Introdução à API do Insight Maker

Aula do Curso de Dinâmica de Sistemas

Por: Nathan de Lima Silva

05 de setembro de 2024

ÍNDICE

Introdução

Sistemas dinâmicos possuem comportamentos que não podem ser explicados ou descritos a partir da explicação ou descrição de seus componentes internos. Isto se deve à estrutura interna do sistema em questão. Logo, a dinâmica de um sistema é resultado da natureza de seus componentes internos e da forma como eles estão organizados (arranjados, conectados).

Existem técnicas conhecidas para descrever esses sistemas, e dentre elas destacamos duas: os sistemas de equações diferenciais e os diagramas de estoque e fluxo. Ambos possuem relação ambivalente, sendo que o segundo pode ser visto como uma linguagem visual para o primeiro, e portanto mais intuitiva. De qualquer forma, estas descrições de um sistema são sempre aproximativas. A precisão muitas vezes está relacionada ao nível de descrição que o modelo oferece. Normalmente, quanto mais preciso, mais complexo um modelo se torna devido ao maior número de detalhes fornecidos.

O objetivo de se desenvolver um modelo para um sistema é poder observar como um certo conjunto arbitrário de variáveis desse sistema evolui ao longo do tempo. Muitas vezes, isso está associado a plotar gráficos do valor de uma variável em relação ao tempo, o que presume o uso de um computador para isso. Como é sabido, computadores são máquinas eletrônicas digitais, o que significa que ela trabalha com quantidades inteiras, inclusive no que diz respeito ao tempo. Portanto, as equações diferenciais em tempo contínuo são calculadas em tempo discreto. Diversos algoritmos são capazes de realizar tal feito. Não é nosso objetivo aqui nos preocuparmos com esses algoritmos. Em vez disso, utilizamos a ferramenta Insight Maker para realizar todo este trabalho.

A forma como interagimos com o Insight Maker até aqui foi através de sua Interface Gráfica (*Graphical User Interface* - GUI), escrevendo os modelos através de diagramas de estoque e fluxo. Nossa intenção neste trabalho é a de apresentar uma segunda forma de interagir com o Insight Maker: através de sua API (*Application Programming Interface*).

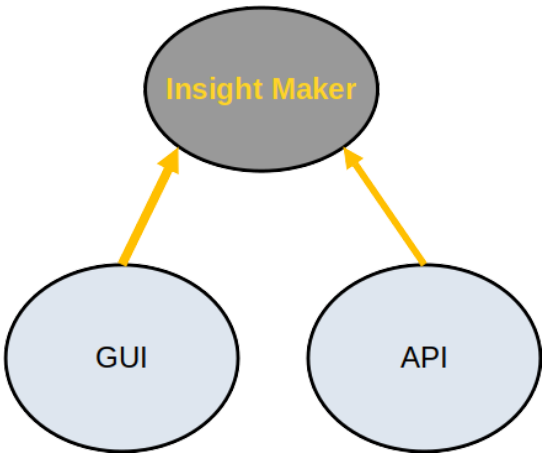

Fig. 1 - Formas de interagir com o Insight Maker

Conceito de APIs

APIs, ou *Application Programming Interfaces*, são conjuntos de interfaces pelas quais interagimos com outro sistema. Essas interfaces são descrições contratuais dos métodos de interação fornecidos pelo próprio sistema. Com isto, deixamos de nos preocupar com a lógica interna do sistema e passamos a tratá-lo como um sistema de caixa-preta, e é uma metodologia que surgiu conceitualmente já há bastante tempo.

Exemplo didático do conceito de API: Serviço de restaurante

Como exemplo didático, vamos ilustrar o serviço de atendimento de um restaurante do ponto de vista da pessoa que nele consome (cliente). Numa primeira modelagem, vamos considerar os seguintes quatro componentes: cliente, garçom, cozinha e pedido. Por enquanto, o pedido pode ser tanto a requisição do mesmo quando é feita ao garçom, quanto também o produto que o garçom traz de volta até o cliente. Aqui então adiantamos que existirão duas classes de pedidos: requisição e resposta. Não necessariamente uma requisição sempre parte do cliente para o garçom, pois pode ser que o garçom solicite algo ao cliente (trocar algum item do pedido que esteja em falta, por exemplo).

