Spis treści

1 HTML 5

W tym rozdziale omówimy sobie wybrane nowe znaczniki, które zostały wprowadzone w HTML 5.

1.1 picture

Znacznik ten zawiera w sobie kilka znaczników *source* oraz jeden znacznik *img* (wymagany). Znaczniki *source* określają różne źródła obrazków dla różnych wielkości ekranów, a znacznik *img* określa defaultowy obrazek.

Dzięki takiemu podejściu możliwa jest łatwa zmiana obrazka w zależności od wielkości ekranu urządzenia.

```
<picture>
  <source media="(min-width: 650px)" srcset="img_pink_flowers.jpg">
  <source media="(min-width: 465px)" srcset="img_white_flower.jpg">
  <img src="img_orange_flowers.jpg" alt="Flowers" style="width:auto;">
</picture>
```

Oprócz tego znacznik *picture* może być używany w sytuacjach gdy:

- gdy mamy ograniczoną przepustowość sieci - nie ma potrzeby ładowania wtedy dużego obrazka, przeglądarka użyje pierwszego obrazka ze znacznika *source*
- wspierane formaty obrazków - nie wszystkie przeglądarki wspierają wszystkie formaty obrazków, przeglądarka użyje pierwszego obrazka z elementu *source* którego format rozpoznaje

1.2 video

Znacznik ten jest standardową metodą zawierania elementów video na stronie. Przed wprowadzeniem tego znacznika aby móc dodawać video do strony konieczne było korzystanie z dodatkowych wtyczek (np. flash).

```
<video width="320" height="240" controls>
  <source src="movie.mp4" type="video/mp4">
  <source src="movie.ogg" type="video/ogg">
Your browser does not support the video tag.
</video>
```

Atrybut *controls* dodaje do naszego video możliwość zatrzymywania, pauzy, wznowienia, regulacji głośności itd.

Znaczniki *source* służą do określania źródła video, najczęściej określa się kilka, przeglądarka wybiera pierwszy format, który rozpoznaje.

1.3 audio

Znacznik ten jest standardową metodą zawierania plików dźwiękowych na stronie. Przed wprowadzeniem tego znacznika aby móc dodawać audio do strony konieczne było (podobnie jak dla video) korzystanie z dodatkowych wtyczek (np. flash).

```
<audio controls>
  <source src="horse.ogg" type="audio/ogg">
  <source src="horse.mp3" type="audio/mpeg">
Your browser does not support the audio element.
</audio>
```

Atrybut *cotrols* dodaje do naszego audio możliwość zatrzymywania, pauzy, wznowienia, regulacji głośności itd.

Znaczniki *source* służą do określania źródła audio, najczęściej określa się kilka, przeglądarka wybiera pierwszy format, który rozpoznaje.

1.4 Znaczniki semantyczne (z przypisanym znaczeniem)

Przed wprowadzeniem html5 znaczniki były pozbawione semantyki, oznacza to, że nie można było powiedzieć nic o zawartości strony/elementu na podstawie znacznika.

Wprowadzane nowych elementów semantycznych w html5 pozwoliło silnikom wyszukiwania na lepszą identyfikację zawartości strony, a developerom na łatwiejszą organizację kodu.

Przykładowe znaczniki semantyczne wprowadzone w html5:

- *nav*
- *section*
- *article*
- *aside*
- *header*
- *footer*
- *main*

1.4.1 nav

Służy do umieszczania w nim najważniejszych linków nawigacyjnych na stronie.

```
<nav>
  <a href="/html/">HTML</a> |
  <a href="/css/">CSS</a> |
  <a href="/js/">JavaScript</a> |
  <a href="/jquery/">jQuery</a>
</nav>
```

1.4.2 section

Definiuje nową sekcję w dokumencie, czyli tematycznie spójną treść, najczęściej z nagłówkiem.

```
<section>
  <h1>WWF</h1>
  <p>The World Wide Fund for Nature (WWF) is....</p>
</section>
```

1.4.3 article

Definiuje niezależną treść na stronie, która może być czytana niezależnie od reszty strony, przykładowo post na blogu, artykuł w gazecie.

```
<article>
  <h1>What Does WWF Do?</h1>
  <p>WWF's mission is to stop the degradation of our planet's natural environment,
  and build a future in which humans live in harmony with nature.</p>
</article>
```

