Programowanie interaktywnych diagramów z wykorzystaniem GoJS

Od dłuższego czasu możemy zauważyć transformację tego, czym jest sieć WWW. Podczas gdy początkowo mieliśmy do czynienia jedynie z prostymi stronami tekstowymi, dziś potrafią być pełnoprawnymi aplikacjami. Programowanie ich, w szczególności graficznych i interaktywnych, nie jest wcale tak prostym zadaniem – jednak z pomocą przychodzi sporo bibliotek JavaScriptowych mających na celu to ułatwić. W niniejszym artykule chciałbym wziąć na tapet GoJS – bibliotekę stworzoną z myślą o interaktywnych wizualizacjach danych.

Wprowadzenie do GoJS

GoJS to rozbudowana biblioteka dla języka JavaScript do tworzenia interaktywnych diagramów oraz wizualizacji wykorzystujących HTMLowy Canvas, działających na różnych nowoczesnych platformach i przeglądarkach. Programista dostaje do dyspozycji pełen zestaw narzędzi do zapewnienia interaktywności, wśród których wymienić można by na przykład wsparcie dla drag'n'drop, obsługę schowka, szeroką gamę gotowych layoutów diagramów, palety, podglądy czy wsparcie dla cofania i ponawiania. Co warto podkreślić, biblioteka jest napisana bez żadnych dodatkowych zależności – ani od zewnętrznych bibliotek, ani od funkcjonalności serwerowych. Dzięki temu nie jesteśmy w żaden sposób ograniczani – możemy z użyciem biblioteki budować dowolny rodzaj aplikacji (frontendową, mobilną, desktopową), a tą wykorzystywać tylko do przeznaczonego jej celu.

Biblioteka jest jednak płatna, aczkolwiek możliwości, jakie oferuje, potrafią zaoszczędzić sporo czasu developerskiego w porównaniu do pracy na ogólnodostępnych darmowych bibliotekach wykorzystywanych w podobnych celach, takich jak D3 czy Fabric.js. Główna różnica to udostępnienie pełnej interaktywności i mnóstwa gotowych narzędzi, dzięki którym bardzo wiele różnego rodzaju aplikacji diagramowych (i nie tylko!) można stworzyć w bardzo krótkim czasie. Sprawia to, że dla programisty aplikacje służące przykładowo do prezentacji przebiegu procesów (w tym z danymi aktualizowanymi w czasie rzeczywistym), edycji UML, tworzenia map myśli, czy też projektowania układów elektronicznych stają się prostsze do zrobienia.

Tworzymy prostą aplikację

W artykule tym chciałbym przedstawić podstawowe możliwości biblioteki poprzez stworzenie bardzo prostej aplikacji (pod względem kodu) do tworzenia niezbyt skomplikowanych diagramów. Punktem wyjściowym dla nas w tym momencie powinien być plik HTML, gdzie zaimportujemy bibliotekę. Możemy ją pobrać (zarówno bezpośrednio ze strony, jak i z npm) bądź wykorzystać CDN. Kolejnym krokiem, jaki wykonujemy, jest dodanie elementu `div`, w którym diagram będzie się znajdować. Co ważne, nie tworzymy sami canvasu, robi to za nas biblioteka. Sam `div` natomiast możemy dowolnie ostylować, by pasował do naszej witryny. Nasz plik powinien wyglądać mniej więcej tak:

Listing 1. Początkowy wygląd pliku index.html

```
<!DOCTYPE html>
<html>
        <head>
                <script
src="https://cdnjs.cloudflare.com/ajax/libs/gojs/1.8.12/go.js"></script>
        </head>
        <body>
```

```
                <div id="diagram-content" style="height: 800px;
border: 1px solid black;"></div>
        </body>
</html>
```

Kolejnym krokiem będzie stworzenie właściwego kodu JavaScript tworzącego diagram. W tym celu utworzymy plik *diagram.js*, w którym zamieścimy całą logikę aplikacji. Możemy tworzyć go w dowolnej konwencji – GoJS nie narzuca nam żadnego konkretnego sposobu budowy źródeł. Osobiście na potrzeby tego przykładu utworzę IIFE (Immediately Invoked Function Expression) o nazwie `diagram`, z udostępnioną na zewnątrz funkcją `initDiagram`. Dzięki zastosowaniu IIFE, będziemy mogli część naszej implementacji ukryć przed dostępem z zewnątrz.