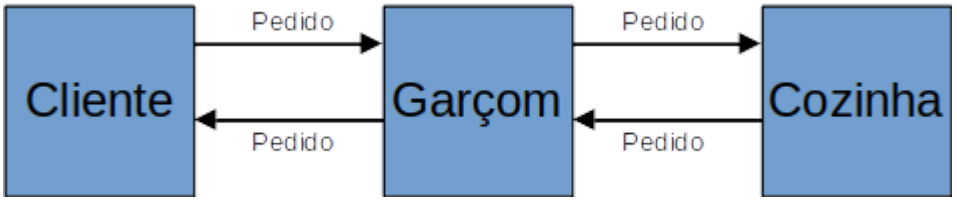

Fig. 2 - Modelagem simplificada de um restaurante

Evidentemente, existem mais processos ocorrendo em um restaurante, como a limpeza, contabilidade, manutenção, depósito, segurança, dentre outros. Nosso objetivo não é o de aprofundar no entendimento desses processos, mas sim utilizar modelos simplificados para expor didaticamente o conceito de APIs.

Na figura 1, a única forma de interação entre o cliente e o garçom é através do pedido. Entretanto, existem outras formas de interação para as quais um bom garçom deve estar preparado para manipular. Por exemplo: solicitar o preço, verificar qual o valor atual da conta e efetuar o pagamento. Em todos esses casos, o cliente não está acessando diretamente a cozinha do restaurante. Ao invés disso, ele se comunica com o garçom utilizando um conjunto finito de funções que tanto o cliente quanto o garçom conhecem. Pelo menos o garçom deve conhecer, e caso seja a primeira vez de um cliente em um restaurante ele pode pedir ao garçom um descritivo das funções que o cliente pode solicitar a ele (ensinar como usar um restaurante).

Na terminologia utilizada em Tecnologia da Informação, o conjunto de funções que eu posso usar para interagir com o garçom é o que se chama de API. Essas funções compõem a interface entre o cliente e o restaurante, e são os serviços que o restaurante oferta aos clientes. O garçom, por sua vez, é o agente que *serve* a refeição ao cliente. Por isso é

chamado de *servidor*. Por ele estar “no meio do caminho” entre o cliente e a cozinha, o garçom também é comumente chamado de *middleware*.

A entidade que trafega entre o cliente e o garçom são as requisições e respostas. Nem tudo o garçom precisará encaminhar até a cozinha, pois ele tem autonomia para executar algumas funções por si mesmo e também possui em sua memória resultados para as requisições que são feitas com mais frequência, como os preços e as composições dos itens do cardápio. Como ele sabe que preços e ingredientes costumam variar todo dia, ele faz apenas a primeira requisição do dia à cozinha e grava em sua memória rápida (cache) esse resultado, com o propósito de aumentar a eficiência do atendimento.

A cozinha, no mesmo contexto da Tecnologia da Informação, está em uma região do modelo chamada de *backend*. No backend é onde estão as lógicas por trás do negócio (do restaurante) que conferem àquele estabelecimento sua própria identidade e qualidade. O usuário não quer saber dessa parte burocrática, mecânica e cheia de detalhes, só quer consumir o pedido.

APIs no contexto da programação

Saindo um pouco do contexto da modelagem de sistemas e arquitetura de software, iremos agora descrever o conceito de API no contexto do desenvolvimento de software, mais especificamente do ponto de vista das linguagens de programação.

As APIs estão por toda parte. Na programação podemos citar dois exemplos emblemáticos: as Bibliotecas e as Web APIs.

Bibliotecas são conjuntos de rotinas encapsuladas em funções que são disponibilizadas ao programador. Por exemplo, quando escrevemos um código *Hello, World!* em linguagem C, precisamos importar a biblioteca stdio.h, que contém a lógica de baixo nível para interagir com o sistema operacional e solicitar a ele que imprima a frase *Hello, World!* na tela. Com isso, podemos compilar o mesmo código tanto em Linux quanto em Windows e o resultado será o mesmo, pois o que mudará é a biblioteca. O nome da biblioteca permanece o mesmo, mas a lógica de baixo nível para interação com o sistema operacional (que muitas vezes é escrita em uma mistura de C e Assembly) é feita pela biblioteca que já vem com o compilador. Isso ocorre pois estamos utilizando uma biblioteca *builtin* (stdio.h) e o padrão ANSI-C. Isso nem sempre é verdade pois muitas vezes utilizamos bibliotecas que não são nativas da linguagem C (isto é, não são *builtin*), e que pode existir apenas para um sistema operacional específico.