1.4.4 aside

Definiuje treść, która jest umieszczona obok treści w której zawarty jest znacznik. Przykładowo boczny pasek nawigacyjny.

```
<p>My family and I visited The Epcot center this summer.</p>

<aside>
  <h4>Epcot Center</h4>
  <p>The Epcot Center is a theme park in Disney World, Florida.</p>
</aside>
```

1.4.5 header

Określa nagłówek dla dokumentu lub sekcji, powinien być używany jako kontener na treść wprowadzającą dla użytkownika. Możemy mieć kilka znaczników tego typu w jednym dokumencie.

```
<article>
  <header>
    <h1>What Does WWF Do?</h1>
    <p>WWF's mission:</p>
  </header>
  <p>WWF's mission is to stop the degradation of our planet's natural environment,
  and build a future in which humans live in harmony with nature.</p>
</article>
```

1.4.6 footer

Definiuje treść, która znajduje się na dole strony lub sekcji. Znacznik ten powinien zawierać treść dotyczącą treści otaczającego go znacznika.

Informacje zawierane w tym znaczniku to przykładowo autor dokumentu, informacje o prawie autorskim, linki do warunków użytkowania, dane kontaktowe itp.

Możemy mieć wiele znaczników *footer* w naszym dokumencie.

```
<footer>
  <p>Posted by: Hege Refsnes</p>
  <p>Contact information: <a href="mailto:someone@example.com">
  someone@example.com</a>.</p>
</footer>
```

1.4.7 main

Definiuje główną treść dokumentu. Treść wewnątrz tego znacznika powinna być unikalna dla całej strony, nie powinna się powtarzać gdziekolwiek na stronie.

W dokumencie nie może być więcej niż jeden znacznik *main*.

Znacznik ten nie może się zawierać w innych elementach semantycznych, jak *article*, *header*, *footer*, *nav* itd.

1.5 canvas

To znacznik, który umożliwia rysowanie grafiki na stronie w trakcie wyświetlania strony użytkownikowi. Zazwyczaj do tego celu używa się JavaScript. Znacznik *canvas* stanowi tylko kontener dla grafiki, aby narysować rzeczywistą grafikę korzystamy z JavaScript.

```
<canvas id="myCanvas"></canvas>

<script>
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
ctx.fillStyle = "#FF0000";
ctx.fillRect(0, 0, 80, 80);
</script>
```

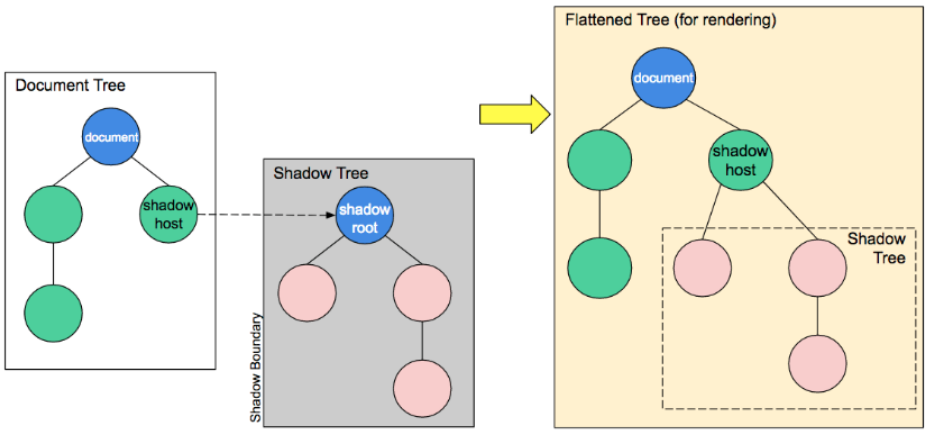
Egzamin

Nowe znaczniki i opisać szczegółowo dwa z nich.

2 Shadow DOM

Shadow DOM to API, które pozwala na realizację enkapsulacji fragmentów kodu HTML w znacznikach HTML. Takie podejście zapewnia nam, że nasz kod HTML nie będzie kolidował z innymi fragmentami kodu, a także nasz kod jest utrzymywany w czystości.

Idea tego API jest taka, że tworzymy sobie poddrzewo DOM, które zawiera się w pewnym znaczniku HTML.



