.

Análise de dependências: Determina quais as expressões que devem ser recalculadas no final da execução de uma ação (no cliente ou servidor), com base nas variáveis ou propriedades dinâmicas modificadas.

Ambas as análises são combinadas de forma a que o cliente e o 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 Listagem 4.10 contém referências a expressões pré-calculadas, por exemplo, a divisão da variável total pelo tamanho da propriedade list (linha 11). A execução da ação AddFriend provoca a alteração do tamanho de

list, o que faz com que a página não apresente conteúdo atualizado. Assim, o cliente deve recalcular a expressão que efetua a divisão. Assumindo, por exemplo, que no servidor é incorporada uma ação que incrementa a variável local total em 2%, então o cliente deve fornecer ao servidor a variável total. Isto porque, essa variável é usada para calcular os 2%, que é incrementado à própria variável. Mas também é usada para recalcular a expressão com a divisão referida, uma vez que uma das variáveis de que depende (total) foi alterada.

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

Em sequência do capítulo 4, onde é apresentada a sintaxe abstrata da linguagem do modelo proposto, no presente capítulo o conjunto $\mathcal{V}$ inclui os elementos v e p_r, apresentados na Figura 4.2 como os identificadores de variáveis e propriedades dinâmicas de *widgets*, respetivamente. Neste contexto, representam os elementos que podem ser manipulados em ações. Relativamente às expressões pré calculadas exp, são propriedades simples p_d, apresentadas na Figura 4.2, onde se observa que a definição do elemento $html$ pode conter referências a identificadores p_d. Neste capítulo, representam os dados consumidos pelo *template* tipo cliente, que devem ser pré calculados no servidor.

5.1 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 ou propriedades dinâmicas modificadas. Introduzem-se 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.

No anexo F.1 são apresentados os factos extraídos do modelo do ecrã Bill, definido na Listagem 4.10. Com base nestes factos definimos um conjunto de regras representadas na Figura 5.2, para exprimir as propriedades pretendidas na análise de dependências entre as ações de um ecrã e as expressões pré-calculadas. São elas,

mod(a, v)	A ação a modifica a variável v.
needEval(a, f)	Após a execução da ação a, a expressão identificada por f deve ser recalculada.

A partir das três regras apresentadas na Figura 5.2, são inferidos os factos dos predicados mod e needEval.

Na regra (1) define-se que as variáveis modificadas numa ação são aquelas que são redefinidas num dos seus nós.

A regra (2) define que, no fim da execução da ação a, é necessário recalcular as expressões pré-calculadas (f). Uma expressão f deve ser recalculada se alguma das variáveis ou propriedades dinâmicas (v) usadas na sua definição (need(f, v)) é modificada num dos nós da ação (mod(a, v)). Por exemplo, no ecrã definido na Listagem 4.10 temos a expressão pré-calculada costPPerson.value, cuja definição usa total e friendsList.list, que se expressa através do facto need com: need(costPPerson.value), total) e need(costPPerson.value, friendsList.list).

A regra (3) determina a transitividade da relação de dependência. No ecrã definido na Listagem 4.10 é instanciada a *widget* ifCount, cujo *placeholder* trueBranch contém a *widget* costPPerson. O `HTML` da *widget* ifCount estabelece um condicionamento da visualização do conteúdo de cada *placeholder*, consoante o valor booleano da propriedade conditionValue. O recálculo da propriedade value de costPPerson está dependente da alteração da expressão pré-calculada conditionValue. É declarado o facto need(costPPerson.value, ifCount.conditionValue), verificando-se needEval(a, ifCount.conditionValue) caso o tamanho da lista modifique, logo verifica-se tambémneedEval(a, costPPerson.value).

As regras da Figura 5.2 são aplicadas também ao nível das *widgets* do ecrã. Observe-se a declaração da *widget* Input, apresentada no anexo E. O seu `HTML` consiste no elemento input com o atributo ng−model do `AngularJS`, o qual permite alterar diretamente a partir da página `HTML` o modelo de dados mantido no cliente. O atributo recebe a referência da propriedade variable, a qual é inicializada na instanciação da *widget*, representada na Listagem 4.10, com referência à variável local total do ecrã. Isto significa que a partir da página no cliente, o utilizador pode alterar o valor da variável total através do elemento input. Posto isto, as expressões pré-calculadas, presentes na página, ficam desatualizadas.

Com recurso ao mecanismo do `AngularJS` espera-se que as expressões sejam automaticamente atualizadas. No entanto, os seus valores são pré-calculados, ou seja, não correspondem a funções que possam ser executadas assim que se deteta uma alteração no modelo. Por isso, é necessário detetar as alterações provocadas através das *widgets* do ecrã, e consequentemente as expressões a recalcular com dependências a variáveis ou propriedades dinâmicas alteradas.

Considerando o mesmo exemplo da Listagem 4.10, com base na definição do `HTML` e nas *widgets* instanciadas, são extraídos os factos: def(Bill_inputTotal,total),

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

Figura 5.2: Análise de dependências.

need(costPPerson.value, total), node(Bill, Bill_inputTotal). A partir destes factos podemos deduzir mod(Bill, total) e consequentemente needEval(Bill, costPPerson.value). Estes resultados permitem deduzir as expressões a recalcular ao nível do ecrã, conforme a parte do modelo de dados que é alterada através da interação do utilizador. No sentido de identificar as expressões que devem ser recalculadas devido à interação com uma *widget*, o facto node extraído da *widget* instanciada é ligeiramente adaptado. Assim são extraídos os mesmos factos, exceto o facto node que corresponde agora a node(Bill_inputTotal, Bill_inputTotal). Desta forma, concluímos mod(Bill_inputTotal, total) e consequentemente needEval(Bill_inputTotal, costPPerson.value), logo quando existe uma interação com o input da *widget* inputTotal, a expressão costPPerson.value é recalculada.

A análise de dependências apresentada tem o objetivo de identificar as expressões que devem ser recalculadas no fim de uma ação, com base nas dependências com os dados que são alterados. Desta forma, minimizamos os valores calculados que são transmitidos do servidor para o cliente.

5.2 Análise *Liveness*

A segunda análise permite determinar o conjunto de variáveis necessárias para a execução de uma ação que, 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, acrescentamos os seguintes:

$\mathsf{use}(n,v)$	A variável v é lida no nó n, ou ocorre na computação 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.
$\mathsf{live}(n,v)$	A variável v é necessária à entrada do nó n.
$\mathsf{live}(s,v)$	A variável v é necessária no ecrã s.

As regras definidas na Figura 5.3 permitem determinar o subconjunto de dados a ser transmitido do servidor para o cliente, e vice-versa. O cliente deve enviar para o servidor

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

Figura 5.3: Análise de *liveness*.

o conjunto de dados determinado pelas regras (4) e (5), no nó de entrada de cada ação invocada.

A regra (4) define que, se uma variável ou propriedade dinâmica v é lida no nó n então o predicado $\mathsf{live}(n, v)$ verifica-se. A regra (5) verifica essa definição para os nós sucessores de n, e caso no nó n a variável/propriedade v não seja escrita, então verifica-se $\mathsf{live}(n, v)$.

Como resposta a um pedido, 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. Note-se que todas as ações são invocadas a partir do cliente, não existindo nenhuma ordem específica para a sua invocação, por isso, os dados fornecidos pelo servidor devem ter em conta os dados necessários por todas as ações. A análise de *liveness* aqui apresentada tem como ponto de partida o nó de entrada do grafo de uma ação (identificado pelo facto entry), com a iteração pelo fluxo de controlo, ou seja com recursividade para os nós sucessores.

A regra (7) é um caso especial do facto use. Note-se que, no final de uma ação, um conjunto de expressões devem ser recalculadas consoante as variáveis ou propriedades modificadas ao longo da ação. Neste sentido, considera-se que uma variável v ocorre no nó n, se não é modificada ao longo da ação a, mas é necessária ao recalcular uma expressão que depende de v e de u, em que u é modificada na 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, e apenas ocorre no sentido do servidor para o cliente. Não se pretende que o cliente envie para o servidor expressões pré calculadas.

5.3 Aplicação das análises

O resultado esperado da aplicação das duas análises é a troca de dados, entre cliente e servidor, estritamente necessários. Quando o cliente invoca uma ação do servidor, é enviado o conjunto de dados determinado pela regra (8) apresentada na Figura 5.4, que obtém o conjunto de variáveis necessárias para a execução da ação, considerando os resultados da análise de *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) apresentadas na Figura 5.4. Obtém assim apenas as variáveis que são necessárias no cliente para executar qualquer ação, excluindo as que não foram modificadas.

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

Figura 5.4: Resultados finais de CtoS (*Client to Server*) e StoC (*Server to Client*).

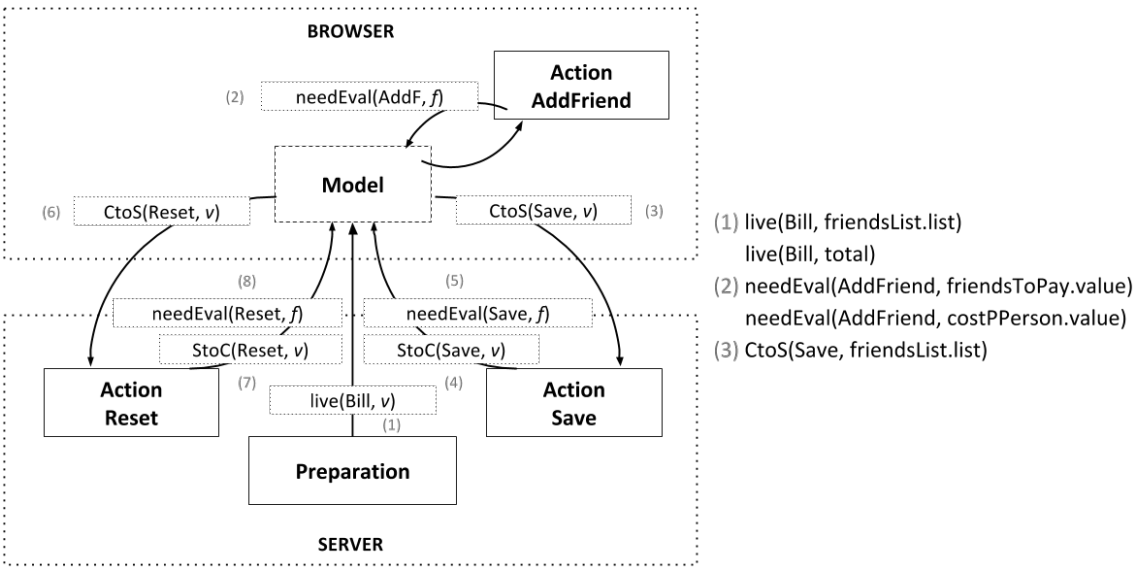


Figura 5.5: Comunicação entre os componentes da aplicação.

A Figura 5.5 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 5.5, a partir dos quais é gerada a estrutura apresentada na secção 4.2.

Durante a geração de código, estes resultados são usados da seguinte maneira: para cada ação do tipo servidor, são determinados, o conjunto de variáveis locais e propriedades dinâmicas alteradas pela ação (predicado $\mathsf{StoC}(A, v)$), e o conjunto das expressões que necessitam de ser recalculadas no final visto serem necessárias no cliente para atualizar a página `HTML` (predicado $\mathsf{needEval}(A, e)$).

No exemplo da aplicação 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 necessita da variável friendsList.list, e no fim são recalculadas as expressões costPPerson.value e friendsToPay.value. A ação Reset opera apenas sobre a base de dados, logo não são transmitidos quaisquer dados.

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 CtoS(A, v), e reavaliação de expressões no servidor, pelo predicado needEval(A, e). A invocação da ação AddFriend implica recalcular as expressões friendsToPay.value e costPPerson.value.

A combinação das duas análises alcança o envio dos dados estritamente necessários, o que contribui não só para a redução do volume de conteúdo transmitido na rede, mas também para a segurança da aplicação. Nas aplicações *OutSystems* as computações de expressões que usem outros dados são sempre efetuadas no servidor, e os valores são incluídos na página `HTML` gerada. Como tal, os dados usados na sua computação nunca são enviados para o cliente, dados esses que poderiam comprometer a segurança da aplicação caso o cliente lhes tivesse acesso. Este comportamento mantém-se na solução proposta neste trabalho, como resultado da análise de dependências.

5.4 Geração de factos

No protótipo desenvolvido neste trabalho, a análise estática é aplicada durante o processo de geração de código. Como mencionado anteriormente, a definição da análise é expressa em `Datalog`, através de um programa composto por factos, regras e interrogações.

O processo de geração de código inclui uma fase que se destina à análise estática sobre o modelo da aplicação, apresentada no capítulo 6. Parte dessa fase corresponde à declaração das regras apresentadas ao longo desta secção. Por exemplo a regra (9) da Figura 5.4 é declarada como a seguinte regra de inferência:

```
StoC(?a, ?v) :- action(?s, ?a), live(?s, ?v), mod(?a, ?v).
```

O predicado StoC é também usado para interrogar a base de conhecimento, formada pelos factos declarados e inferidos por outras regras, para obter que variáveis devem ser enviadas do servidor para o cliente em sequência da execução da ação AddFriend:

```
?-StoC('AddFriend', ?v).
```

Ao longo da fase de análise estática do processo de geração de código, tudo o que é declarado (regras) e gerado a partir do modelo da aplicação (factos e interrogações), é concatenado numa *string*. A *string* equivale ao programa `Datalog`, interpretado pela ferramenta `IRIS`, que produz o conjunto de resultados das interrogações à base de conhecimento.

Nesta secção apresenta-se o algoritmo responsável por gerar os factos, que em conjunto com o conjunto de regras apresentadas, constituem o programa `Datalog` obtido no fim do processo de geração de código.

A extração de factos da definição do modelo de uma aplicação é baseada na iteração da árvore de elementos definidos no modelo, que se divide em: primeiro, as *widgets* do

ecrã, que inclui iterar a definição do HTML de todas *widgets* instanciadas, inclusive as contidas nos seus *placeholders*; segundo, as instruções das ações de um ecrã, inclusive as restantes instruções no seu corpo, como é o caso do if else e forEach.

A apresentação do algoritmo usa uma anotação de conjuntos, formados pelos factos gerados ao longo da iteração pelo modelo da aplicação, e cuja declaração é extraída diretamente da definição do modelo.

A sintaxe abstrata da linguagem modelo, apresentada nas Figuras 4.1 e 4.2, define a ação como um conjunto de variáveis locais e instruções. Cada instrução equivale a um nó no sub conjunto do CFG respetivo à ação, podendo variar entre um conjunto de operações possíveis.

Os factos a gerar são: node, def, use, exp e need. Em particular, ao nível das ações do ecrã, são também gerados os factos succ, action e entry. Os restantes predicados são inferidos através das regras declaradas.

A necessidade de gerar todos estes factos deve-se às duas análises apresentadas anteriormente. As interrogações à base de conhecimento devem ser deterministas e, para obter os resultados esperados, é necessário distinguir os elementos do modelo da aplicação. Por exemplo, quando efetuada a interrogação CtoS('AddFriend', ?v) é necessário conhecer qual o ponto de entrada da ação AddFriend, que é identificado pelo facto entry. Se o facto for declarado, assume-se que ?v corresponde agora a AddFriend_0. Depois é intersetado com os factos deduzidos pela regra live('AddFriend_0', ?v), e assim obtemos quais a variáveis a enviar do cliente para o servidor.