Programa 1 - *Hello, World!* em C

```
#include <stdio.h>
int main() {
        printf("Hello, World!\n");
        return 0;
}
```

Não apenas podemos importar bibliotecas que são disponibilizadas pelos fornecedores, como podemos também encapsular trechos de códigos em funções, e agrupá-las em nossas próprias bibliotecas. Por exemplo, vamos criar em linguagem Python uma biblioteca com uma função que imprime *Hello, World!* na tela:

Programa 2 - Biblioteca Utils.py, em Python

```
# Utils.py

def imprime_hello_world():
        print("Hello, World!")
```

Em seguida, vamos criar um script que faz uso dessa biblioteca criada:

Programa 3- Uso da biblioteca Utils.py

```
# main.py

from Utils import imprime_hello_world

imprime_hello_world()
```

ou:

Programa 3.b - Uso da biblioteca Utils.py

```
# main.py

import Utils

Utils.imprime_hello_world()
```

Presume-se que as bibliotecas estão dentro de uma mesma plataforma, ou seja, dentro de um mesmo computador. Mesmo que para isso seja feito *download* das mesmas durante o processo de *build* da aplicação. Existem também os sistemas dispersos, ou distribuídos.

Vamos supor um programa que interage com a API da Receita Federal para emissão de NFe. E vamos supor também que o servidor da Receita Federal (RF) deverá fazer algumas consultas nos servidores da Secretaria de Fazenda (SeFaz).

Esse programa estará consumindo uma API cujos processos estão alocados fisicamente em outro local, em outro servidor. Nesse caso, as requisições e respostas ocorrem por meio da internet. Existem várias tecnologias para fazer o transporte dessas requisições e respostas, como o TCP, IP, HTTP, RPC, SOAP, REST, dentre outros. Isto para mencionar apenas os casos básicos, pois sistemas baseados em microsserviços comumente são implantados por meio de orquestradores de containers, e normalmente esses orquestradores como o Kubernetes possuem serviços internos de mensageria diferentes da pilha TCP/IP.

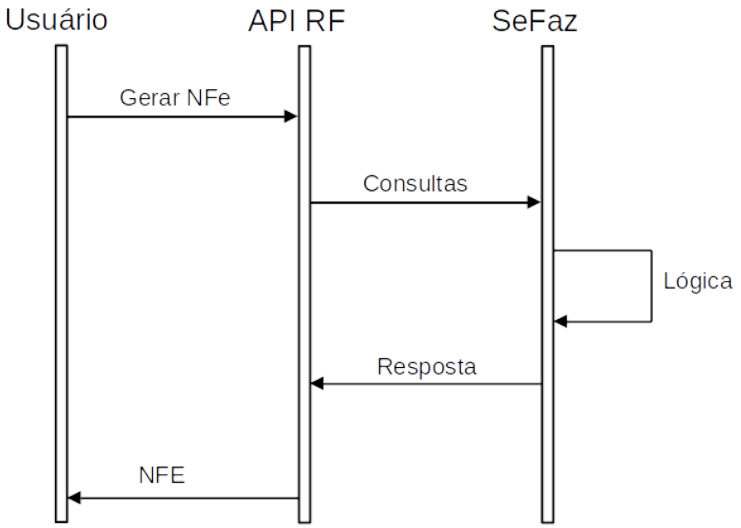

Fig. 3 - Diagrama de sequência de uma requisição de emissão de NFe

Algoritmos e Pipelines

Nesta seção descreveremos o que são algoritmos e pipelines. Iremos abordar estes dois tópicos do ponto de vista da tecnologia, e para isto vamos desenvolver um ponto de vista sobre o que é a tecnologia. Começaremos com alguns exemplos.