Przykładem takiego podejścia jest znacznik *video* wprowadzony w html5, to co widzimy w kodzie to:

```
<video width="400" controls>
  <source src="mov_bbb.mp4" type="video/mp4">
  <source src="mov_bbb.ogg" type="video/ogg">
  Your browser does not support HTML5 video.
</video>
```

To co widzimy w przeglądarce to:

Skąd się wzięły tutaj wszystkie przyciski do kontroli wideo ? To właśnie zasługa Shadow DOM, fragment kodu HTML odpowiedzialny za przyciski i ich style został zawarty w znaczniku *video*, jedyne co zrobiliśmy to dodanie atrybutu *controls*.

Shadow Root możemy dodać do jakiegokolwiek elementu w bardzo prosty sposób:

```
let shadow = elementRef.attachShadow({mode: 'open'});
\\ lub
let shadow = elementRef.attachShadow({mode: 'closed'});
```

Powyższa funkcja dodaje do elementu *elementRef* korzeń naszego drzewa shadow DOM. *mode* określa czy mamy dostęp do manipulowania tym drzewem z zewnątrz (z kontekstu całej strony) czy tylko z

wewnątrz.

Przykład użycia:

```
let shadow = this.attachShadow({mode: 'open'});

var para = document.createElement('p');
shadow.appendChild(para);
// etc.
```

Możemy też utworzyć nasz własny element html, który będzie zawierał w sobie shadow DOM:

```
class PopUpInfo extends HTMLElement {
  constructor() {
    // Always call super first in constructor
    super();

    // write element functionality in here

    ...

    var shadow = this.attachShadow({mode: 'open'});

    // Create spans
  var wrapper = document.createElement('span');
  wrapper.setAttribute('class','wrapper');
  var icon = document.createElement('span');
  icon.setAttribute('class','icon');
  icon.setAttribute('tabindex', 0);
  var info = document.createElement('span');
  info.setAttribute('class','info');

  // Take attribute content and put it inside the info span
  var text = this.getAttribute('text');
  info.textContent = text;

  // attach the created elements to the shadow dom
  shadow.appendChild(wrapper);
  wrapper.appendChild(icon);
  wrapper.appendChild(info);

  }
}
```

```
// Define the new element
customElements.define('popup-info', PopUpInfo);
```

```
<popup-info img="img/alt.png" text="Your card validation code (CVC) is an extra
                                    security feature - it is the last 3 or 4
                                    numbers on the back of your card.">
```

Na podstawie `https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM`.

Egzamin

Shadow DOM

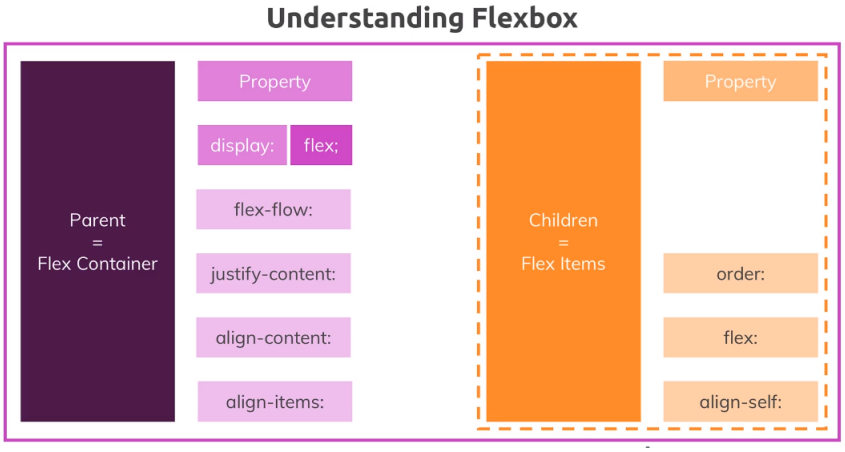
3 CSS

W tym rozdziale omówimy sobie tylko zaawansowane metody stosowane w CSS do pozycjonowania elementów na stronie.

3.1 Flexbox

flex to nowa wartość dla atrybutu display (inne to *inline*, *block*, *inline-block*, a także *grid* o którym później). Przez przypisanie tej własności nasz kontener staje się kontenerem flex, który używamy do pozycjonowania elementów.

Flexbox opiera się na dwóch rodzajach elementów - flex container i flex item. Do każdego z nich możemy przypisywać atrybuty, które opisują jak elementy mają być położone.