Facto DEF

Apresenta-se agora o algoritmo para a geração dos factos def ao nível das instruções de cada ação de um ecrã, aplicado ao longo da iteração das ações de um ecrã, e sucessivamente das instruções de cada ação.

Cada instrução equivale a um nó no CFG necessitando, como tal, de uma identificação, como é ilustrado na Figura 5.1, com o CFG da ação AddFriend. Esta identificação consiste em concatenar o nome da ação com o índice da instrução, separados por _. Caso a instrução da ação a inclua outras instruções no seu corpo, a identificação das sub instruções são identificadas por a_i_j, sendo j o índice da sub instrução, e i o índice da instrução. A identificação do nó corrente é transmitida ao longo do algoritmo como n, e.g., DEF_n.

O algoritmo apresenta $\text{DEF}_n(arg)$ como uma função recursiva, cujo retorno é composto pelos factos def, e arg corresponde aos vários tipos de instruções de uma ação, ou a um identificador, que são iterados ao longo do algoritmo apresentado na Figura 5.6.

Ao nível das *widgets* instanciadas num ecrã podem também ser extraídos factos def, resumindo-se à situação em que é possível alterar variáveis ou propriedades dinâmicas através da página HTML. Essa afetação é permitida através do atributo ng−model do

$$
\begin{array}{rcl}
\textsc{Def}(a) & = & \bigcup_{i=0}^{j} \textsc{Def}_{a\_i}(st_i) \quad , st_i \in fbody \in a \\
\textsc{Def}_n(id = exp) & = & \textsc{Def}_n(id) \\
\textsc{Def}_n(listop(id, exp)) & = & \textsc{Def}_n(id) \\
\textsc{Def}_n(\mathsf{foreach}(id)\{\overline{st}\}) & = & \bigcup_{i=0}^{j} \textsc{Def}_{n\_i}(st_i) \\
\textsc{Def}_n(\mathsf{if}(exp)\{\overline{st}\}\mathsf{else}\{\overline{st}\}) & = & \bigcup_{i=0}^{j} \textsc{Def}_{n\_i}(st_i) \\
\textsc{Def}_n(\mathsf{refreshQuery}(id)) & = & \textsc{Def}_n(id) \\
\textsc{Def}_n(\mathsf{refreshDataSource}(z)) & = & \textsc{Def}_n(z.p_r) \quad , init \in w \wedge p_r \in init \\
\textsc{Def}_n(id) & = & \begin{cases} \mathsf{def}(n, id) & , se\ id = (v \vee z.p_r) \\ \emptyset \end{cases}
\end{array}
$$

Figura 5.6: Algoritmo de geração dos factos DEF ao nível das ações dos ecrãs.

`AngularJS`. Neste caso, as *widgets* representam também nós no CFG, onde o identificador n corresponde à concatenação do nome do ecrã com o nome da instância da *widget*, separados também por _. Para esta geração são iteradas as *widgets* instanciadas num ecrã s e é aplicado o algoritmo apresentado na Figura 5.7

$$
\begin{array}{rcl}
\textsc{Def}_n(z) & = & \bigcup_{i=0}^{j} \textsc{Def}^{z}_{n\_z}(html_i) \\
\textsc{Def}^{z}_{n}(id) & = & \mathsf{def}(n, id) \quad , se\ id = (z.p_d \vee z.p_r) \\
\textsc{Def}^{z}_{n}(html) & = & \{\bigcup_{i=0}^{j} \textsc{Def}^{z}_{n}([ng_{attr} = p_{d,r}]_i)\} \cup \{\bigcup_{i=0}^{j} \textsc{Def}^{z}_{n}(html_i)\} \\
\textsc{Def}^{z}_{n}(ng_{attr} = p_{d,r}) & = & \begin{cases} \textsc{Def}^{z}_{n}(p_{d,r}) & , se\ ng_{attr} = \mathsf{ng\text{-}model} \\ \emptyset \end{cases} \\
\textsc{Def}^{z}_{n}(p_h) & = & \bigcup_{i=0}^{j} \textsc{Def}_{n}(z_i) \quad , se\ p_h(\overline{Winst}) \in z \wedge z_i \in \overline{Winst}
\end{array}
$$

Figura 5.7: Algoritmo de geração dos factos DEF ao nível das *widgets* dos ecrãs.

Nota. O identificar n antes da iteração das *widgets* instanciadas é inicializado com o nome do ecrã. Ao longo da iteração, para além do nome da *widget* concatenado com o nome do ecrã, pode ainda ser concatenado com o nome de *widgets* filhas.

Facto USE

O algoritmo da geração dos factos use é apresentado na Figura 5.8 , que surge da iteração das ações de cada ecrã. Tal como na geração dos factos def, os nós do CFG correspondentes às instruções da ação são identificados da mesma forma.

Note-se que o *template* tipo cliente gerado ao nível de um ecrã também consome dados e, por isso, é necessário extrair os factos use das *widgets* de um ecrã, pois a definição do seu `HTML` é usada na criação do *template*.

Recordando as regras apresentadas anteriormente, os factos use ao nível do *template*

$$
\begin{array}{rcl}
\textsc{Use}(a) & = & \bigcup_{i=0}^{j} \textsc{Use}_{a\_i}(st_i) \quad , st_i \in fbody \in a \\
\textsc{Use}_n(id) & = & \mathsf{use}(n, id) \quad , se\ id = (v \vee z.p_d \vee z.p_r) \\
\textsc{Use}_n(exp_1\ op\ exp_2) & = & \textsc{Use}_n(exp_1) \cup \textsc{Use}_n(exp_2) \\
\textsc{Use}_n(f(\overline{exp})) & = & \bigcup_{i=0}^{j} \textsc{Use}_{n\_i}(exp_i) \\
\textsc{Use}_n(\mathsf{length}(id)) & = & \textsc{Use}_n(id) \\
\textsc{Use}_n(\mathsf{foreach}(id)\{\overline{st}\}) & = & \textsc{Use}_n(id) \cup \{\bigcup_{i=0}^{j} \textsc{Use}_{n\_i}(st_i)\} \\
\textsc{Use}_n(\mathsf{if}(exp)\{\overline{st}\}\mathsf{else}\{\overline{st}\}) & = & \textsc{Use}_n(exp) \cup \{\bigcup_{i=0}^{j} \textsc{Use}_{n\_i}(st_i)\} \\
\textsc{Use}_n(listop(id, exp)) & = & \textsc{Use}_n(id) \cup \textsc{Use}_n(exp) \\
\textsc{Use}_n(\mathsf{refreshDataSource}(z)) & = & \textsc{Use}_n(z.p_d) \quad , init \in w \wedge p_d \in init \\
\textsc{Use}_n(dbop(e, exp)) & = & \begin{cases} \textsc{Use}_n(exp) & , se\ dbop = \mathsf{dbcreate} \vee \mathsf{dbupdate} \\ \emptyset & \end{cases}
\end{array}
$$

Figura 5.8: Algoritmo de geração dos factos USE ao nível das ações dos ecrãs.

são necessários para inferir $\mathsf{live}(s, v)$, tal como os factos def gerados a partir das *widgets* do ecrã. O algoritmo para gerar os factos use é apresentado na Figura 5.9.

Aqui é necessário transmitir, ao longo das chamadas recursivas, o identificador da *widget*, correspondente a n (composto pelo nome do ecrã e nome da instância da *widget*), e o nome da instância da *widget*, para que seja possível reconhecer a instância da *widget* que se está a iterar, e dessa forma aceder às suas propriedades.

$$
\begin{array}{rcl}
\textsc{Use}^z_n(id) & = & \mathsf{use}(n, id) \quad , id = v \vee z.p_r \\
\textsc{Use}_n(id = exp) & = & \textsc{Use}_n(exp) \\
\textsc{Use}_n(z) & = & \{\bigcup_{i=0}^{j} \textsc{Use}^z_{n\_z}([p_d = exp]_i)\} \cup \{\bigcup_{i=0}^{j} \textsc{Use}^z_{n\_z}(html_i)\} \\
\textsc{Use}^z_n(html) & = & \{\bigcup_{i=0}^{j} \textsc{Use}^z_n([html_{attr} = p_{d,r}]_i)\} \cup \\
 & & \{\bigcup_{i=0}^{j} \textsc{Use}^z_n([ng_{attr} = p_{d,r}]_i)\} \cup \{\bigcup_{i=0}^{j} \textsc{Use}^z_n([html]_i)\} \\
\textsc{Use}^z_n(html_{attr} = p_{d,r}) & = & \textsc{Use}^z_n(p_{d,r}) \\
\textsc{Use}^z_n(\{\{p_{d,r}\}\}) & = & \textsc{Use}^z_n(p_{d,r}) \\
\textsc{Use}^z_n(p_h) & = & \bigcup_{i=0}^{j} \textsc{Use}_n(z_i) \quad , se\ p_h(\overline{Winst}) \in z \wedge z_i \in \overline{Winst} \\
\textsc{Use}^z_n(ng_{attr} = p_{d,r}) & = & \begin{cases} \textsc{Use}^z_n(p_{d,r}),\ ng_{attr} = (\mathsf{ng\text{-}if} \vee \mathsf{ng\text{-}repeat}) \\ \emptyset \end{cases}
\end{array}
$$

Figura 5.9: Algoritmo de geração dos factos USE ao nível das *widgets* dos ecrãs.

Nota. O identificar n antes da iteração das *widgets* instanciadas é inicializado com o nome do ecrã. Ao longo da iteração, para além do nome da *widget* concatenado com o nome do ecrã, pode ainda ser concatenado com o nome de *widgets* filhas.

Facto SUCC

Os factos succ definem a noção de sucessor entre as instruções (nós do grafo), no contexto de uma ação. Embora seja ocultado na linguagem, a computação de expressões pré-calculadas na análise estática - desejando-se a transparência para o programador - deve ser representada. Para efetuar a computação de expressões no servidor, necessita de dados extra que devem ser inferidos através da análise. Nesse sentido, acresce às instruções de uma ação uma outra instrução, que representa a computação de expressões pré-calculadas, identificadas pela análise de dependências. Desta forma, o último nó de uma ação corresponde sempre à avaliação de expressões.

Os factos node representam o nó de cada instrução ou *widget* do ecrã e, em particular, o facto entry define um nó como o ponto de entrada de uma ação, ou seja, a primeira instrução.

Os factos action são gerados para cada ação do ecrã, criando a relação entre ações e as suas instruções, agora representadas por nós.

O algoritmo que gera os factos succ, action e node é apresentado na Figura 5.10, e sucede-se ao iterar as instruções das ações.

$$
\begin{array}{rcl}
\textsc{Succ}_s(a) & = & \textsc{Succ}_a(fbody) \cup \mathsf{action}(s,a) \\
\textsc{Succ}_a(fbody) & = & \bigcup_{i=0}^{j} \textsc{Succ}_{a\_i}(st_i) \cup \forall_{i<j}\mathsf{succ}(a\_i, a\_(i+1)) \cup \\
 & & \mathsf{succ}(a, a\_0) \cup \mathsf{node}(a, a\_i) \cup \\
 & & \mathsf{succ}(a\_j, a\_eval) \quad , st_i \in fbody \\
\textsc{Succ}_n(\mathsf{foreach}(id)\ \overline{st}\ \mathsf{end}) & = & \bigcup_{i=0}^{j} \textsc{Succ}_{n\_i}(st_i) \cup \forall_{i<j}\mathsf{succ}(n\_i, n\_(i+1)) \cup \\
 & & \mathsf{succ}(n, n\_0) \cup \mathsf{node}(n, n\_i) \\
\textsc{Succ}_n(\mathsf{if}\ exp\ \overline{st1}\ \mathsf{else}\ \overline{st2}) & = & \{\bigcup_{i=0}^{j} \textsc{Succ}_{n\_i}(st1_i) \cup \forall_{i<j}\mathsf{succ}(n\_i, n\_(i+1))\} \cup \\
 & & \{\bigcup_{q=j}^{r} \textsc{Succ}_{n\_i}(st2_q) \cup \forall_{q<j}\mathsf{succ}(n\_q, n\_(q+1))\} \cup \\
 & & \mathsf{node}(n, n\_i, q) \cup \mathsf{succ}(n, n\_0) \cup \mathsf{succ}(n, n\_j)
\end{array}
$$

Figura 5.10: Algoritmo de geração dos factos SUCC ao nível das ações dos ecrãs.

Nota. As quantificações presentes ao longo do algoritmo são usadas para limitar a geração de factos succ, não gerando identificadores de nós sucessores não existentes. Note-se que existe um incremento sempre que é definido o facto succ do corpo de uma ação ou instrução. O último nó de uma ação é definido como a computação das expressões pré calculadas ao gerar o facto $\mathsf{succ}(a\_j, a\_eval)$, onde j é o índice da última instrução.

Facto NEED

O algoritmo que gera os factos need e exp, ao longo da iteração pelas *widgets* instanciadas num ecrã, é apresentado na Figura 5.11. Os factos exp permitem distinguir entre todos os identificadores de elementos da aplicação os que representam expressões pré-calculadas.

$$
\begin{array}{rcl}
\textsc{Need}_g(z) & = & \textsc{Need}^z_g(w) \\
\textsc{Need}^z_g(w) & = & \bigcup_{i=0}^{j} \textsc{Need}^z_g(html_i) \\
\textsc{Need}^z_g(html) & = & \begin{cases}
\textsc{Need}^z_p(html_i) \cup \\
\{\bigcup_{i=0}^{j} \textsc{Need}^z_p([ng_{attr} = p]_i)\} \cup \\
\{\bigcup_{i=0}^{j} \textsc{Need}^z_g([html_{attr} = p]_i)\}, \exists\, \textsf{ng-if} = p \in html \\
\\
\textsc{Need}^z_g(html_i) \cup \{\bigcup_{i=0}^{j} \textsc{Need}^z_g([html_{attr} = p_{d,r}]_i)\}
\end{cases} \\
\textsc{Need}^z_g(ng_{attr} = p) & = & \textsc{Need}_{p,g}(\textsc{ExpMap}(z.p)) \quad , se\, p = p_d \\
\textsc{Need}^z_g(html_{attr} = p) & = & \textsc{Need}_{p,g}(\textsc{ExpMap}(z.p)) \quad , se\, p = p_d \\
\textsc{Need}^z_g(\{\{p\}\}) & = & \textsc{Need}_{p,g}(\textsc{ExpMap}(z.p)) \quad , se\, p = p_d \\
\textsc{Need}^z_g(p_h(\overline{Winst})) & = & \textsc{Need}_g(z_i) \quad , se\, z_i \in \overline{Winst} \\
\textsc{Need}_{f,g}(exp) & = & \begin{cases}
\textsc{Need}_{f,g}(exp_1) \cup \textsc{Need}_{f,g}(exp_2) & , exp = exp_1\ op\ exp_2 \\
\textsc{Need}_{f,g}(exp_i) & , exp = f(\overline{exp}) \\
\textsc{Need}_{f,g}(id) & , exp = \textsf{length}\ id \\
\textsc{Need}_{f,g}(id) & , exp = \textsf{currentRow}\ id \\
\textsf{need}(f, id) \cup \textsf{exp}(f) \cup \textsf{need}(f, g) & , exp = id \wedge g \neq \varnothing \\
\textsf{need}(f, id) \cup \textsf{exp}(f) & , exp = id \\
\emptyset &
\end{cases}
\end{array}
$$

Figura 5.11: Algoritmo de geração dos factos NEED ao nível das *widgets* dos ecrãs.