W GoJS podstawą, od której zaczynamy tworzenie wszelkich obiektów, jest funkcja `go.GraphObject.make`, która wywołuje metodę fabrykującą wybranej klasy biblioteki. Jest ona na tyle często wykorzystywana, że warto przypisać ją sobie do zmiennej o krótszej nazwie. Ogólnie przyjętą konwencją jest stosowanie do tego znaku dolara (warto uważać na to w przypadku korzystania równolegle z jQuery). Funkcja ta przyjmuje jako pierwszy argument klasę, której instancja ma zostać utworzona, a jako kolejne wartości poszczególnych pól obiektu. Warto zaznaczyć, że zwykle jako drugi argument jest podawany string z wartością jednego wybranego pola, zdefiniowanego w obiekcie – zwykle którego wartość jest najbardziej istotna w danym kontekście, np. rodzaj tworzonej figury.

Pierwsze, co chcemy utworzyć, to sam diagram. Znajduje się on pod klasą `go.Diagram`, a najważniejszym parametrem jest id elementu, w którym ma zostać umieszczony. Stąd też będziemy wywoływać `$(go.Diagram, 'diagram-content')`. Warto przejrzeć w dokumentacji, jakie pola posiada obiekt diagramu, aby lepiej sprecyzować jego działanie. Możemy na przykład przypisać polu `initialContentAlignment` wartość `go.Spot.Center`, dzięki czemu elementy diagramu na starcie aplikacji będą umieszczone na jego środku (domyślnie lewy górny róg). Zamieśćmy to w naszym `initDiagram`. Następnie pamiętajmy, aby wywołać tę funkcję w naszym pliku HTML. W rezultacie powinniśmy otrzymać następujące pliki JS i HTML:

Listing 2. Szkielet diagram.js i tworzenie pustego diagramu

```
diagram = (function(){
        var $ = go.GraphObject.make;
        var diagram;
        var initDiagram = function () {
                diagram = $(go.Diagram,
                        'diagram-content', {
                                initialContentAlignment: go.Spot.Center
                });
        }
        return { initDiagram };
})();
```

Listing 3. Wywołanie tworzenia diagramu

```
<!DOCTYPE html>
<html>
        <head>
                <script
src="https://cdnjs.cloudflare.com/ajax/libs/gojs/1.8.12/go.js"></script>
                <script src="diagram.js"></script>
        </head>
```

```
        <body>
                <div id="diagram-content" style="height: 800px;
border: 1px solid black;"></div>
                <script>
                        window.onload = function() {
                                diagram.initDiagram();
                        };
                </script>
        </body>
</html>
```

Na tę chwilę niestety nie zobaczymy jeszcze żadnego działania, gdyż nasz diagram jest pusty. Jedyne, po czym rozpoznamy, że GoJS działa, to blokada domyślnej akcji kliknięcia prawego przycisku myszy na diagramie.

Prezentacja danych na diagramie

Podstawą dla diagramu jest utworzenie modelu danych, na którym będzie on operować. Model składa się z dwóch tablic – `nodeDataArray` (dane węzłów) i `linkDataArray` (dane krawędzi). Węzły to główne elementy diagramu reprezentujące graficznie obiekty (np. encja na diagramie związków encji). Krawędzie (linki) to połączenia między węzłami (skierowane, bądź nie), które w zależności od typu diagramu mogą mieć różne znaczenie (np. relacja między encjami na diagramie związków encji).

Poszczególne węzły definiujemy jako obiekty, gdzie jedynym wymaganym polem jest `key`, czyli unikalny identyfikator. Oprócz tego możemy definiować opcjonalnie kategorie obiektów (`category`), a także uzupełnić dowolnymi własnymi danymi, które będą przechowywane wraz z węzłem na diagramie. Krawędzie definiujemy również jako obiekty, które mają wymagane dwa pola – `from` i `to`, których wartości określają, jakie węzły są łączone. Podobnie jak w przypadku węzłów, możemy dopisać także własne dodatkowe dane.

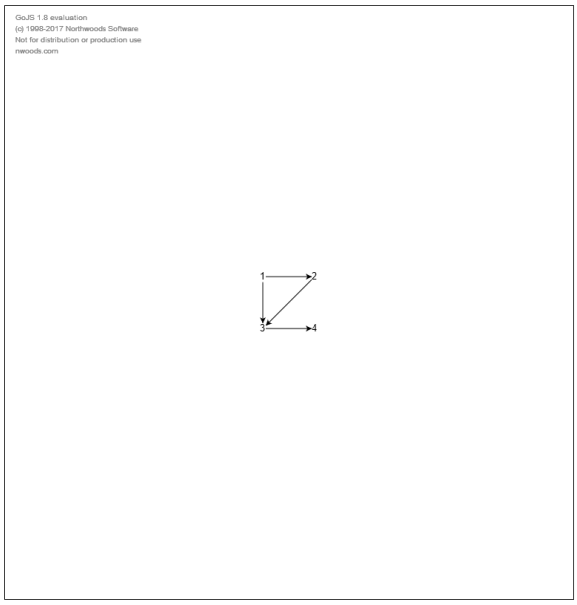
Model diagramu znajduje się pod polem `model` obiektu diagramu. Najlepiej zdefiniować go, tworząc obiekt typu `go.GraphLinksModel`, do którego przekazujemy węzły i krawędzie. Przykładowa definicja modelu może wyglądać następująco:

Listing 4. Definicja modelu diagramu

```
diagram.model = $(go.GraphLinksModel, {
        nodeDataArray: [
                {key: 1, category: 'first'},
                {key: 2, category: 'second'},
                {key: 3, category: 'second'},
                {key: 4, category: 'third'}
        ],
        linkDataArray: [
                {from: 1, to: 2},
                {from: 2, to: 3},
                {from: 1, to: 3},
                {from: 3, to: 4}
        ]
});
```

Po zamieszczeniu jej w naszym `initDiagram` powinniśmy otrzymać po uruchomieniu bardzo prosty diagram, gdzie zobaczymy wartości wpisane w polach `key` połączone między sobą krawędziami.

Rysunek 1. Nasz pierwszy diagram



Kolejnym krokiem, jaki powinniśmy podjąć, jest nadanie naszym obiektom wyglądu. Wygląd tworzymy poprzez utworzenie obiektu typu `go.Node`, w którym następnie tworzymy kolejne obiekty definiujące kształt (`go.Shape`). Najprostsza definicja wyglądu obiektu (template) wyglądałaby więc tak:

Listing 5. [Javascript] Najprostsza definicja wyglądu obiektu

```
diagram.nodeTemplate = $(go.Node, 'Auto', $(go.Shape, 'Circle'));
```

W tym przypadku utworzymy czarne koło, które będzie reprezentować obiekt. `Auto` oznacza sposób rozmieszczania poszczególnych kształtów. W przypadku `Auto` każdy kształt, jaki byśmy zdefiniowali wewnątrz `go.Node`, będzie układany warstwowo na środku obiektu. W przypadku kształtów wybrany został tutaj `Circle`, aczkolwiek jest udostępnionych wiele różnych, które możemy znaleźć pod adresem: https://gojs.net/latest/samples/shapes.html. Moglibyśmy także w obiekcie kształtu ustawić wartości pól – moglibyśmy wówczas określić takie rzeczy jak wypełnienie, kontur czy rozmiar. Dodatkowo wartości te możemy pobierać z modelu danych – wykorzystujemy wówczas do tego klasy `go.Binding`.

Inny sposób definicji wyglądu to stworzenie kształtu z wykorzystaniem definicji ścieżek SVG. Wtedy taką ścieżkę zamieszczamy jako wartość pola `geometryString`. Przykładowa definicja template wyglądałaby wtedy następująco (dodatkowo ustawimy rozmiar i kolor wypełnienia):

Listing 6. Definicja wyglądu wykorzystująca ścieżkę SVG

```
diagram.nodeTemplate = $(go.Node,
        'Auto',
        $(go.Shape, {
                geometryString:
'F M0 0 L100 0 Q150 50 100 100 L0 100 Q50 50 0 0z',
                fill: 'white',
                width: 100,
                height: 100
        }));
```

Możemy także definiować wygląd za pomocą obrazków w dowolnym formacie (w tym SVG). Wówczas, zamiast z `go.Shape`, korzystamy z `go.Picture`. Co ciekawe, `go.Picture` może

przechowywać nie tylko obrazki, ale także inne elementy typu canvas, dzięki czemu możemy tworzyć jeszcze bardziej zaawansowane wizualizacje.

Kolejnym sposobem na definicję wyglądu obiektów jest składanie wielu różnych kształtów w jeden. Zaletą takiego podejścia jest to, że po podzieleniu naszego obiektu na kilka mniejszych, możemy definiować różne zachowania dla każdej z części (np. jeden z kształtów może być przyciskiem, na którym zdefiniujemy akcję kliknięcia - tylko ten fragment wówczas będzie miał przypisane to zdarzenie) W celu tworzenia złożonych obiektów warto zapoznać się ze sposobami, jak mogą być rozmieszczane kształty na obiekcie. Do tej pory wykorzystywaliśmy `Auto`, czyli umieszczanie wszystkiego na środku. Oprócz niego mamy do dyspozycji także inne, z czego najbardziej podstawowe to: `Horizontal` (od lewej do prawej), `Vertical` (z góry na dół), `Spot` (umieszczanie na odgórnie określonych pozycjach jak Center, Top, Bottom itd.). Kolejne kształty przekazujemy jako kolejne argumenty funkcji tworzącej `Node` i w takiej też kolejności są umieszczane. Na przykład, dwa prostokąty, każdy o innym wyglądzie, położone jeden na drugim, byłyby zdefiniowane w następujący sposób:

Listing 7. Definicja wyglądu składającego się z kilku kształtów

```
diagram.nodeTemplate = $(go.Node,
        'Vertical',
        $(go.Shape,
                'Rectangle', {
                        width: 50,
                        height: 50,
                        strokeWidth: 0,
                        fill: 'yellow',
                }),
        $(go.Shape,
                'Rectangle', {
                        width: 100,
                        height: 50,
                        strokeWidth: 5,
                        stroke: 'red',
                        fill: 'green'
                }));
```

Co jednak, jeżeli chcielibyśmy różne rodzaje obiektów? W tym celu zamiast przypisywać wygląd do `nodeTemplate`, uzupełniamy mapę `nodeTemplateMap`, gdzie kluczem jest kategoria obiektu (category, które definiowaliśmy w modelu), a wartością template. Na potrzeby tego przykładu załóżmy, że wykorzystamy wyżej opisane definicje dla naszych kategorii: `first` będzie kołem, `second` kształtem SVG, `third` dwoma prostokątami. Ze względów praktycznych warto jednak nie przypisywać template bezpośrednio do mapy, tylko stworzyć funkcję zwracającą, jaki template przypiszemy do jakiej kategorii, i następnie uzupełniać danymi z niej mapę `nodeTemplateMap`. Powód tego wyjaśnię w dalszej części artykułu. Warto zwrócić uwagę, że mapę tą należy uzupełnić, zanim zasilimy diagram modelem. Na przykład mogłoby to wyglądać następująco (ze względu na czytelność pominięto niektóre fragmenty kodu):

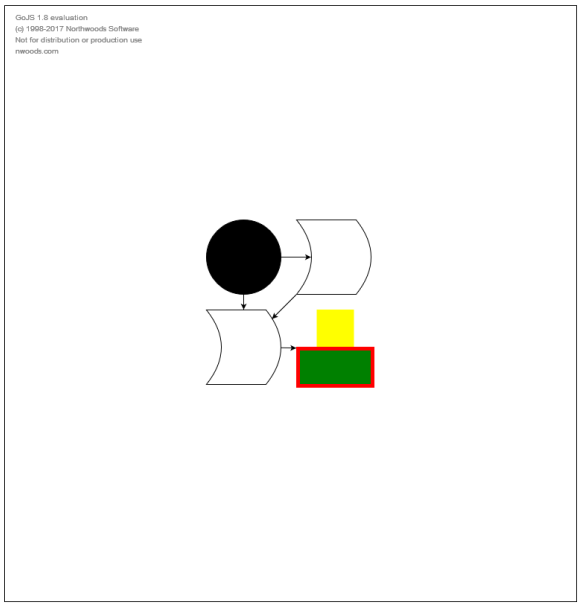
Listing 8. Propozycja sposobu korzystania z nodeTemplateMap

```
var getTemplates = function () {
        return [{
                        category: 'first',
                        template: $(go.Node, 'Auto', $(go.Shape, 'Circle'))
                }, {
```

```
                        category: 'second',
                        template: $(go.Node,
                                'Auto',
                                $(go.Shape, {
                                        // ...
                                }))
                }, {
                        category: 'third',
                        template: $(go.Node,
                                'Vertical',
                                // ...
                                }))
                }];
}
var initDiagram = function () {
        // ...
        getTemplates().forEach(x =>
diagram.nodeTemplateMap.add(x.category, x.template));
        diagram.model = $(go.GraphLinksModel, {
                // ...
        });
}
```

Po napisaniu tego powinniśmy uzyskać następujący efekt:

Rysunek 2. Diagram z trzema różnymi template dla węzłów



Jak widać, nasz diagram zaczyna nabierać kształtów. Dodatkowo warto wspomnieć, że oprócz wyglądu węzłów można także ustalać wygląd krawędzi. Definiuje się go analogicznie, z tą różnicą, że zamiast `nodeTemplate` czy `nodeTemplateMap` mamy `linkTemplate` i `linkTemplateMap`, a przy definiowaniu template zamiast tworzyć obiekt `go.Node`, tworzymy obiekt `go.Link`.

Ustawianie i edycja krawędzi

Nasz diagram zaczyna wyglądać coraz lepiej, jednak można zauważyć, że po pierwsze, brakuje mu możliwości edycji (jedyne, co można, to przesuwać węzły i je klonować z użyciem klawisza CTRL), a po drugie – moglibyśmy chcieć, aby krawędzie rysowały się między konkretnymi punktami naszych węzłów. Szczególnie widać to na obiektach, które powstały ze ścieżek SVG – najbardziej pożądane

byłoby ustawienie tutaj wejścia z lewej strony i wyjścia z prawej. Aby to osiągnąć, musimy zdefiniować porty.