Dla zupełności na powyższym rysunku powinny się znajdować jeszcze dwie dodatkowe własności dla flex container oraz trzy dodatkowe dla flex item:

- dla flex container - *flex-direction* oraz *flex-wrap*, ale *flex-flow* jest skrótem dla nich dwóch, ustawiając *flex-flow* ustawiamy zarówno *flex-direction* jak i *flex-wrap*
- dla flex item - *flex-grow*, *flex-shrink*, *flex-basis*, ale własność *flex* jest skrótem dla ustawienia każdej z nich

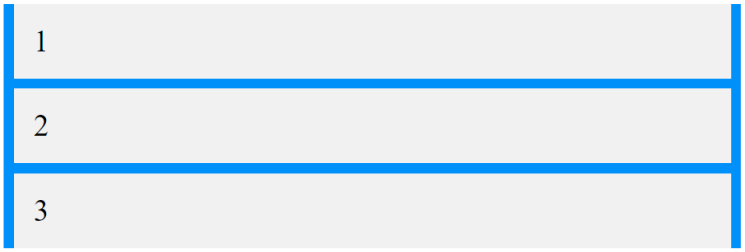
Samo utworzenie kontenera flex, a więc ustawienie display na wartość *flex* powoduje, że elementy tego kontenera będą wyświetlane inaczej niż zwykle:

```
<style>
.flex-container {
  display: flex;
  background-color: DodgerBlue;
}

.flex-container > div {
  background-color: #f1f1f1;
  margin: 10px;
  padding: 20px;
  font-size: 30px;
}
</style>

<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
```

Dla porównania bez ustawionego display *flex* mielibyśmy taki layout:



Dzieje się tak dlatego, że ustawienie display flex powoduje, że elementy w flex container są stackowane jeden obok drugiego.

3.2 Własności flex container

3.2.1 flex-direction

To własność, która określa w którym kierunku nasze elementy mają być stackowane, defaultowa wartość to row, nasze elementy są ustawiane w wierszu jak widzieliśmy na rysunku powyżej. Inne możliwe wartości to:

- *row*
- *row-reverse*
- *column*
- *column-reverse*

Przykład

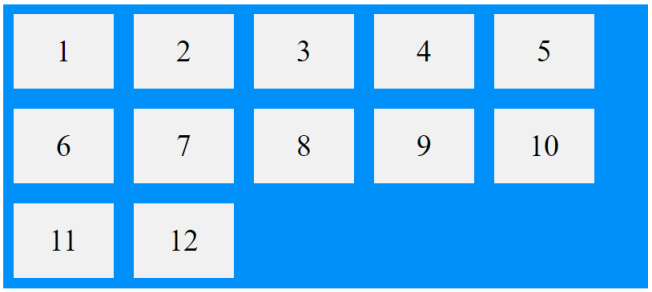
```
.flex-container {
  display: flex;
  flex-direction: column;
}
```

3.2.2 flex-wrap

Własność *flex-wrap* określa czy elementy powinny być przenoszone do następnej linii gdy brakuje dla nich miejsca w obecnej. Możliwe wartości to:

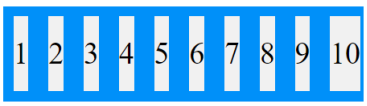
- *wrap*

```
.flex-container {
  display: flex;
  flex-wrap: wrap;
  background-color: DodgerBlue;
}
```



- *nowrap*

```
.flex-container {
  display: flex;
  flex-wrap: nowrap;
  background-color: DodgerBlue;
}
```



- *wrap-reverse*

3.2.3 flex-flow

To własność, która jest skrótem do ustawiania dwóch własności naraz: *flex-direction* i *flex-wrap*.

```
.flex-container {
  display: flex;
  flex-flow: row wrap;
}
```

Jeśli nie ustawimy żadnej wartości, to defaultowo są przyjmowane row dla *flex-direction* oraz nowrap dla *flex-wrap*.

3.2.4 justify-content

To własność służąca do wyrównania elementów do pewnej pozycji wewnątrz kontenera, np. do środka, lewej, prawej itd. Możliwe wartości to:

- *center* - wyrównania do środka
- *flex-start* - wyrównania do początku kontenera
- *flex-end* - wyrównania do końca kontenera
- *space-around* - utworzenie równej pustej przestrzeni z każdej strony elementu
- *space-beetween* - utworzenie równych przestrzeni pomiędzy elementami