As propriedades simples são inicializadas aquando da sua instanciação. Essas propriedades podem ser referenciadas na definição do `HTML` da *widget*. A fim de simplificar o algoritmo, assume-se que existe um dicionário EXPMAP que mapeia as propriedades simples de *widgets* com os valores de inicialização. Os valores contidos no dicionário são expressões que, na sintaxe da linguagem apresentada no capítulo 4, correspondem a exp. Uma consulta $\textsc{ExpMap}(z.p)$ devolve portanto um elemento exp. Desta forma, o dicionário é consultado ao iterar o `HTML` da *widget* instanciada e, quando existem atributos do `AngularJS` ou do elemento `HTML` a serem inicializados com propriedades da *widget*, é consultado diretamente com que valores a propriedade foi inicializada na instanciação. Assim, obtemos uma relação direta entre dependências de valores. Considere-se uma *widget* com a propriedade $condition$, e cuja definição do `HTML` contém o atributo ng-model que referencia a propriedade anterior. Ao instanciar a *widget*, inicializamos a propriedade com a variável $test$ do ecrã. Temos:

$$
\begin{array}{rcll}
condition & = & test & \text{, inicialização da propriedade na instanciação da } \textit{widget} \\
\textsf{ng-if} & = & condition & \text{, inicialização de atributo na definição do } \texttt{HTML} \text{ da } \textit{widget}
\end{array}
$$

Perante isto, deve ser gerado o facto $\textsf{need}(condition, test)$. Para tal é necessário ter o conhecimento com que valor foi a propriedade inicializada na instanciação da *widget* corrente da iteração.

Quando as expressões de *widgets* dependem de uma expressão da *widget* pai que, condicione a sua visualização (ver secção 5.1), é necessário manter a expressão condicional ao longo da iteração, identificada por g no algoritmo. O identificador g inicia com valor vazio mas, se ao longo da iteração pelo `HTML` existir um atributo ng−if do `AngularJS`, então a propriedade com que é inicializada deve ser considerada como o valor g.

Ao longo do algoritmo, apresentado na Figura 5.11, assume-se:

$$\begin{aligned} \text{EXPMAP}(z.p_d) &= exp \quad \text{, propriedades simples inicializadas na } \textit{widget} \text{ instanciada } z \\ g &= \varnothing \quad \text{, é inicializada vazia} \end{aligned}$$

Nota. A expressão $f(\overline{exp})$ corresponde à invocação de uma função da aplicação. Neste contexto, f corresponde ao nome da função e não ao conjunto E que contém as expressões pré-calculadas.

Note-se que a geração do facto exp para a expressão g é gerada numa primeira iteração. Uma vez que g é uma expressão condicional da *widget* pai, o facto foi gerado numa iteração anterior.

5.5 Sumário

A técnica de análise estática apresentada permite minimizar a transmissão de dados e garantir que a informação necessária para atualizar as interfaces, é transmitida do cliente para o servidor.

Com esta abordagem o campo *ViewState*, usado atualmente na plataforma *OutSystems* para manter o estado ao longo dos pedidos entre cliente e servidor, é substituído por uma estrutura de dados ainda mais otimizada. Os dados enviados entre cliente e servidor correspondem ao conjunto diferencial e mínimo dependendo da ação invocada, enquanto que na plataforma o envio do *ViewState* inclui os dados necessários por todas as ações possíveis de executar.

Ao manter uma separação estrita entre o modelo de dados e a interface, a análise de dependências permite identificar que expressões da interface devem ser recalculadas após a execução de uma determinada ação, considerando que variáveis e propriedades foram modificadas.

6

Implementação

O desafio proposto neste trabalho é otimizar ainda mais as aplicações geradas pela plataforma *OutSystems*, adaptando a sua arquitetura atual. Para isso, o processo de geração de código, implementado no compilador *OutSystems*, deve ser alterado. Nesse sentido, foram realizadas algumas experiências no contexto do compilador, apresentadas neste capítulo.

Os artefactos gerados atualmente ao nível do ecrã foram analisados e definiu-se a geração de novos artefactos, necessários para produzir a nova arquitetura. Com o intuito de gerar a arquitetura idealizada, foram encontrados diversos desafios técnicos fora do âmbito deste trabalho, que começaram a comprometer a definição e formalização de uma solução. Como tal, optou-se por desenvolver um protótipo independente do compilador *OutSystems*, cuja implementação é apresentada neste capítulo.

6.1 Adaptação do compilador *OutSystems*

O compilador *OutSystems* é responsável pela geração das aplicações a partir do modelo de uma aplicação desenvolvida no *Service Studio*. Os artefactos gerados ao nível de um ecrã são o *template* tipo servidor, com base nas *widgets* instanciadas, e o código associado ao *template*, onde é implementada a lógica da aplicação.

Modificar a arquitetura das aplicações geradas pela plataforma *OutSystems* significa que os artefactos gerados devem ser adaptados. Considere-se o exemplo da aplicação Biblioteca Musical, em particular o seu ecrã com a apresentação de um conjunto de músicas. Os artefactos gerados são: Songs.aspx (*template* tipo cliente) e Songs.aspx.cs (código das ações, inclusive a *Preparation*).

(Original)	Songs.aspx	*Template* tipo servidor de um ecrã
(Novo)	Songs.html	*Template* tipo cliente de um ecrã
(Novo)	Songs.data.aspx	*Template* para gerar o JSON dados de um ecrã
(Original)	Songs.aspx.cs	*Code behind* do *template* de um ecrã
(Novo)	Songs.js	*Script* no cliente para solicitar os dados

Tabela 6.1: Ficheiros gerados ao nível de um ecrã, com a geração de JSON através de um *template* ASPX.

Para gerar a arquitetura proposta neste trabalho é necessário gerar novos artefactos, nomeadamente, o *template* tipo cliente para cada ecrã, e o código executado no cliente para submeter pedidos ao servidor. O servidor deve fornecer ao cliente o *template* do ecrã e, no fim de cada ação, enviar o conjunto de dados necessários para o cliente. Por exemplo, os dados fornecidos no final da ação especial *Preparation* são os necessários para instanciar o *template* no cliente, e ainda aqueles que o cliente deve fornecer ao servidor ao desencadear uma outra ação, visto que o servidor não mantém o estado do ecrã. As tabelas 6.1 e 6.2, representam abordagens diferente para produzir esta arquitetura. Apresentam os artefactos gerados na arquitetura original, os artefactos novos, e os que foram adaptados, no contexto do compilador *OutSystems*.

O *template* tipo servidor gerado pelo compilador *OutSystems*, é substituído pela geração do *template* tipo cliente, anotado segundo a *framework* AngularJS. No primeiro pedido do ecrã, o servidor fornece o *template* ao cliente, que sendo um documento estático, é mantido em cache no navegador. No cliente, o *template* é instanciado com os dados fornecidos também pelo servidor.

Com a geração do *template* tipo cliente, o passo seguinte é adaptar o código do servidor, de forma a fornecer os dados para instanciar o primeiro. Como tal, os dados originalmente apresentados na página HTML devem ser estruturados num formato, como por exemplo JSON, reconhecido pelo cliente e servidor.

Foram discutidas duas abordagens para a geração desta estrutura de dados com base nos dados consumidos pelas *widgets* instanciadas no ecrã.

A primeira abordagem adapta o *template* ASPX de um ecrã, usado para construir as páginas HTML, para produzir a estrutura JSON dos dados representativos do que consome um ecrã. Na Tabela 6.1 este *template* adaptado é representado como Songs.data.aspx. A definição de um ASPX consiste num conjunto de controlos que, neste contexto, correspondem às *widgets* do ecrã. Como tal, para o subconjunto de *widgets* que se deseja suportar na implementação da nova arquitetura, implementa-se a geração de JSON nos respetivos controlos, em paralelo com a geração de HTML. Os dados acabam por ser gerados segundo a estrutura JSON em vez da estrutura HTML. Assim, esta abordagem mantém o envio dos mesmos dados, reaproveitando a maioria do código gerado originalmente.

Cada *widget* mantém a geração do conteúdo, em HTML, ou apenas dos dados que consome, em JSON. Assim, é possível manter as duas arquiteturas em simultâneo. Essa possibilidade é vantajosa, no sentido de que nem sempre a construção de páginas no

(Original)	Songs.aspx	*Template* tipo servidor de um ecrã
(Novo)	Songs.html	*Template* tipo cliente de um ecrã
(Novo)	Songs.controller.cs	Código no servidor que mantém a árvore de *widgets* e gera o `JSON` de dados
(Novo)	Songs.data.aspx	*Template* para gerar o `JSON` de um ecrã
(Original)	Songs.aspx.cs	*Code behind* do *template* de um ecrã
(Novo)	Songs.js	*Script* no cliente para solicitar os dados

Tabela 6.2: Ficheiros gerados ao nível de um ecrã, com a geração de `JSON` no código do servidor.

cliente revela maior performance.

A segunda abordagem (Tabela 6.2) envolve desenvolver mais código no compilador. Através das bibliotecas `JSON.NET` e `ApiController`, é definida a geração do código do servidor como um serviço de dados, onde cada método corresponde a uma ação do ecrã, e é invocado pelo cliente. Para além disso, o código executado no servidor passa a manter uma árvore de *widgets*, para que através da sua iteração em tempo de execução seja criado o `JSON` de dados. Isto envolve criar um novo artefacto que implementa o serviço de dados do servidor ao nível de cada ecrã, em vez de se adaptar o código do servidor gerado originalmente. A vantagem desta abordagem é que, independentemente do *front end* do servidor ser `JAVA` ou `.NET`, não existem controlos de *widgets* (necessários na primeira abordagem do lado do servidor) para gerar o `JSON` de dados de cada *widget*. Ainda assim, esta abordagem implica alguns desafios técnicos que já eram resolvidos pela arquitetura original do compilador.

Aplicar estas alterações num espaço de tempo reduzido, adotando qualquer abordagem, sobre um projeto como o compilador da *OutSystems*, obriga a restrição do tipo de aplicações a suportar no novo processo de geração de código.

Para testar a implementação desta nova arquitetura seleciona-se um conjunto de *widgets* que devem suportar a geração do *template* tipo cliente, e geração do `JSON` com os dados consumidos por estas. Esse conjunto divide-se em dois grupos: o primeiro, engloba listas de registos, expressões, texto, e blocos condicionais; o segundo, engloba os *inputs*, e botões para o desencadeamento de ações.

Para aplicações pouco sofisticadas, construídas por *widgets* do primeiro grupo, produziu-se com sucesso a arquitetura desejada. O cliente solicita a página, o servidor fornece o *template* tipo cliente, e separadamente os dados que o instanciam. O cliente, através dos mecanismos do `AngularJS`, gera a página `HTML`.

Para desenvolver aplicações com comunicação assíncrona são necessárias *widgets* do segundo grupo, por exemplo, a submissão de um pedido ao servidor através da interação do utilizador com um botão na interface. Nesses pedidos assíncronos está inerente a troca de dados entre cliente e servidor. Para essa troca de dados é necessário um reconhecimento da estrutura que é enviada na rede. Assume-se que o cliente recebe no primeiro pedido uma estrutura com todos os dados necessários, a partir dos quais se

define o estado do ecrã. Essa estrutura é enviada na totalidade para o servidor, quando é desencadeada uma ação. Por sua vez, o servidor necessita alterar dados conforme as operações da ação, devendo, assim, ter conhecimento do formato da estrutura para que, ao alterar a mesma, possa envia-la de volta ao cliente. No entanto, a transmissão de toda a estrutura na rede vai contra um dos objetivos deste trabalho: reduzir o volume dos dados transmitidos. Portanto, o objetivo é enviar parcialmente a estrutura com os dados estritamente necessários, através da aplicação de análise estática.

A implementação da análise estática no contexto do compilador envolve alterações complexas ao nível da geração de código. O compilador não nos proporciona a agilidade necessária para explorar os resultados que se deseja obter com a análise. Desenvolveu-se por isso um protótipo, que permite criar uma solução mais objetiva, com abstração de vários detalhes de desenvolvimento. Ainda assim, a exploração do compilador *OutSystems* permitiu adquirir contextualização sobre a linguagem da plataforma *OutSystems*, o que foi útil na definição do modelo proposto neste trabalho.

6.2 Implementação do Protótipo

O protótipo desenvolvido no âmbito deste trabalho usou como recurso ferramentas de definição de linguagens de programação textuais, mais precisamente o `Xtext`, integrada no ambiente de desenvolvimento `Eclipse`. O desenho deste protótipo envolve a criação de uma linguagem e respetivo compilador.

A linguagem criada baseia-se na representação abstrata de uma aplicação tendo como inspiração, a representação usada pela plataforma *OutSystems*. Implementou-se o compilador 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`, com suporte à comunicação assíncrona, e `AngularJS`, para suportar uma interface de utilizador que é automaticamente atualizada em resposta às alterações dos dados que apresenta.

6.2.1 Linguagem

A ferramenta `Xtext` permite definir a sintaxe da linguagem destinada à especificação do modelo de uma aplicação. A sintaxe da linguagem é definida por um conjunto de regras. A regra apresentada na Listagem 6.1 define a sintaxe concreta da definição de uma aplicação. Os elementos da linguagem que compõem uma aplicação, neste caso representados por imports, datamodel, functions e screens, são acessíveis no gerador de código, e é a partir destas que são iterados os conjuntos de elementos definidos no modelo de uma aplicação. O código é gerado por fases: análise estática; *template* tipo cliente; código do cliente; código do servidor. Como exemplo, a Listagem 6.2 apresenta a função, em versão

```
App:
  imports += Import*
  'app' name = ID '{'
    datamodel = DataModel
    functions += Function*
    screens += Screen*
  '}'
```

Listagem 6.1: Regra em `Xtext` que define a sintaxe da criação de uma aplicação.

```
def generateServerScript(...) {
  generateFile('app.js',
      screens.map[ s |
        s.generateScreenPreparation() +
        s.generateServerSideScreenActions()
      ].join("\n")
}
```

Listagem 6.2: Função de geração do artefacto com o código do servidor, em `Xtend`.

simplificada, que gera o artefacto com o código a executar no servidor.

Nas secções seguintes é usada a aplicação exemplo Split the Bill, cujo ecrã é apresentado na Listagem 6.3, como suporte aos exemplos do código gerado ao nível da análise estática e restantes artefactos gerados.

6.2.2 Análise estática

Normalmente, a análise estática é implementada através dos tradicionais algoritmos de ponto fixo e de um grafo do fluxo de controlo. Neste trabalho desejámos algo mais declarativo e com uma implementação mais rápida, preservando a complexidade dos algoritmos anteriores. Como tal, ao conjunto de ferramentas usado, foi adicionado o interpretador de `Datalog`, `IRIS`, que possibilita a implementação da análise estática definida no Capítulo 5.