Porty są częściami węzłów, które służą do niczego innego jak ustawiania krawędzi względem reszty obiektu. Mogą nimi być dowolne już istniejące elementy węzła (nawet cały węzeł), jak i specjalnie do tego celu utworzone nowe elementy. Definiuje się je poprzez ustawienie wartości pola `portId`. Oprócz tego możemy m.in. także definiować, czy możemy z portu tworzyć nowe linki (pole `fromLinkable`) oraz czy możemy do niego dołączać nowe linki (`toLinkable`). Na potrzeby przykładu porty będą definiowane tutaj jako widoczne kształty, aczkolwiek mogą nimi być dowolne obiekty (w tym niewidzialne).

W przypadku pierwszej kategorii chcielibyśmy port umieścić w samym środku. Aby było widać, że krawędzie tam trafiają, zmienimy przy okazji kolor koła, aby było białe, oraz ustalimy mu stały rozmiar. Od razu ustawmy też, żeby można było tworzyć z tego portu połączenia w obie strony. Definicja kształtu będzie wówczas wyglądać następująco:

Listing 9. Definicja kształtu rozbudowana o port

```
$(go.Node, 'Auto',
        $(go.Shape,
                'Circle', {
                        width: 100,
                        height: 100,
                        fill: 'white'
        }),
        $(go.Shape,
                'Rectangle', {
                        portId: '',
                        width: 20,
                        height: 20,
                        fill: 'black',
                        fromLinkable: true,
                        toLinkable: true
        }))
```

Jak wspomniałem wcześniej, istotne jest, aby pole `portId` miało ustawioną wartość, jednak nie zabrania nam to użycia pustego stringa. W przypadku nadania pustego stringa port ten będzie portem domyślnym.

Dla drugiej kategorii, chcielibyśmy zdefiniować port po jej lewej i prawej stronie. Możemy to osiągnąć na dwa sposoby: zmieniając rozmieszczanie elementów na `Horizontal` i ustawiając kształty w kolejności port-główny kształt-port lub zmieniając na `Spot` i wówczas ustawiamy, że porty mamy w pozycjach `Left` i `Right`, a główny kształt w `Center`. Bardziej elastyczny jest ten drugi sposób i też z niego skorzystamy. Z lewej strony umieśćmy taki sam prostokąt jak na pierwszym kształcie, ustawiając, że można jedynie linkować do niego, natomiast z prawej taki, że możemy jedynie wyprowadzać krawędzie z niego. Dla rozróżnienia ich powinniśmy nadać im inne identyfikatory niż pusty string (np. `entry` i `exit`). Taka definicja wyglądałaby wówczas następująco:

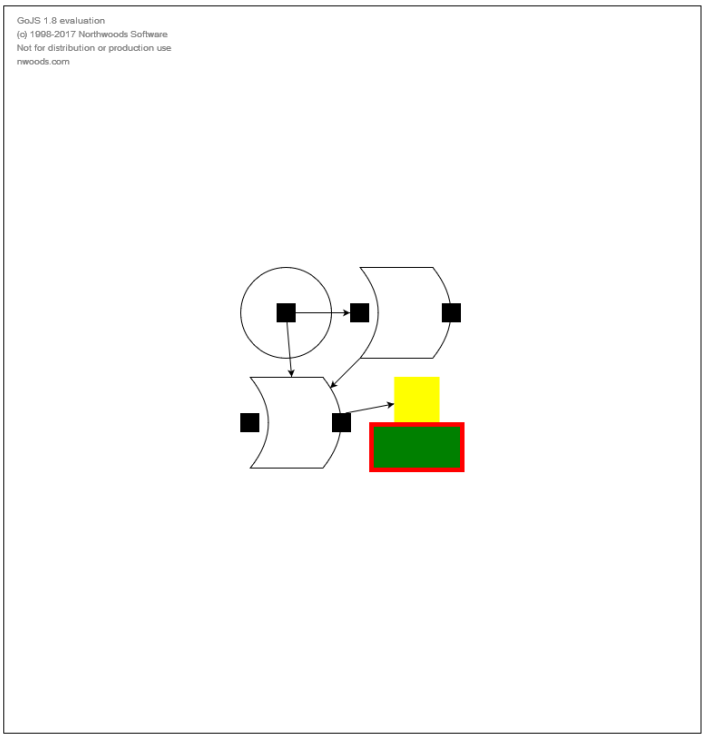
Listing 10. Definicja kształtu rozbudowana o dwa porty

```
$(go.Node,
        'Spot',
        $(go.Shape, {
                // ...
        }),
        $(go.Shape,
                'Rectangle', {
```

```
                    portId: 'entry',
                    width: 20,
                    height: 20,
                    fill: 'black',
                    alignment: go.Spot.Left,
                    fromLinkable: false,
                    toLinkable: true
        }),
        $(go.Shape,
                'Rectangle', {
                portId: 'exit',
                width: 20,
                height: 20,
                fill: 'black',
                alignment: go.Spot.Right,
                fromLinkable: true,
                toLinkable: false
        }))
```