Stack tecnológica

Quando utilizamos aplicativos em um smartphone, como o Uber ou o Ifood, notamos que eles basicamente são compostos em cima de uma reunião de tecnologias que já existiam: GPS, Internet e sistemas de pagamento (sistemas bancários e fiscais). Essas tecnologias, por sua vez, são construídas sobre outras tecnologias, como o sistema de comunicação (WiFi, 4G, 5G), o hardware referente ao GPS e a lógica de counicação segura (HTTP, SSL, TCP, IP) e consumo de web APIs (REST, RPC, SOAP, etc) para a parte do pagamento. Podemos ainda dizer que essas tecnologias são construídas a partir da existência de tecnologias anteriores, e assim por diante. Chamamos a esse “empilhamento” de tecnologias de *stack*.

Podemos descrever as *stacks* como uma “pilha de caixas”, ou o empilhamento de componentes. Na vida real, esse empilhamento nem sempre é bem definido. Cada um dos componentes também pode (e normalmente tem) uma *stack* intrínseca. Por exemplo, o aplicativo do Uber possui internamente componentes separados para lidar com requisições aos servidores, interface gráfica e lógicas de funcionamento. Cada um desses componentes demandará bibliotecas para efetuar as requisições, UI, gerenciamento de aplicação, etc.

Sem mencionar que tudo isso é desenvolvido utilizando linguagens de programação e portanto possuem sistemas de build e runtime com outras complexidades intrínsecas.

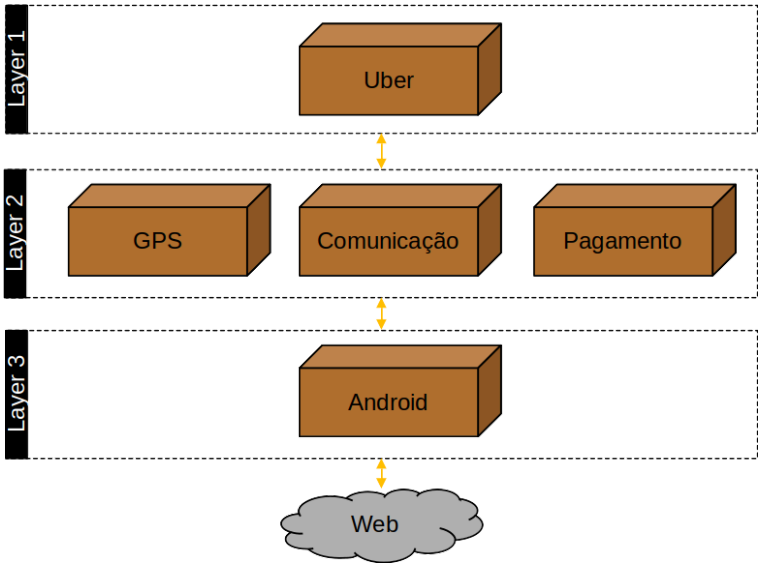

Fig.4 - Exemplo de *stack* para o aplicativo Uber.

Pipelines

Agora que já foi discutido o conceito de pilha, ou *stack*, vamos falar do conceito de Pipelines. Podemos adiantar que elas também estão relacionadas ao conceito de pilha, mas vamos expor um segundo conceito: o de linha de montagem.

Pipelines são sequências de ações com o objetivo de montar um produto final. Este conceito vem do Toyotismo, aplicado nas linhas de montagem de automóveis. Essas linhas de montagem são compostas por estágios interdependentes entre si, onde cada estágio recebe do estágio anterior o produto parcial que está sendo montado e entrega ao estágio seguinte o produto parcial em um novo estado de montagem.

Quando um software é desenvolvido, o produto final dele são os artefatos. Quando um usuário clica em um ícone na área de trabalho para abrir um programa, implicitamente esta ação irá executar o executável daquele programa (por exemplo um arquivo com extensão .exe). Este executável é o produto final do desenvolvimento daquele software, ou seja, o artefato que foi gerado. Entretanto, nem sempre o artefato é um executável. Em softwares de servidor, normalmente os artefatos são módulos que são implantados em servidores de aplicação. Por exemplo, um programa escrito em Java que será executado em um servidor Weblogic, terá como artefato um arquivo com extensão .war.