A partir do modelo, iterando as *widgets* e ações do ecrã, são extraídos os factos e interrogações, conforme o algoritmo apresentado na secção 5.4. A Figura 6.4 ilustra a produção do programa de `Datalog` através da definição das *widgets* e ações, o qual equivale a uma *string*. O programa é interpretado e executado pela biblioteca `IRIS`, produzindo um conjunto de resultados. A biblioteca `IRIS` para além de um interpretador, fornece também uma API que permite criar o programa de `Datalog` através de objetos. No entanto, a opção pelo interpretador do programa representado por uma *string* revelou-se mais adequada. Usar diretamente a linguagem `Datalog` para declarar factos, regras e interrogações que devem ser gerados ao iterar o modelo de uma aplicação, revelou-se mais legível e, consequentemente, menos propícia a erros, sendo mais fácil identificar o que é gerado em cada caso.

```
screen Bill {
  variables { ... }
  widgets {
    <Input variable=total/>
    <If name=ifCount conditionValue = (friendsList.list.Length > 0)>
        <trueBranch>
            <Expression name=costPPerson value=(total / friendsList.list.Length) />
            <Expression name=friendsToPay value=CountUncheck(friendsList.list) />
        </trueBranch>
        <falseBranch/>
    </If>
    <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 { ... }
  actions { ... }
}
```

Listagem 6.3: Definição de um ecrã para a aplicação Split the Bill.

Todos os factos gerados são compostos por literais que, neste contexto, correspondem aos identificadores do modelo incluindo nomes de ecrãs, ações, entidades e atributos, variáveis, *widgets* e propriedades, e identificadores sequenciais para as instruções de cada ação. Por exemplo, o facto need(ifCount.conditionValue, friendsList.list) contém os literais ifCount.conditionValue e friendsList.list.

Os resultados obtidos após o programa de `Datalog` ser interpretado, são organizados em estruturas devidamente indexadas. Por exemplo, a interrogação ?-StoC('actionName', ?v), onde actionName representa o nome de uma ação, obtém o conjunto de variáveis e propriedades de *widgets* que devem ser enviadas do servidor para o cliente no final dessa ação. Como tal são inseridos na estrutura StoC no índice de actionName. Desta forma, ao longo da geração do código que será executado no cliente e servidor, é possível construir a estrutura de suporte trocada entre cliente e servidor, baseada nestes resultados. Existe, por isso, uma associação entre os identificadores usados na análise e a identificação dos elementos da estrutura, apresentada na Figura 4.12. No código gerado, a estrutura corresponde a um objeto `JavaScript` é traduzido diretamente para `JSON`. O acesso à propriedade ifCount.conditionValue é efetuado através do caminho model.EvalData.ifCount.conditionValue.

A fim de obter uma tradução direta entre os resultados da análise estática e

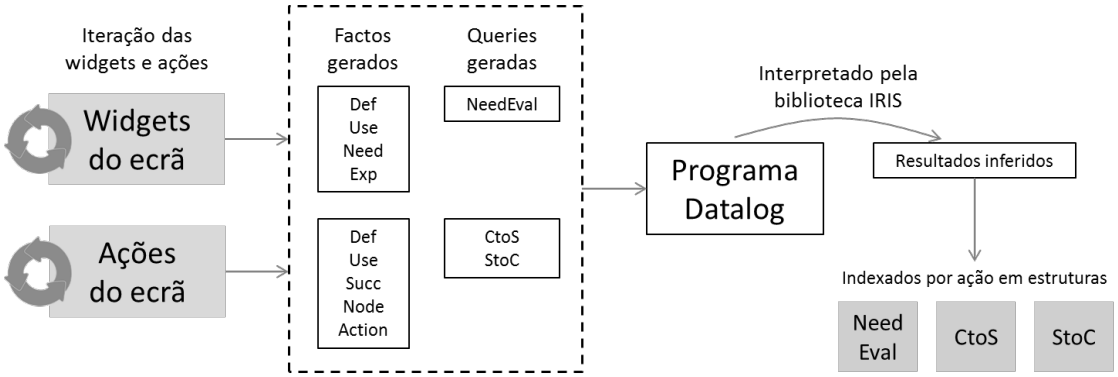


Figura 6.4: Fase de compilação que aplica a análise estática sobre a definição do modelo de uma aplicação.

a estrutura de suporte gerada, todos os identificadores são gerados com um prefixo equivalente ao caminho de acesso no objeto que representa a estrutura, ou seja, need(model.EvalData.ifCount.conditionValue, model.friendsList_list). A lista dinâmica de uma *widget* deve ser representada como um elemento único da estrutura de dados, o que justifica a alteração entre friendsList.list e friendsList_list, caso contrário seria criado um elemento friendsList composto pelo elemento filho list. Os mesmos identificadores são usados na parametrização do *template* tipo cliente, que consome diretamente os dados do objeto $scope fornecido pelo `AngularJS`, que contém a estrutura de suporte model com todos os dados necessários. A notação usada na identificação dos elementos de uma aplicação, quando inseridos na estrutura, é apresentada na Tabela 6.3.

No anexo E é apresentado o programa de `Datalog` gerado ao nível do ecrã da Listagem 6.3. Note-se que todos os factos são gerados conforme o algoritmo apresentado na secção 5.4.

6.2.3 *Template* tipo cliente

O *template* tipo cliente é gerado percorrendo em profundidade as *widgets* instanciadas no ecrã, inclusive as que são instanciadas nos seus *placeholders*. O código `HTML` do *template* produzido tem como base o `HTML` definido por cada *widget*. Observando a declaração das *widgets* apresentada em anexo, é possível associar o `HTML` definido para cada uma com o excerto do *template* gerado, apresentado na Listagem 6.5, conforme as *widgets* instanciadas no ecrã da Listagem 6.3. Ao nível das referências a propriedades, ou seja, dos parâmetros do *template* produzido, é aplicada uma pequena otimização.

Atente-se que os valores das propriedades referenciadas no `HTML` podem ser valores atómicos (referências para propriedades ou variáveis, ou até valores literais), ou valores com expressões aritméticas compostas pelos valores anteriores que necessitam de ser calculadas (referidas neste documento como expressões pré-calculadas). A este nível otimizou-se a parametrização de dados no *template*, com base no tipo dos valores:

model.EvalData.i.pd	Expressão pré-calculada correspondente à propriedade simples p_d da *widget* i.
model.RuntimeProps.i.pr	Propriedade dinâmica p_r da *widget* i.
model.v	Variável v local ao ecrã.
[List].model.i_pr	Propriedade dinâmica p_r da *widget* i, do tipo Lista. É anotado com [List] de forma a que o gerador saiba que o elemento a ser inserido na estrutura de suporte deve ser gerado de forma diferente, que neste caso é um array composto por Item+RuntimeProps+EvalData. Os dois últimos elementos contêm as propriedades e expressões pré-calculadas das *widgets* filhas, instanciadas por cada linha da lista.
[List].model.i_pr.Item.e.eattr	Valor do atributo e_{attr} da entidade e, contido no Item da lista dinâmica p_r da widget i.
actionName_j	Instrução número j da ação com nome $actionName$. O valor j é gerado sequencialmente ao longo da iteração pelas instruções da ação.
actionName_j_q	Instrução número q do corpo de uma instrução número j de uma ação com nome $actionName$.
actionName_eval	Última instrução da ação com nome $actionName$, onde são recalculadas as expressões.

Tabela 6.3: Notação dos identificadores dos dados do modelo de uma aplicação.

1. Caso sejam valores literais, são inseridos diretamente no *template*, uma vez que os valores representam propriedades simples, que nunca são manipuladas ao longo da aplicação.

2. Caso sejam referências, são colocados diretamente no *template*. Se uma propriedade simples referencia uma variável, essa pode ser manipulada. A interface é atualizada uma vez que o *template* consome diretamente a estrutura de dados, que contém a variável.

3. Caso contrário, o valor final corresponde a um sub elemento do EvalData na estrutura de suporte, e no *template* é inserido o respetivo identificador, com a notação apresentada na Tabela 6.3.

As *widgets* instanciadas têm conhecimento da inicialização das suas propriedades. Ao iterar as *widgets* é construído um mapa, que mantém o valor das propriedades de cada uma. Esta informação deve ser mantida durante toda a iteração. Como as propriedades podem ser inicializadas com referências a propriedades de outras *widgets*, quando se gera o `HTML` o mapa é consultado de forma a obter o valor ou parâmetro a inserir no *template* tipo cliente. Esta otimização evita casos em que existam referências para outras referências e assim sucessivamente, provocando a existência de elementos desnecessários na estrutura. Em vez disso, usa-se uma referência direta para o elemento da estrutura de

```
<input ng−model ="model.total" ng−change="eval_inputTotal()" placeholder="Enter total amount">
</input>
<span>
  <span ng−if ="model.EvalData.ifCount.conditionValue">
    {{ model.EvalData.costPPerson.value }}
    {{ model.EvalData.friendsToPay.value }}
  </span>
  <span ng−if ="!model.EvalData.ifCount.conditionValue" ></span>
</span>
<ul>
  <li ng−repeat= "current in model.friendsList_list">
    <input ng−model ="current.Item.Friend.Check" ng−change="eval_friendCheck($index)"
      type="checkbox" class="current.EvalData.friendCheck.classInput"
      placeholder="current.EvalData.friendCheck.placeholder" >
    </input>
    {{ current.Item.Friend.Name }}
  </li>
</ul>
```

Listagem 6.5: *Template* tipo cliente gerado.

dados.

A instanciação do *template* com os dados é efetuada através do objeto $scope do `AngularJS`, que contém a estrutura de suporte model com todos os dados necessários, fornecidos inicialmente pelo servidor. Os mecanismos do `AngularJS` permitem a coerência entre o estado do objeto $scope e a interface. Assim, quando o cliente recebe uma estrutura atualizada do servidor, a interface é automaticamente atualizada. O mesmo acontece quando o utilizador insere dados num formulário e, automaticamente, ficam disponíveis na estrutura que é enviada para o servidor.

6.2.4 Código do servidor

O código executado no servidor, incorpora uma API REST que é disponibilizada ao cliente. Cada método corresponde a uma ação executável no servidor, inclusive a ação especial *Preparation*. A Figura 6.6 ilustra a geração dessa API com vários métodos. As ações do ecrã são iteradas, e para cada ação que seja executável no servidor, é gerado um método, onde é implementada a ação. A estrutura NeedEval, presente na mesma figura, contém as expressões que devem ser recalculadas no final de cada ação. Como tal, no final do código que implementa a ação, acresce o cálculo de expressões, que na Figura 6.6 é representado como ExprEval. Por fim, é necessário preparar o conjunto de dados a enviar para o cliente, o que corresponde à consulta da estrutura StoC (*ServerToClient*). Nessa estrutura estão contidos os resultados inferidos pela análise estática, indexados por ação, indicando quais os dados a enviar do servidor para o cliente no fim da ação.

A ação *Preparation* é a primeira a executar quando se efetua um pedido da página, e por isso deve inicializar a estrutura de dados necessários no cliente, enviada depois na

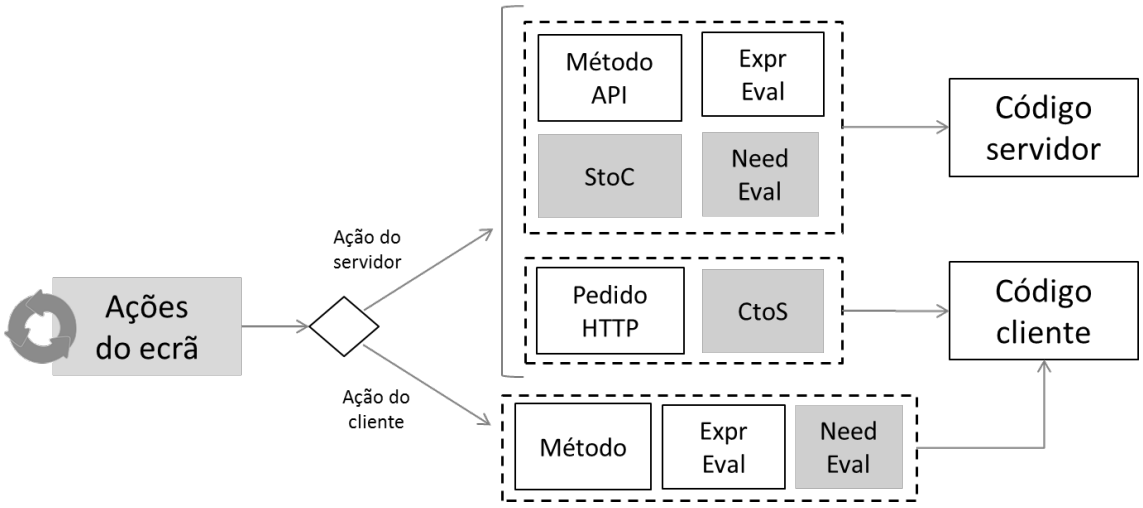


Figura 6.6: Geração do código executado no cliente e servidor através da iteração das ações do ecrã.

```
1  var Friend = [{Friend: {Name: "Sara"}},{Friend: {Name: "Hugo"}}];
2  function CountUncheck (list, model){ ... }
3  app.get('/Bill/GetData', function(req,res) {
4    Bill_initModel(model);
5    Bill_Preparation(model);
6    Bill_evalDataModel(model);
7    res.send(model);
8  });
9  app.post('/Bill/Reset', function(req,res) {
10   var model = req.body.model;
11   Bill_Reset(model);
12   var serverToClient = modelServerToClient_Reset(model);
13   res.send(JSON.stringify(serverToClient));
14 });
```

Listagem 6.7: Excerto do código do servidor gerado.

resposta ao cliente.

Um excerto do código do servidor gerado ao nível do ecrã da Listagem 6.3 é apresentado na Listagem 6.7. Por exemplo, a função Bill_evalDataModel encarrega-se da computação das expressões, e a função modelServerToClient_Reset define que parte da estrutura de dados deve ser enviada para o cliente após a execução da ação Reset.

Ainda no contexto de geração de código para uma ação executável no servidor, é necessário gerar código para o cliente, de forma a que esse seja capaz de invocar a ação do servidor. Como tal, na Figura 6.6 é ilustrado que é gerado o código para submeter um pedido `HTTP`, o que envolve preparar o conjunto necessário de dados para a execução da ação invocada. Esses dados estão contidos na estrutura CtoS (*ClientToServer*), obtidos pela análise estática. É sobre esses dados que o servidor executa a ação.

Para além da estrutura de dados trocada entre cliente e servidor constantemente,

```
1  function controller($scope, $http) {
2      $http({ url: 'Bill/GetData', method: "GET", responseType: "json" }).success(function (model) {
3      $scope.model = model;
4      updateModel(model);
5    });
6    $scope.addFriend_onClick = function(){
7      AddFriend($scope.model);
8    }
9    $scope.reset_onClick = function(){
10     var clientToServer = new Object();
11     set(clientToServer, "total", $scope.model.total);
12     updateModelClientToServer(clientToServer);
13     $http({ url: 'Bill/Reset', method: "POST", data: {"model": clientToServer}}).success(
14     function (model) {
15       updateModelServerToClient(model);
16       diffModel(model, $scope.model);
17       updateModel($scope.model);
18     });
19   }
20 }
```

Listagem 6.8: Excerto do código do cliente gerado.

existe um conjunto de dados persistentes da aplicação que simulam uma base de dados, definida na linha 1 da Listagem 6.7. No sentido de simplificar a implementação do protótipo optou-se por manter esses dados na memória do servidor, que correspondem ao conjunto de entidades declarado, inicializado com os registos declarados no modelo da aplicação.

6.2.5 Código do cliente

O código do cliente é gerado com base na arquitetura de uma aplicação implementada segundo a *framework* `AngularJS`. Contém as funções de todas as ações que é possível desencadear a partir da interface, com alguma diferença entre as ações executáveis no cliente ou no servidor.