Dla trzeciej kategorii jedynie zmodyfikujmy kwadrat na górze, aby stał się on portem, poprzez dopisanie `portId` z pustym stringiem oraz `fromLinkable` i `toLinkable` ustawionych na `true`. Po stworzeniu portów diagram powinien prezentować się następująco:

Rysunek 3. Diagram ze wstępną konfiguracją portów



Jak widać na powyższym rysunku, w przypadku pierwszej i trzeciej kategorii porty działają doskonale, natomiast nie działają dla drugiej. Podobnie jest z rysowaniem linków. Bez problemu działa rysowanie między pierwszą i trzecią kategorią, natomiast w drugiej krawędzie wyskakują z portów. Powodem są dwie kwestie. Po pierwsze, w modelu linków nie zdefiniowaliśmy, między jakimi portami mają zachodzić połączenia. Po drugie, nie zostało w modelu ustawione, jak przechowywać połączenia między portami. Przenieśmy się więc z powrotem do `initDiagram` i definicji modelu. Do pól `GraphLinksModel` musimy dopisać dwa pola: `linkFromPortIdProperty` oraz `linkToPortIdProperty`, którymi ustalamy, jak w modelu definiujemy wykorzystywane porty.

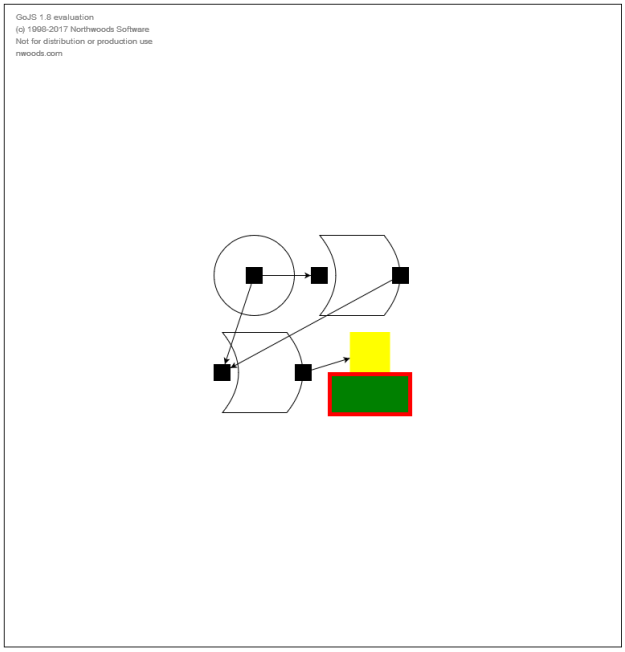
Dodatkowo w `linkDataArray` powinniśmy zdefiniować, jak prowadzone będą krawędzie dla węzłów 2 i 3 (czyli tych, co należą do drugiej kategorii). Zmiany te powinny wyglądać następująco:

Listing 11. Zmiany w modelu umożliwiające wykorzystanie portów

```
diagram.model = $(go.GraphLinksModel, {
        linkFromPortIdProperty: 'fromPort',
        linkToPortIdProperty: 'toPort',
        nodeDataArray: [
                // ...
        ],
        linkDataArray: [
                {from: 1, to: 2, toPort: 'entry'},
                {from: 2, to: 3, fromPort: 'exit', toPort: 'entry'},
                {from: 1, to: 3, toPort: 'entry'},
                {from: 3, to: 4, fromPort: 'exit'}
        ]
});
```

Teraz krawędzie na diagramie wyglądają tak jak powinny:

Rysunek 4. Diagram po prawidłowym skonfigurowaniu portów



Paleta i końcowe szlify

Aby nasz diagram był w pełni interaktywny, brakuje nam jeszcze możliwości wstawiania elementów. Do tego celu utworzymy wbudowaną w GoJS paletę. Paleta to oddzielny canvas, który działa w taki sposób, że wyświetla wszystkie zdefiniowane w niej węzły i umożliwia przeniesienie ich na główny diagram. Konfigurujemy ją analogicznie do diagramu, jednak poprzez utworzenie obiektu typu `go.Palette`. Aby jednak to zrobić, najpierw powinniśmy dodać do pliku HTML dodatkowy div, w którym paleta zostanie umieszczona. Do tego możemy dopisać w pliku JS funkcję do inicjalizacji palety, którą również udostępnimy na zewnątrz (`initPalette`). W naszym przykładzie umieścimy paletę obok diagramu, więc plik HTML możemy zedytować w następujący sposób:

Listing 12. Dodanie palety do pliku HTML

```
<div id="palette-content" style="height: 800px; width: 200px;
display: inline-block; border: 1px solid black"></div>
```

```
<div id="diagram-content" style="height: 800px; width: calc(100% - 220px);
display: inline-block; border: 1px solid black;"></div>
<script>
        window.onload = function() {
                diagram.initDiagram();
                diagram.initPalette();
        };
</script>
```

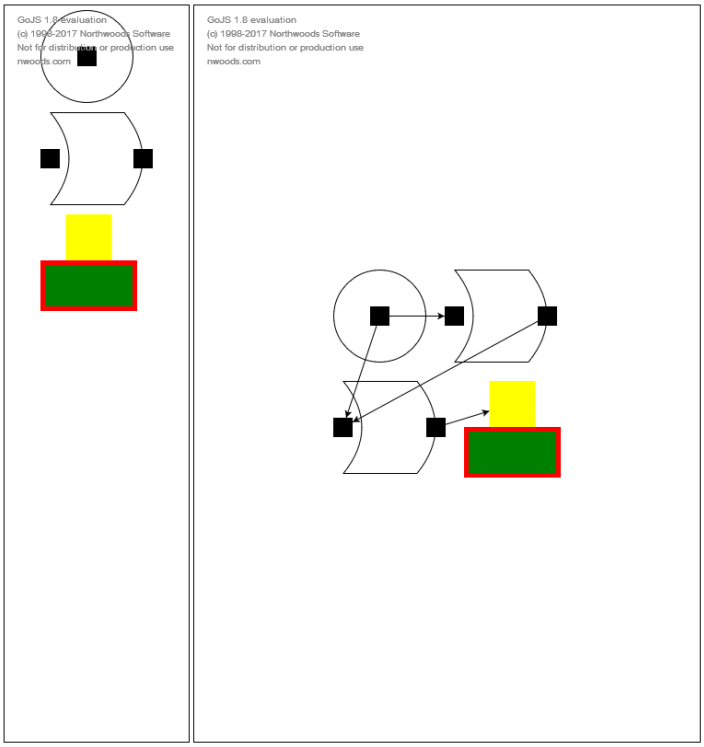
Kolejnym krokiem będzie uzupełnienie utworzonej przez nas funkcji `initPalette`. Jak wspomniałem wcześniej, inicjalizacja palety niewiele się różni od inicjalizacji diagramu. Właśnie w tym momencie wykorzystamy po raz drugi naszą funkcję `getTemplates()` – dzięki temu nie musimy definiować wyglądu kształtów po raz kolejny, oddzielnie dla palety. Jednak warto zaznaczyć, że możemy utworzyć inny – przy przenoszeniu na diagram znaczenie ma kategoria elementu, a nie jej wygląd. Dodatkowo, aby umożliwić przenoszenie z palety na diagram, musimy do inicjalizacji diagramu dopisać pole `allowDrop` i ustawić mu wartość `true`. Kod powinien prezentować się następująco:

Listing 13. Inicjalizacja palety

```
var initDiagram = function () {
    diagram = $(go.Diagram,
        'diagram-content', {
            initialContentAlignment: go.Spot.Center,
            allowDrop: true
    });
    // ...
}
var initPalette = function () {
    palette = $(go.Palette, 'palette-content');
    getTemplates().forEach(x => palette.nodeTemplateMap.add(x.category,
x.template));
    palette.model.nodeDataArray = [
        {key: 1, category: 'first'},
        {key: 2, category: 'second'},
        {key: 3, category: 'third'}
    ]
}
```

A aplikacja będzie prezentować się teraz tak:

Rysunek 5. Diagram z dodaną paletą



Dodatkowo na sam koniec, aby nieco poprawić wygląd aplikacji, dodajmy jeszcze dodatkowo dwie rzeczy: siatkę, do której będą przylegać elementy, oraz ich autopozycjonowanie.

Zacznijmy od siatki. Tą znajdziemy pod `diagram.grid` i jedyne, co musimy zrobić, to zmienić jej właściwość `visible` na `true`. Natomiast aby dodać przyciąganie, należy je ustawić we wbudowanym narzędziu przeciągania dostępnym pod `diagram.toolManager.draggingTool`. Możemy tu wykorzystać dwie właściwości: `isGridSnapEnabled` do włączenia przyciągania oraz `gridSnapCellSize` do zdefiniowania wielkości siatki (dla naszego przykładu zdefiniujmy rozmiar 50x50). Jeżeli korzystalibyśmy także z modyfikacji rozmiaru węzłów (`resizingTool`), to ustawilibyśmy także w nim analogiczne pola. Kod odpowiedzialny za ustawienie siatki powinien wyglądać następująco:

Listing 14. Ustawienie siatki na diagramie

```
diagram.grid.visible = true;
diagram.toolManager.draggingTool.isGridSnapEnabled = true;
diagram.toolManager.draggingTool.gridSnapCellSize = new go.Size(50, 50);
```