Um script escrito como uma sequência de comandos também pode ser visto como uma pipeline. Por exemplo, vamos criar um script em *Shell Script* que irá efetuar os seguintes estágios:

1. Mudar o diretório atual para a pasta *home*;
2. Criar um arquivo chamado “teste.txt”
3. Criar uma pasta chamada “pasta_teste”
4. Mover o arquivo para a pasta

Programa 4 - Script em forma de pipeline

```
cd ~
touch teste.txt
mkdir pasta_teste
mv teste.txt pasta_teste
```

Uma vez que as pipelines são compostas por estágios sequenciais, muitas vezes o símbolo das pipelines é uma pilha, indicando que é um empilhamento de estágios. A propósito, o nome “pilha” é bastante sugestivo pois quando falamos de pipelines, pois cada estágio de uma pipeline só pode ser executado se o estágio anterior for executado. Da mesma forma, uma caixa só pode ser retirada de uma pilha de caixas caso a caixa de cima já tenha sido retirada antes.

Fig. 5 - Símbolo geralmente usado para representar Pilhas e Pipelines.

Algoritmos

Algoritmos são conjuntos de instruções bem definidas e organizadas que são seguidas para resolver um problema ou executar uma tarefa específica. Eles são fundamentais em ciência da computação, pois fornecem uma abordagem sistemática para processar dados e tomar decisões. Um algoritmo pode ser simples, como uma receita de bolo que segue uma sequência linear de passos, ou complexo, envolvendo estruturas condicionais, loops, e operações paralelas. A flexibilidade dos algoritmos permite que sejam aplicados em diversas áreas, desde a ordenação de listas até a inteligência artificial.

Pipelines, por outro lado, são mais restritos em sua estrutura, focando em uma sequência linear de estágios ou comandos, onde a saída de um estágio serve como entrada para o próximo. Eles são amplamente utilizados em processos automatizados e scripts, especialmente na integração e entrega contínua (CI/CD) em desenvolvimento de software. Embora os pipelines sejam eficientes para tarefas repetitivas e bem definidas, eles geralmente não suportam a complexidade e a flexibilidade de um algoritmo completo, sendo limitados na capacidade de executar operações paralelas ou lidar com loops complexos.

A principal diferença entre algoritmos e pipelines reside na flexibilidade e complexidade das operações que cada um pode realizar. Enquanto um pipeline é semelhante a uma linha de produção, seguindo uma ordem pré definida de operações, um algoritmo pode se adaptar a diferentes condições e caminhos, podendo incluir módulos que operam em paralelo, funções que separam partes do processo, e loops que repetem operações conforme

necessário. Isso torna os algoritmos mais adequados para resolver problemas complexos e dinâmicos, enquanto os pipelines são ideais para processos lineares e previsíveis.

De forma didática, um algoritmo pode ser visto como uma “receita de bolo”, onde seguimos uma sequência de etapas (que não necessariamente é linear) para obter um resultado final. Na prática, vários algoritmos irão interagir entre si para resolver um problema do mundo real. Ou seja, Problemas reais de computação são resolvidos a partir do uso de um grande número de pequenos algoritmos.

HTML

HTML, sigla para HyperText Markup Language, é a linguagem padrão utilizada para criar e estruturar páginas na web. Criada nos primórdios da internet, em 1991, por Tim Berners-Lee, o HTML foi inspirado no formato dos jornais impressos da época. Assim como os jornais usam títulos, parágrafos, imagens e links para organizar o conteúdo, o HTML utiliza uma série de elementos e tags para definir a estrutura de uma página, como cabeçalhos, parágrafos, listas e links, permitindo que os navegadores exibam o conteúdo de forma organizada e acessível para os usuários.

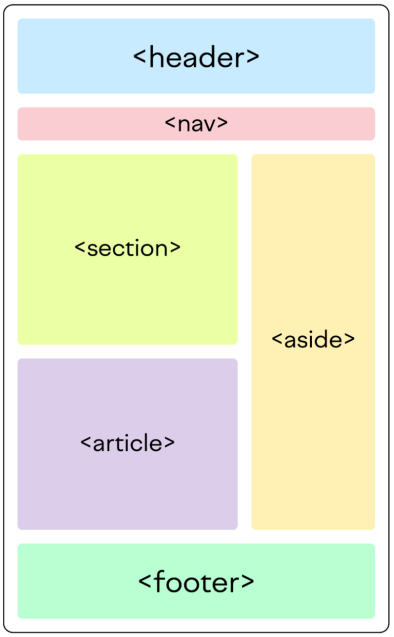

Fig. 5 - Estrutura básica de uma página HTML, inspirada nos jornais impressos e chamado de Semantic HTML.