A Figura 6.6 apresenta a interação das ações, a partir das quais é gerado o código do cliente. Caso a ação seja executável no cliente, então é gerado o código que executa a ação. Tal como as ações do servidor, no final é necessário recalcular as expressões que estão contidas na estrutura NeedEval. Caso a ação seja executável no servidor, então é necessário gerar o código que submete ao servidor um pedido `HTTP` para executar a ação, como já foi referido anteriormente. Como resposta é obtido um novo conjunto de dados, que atualiza o modelo de dados mantido no cliente. Por exemplo, na Listagem 6.8, a implementação da ação AddFriendexecutável no cliente, que é declarada no ecrã da Listagem 6.3, não efetua qualquer pedido ao servidor.

Após o primeiro pedido que invoca a ação *Preparation*, o código do cliente recebe a estrutura de dados que é mantida ao longo do ciclo de vida do ecrã. Como tal, são geradas

funções que ajudam na manutenção da estrutura de suporte. Quando o cliente recebe uma nova estrutura, não é suficiente fazer a substituição da estrutura que esse mantém, visto que os dados transmitidos na rede não correspondem à estrutura completa, devido às otimizações da análise estática. Em vez disso a estrutura local deve ser atualizada. Por outro lado, quando o cliente efetua alterações sobre a estrutura através da execução de ações no cliente, é necessário garantir que são alterados os elementos corretos do modelo.

A reavaliação de expressões também é suportada como consequência da interação com a interface. Considere-se uma widget *input*, com uma propriedade que referencia uma variável. Quando o utilizador insere um valor, a variável no modelo assume automaticamente o novo valor. Mas, outras expressões podem estar contidas na página e dependerem da variável que foi alterada. O `AngularJS` permite que essas expressões sejam automaticamente atualizadas, desde que sejam devidamente inseridas no *template*, como chamadas a funções no modelo. Assim, quando uma mudança ocorrer, a função é executada atualizando o valor da expressão. Note-se que o nosso modelo de dados para além de ser consumido diretamente pelo *template*, também é enviado na rede entre cliente e servidor. Descarta-se, por isso, a possibilidade de manter funções como elementos da estrutura de dados. A nossa implementação consiste, assim, em eventos desencadeados pela deteção de alterações dos elementos `HTML` (diretiva ng-change do `AngularJS`), que com o auxílio dos resultados obtidos pela análise estática são reavaliadas somente as expressões necessárias com base nas variáveis alteradas. Na Listagem 6.5 é visível em cada elemento *input* o atributo ng-change.

6.2.6 Estrutura de suporte

É gerada uma estrutura de suporte que permite estabelecer quais os dados a transmitir entre cliente e servidor, como auxílio à ausência de estado do ecrã no servidor. Esta estrutura está presente nos métodos incorporados no código do cliente e servidor, e no *template* tipo cliente.

A sua geração depende estritamente dos resultados obtidos da análise estática, que são mantidos nas estruturas SToCVars, CToSVars, needEvalExps, liveScreenVars. As estruturas são preenchidas após a execução do programa de `Datalog`, que produz os resultados perante as interrogações à base de conhecimento, composta por factos e regras. Note-se que uma interrogação em Datalog, como ?-CToS('action_name', ?v), é formada pelo predicado CToS e pelas variáveis que podem ser inferidas ou afetadas com um valor de forma a restringir os resultados. O nome da estrutura corresponde portanto ao nome do predicado de cada interrogação.

No fim da execução de uma ação no servidor, são definidos quais os dados estritamente necessários a enviar. Isto corresponde à iteração do valor mapeado em StoCVars com o nome da ação, e em needEvalExps.

Os dados enviados do cliente para o servidor, representativos do estado do ecrã, estão contidos na estrutura CToSVars.

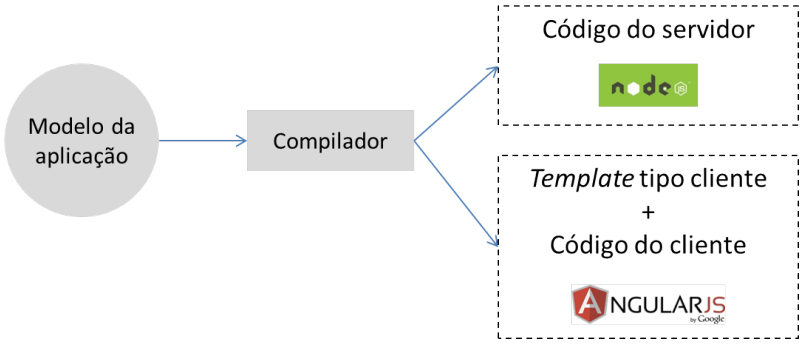


Figura 6.9: Artefactos gerados pelo compilador através da definição do modelo de uma aplicação.

A inicialização da estrutura, que se sucede no primeiro pedido, consiste em adicionar as variáveis do ecrã, e os objetos RuntimeProps e EvalData. A seleção de variáveis, tal como os elementos dos objetos, necessários no ecrã, é definida pela estrutura liveScreenVars, indexada pelo nome dos ecrãs. Esta estrutura engloba os dados consumidos pelo *template*, e os dados necessários para a execução das ações.

6.2.7 Aplicação gerada

Após definir o modelo de uma aplicação no `Eclipse`, com a linguagem criada, é executada a aplicação. Os artefactos gerados são organizados numa hierarquia de ficheiros e diretorias, comum em aplicações da plataforma `Node.js`. Para executar o servidor invoca-se na diretoria da aplicação gerada, o comando node app e a aplicação fica disponível, neste caso num servidor local, acessível através do *browser*.

A arquitetura da aplicação gerada consiste num MVC do lado do cliente, e um MVC no servidor. No cliente temos uma aplicação standard `AngularJS`, no servidor uma aplicação standard `Node.js`, como apresenta a Figura 6.9. O servidor disponibiliza uma API cujos métodos são invocados pelo cliente para a execução de ações, e como resposta recebe dados. No primeiro pedido, quando o cliente solicita a página, o servidor prepara a estrutura de dados que é enviada para o cliente, e posteriormente mantida por ele. Nos pedidos seguintes, a estrutura continua a ser partilhada entre o cliente e servidor, composta apenas pelos dados estritamente necessários. Portanto, o componente *Model* dos dois MVC's é partilhado, através da troca da estrutura de dados que o representa. Essa estrutura é manipulada por ações no cliente ou no servidor, interação do utilizador com a interface, e é consumida pelo *template*.

6.3 Sumário

Neste capítulo é apresentada em resumo a implementação parcial de uma nova arquitetura no compilado. Parcial pois, conduziu a vários desafios técnicos fora do âmbito deste

trabalho. Como solução, apresentamos o protótipo, cujo desenho envolve a criação de uma linguagem textual e a implementação do seu compilador.

O protótipo implementado tem como input um modelo de uma aplicação, e gera uma aplicação `Node.js` para o servidor, e `AngularJS` para o cliente, que ao longo da execução comunicam entre si. A aplicação de análise estática permitiu-nos otimizar a estrutura de dados enviada entre cliente e servidor de forma a que, apenas os dados estritamente necessários sejam transmitidos na rede.

Em suma, obtivemos um modelo simplificado da linguagem *OutSystems*, a partir do qual geramos aplicações mais fluídas, com exploração de ações no cliente, e onde a transmissão de dados na rede é otimizada.

7

Resultados

Este trabalho tem como objetivo otimizar as aplicações geradas pela plataforma *OutSystems*. O aumento da fluidez da interação do utilizador com a aplicação e a redução do volume de conteúdo transmitido, é o que se pretende obter como resultado final deste trabalho.

Uma análise à comunicação entre cliente e servidor ao longo da execução das aplicações *OutSystems* permitiu observar que todos os ciclos de pedido-resposta transmitem uma página inteira ou fragmentos `HTML`, em conjunto com o campo *ViewState* que suporta a manutenção do estado do ecrã. Esse código `HTML` é maioritariamente composto por informação estática, relativa à estrutura e comportamento da página, que não é possível manter em cache, pois o conteúdo é gerado dinamicamente no servidor.

O modelo proposto pretende abstrair vários detalhes das aplicações *OutSystems* e gerar uma nova arquitetura através do modelo proposto. No sentido de reduzir o volume de conteúdo transmitido, o envio de páginas completas `HTML` é substituído pelo envio dos dados estritamente necessários na comunicação entre cliente e servidor. Nesse sentido, é delegado no cliente o processo de geração das páginas `HTML`, instanciando-as com os dados fornecidos pelo servidor. O tempo de latência reduz, pois a resposta do servidor deixa de envolver esse processamento, o que também influencia de forma positiva a fluidez da interação. A fim de aumentar essa fluidez, o modelo proposto também estende a situação atual com a possibilidade de anotar ações como sendo executáveis diretamente no cliente.

Neste capítulo são apresentados os resultados de validação do trabalho proposto. Analisam-se algumas métricas sobre o ciclo de vida de uma aplicação gerada no modelo proposto, quando comparadas com a plataforma *OutSystems* atual. Em primeiro lugar, são apresentados resultados dos testes que medem o volume de conteúdo transmitido

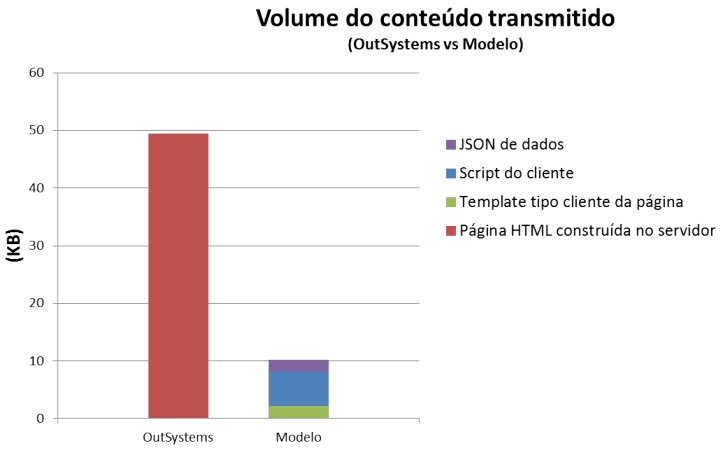


Figura 7.1: Comparação do volume do conteúdo transmitido no primeiro pedido da página de músicas desenvolvida em *OutSystems* e no modelo proposto.

na rede. De seguida, são apresentados os resultados dos testes que medem o tempo de latência e de transmissão, e os tempos de visualização das páginas, que decorrem desde o instante em que o cliente efetua o pedido até concluir o desenho da página no navegador.

A aplicação utilizada nestes testes foi a da Biblioteca Musical, introduzida na Secção 3.3.3, e que é composta por um ecrã que lista um conjunto de músicas. Para efeitos dos testes, esse ecrã foi modificado e estendido com ações assíncronas. As ações correspondem à adição de uma nova música, à remoção de uma música e à edição dos detalhes de uma música. Em termos de interface, a informação das músicas é agora apresentada utilizando elementos `HTML` do tipo *input*, o que permite a sua edição de forma integrada.

Note-se a aplicação utilizada nos testes é alocada num servidor não local, de forma a simular uma situação real da comunicação entre cliente e servidor. É também importante referir que as aplicações desenvolvidas sobre o modelo proposto são limitadas comparativamente às aplicações desenvolvidas na plataforma *OutSystems*. As limitações devem-se aos detalhes abstraídos pelo modelo, e aos que não foram suportados por estarem fora do âmbito do trabalho.

Volume de dados O primeiro teste compara o volume do conteúdo transmitido no primeiro carregamento da página que lista um conjunto de 500 músicas, entre a aplicação gerada pela plataforma *OutSystems* e a aplicação gerada sobre o modelo proposto. Os valores são apresentados na Figura 7.1. No caso da aplicação gerada em *OutSystems*, a página `HTML` é construída totalmente no servidor, e enviada para o cliente. Os artefactos que são transmitidos na versão gerada sobre o modelo proposto apresentam um volume 79% menor.

Observa-se, assim, os primeiros ganhos da nova arquitetura baseada em *templates* tipo cliente. Os ganhos são ainda maiores nos carregamentos da página que se seguem. Na versão *OutSystems*, a página `HTML` continua a ser transmitida na rede, mas na versão

Operações	**Página HTML** (KB)	**JSON dados** (KB)
GetData	49,5	2,13
Add	95,6	2,24
Remove	98,3	2,21
Update	5,1	0

Tabela 7.1: Tamanhos relativos entre a página `HTML` construída no servidor, na versão *OutSystems*, e a estrutura do modelo de dados em `JSON`, na versão prototipada.

prototipada, apenas o conjunto de dados é que necessita de ser transmitido. O *template* tipo cliente e o *script* são documentos estáticos, que permanecem em cache no cliente.

As limitações das aplicações desenvolvidas sobre o modelo proposto podem ter influência nestes resultados. O código `HTML` gerado no servidor contém informação extra que não é abrangida no modelo proposto, como por exemplo a identificação de todos os elementos `HTML` da página. Para validar a necessidade dessa informação extra seria necessário estudar em maior profundidade o escopo das aplicações desenvolvidas na plataforma *OutSystems*.

Volume de dados em ações locais Analisa-se a seguir o volume de dados que é transmitido entre cliente e servidor, para pedidos da mesma página, ou seja, como resultado da invocação das ações locais a cada ecrã.

A Tabela 7.1 apresenta um conjunto de operações que surgem da interação de um utilizador com a aplicação referida. Neste exemplo, a operação *GetData* representa o primeiro pedido ao ecrã que apresenta a listagem de 500 músicas. Os resultados obtidos mostram uma redução drástica do conteúdo transmitido na rede. Essa redução deve-se, em grande parte, à substituição do envio de `HTML` pelo envio do conjunto de dados. Deve-se também à otimização da transmissão dos dados estritamente necessários, quando uma ação é invocada. Nos testes realizados, o tamanho relativo ao `HTML` inclui também o *ViewState* com o estado atualizado do ecrã.

Note-se que a diferença dos valores apresentados na Tabela 7.1 também está relacionada com o facto de o modelo proposto não cobrir os mesmos detalhes que a plataforma *OutSystems*. Ou seja, as páginas `HTML` que são construídas no servidor contêm informação relativa ao comportamento da aplicação que foi considerada fora do âmbito deste trabalho.

No modelo proposto, o *ViewState* é substituído pelo envio da estrutura de suporte, apresentada na secção 4.2. A transmissão do estado nas aplicações *OutSystems* já é otimizada, na medida em que, contém os dados necessários para a execução de qualquer ação. No entanto, em todos os pedidos e respostas é sempre enviado o *ViewState* com o mesmo conjunto de dados. A estrutura de suporte contém apenas os dados, que através de análise estática (capítulo 5), são considerados necessários para a execução das ações do ecrã, e para a instanciação do *template* tipo cliente. Desta forma a estrutura contém

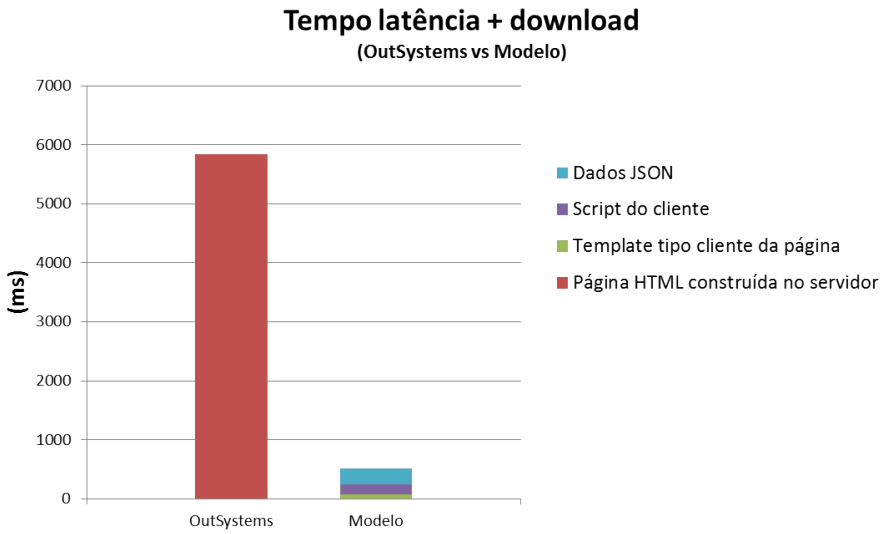


Figura 7.2: Comparação dos tempos médios (latência + *download*) no pedido da página de músicas desenvolvida em *OutSystems* e no modelo proposto.

os dados que representam não só o conteúdo apresentado pela página, como também a informação do estado da página.