Pozycjonowanie elementów najłatwiej osiągnąć poprzez zmianę layoutu diagramu. Domyślny layout nie definiuje w żaden sposób automatycznego rozmieszczania elementów (jedynie układa je tak, aby nie nachodziły na siebie). Oferują je natomiast inne: `GridLayout` (układnie na siatce), `TreeLayout` (drzewo), `ForceDirectedLayout` (grafy z odpychaniem od siebie elementów), `LayeredDigraphLayout` (graf rozmieszczony w kolumny i wiersze) oraz `CircularLayout` (rozmieszczenie węzłów na okręgu). Layout ustawiamy jako kolejne pole przy inicjalizacji diagramu – `layout`, któremu przekazujemy nową instancję wybranego layoutu wraz z jego ewentualną konfiguracją. Na przykład możemy to zrealizować następująco dla `LayeredDigraphLayout`:

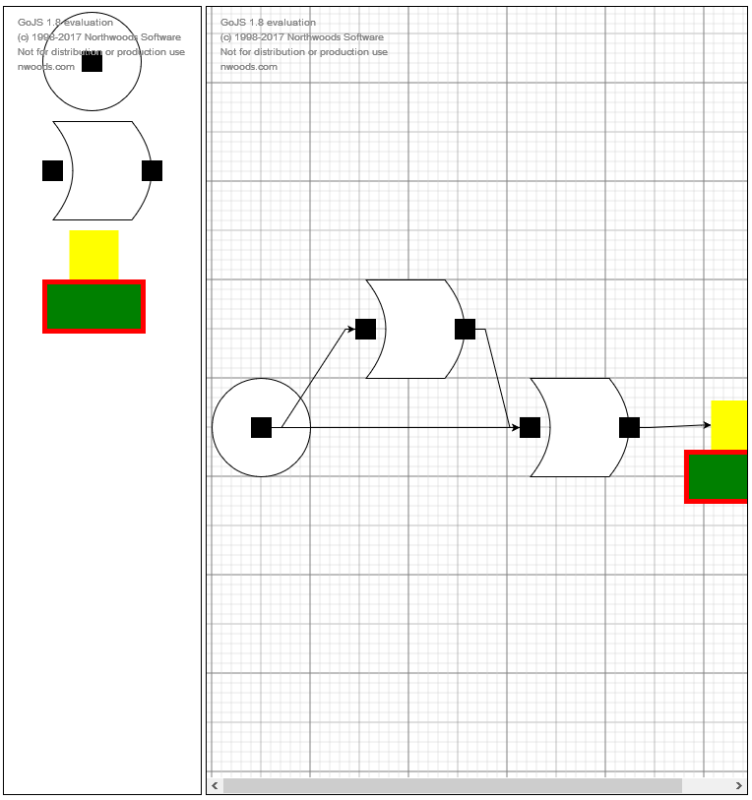
Listing 15. Ustawienie layoutu Layered Digraph dla diagramu

```
diagram = $(go.Diagram,
    'diagram-content', {
```

```
        initialContentAlignment: go.Spot.Center,
        allowDrop: true,
        layout: $(go.LayeredDigraphLayout)
});
```

Efekt jest następujący:

Rysunek 6. Diagram z włączoną siatką i ustawionym LayeredDigraphLayout



Można także posprawdzać inne dostępne layouty, jednak należy pamiętać o ich ograniczeniach. Na przykład `TreeLayout` nie poradzi sobie z naszą sytuacją, gdzie do jednego węzła prowadzą dwie krawędzie.

Podsumowanie

Jak można było się przekonać w trakcie lektury tego artykułu, bardzo niewielkim nakładem pracy uzyskaliśmy dość rozbudowaną aplikację, korzystając jedynie z udostępnionej funkcjonalności biblioteki GoJS, bez dodatkowego hackowania. Całość kodu JavaScript zamknęła się w około 120 liniach kodu, co tym bardziej pokazuje, że nie robiliśmy nic złożonego. GoJS pozwala nam zaoszczędzić sporo czasu, który zapewne poświęcilibyśmy na stworzenie chociażby palety, obsługi elementów na canvasie czy też nietrywialnych rzeczy jak automatyczne pozycjonowanie elementów. Plusami tej biblioteki są także przejrzysta dokumentacja, mnóstwo gotowych przykładów oraz bardzo dobry support producenta. Zwolennicy TypeScript także będą zadowoleni, ponieważ są udostępnione oficjalne typingi, a twórcy także udostępniają za darmo rozszerzenia biblioteki napisane właśnie w TypeScript.

Jednak w trakcie tego krótkiego artykułu nie sposób było przedstawić wszystkich możliwości biblioteki, co warto zrobić na własną rękę poprzez zapoznanie się z przykładami użycia dostępnymi na stronie producenta.