Apesar de ser chamado de linguagem, o HTML não é considerado uma linguagem de programação, mas sim uma linguagem de marcação. Isso ocorre porque o HTML não possui lógica computacional, como condições, loops ou variáveis, que são características essenciais das linguagens de programação. Ao invés de realizar operações ou processar dados, o HTML apenas descreve a estrutura e o conteúdo de uma página. Seu principal objetivo é definir como os elementos de uma página devem ser apresentados, sem a capacidade de manipular dados ou tomar decisões lógicas por conta própria.

O conceito de HTML semântico refere-se ao uso de tags que têm um significado claro e específico em relação ao conteúdo que envolvem. Ao invés de usar apenas divs e spans genéricos para criar a estrutura de uma página, o HTML semântico emprega tags como `<article>`, `<header>`, `<footer>`, e `<nav>`, que indicam claramente o propósito de cada seção do conteúdo. Isso não só melhora a acessibilidade para usuários e mecanismos de busca, mas também torna o código mais legível e fácil de manter. O uso de HTML semântico é uma prática recomendada, pois facilita a interpretação do conteúdo tanto por humanos quanto por máquinas.

A criação do HTML e sua evolução para incorporar elementos semânticos refletem a necessidade de adaptar a web para ser mais intuitiva e acessível. Nos primeiros dias da internet, o foco principal era simplesmente exibir informações de maneira acessível, imitando o formato dos jornais impressos, com blocos de texto e links. À medida que a web se desenvolveu, a necessidade de criar páginas que fossem compreensíveis por diversos tipos de agentes, como motores de busca e leitores de tela, levou ao desenvolvimento de práticas semânticas, que melhoram tanto a experiência do usuário quanto a eficiência na indexação de conteúdo.

Em resumo, HTML é a espinha dorsal da web, estruturando e organizando o conteúdo das páginas de forma que possam ser apresentadas corretamente pelos navegadores. Embora não seja uma linguagem de programação, seu papel é crucial na criação de páginas acessíveis e bem organizadas. O advento do HTML semântico trouxe um nível adicional de clareza e acessibilidade, alinhando-se à evolução da internet, que busca cada vez mais oferecer uma experiência mais rica e inclusiva para todos os usuários.

Javascript

JavaScript é uma linguagem de programação amplamente utilizada para criar e controlar o comportamento dinâmico de páginas web. Ao contrário do HTML, que é responsável por estruturar o conteúdo da página, e do CSS, que define o estilo e a apresentação visual, o JavaScript permite que as páginas web sejam interativas e respondam a ações dos usuários, como cliques, toques ou entrada de dados. Com JavaScript, é possível realizar tarefas como validar formulários, criar animações, atualizar conteúdos em tempo real sem precisar recarregar a página, e até mesmo desenvolver aplicações complexas como jogos e sistemas web completos.

A relação entre JavaScript e HTML é complementar e fundamental para a criação de páginas web modernas. Enquanto o HTML define a estrutura e o conteúdo da página, o JavaScript interage com essa estrutura para manipular os elementos do HTML dinamicamente. Por exemplo, um script em JavaScript pode ser utilizado para alterar o texto de um parágrafo, modificar a cor de um botão ao passar o mouse por cima, ou até mesmo adicionar novos elementos ao HTML em resposta a uma ação do usuário. Essa interação é feita através do DOM (Document Object Model), uma interface que permite que o JavaScript acesse e manipule os elementos do HTML de forma programática.

JavaScript é uma das três tecnologias fundamentais para o desenvolvimento web, ao lado do HTML e do CSS. Enquanto o HTML fornece a estrutura e o CSS cuida do estilo, o JavaScript adiciona a camada de interatividade e comportamento dinâmico. Essa

combinação permite que os desenvolvedores criem experiências ricas e envolventes para os usuários, transformando simples páginas estáticas em aplicações web sofisticadas e responsivas. Além disso, com o avanço de tecnologias como Node.js, o JavaScript expandiu seu uso para fora dos navegadores, possibilitando o desenvolvimento de servidores, aplicações desktop e até mesmo programação de hardware.