A estrutura de suporte proposta é diferente do *ViewState*, pois a análise estática aplicada sobre o modelo da aplicação permite identificar, ao nível da ação, os dados estritamente necessários pelo servidor, o que filtra ainda mais a quantidade de dados transmitida na rede. Note-se que no primeiro pedido (ação *Preparation*), a estrutura enviada contém todos os dados necessários para qualquer ação que seja executada, a qual é mantida no cliente. Nos pedidos seguintes, é trocado com o servidor apenas um subconjunto da estrutura, composto pelos resultados obtidos da análise estática.

Relativamente à ação *Update*, na versão desenvolvida no modelo proposto, o conjunto de dados é vazio, pois o servidor apenas atualizou a música na base de dados, e a alteração ao nível da interface manteve-se no modelo de dados no cliente, não sendo portanto necessário transmitir dados de volta para o cliente.

Os tamanhos relativos aos volumes de conteúdo obtidos nestes testes são na ordem de `KB`, o que pode parecer irrelevante. Ainda assim, quando o utilizador corre a aplicação num contexto *mobile* através de uma rede `3G`, por exemplo, a largura de banda disponível é limitada. Nesse cenário, a variação do consumo de `KB` faz toda a diferença, e como tal não deve ser desprezada.

Latência e transmissão O envio de conteúdo na rede tem associado um tempo de latência e de transmissão (*download*). A latência representa o tempo da resposta do servidor ao cliente. A Figura 7.2 apresenta uma comparação dos tempos do primeiro carregamento da página que lista 500 páginas, entre a página gerada em *OutSystems* e no modelo proposto. Os tempos são na ordem de mili segundos, com grande tendência para

variarem, por isso o primeiro carregamento foi repetido 5 vezes, cuja média é apresentada na Figura 7.2. Ainda assim, a diferença obtida entre a versão *OutSystems* e a versão prototipada permite observar os ganhos deste trabalho, que correspondem a uma diminuição de 91%. O tempo de latência representa a maior porção do tempo total na versão *OutSystems*, pois é necessário esperar que o servidor construa toda a página `HTML` a ser transmitida para o cliente. O tempo de *download* é proporcional ao volume do conteúdo transmitido.

Os testes foram realizados utilizando uma rede de internet sem fios, com uma velocidade nominal de 300Mbps. Mas se considerarmos um cenário em que a aplicação é acedida através de uma rede `3G/4G`, a velocidade será menor, e consequentemente o tempo de transmissão aumenta. Os acessos a aplicações através de redes móveis existem cada vez mais, por isso a diferença de tempos em mili segundos apresentada nos resultados é ainda mais relevante, pois num contexto móvel os tempos aumentam.

É importante realçar que, nas aplicações geradas sobre o modelo proposto, em contraste com a *OutSystems*, o servidor não envolve o processamento na construção de páginas `HTML`. No entanto, como a página é construída agora no navegador, existe também um tempo acrescido relativo à instanciação do *template* tipo cliente.

Visualizar a página final Apresentam-se, de seguida, os resultados dos tempos de visualização das páginas, que decorrem desde o instante em que o cliente efetua o pedido até concluir o desenho da página no navegador. A Figura 7.3 apresenta a média dos valores obtidos nas 5 repetições do primeiro carregamento da página de músicas. O gráfico apresenta a comparação do tempo entre o instante em que é efetuado o pedido ao servidor, e o instante em que é possível visualizar a página no navegador. Nesses tempos é incluído o tempo de latência e transmissão, uma vez que é contabilizado a partir do instante em que é efetuado o pedido.

Nos testes anteriores concluiu-se que no modelo proposto o cliente recebe 91% mais rápido todos os artefactos, em comparação com a versão da plataforma *OutSystems*. No entanto, é necessário contabilizar o tempo de processamento que agora é exigido ao cliente para gerar a página `HTML`. O gráfico da Figura 7.3 divide o tempo até o utilizador visualizar a página entre: O desenho da estrutura da página, que é definida pelo *template* tipo cliente; A execução de código no cliente, inclusive a função `$digest` do `AngularJS`; E o desenho da página final com os dados. Observa-se que a função `$digest`, responsável por atualizar a interface perante alterações do modelo de dados que se mantém no cliente, é a que consome mais tempo de processamento do cpu. Os valores apresentam, ainda assim, uma redução de 39% do tempo que o utilizador espera até ver a página final.

Apesar de tudo a *framework* `AngularJS` apresenta alguns problemas de performance relativamente à atualização entre dados e interface, quando a complexidade da página aumenta. Em paralelo a este teste, observou-se que o tempo de processamento da função `$digest` diminui proporcionalmente com a redução do número de músicas apresentadas

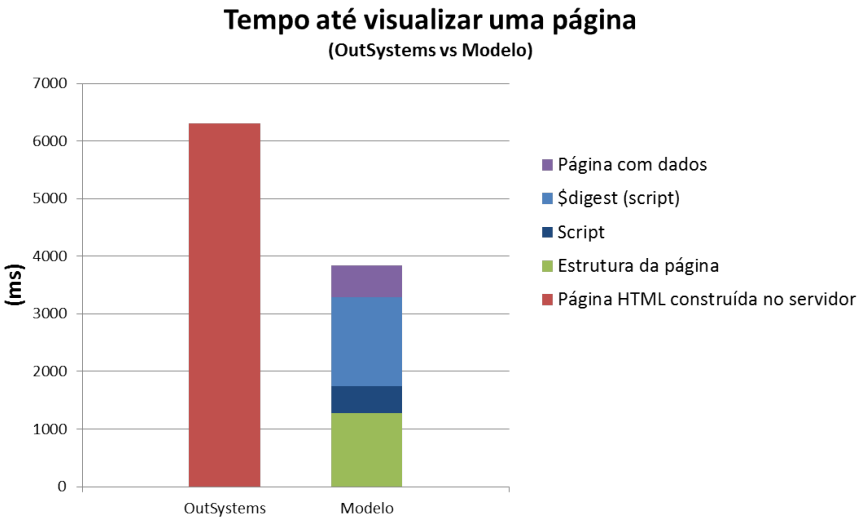


Figura 7.3: Comparação dos tempos médios parciais desde o momento em que inicia o pedido até ao momento em que a página é visualizada.

na lista. Os valores obtidos do tempo gasto por esta função foram na ordem de um segundo no caso de uma lista de 500 músicas. A análise de uma lista de 50 músicas verificou que esse tempo reduzia para uma média de 100 mili segundos. Note-se que esta função do `AngularJS` é processada no dispositivo cliente, que nestes testes foi um *desktop*. Em relação com um *smartphone*, o tempo de um segundo obtido anteriormente aumenta, pois o poder computacional do dispositivo é menor.

Quando o utilizador navega entre páginas de uma aplicação gerada em *OutSystems*, visualiza um ecrã branco entre elas. Esse comportamento provoca quebras na fluidez da interação do utilizador com a aplicação. Contudo, na nova arquitetura das aplicações geradas no modelo proposto, o desenho da página ocorre em dois momentos: o primeiro, é quando o cliente recebe o *template* tipo cliente, permitindo ao utilizador visualizar a estrutura estática da página final; o segundo, é quando o cliente recebe os dados que vão permitir instanciar o *template*, e de seguida desenhar a página com dados.

Os resultados mostram que os dois momentos do desenho da página final ocorrem num espaço de tempo menor do que o desenho da página `HTML` construída no servidor, na versão da aplicação gerada pela plataforma *OutSystems*. No entanto, os problemas de performance do `AngularJS` podem provocar um aumento do tempo até desenhar a página final, caso a página seja muito complexa. Mas como o desenho da estrutura da página sem dados ocorre num curto espaço de tempo, o utilizador pode visualizar parcialmente a página final. A visualização parcial da página pode ainda ser auxiliada com recurso a `CSS` e imagens, que são fornecidas pelo servidor de forma assíncrona de forma a não bloquear o desenho da página.

	Modelo	*OutSystems*
Ações no cliente	✓	-
Ações no servidor	✓	✓
Atualização automática entre dados e interface	✓	-
Suporte de todos os componentes de uma aplicação	-	✓
Camada de abstração sobre as aplicações geradas	✓	✓
Declarar as próprias *widgets*	✓	-
Geração do `HTML` no cliente	✓	-
Cache da estrutura e comportamento da página	✓	-
Manutenção do estado do servidor	✓	✓
Manutenção diferencial do estado do servidor	✓	-

Tabela 7.2: Comparação dos detalhes de desenvolvimento e execução de aplicações entre o modelo proposto e a plataforma *OutSystems*.

Ações no cliente Com o intuito de reduzir o tempo de espera ao longo da interação com as aplicações, estendeu-se a linguagem criada com uma notação nas ações que, permite delegar no cliente a execução de ações e, por sua vez eliminar, o tempo de latência.

A plataforma `AngularJS` suporta uma arquitetura MVC, o que simplifica a execução de lógica da aplicação no cliente. Note-se que todas as ações das aplicações definidas na linguagem *OutSystems* são executadas no servidor, exigindo a comunicação com o servidor mesmo quando é possível evitar. Assim, reduzimos o número de comunicações desnecessárias com o servidor para executar ações, que podem ser implementadas no cliente. Por exemplo, quando se deseja adicionar uma nova música apenas ao nível da interface, sem impacto na base de dados. Ainda assim, não se dispensa a necessidade de executar ações no servidor, nomeadamente ações que manipulem a base de dados, acessível apenas através do servidor.

A arquitetura das aplicações produzidas a partir do modelo proposto pode nem sempre ser vantajosa. A execução de uma aplicação num dispositivo móvel, com inferior poder computacional, pode ter um desempenho superior se optar pela construção das páginas no servidor e, ao mesmo tempo optar pela execução de todas as ações também no servidor.

A criação de uma linguagem, inspirada na linguagem *OutSystems*, permitiu obter um modelo simplificado da definição de uma aplicação. Com a abstração de vários detalhes das aplicações *OutSystems* e, com a idealização da arquitetura para as aplicações geradas, permitiu também simplificar o processo de desenvolvimento. A simplificação de maior importância diz respeito à operação AjaxRefresh, usada explicitamente nas ações definidas em *OutSystems*, para atualizar explicitamente *widgets* do ecrã. No modelo proposto, a operação deixa de ser necessária, pois através do mecanismo do `AngularJS` é feita a sincronização automática entre os dados de um ecrã e a interface que os apresenta.

Ao nível dos tempos de compilação é possível averiguar que a análise estática aplicada sobre modelo proposto revela tempos desprezáveis. Note-se que a complexidade da

interpretação de um programa `Datalog` é a mesma que os algoritmos de ponto fixo implementados usualmente nos compiladores. Contudo, o tempo de compilação está fora do âmbito deste trabalho, cujo objetivo é otimizar as aplicações geradas, e não otimizar a sua geração.

Em suma, a Tabela 7.2 apresenta uma comparação entre as capacidades do modelo proposto neste trabalho e a plataforma *OutSystems*.

8

Conclusões

Este trabalho apresenta uma abordagem inovadora que otimiza as aplicações web geradas pela plataforma *OutSystems*. Nesse sentido, foi proposto um modelo sobre o qual são definidas aplicações, abstraindo vários detalhes de desenvolvimento. A representação abstrata de uma aplicação permitiu-nos obter um modelo simplificado e explorar a extensão com semântica extra, a fim de otimizar as aplicações geradas. Através do modelo são geradas automaticamente aplicações otimizadas, baseadas em comunicação assíncrona de dados.

O modelo proposto é expresso pela linguagem apresentada no capítulo 4, inspirada na linguagem da *OutSystems* e na representação abstrata de uma aplicação. Sobre esse modelo são aplicadas duas análises estáticas do fluxo de dados (capítulo 5), que minimizam a transmissão de dados na rede. Garante-se o envio dos dados estritamente necessários para a atualização da interface e para a execução das ações que podem ser executadas.

A geração das aplicações a partir da definição do modelo é concretizada pelo compilador da linguagem, cuja implementação é apresentada no capítulo 6. No fim do processo de compilação, obtêm-se aplicações com um MVC no cliente e um MVC no servidor. Em contraste com a arquitetura atual das aplicações geradas pela plataforma *OutSystems*, as páginas são construídas no cliente, através da instanciação do *template* tipo cliente, com os dados fornecidos pelo servidor. O modelo proposto permite, ainda, distribuir a execução da lógica da aplicação, entre cliente e servidor. Essa distribuição é estabelecida no modelo pela notação da declaração de ações como sendo executáveis no servidor ou cliente.

8.1 Contribuições

Este trabalho contribui com um modelo da representação abstrata de uma aplicação Web, a partir do qual são geradas automaticamente aplicações otimizadas. Os resultados apresentados no capítulo 7, validam essa otimização, exibindo uma redução de 71% do volume de conteúdo transmitido na rede, 91% do tempo de latência, e 31% do tempo que o utilizador espera até visualizar a página final.

A delegação da geração das páginas `HTML` no cliente é um fator na redução do volume de conteúdo transmitido. A análise estática sobre o modelo assegura, ainda, que o conteúdo transmitido é apenas o estritamente necessário.

As aplicações geradas pela plataforma *OutSystems* exigem que o estado do ecrã seja mantido explicitamente através do campo *ViewState*, com a implicação do envio deste conjunto de dados em todos os ciclos de pedido-resposta. Esses dados são os necessários para a execução de qualquer ação, ou seja, independentemente da ação invocada é sempre enviado o mesmo conjunto para o servidor e consequentemente para o cliente.

A transmissão de dados entre cliente e servidor, que deriva da invocação de ações, é isolada e otimizada pelo modelo proposto neste trabalho. Através da análise estática aplicada sobre a definição do modelo de uma aplicação, obtemos o conjunto de dados estritamente necessário na invocação de uma ação em particular. O resultado final são aplicações que fazem uma manutenção diferencial do estado do ecrã ao longo da comunicação entre cliente e servidor.

As contribuições deste trabalho foram publicadas no simpósio de informática INForum 2014, realizado na Universidade do Porto, através do artigo [5], o qual esteve entre os nomeados para melhor artigo.

8.2 Trabalho futuro

As direções de trabalho futuro apontam para a extensão da linguagem para que seja possível anotar os dados que não devem ser diretamente transmitidos na rede por razões de segurança, e determinar automaticamente sobre que parte da lógica aplicacional pode ser incorporada no cliente ou deve ser mantida no servidor. Ao longo do desenvolvimento de uma aplicação sobre o modelo proposto, quando é criada uma ação executável no cliente o programador deve estar consciente de quais as implicações. A execução de uma ação no cliente envolve não só o envio dos dados necessários pela ação, como outros dados necessários para o cálculo de expressões, que devem ser reavaliadas no fim da ação. Se o programador não tiver conhecimento de que esses dados são transmitidos na rede, pode comprometer a segurança da aplicação caso sejam dados confidenciais.

O particionamento automático das ações de uma aplicações, entre cliente e servidor, com base na identificação dos dados confidenciais que seriam necessários transmitir na rede, revela o potencial como trabalho a ser desenvolvido, pois eliminaria a preocupação do programador sobre onde as ações devem ser executadas e a transmissão de dados

subjacente.

Como complemento à redução da transmissão de dados, poderia ser usada uma *framework*, como por exemplo a `MessagePack` [29], que permite reduzir o volume de um `JSON`, através da sua serialização binária.

O modelo proposto só endereça uma página de cada vez, o que implica a transmissão do seu *template* (no primeiro carregamento), a transmissão dos seus dados, a partir dos quais é gerada a página final. Seria interessante analisar também o desenvolvimento de aplicações de página única (SPA's). Neste trabalho, esse conceito não foi abordado tendo em conta as grandes alterações que seriam feitas sobre a linguagem. Este tipo de aplicações apresentam potencial num contexto *mobile*, na medida em que, o contexto de utilização não altera entre as páginas. A navegação entre "páginas" é efetuada pela atualização de fragmentos da única página da aplicação.

Propõe-se também como trabalho futuro a extensão da análise estática, a fim de otimizar o envio das listas de dados contidas na estrutura enviada entre cliente e servidor. Essa otimização endereça um problema difícil, cuja solução não esteve no âmbito deste trabalho. Considere-se o cenário em que o servidor efetua o *append* de um novo elemento a uma lista. No modelo proposto o cliente necessita de enviar a lista para o servidor, para que este a atualize com o novo elemento. Ou seja, a lista inteira é enviada ao longo da comunicação entre cliente e servidor. De certa forma, isto podia ser evitado, bastaria detetar que a operação realizada sobre a lista é a adição de um novo elemento à cauda. Assim, o cliente não necessitaria de enviar a lista, e o servidor apenas enviaria o novo elemento e a informação que este seria adicionado à cauda da lista. A dificuldade do problema surge quando se considera e analisa as várias condicionantes. A operação *append* não é a única possível de efetuar sobre uma lista. No modelo proposto existem também a remoção e atualização dos elementos. Tudo isto dificulta a análise sobre quais os índices alterados, ou elementos adicionados e removidos, de forma a manter uma lista consistente no cliente. Propõe-se, por isso, como trabalho futuro, a extensão da análise estática a fim de otimizar o envio de listas de dados entre cliente e servidor.

Bibliografia

Referências

[1] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann e J. Kato. "It's Alive! Continuous Feedback in UI Programming". Em: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 95–104. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462170. URL: http://doi.acm.org/10.1145/2491956.2462170.

[2] R. T. Fielding e R. N. Taylor. "Principled Design of the Modern Web Architecture". Em: *ACM Trans. Internet Technol.* 2.2 (mai. de 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185.

[3] F. García, R. Izquierdo Castanedo e A. Juan Fuente. "A Double-Model Approach to Achieve Effective Model-View Separation in Template Based Web Applications". English. Em: *Web Engineering*. Ed. por L. Baresi, P. Fraternali e G.-J. Houben. Vol. 4607. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 442–456. ISBN: 978-3-540-73596-0. DOI: 10.1007/978-3-540-73597-7_37. URL: http://dx.doi.org/10.1007/978-3-540-73597-7_37.

[4] F. J. García, R. I. Castanedo e A. A. J. Fuente. "A Double-model Approach to Achieve Effective Model-view Separation in Template Based Web Applications". Em: *Proceedings of the 7th International Conference on Web Engineering*. ICWE'07. Como, Italy: Springer-Verlag, 2007, pp. 442–456. ISBN: 978-3-540-73596-0. URL: http://dl.acm.org/citation.cfm?id=1770588.1770634.

[5] S. Gonçalves, J. C. Seco, H. Lourenço e S. Silva. "Otimização automática de aplicações web usando templates client side". Em: *INForum 2014 Atas do 6º Simpósio de Informática*. Ed. por S. P. Abreu e J. P. Faria. 2014, pp. 110–125. ISBN: 978-972-752-171-5. URL: http://inforum.org.pt/INForum2014/docs/atas-do-inforum2014.

[6] F. Heidenreich, J. Johannes, M. Seifert, C. Wende e M. Böhme. "Generating Safe Template Languages". Em: *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*. GPCE '09. Denver, Colorado, USA: ACM, 2009, pp. 99–108. ISBN: 978-1-60558-494-2. DOI: 10.1145/1621607.1621624. URL: http://doi.acm.org/10.1145/1621607.1621624.

[7] M. B. Jovan Milosevic Milos Glisic. "System and method for client side rendering of a web page". Patent US 2006/0248166 A1 (Toronto). Nov. de 2006. URL: http://www.lens.org/lens/patent/US_2006_0248166_A1.

[8] A Leff e J. Rayfield. "Web-application development using the Model/View/Controller design pattern". Em: *Enterprise Distributed Object Computing Conference, 2001. EDOC '01. Proceedings. Fifth IEEE International*. 2001, pp. 118–127. DOI: 10.1109/EDOC.2001.950428.

[9] A. Leff e J. T. Rayfield. "Web-Application Development Using the Model/View/-Controller Design Pattern". Em: *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*. EDOC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 118–. ISBN: 0-7695-1345-X. URL: http://dl.acm.org/citation.cfm?id=645344.650161.

[10] F. Nielson, H. R. Nielson e C. Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[11] T. J. Parr. "Enforcing Strict Model-view Separation in Template Engines". Em: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 224–233. ISBN: 1-58113-844-X. DOI: 10.1145/988672.988703. URL: http://doi.acm.org/10.1145/988672.988703.

[12] F. Pfenning. "Lecture Notes on Dataflow Analysis". Em: (2008).

[13] W. Pree e H. Sikora. "Design Patterns for Object-oriented Software Development (Tutorial)". Em: *Proceedings of the 19th International Conference on Software Engineering*. ICSE '97. Boston, Massachusetts, USA: ACM, 1997, pp. 663–664. ISBN: 0-89791-914-9. DOI: 10.1145/253228.253810. URL: http://doi.acm.org/10.1145/253228.253810.

[14] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. 5th. McGraw-Hill Higher Education, 2001. ISBN: 0072496681.

[15] M. I. Schwartzbach. "Lecture notes on static analysis". Em: *Basic Research in Computer Science, University of Aarhus, Denmark* (2008).

[16] Y. Smaragdakis e M. Bravenboer. "Using Datalog for Fast and Easy Program Analysis". Em: *Proceedings of the First International Conference on Datalog Reloaded*. Datalog'10. Oxford, UK: Springer-Verlag, 2011, pp. 245–251. ISBN: 978-3-642-24205-2. DOI: 10.1007/978-3-642-24206-9_14. URL: http://dx.doi.org/10.1007/978-3-642-24206-9_14.

[17] M. Tatsubori e T. Suzumura. "HTML Templates That Fly: A Template Engine Approach to Automated Offloading from Server to Client". Em: *Proceedings of the 18th International Conference on World Wide Web*. WWW '09. Madrid, Spain: ACM, 2009, pp. 951–960. ISBN: 978-1-60558-487-4. DOI: 10.1145/1526709.1526837. URL: http://doi.acm.org/10.1145/1526709.1526837.

[19] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke e J. Shanmugasundaram. "A Unified Platform for Data Driven Web Applications with Automatic Client-server Partitioning". Em: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 341–350. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242619. URL: http://doi.acm.org/10.1145/1242572.1242619.

Recursos online

[20] *AngularJS*. (acedido em 04-10-2013). URL: http://angularjs.org.

[21] *AngularJS: suporte a serviços REST*. (acedido em 02-10-2013). URL: http://docs.angularjs.org/api/ngResource.

[22] *Backbone.js*. (acedido em 09-10-2013). URL: http://backbonejs.org/.

[23] V. Basavaraj. *The client-side templating throwdown: mustache, handlebars, dust.js, and more*. (acedido em 20-11-2013). URL: http://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more.

[24] *EmberJS*. (acedido em 01-09-2014). URL: http://emberjs.com.

[25] *Gallery of applications built with AngularJS*. (acedido em 29-11-2013). URL: http://builtwith.angularjs.org.

[26] *HTTP Archive - Trends*. (acedido em 22-01-2014). URL: http://httparchive.org/trends.php.

[27] B. Kiefer. *The Trello Tech Stack*. (acedido em 29-11-2013). URL: http://blog.fogcreek.com/the-trello-tech-stack/.

[28] *Knockout*. (acedido em 09-10-2013). URL: http://knockoutjs.com.

[29] *MessagePack - It's like JSON, but fast and small*. (acedido em 15-09-2014). URL: http://msgpack.org/.

[30] *Mustache*. (acedido em 09-10-2013). URL: http://mustache.github.io/.

[31] *TodoMVC - Helping you select an MV* framework*. (acedido em 02-10-2013). URL: http://todomvc.com.

[32] *Top JavaScript MVC Frameworks*. (acedido em 20-11-2013). URL: http://www.infoq.com/research/top-javascript-mvc-frameworks.

[33] D. Webb. *Improving performance on twitter.com*. (acedido em 22-11-2013). URL: https://blog.twitter.com/2012/improving-performance-twittercom.

[34] *Xtext - Language Development Made Easy*. (acedido em 15-09-2014). URL: http://eclipse.org/Xtext/.

Exemplo ASP.NET vs AngularJS

As Listagens A.1 e A.2 mostram o contraste entre um *template* tipo servidor e tipo cliente, definidos respetivamente em `ASP.NET` e a *framework* `JavaScript AngularJS`, gerando ambos a mesma tabela `HTML`.

A estrutura do *template* tipo servidor, na Listagem A.1, é definida por controladores da linguagem `ASP.NET`, todos implementados num pacote específico. O elemento <*asp:DataGrid*> exemplifica como se instancia o controlador *DataGrid*, que por sua vez se traduz em elementos `HTML` quando a página Web é gerada. Cada controlador oferece um conjunto de propriedades que permite definir a apresentação do conteúdo (e.g., <*asp:ItemTemplate*>). Os delimitadores <%# ... %> são usados para incluir código, que produz conteúdo no processo de geração da página, inserido no `HTML` consoante a definição do controlador que estiver em volta. Nesse código podem ser invocadas funções, implementadas no *code-behind*, associado ao *template* onde é definido o código que implementa a lógica aplicacional da página Web.

O *template* tipo cliente, da Listagem A.2, é definido em `HTML` estendido com a anotação do `AngularJS`, que consiste em atributos com nomes predefinidos, e {{ ... }} como a alternativa mais direta para especificar a inserção de conteúdo. Associado ao *template* existe um *script* que liga os dados fornecidos pelo servidor aos parâmetros usados no *template*. Na Listagem A.3 é apresentado o exemplo do *script* associado ao *template* A.2, onde se assume que existe um serviço REST [2] para fornecer os dados, cuja comunicação é simplificada pela *framework* `AngularJS` [21].

Comparando os dois tipos de *template*, o *template* tipo cliente revela os seguintes pontos positivos: menos linhas de código; perceção mais direta do conteúdo que será apresentado na página Web; associação simples com fonte de dados, sem necessidade de invocar funções; e a especificação em `HTML` simplifica a criação do *template*.

```
<asp:DataGrid>
  <Columns>
    <asp:TemplateColumn>
      <HeaderTemplate>
        <asp:PlaceHolder runat="server"> <%# "Name" %> </asp:PlaceHolder>
      </HeaderTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn>
      <ItemTemplate>
        <asp:PlaceHolder runat="server">
          <%# expressionExpSongName() %>
        </asp:PlaceHolder>
      </ItemTemplate>
    </asp:TemplateColumn>
  </Columns>
</asp:DataGrid>
```

Listagem A.1: *Template* tipo servidor em `ASP.NET` com tabela dos nomes das músicas.

```
  <table>
    <thead>
      <th>Name</th>
    </thead>
    <tbody>
      <tr ng−repeat="song in songs">
        <td> {{song.songName}} </td>
      </tr>
    </tbody>
  </table>
```

Listagem A.2: *Template* tipo cliente em `AngularJS` para gerar tabela com nomes de músicas.

```
function SongsController ($scope,resource) {
  var Songs = $resource('/MusicLibrary/songs')$;scope.songs = Songs.query();
}
```

Listagem A.3: *Script* para associar os dados ao *template* em `AngularJS`.

B

Ecrã no Service Studio

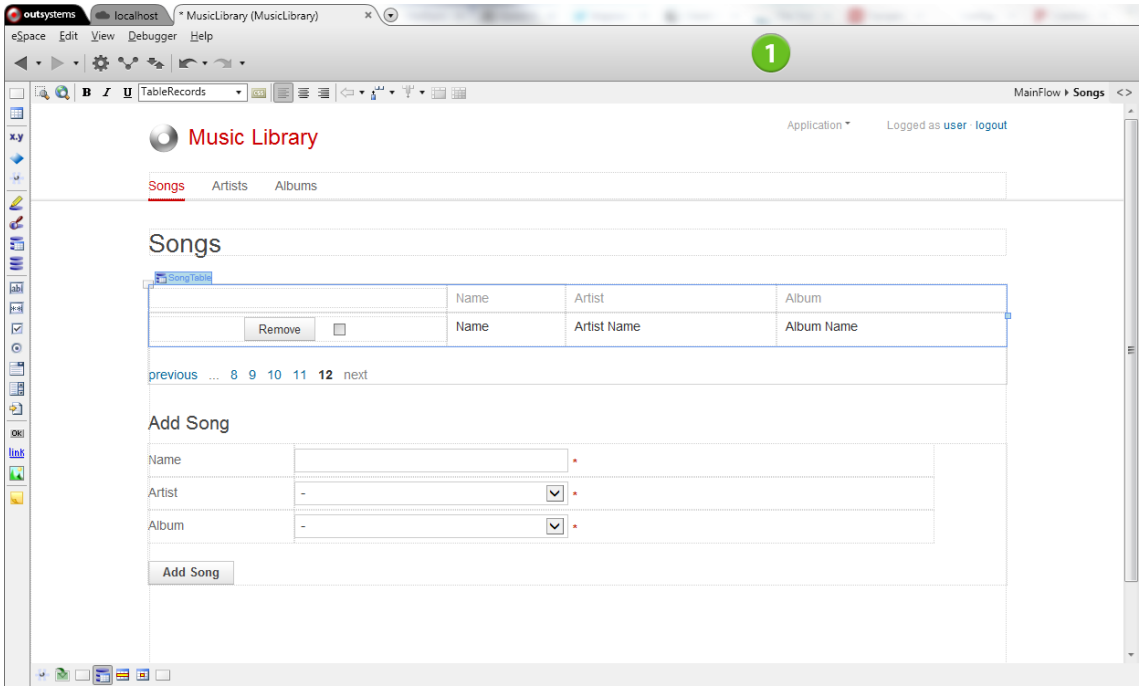


Figura B.1: Desenvolvimento do ecrã de músicas da aplicação Biblioteca Musical no *Service Studio*.

C

Frameworks JavaScript

C.1 AngularJS

Os *templates* definidos em `AngularJS` são documentos `HTML` estendidos com diretivas cuja anotação consiste na inserção de atributos específicos aos elementos `HTML`, e que definem como será gerado o `HTML` relativo ao aspeto final da página web. As diretivas mais comuns no desenvolvimento de aplicações, são a ngBind, ngModel, ngRepeat, ngView e ngController. Segue-se uma descrição de cada uma:

ngBind Substitui o conteúdo do elemento `HTML` pelo valor da expressão indicada, e atualiza o conteúdo caso o valor da expressão altere. Esta diretiva tem a particularidade de que tanto pode ser referida como atributo do elemento `HTML`, ou então abreviada como <element> ... {{ *expressão* }} ... </element>.

ngModel Usada em elementos como *input*, *select* ou *textarea*, permitindo alterações ao modelo da aplicação consoante os dados inseridos ou manipulados pelo utilizador. Esta diretiva e a anterior constituem a propriedade de *two-way data binding* (fornecida pelo `AngularJS`) que consiste no fluxo de dados entre a camada de apresentação e modelo. Ou seja, se os dados são alterados no modelo, o `HTML` é atualizado, se o utilizador insere ou manipula dados através da interface Web, o modelo é atualizado.

ngRepeat Tem a mesma função do ngBind, mas destina-se à iteração sobre um conjunto de dados, gerando `HTML` para cada elemento no conjunto.

ngView Inerente a esta diretiva está o uso do serviço de *routing*. Em aplicações de várias

páginas, que partilham uma estrutura, alternando uma parte do conteúdo, consoante a página corrente, associada a uma *route*, o conteúdo do respetivo elemento é substituído pelo fragmento `HTML` que representa o conteúdo dessa página, de acordo com a configuração de *routing*.

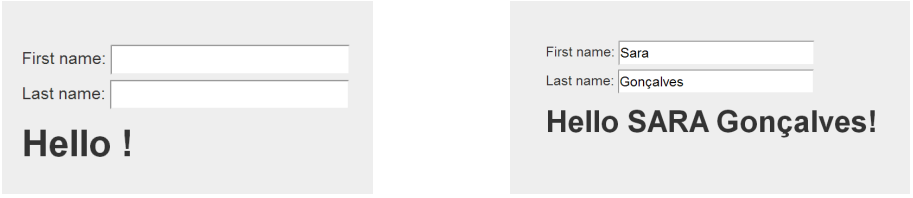
ngController Permite associar um controlador a um fragmento da página, criando um novo objeto *scope*, ou seja, um modelo, responsável pelos dados apresentados nesse fragmento.

O uso das diretivas ngModel e ngBind é demonstrado no exemplo Hello World, apresentado na Listagem C.1, cujo objetivo é: conforme o utilizador insere o primeiro e último nome nos *inputs*, é visualizado a adição do nome completo ao texto *Hello <name>!*, sem ser necessário um botão ou tecla para submissão de dados. As expressões *firstName* e *lastName* tornam-se propriedades do objeto *$scope*, manipuladas no input através da diretiva ngModel, e apresentadas através da diretiva ngBind (abreviada por {{ .. }}). Para esta página não é necessário um controlador, pois a substituição das expressões, inclusive a formatação de *firstName* para *upperCase* é especificada no *template* como {{*firstName* | *uppercase*}}, automaticamente processada pelo `AngularJS`. O resultado desta página, antes e depois da inserção do nome, é apresentada na Figura C.2.

O `AngularJS` usa um mecanismo, ao qual se dá o nome de *dirty checking*, que torna automático a atualização de dados nos dois sentidos, ou seja do modelo para a visualização da página, e da página para o modelo. Este mecanismo baseia-se na procura por alterações em todas as propriedades do objeto *$scope*.

```
1  <html ng-app>
2    <head>
3      <script src="angular.js"></script>
4    </head>
5    <body>
6      <div>
7          <label>First name: <input type="text" ng-model="firstName" /></label>
8          <label>Last name: <input type="text" ng-model="lastName" /></label>
9
10         <h1>Hello {{firstName | uppercase}} {{lastName}}!</h1>
11     </div>
12   </body>
13 </html>
```

Listagem C.1: *Template* em `AngularJS` da página Hello World



(a) Input vazio. (b) Input preenchido.

Figura C.2: Página *Hello World* visualizada no browser.

C.2 KnockoutJS

Os *templates* definidos em `KnockoutJS` são documentos `HTML` com a adição do atributo *data-bind* aos elementos `HTML`, de forma a definir o comportamento do documento. Mantendo o exemplo usado anteriormente, o Hello World, na Listagem C.3 podemos observar o uso do atributo *data-bind* com os valores *text* e *value*, sendo o primeiro usado para a visualização do valor da expressão, *fullName* neste caso, e o segundo para manipular o valor das expressões *firstName* e *lastName*.

No `KnockoutJS` é sempre necessário definir código `JavaScript` para especificar a componente *ViewModel*, onde os objetos e variáveis do modelo (os dados da página web) devem ser explicitamente declarados. Subjacente a esta declaração, estão as propriedades *observable* e *computed* atribuídas às variáveis. É a partir da dependência entre variáveis criada por estas propriedades que o `KnockoutJS` efetua a sua verificação por alterações aos dados do modelo, a fim de atualizar a visualização da página.

observable Esta propriedade permite o `KnockoutJS` atualizar automaticamente uma página Web conforme as alterações às variáveis do modelo. A declaração de uma variável como *observable* gera uma notificação, provocando a verificação das variáveis dependentes desse, a fim de atualizar também a sua apresentação na página Web.

computed Uma variável *computed* corresponde a uma função que computa o valor com dependência de uma ou mais variáveis *observable*. Quando as *observable* são alteradas, é detetada a dependência, e a variável *computed* é computada de novo.

A junção das propriedades *observable* e *computed* com os valores *text* ou *value* do atributo *data-bind* permitem a propriedade de *two-way data binding* (apresentada na secção C.1) no `KnockoutJS`.

Na Listagem C.4 observa-se a declaração das variáveis *firstName* e *lastName* como *observable*, uma vez que o utilizador pode manipular os seus valores através dos *inputs*, e devem por isso ser vigiadas a fim de detetar alterações, e consequentemente atualizar variáveis dependentes, neste caso a variável *fullName*, refletindo-se na visualização da página. Tal como no exemplo usado anteriormente, pretende-se que a expressão *firstName* seja formatada para *upperCase* quando visualizada entre o texto *Hello*

<name>!. Em `KnockoutJS` não é possível especificar tal formatação diretamente no *template*, sendo necessário especificar no *script*, como é visualizado na Listagem C.4 em *this.firstName().toUpperCase()*

Ao contrário do `AngularJS`, este exemplo implementado em `KnockoutJS` exige que o utilizador prima fora do input, ou prima *enter*, de forma que os dados inseridos no *input* sejam processados para que seja efetuada a verificação e consequentemente se visualize alterações na página.

Ainda no código apresentado na listagem C.4, a função *applyBindings* é o que ativa o `KnockoutJS` para analisar os atributos *data-bind* expressos no *template*, para no fim gerar a página com o respetivo conteúdo.

O resultado desta página, antes e depois da inserção do nome, é o mesmo apresentado na Figura C.2.

```
<html>
  <head>
    <script src="knockout.js"></script>
    <script src="helloworld.js"></script>
  </head>
  <body>
    <div>
      <label>First name: <input data-bind="value: firstName" /> </label>
      <label>Last name: <input data-bind="value: lastName" /> </label>

      <h1>Hello <span data-bind="text: fullName"></span>! </h1>
    </div>
  </body>
</html>
```

Listagem C.3: *Template* em KnockoutJS da página Hello World.

```
var AppViewModel = function(name) {
    this.firstName = ko.observable("");
    this.lastName = ko.observable("");

    this.fullName = ko.computed(function() {
        return this.firstName().toUpperCase() + " " + this.lastName();
    }, this);
};
ko.applyBindings(new AppViewModel());
```

Listagem C.4: Código da definição do *ViewModel* da página Hello World.

D

Exemplo código gerado ação *Preparation*

```
1  public class ScrnSongs {
2    SongRecordList queryGetSongs_outParamList =
3      new SongRecordList();
4      ...
5    public void Preparation(...){
6      queryGetSongs_outParamList = FuncssPreparation.QueryGetSongs(...);
7      queryGetAlbums_outParamList = FuncssPreparation.QueryGetAlbums(...);
8    }
9  }
10 public static class FuncssPreparation{
11   public static SongRecordList QueryGetSongs(...){
12     string sql = "";
13     string sqlSelect = "SELECT ENArtist.NAME, ENSong.NAME, ENAlbum.NAME ";
14     string sqlFrom = " FROM ENArtist INNER JOIN
15                        ENSong ON (ENSong.ARTISTID = ENArtist.ID) INNER JOIN
16                        ENAlbum ON (ENSong.ALBUMID = ENAlbum.ID)";
17     sql = sqlSelect + sqlFrom;
18     ArtistSongAlbumRecordList outParamList = new ArtistSongAlbumRecordList();
19     // Execution of query
20     return outParamList;
21   }
22   public static AlbumRecordList QueryGetAlbums(...){ ... }
23 }
```

Listagem D.1: Código gerado (simplificado) respetivo à ação Preparação.

Widgets

```
widget ListRecords {
  designtimeProperties { List src }
  runtimeProperties { List list }
  refreshDataSource { list = src }
  placeholders { item }
  html {
    <ul>
      <li ng-repeat = list> item  </li>
    </ul>
  }
}
widget If {
  designtimeProperties { Boolean conditionValue }
  placeholders { trueBranch, falseBranch }
  html {
    <span>
      <span ng-if = conditionValue> trueBranch </span>
      <span ng-if = !conditionValue> falseBranch </span>
    </span>
  }
}
widget Expression {
  designtimeProperties { Exp value }
  html { <div>{{ value }}</div> }
}
widget Button {
  designtimeProperties {
```

```
28      Text label
29      Destination onClick
30    }
31    html {
32      <button ng-click = onClick style='display: inline'> {{ label }} </button>
33    }
34  }
35  widget Input {
36    designtimeProperties {
37      Exp variable
38      Text typeInput
39      Text classInput
40      Text textholder
41    }
42    runtimeProperties {
43      Boolean valid = true
44      Text validationMessage = ""
45    }
46    html {
47      <input ng-model=variable type=typeInput class=classInput placeholder=textholder />
48    }
49  }
50  widget TextField {
51    designtimeProperties { Text field }
52    html { {{ field }} }
53  }
```

Listagem E.1: Declaração de algumas *widgets* no modelo proposto, que existem na plataforma *OutSystems*.

F

Programa de Datalog

```
1  def('Bill_inputTotal','model.total').
2  def('Bill_inputFriend','model.newFriend.Friend.Name').
3  def('Bill_friendsList_friendCheck','[List].model.friendsList_list').
4  use('Bill','[List].model.friendsList_list.Item.Friend.Name').
5  use('Bill', 'model.total').
6  exp('model.EvalData.ifCount.conditionValue').
7  need('model.EvalData.ifCount.conditionValue','[List].model.friendsList_list').
8  exp('model.EvalData.costPPerson.value').
9  need('model.EvalData.costPPerson.value','model.EvalData.ifCount.conditionValue').
10 need('model.EvalData.costPPerson.value','model.total').
11 need('model.EvalData.costPPerson.value','[List].model.friendsList_list').
12 exp('model.EvalData.friendsToPay.value').
13 need('model.EvalData.friendsToPay.value','model.EvalData.ifCount.conditionValue').
14 need('model.EvalData.friendsToPay.value','[List].model.friendsList_list').
15 // Actions
16 action('Bill', 'AddFriend').
17 entry('AddFriend','AddFriend_0').
18 node('AddFriend', 'AddFriend_0').
19 use('AddFriend_0', 'model.newFriend.Friend.Name').
20 use('AddFriend_0_0','[List].model.friendsList_list').
21 use('AddFriend_0_0', 'model.newFriend').
22 node('AddFriend', 'AddFriend_0_0').
23 use('AddFriend_0_0','[List].model.friendsList_list').
24 def('AddFriend_0_0','[List].model.friendsList_list').
25 succ('AddFriend_0','AddFriend_0_0').
26 succ('AddFriend_0','AddFriend_0_1').
27 succ('AddFriend_0','AddFriend_eval').
```

```
28 node('AddFriend','AddFriend_eval').
29
30 action('Bill', 'Save').
31 node('Bill_save','Bill_save').
32 entry('Save','Save_0').
33 node('Save', 'Save_0').
34 succ('Save_0','Save_1').
35 node('Save', 'Save_1').
36 use('Save_1','[List].model.friendsList_list').
37 use('Save_1_0','[List].model.friendsList_list.Item.Friend').
38 node('Save', 'Save_1_0').
39 succ('Save_1','Save_1_0').
40 succ('Save_1_0','Save_1').
41 succ('Save_1','Save_eval').
42 node('Save','Save_eval').
43
44 action('Bill', 'Reset').
45 node('Bill_reset','Bill_reset').
46 entry('Reset','Reset_0').
47 node('Reset', 'Reset_0').
48 succ('Reset_0','Reset_1').
49 node('Reset', 'Reset_1').
50 def('Reset_1','model.queryFriends').
51 succ('Reset_1','Reset_2').
52 node('Reset', 'Reset_2').
53 use('Reset_2','model.queryFriends').
54 def('Reset_2','[List].model.friendsList_list').
55 succ('Reset_2','Reset_eval').
56 node('Reset','Reset_eval').
57
58 // Rules
59 mod(?a, ?v) :− node(?a, ?n), def(?n, ?v).
60 use(?n, ?v) :− not exp(?v), not mod(?a, ?v), node(?a, ?n), needEval(?a, ?f), need(?f, ?v).
61 needEval(?a, ?f) :− mod(?a, ?v), need(?f, ?v).
62 needEval(?a, ?f) :− need(?f, ?g), needEval(?a, ?g).
63 live(?n, ?v) :− use(?n, ?v).
64 live(?n, ?v) :− not def(?n, ?v), succ(?n, ?m), live(?m, ?v).
65 live(?s, ?v):− action(?s, ?a), entry(?a, ?n), live(?n, ?v).
66 CToS(?a, ?v) :− entry(?a, ?n), live(?n, ?v).
67 SToC(?a, ?v) :− action(?s, ?a), live(?s, ?v), mod(?a, ?v).
68
69 //Queries
70 ?−needEval('Bill_addLabel', ?e).
71 ?−needEval('Bill_inputTotal', ?e).
72 ?−needEval('Bill_ifCount_costPPerson', ?e).
73 ?−needEval('Bill_ifCount_friendsToPay, ?e).
```

```
74 ?−needEval('Bill_ifCount', ?e).
75 ?−needEval('Bill_inputFriend', ?e).
76 ?−needEval('Bill_friendsList_friendCheck', ?e).
77 ?−needEval('Bill_friendsList_friendName', ?e).
78 ?−needEval('Bill_friendsList', ?e).
79 ?−live('Bill', ?v).
80 ?−needEval('AddFriend', ?e).
81 ?−CToS('AddFriend', ?v).
82 ?−SToC('AddFriend', ?v).
83 ?−needEval('Reset', ?e).
84 ?−CToS('Reset', ?v).
85 ?−SToC('Reset', ?v).
86 ?−needEval('Save', ?e).
87 ?−CToS('Save', ?v).
88 ?−SToC('Save', ?v).
```

Listagem F.1: Programa de `Datalog` gerado a partir da definição do modelo da aplicação SplitTheBill.