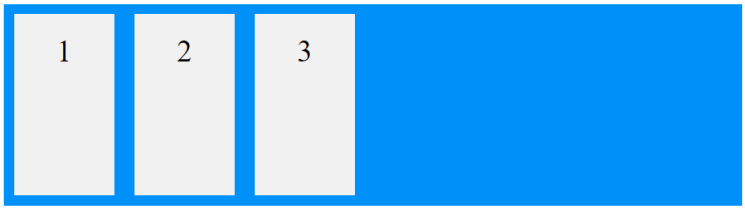
```
.flex-container {
  display: flex;
  justify-content: space-around;
  background-color: DodgerBlue;
}
```

3.2.5 align-items

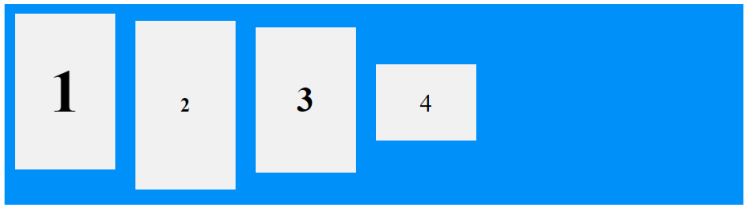
To własność, która odpowiada za wyrównanie elementów, ale pionowo, to odróżnia ją od *justify-content*, która działa poziomo. Możliwe wartości dla *align-items* to:

- *center* - środek
- *flex-start* - początek kontenera
- *flex-end* - koniec kontenera
- *stretch* (default) - rozciągnięcie elementów tak aby wypełniły kontener (zachowanie defaultowe)
- *baseline* - wyrównanie względem baseline każdego z elementów

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: stretch;
}
```




```
.flex-container {
  display: flex;
  height: 200px;
  align-items: baseline;
}
```



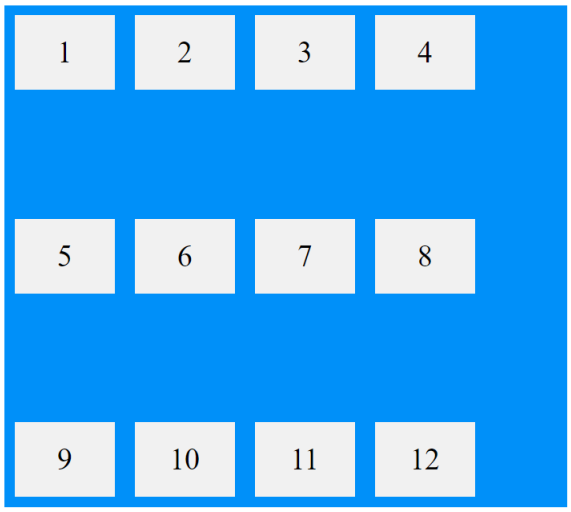
3.2.6 align-content

To własność, która służy do pozycjonowania tzw. *flex lines*, czyli wierszów w których znajdują się elementy. Możliwe wartości to:

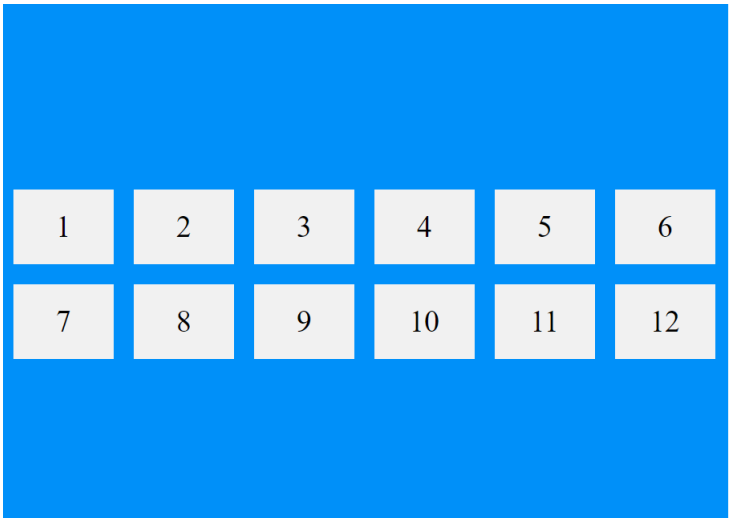
- *space-between* - linie są od siebie oddalone w takiej samej odległości

- *space-around* - każda z linii ma równą odległość przed i po niej
- *stretch* (default) - rozszerza linie aby zajmowały całą przestrzeń kontenera
- *center* - środek
- *flex-start* - początek kontenera
- *flex-end* - koniec kontenera

```
.flex-container {
  display: flex;
  height: 500px;
  flex-wrap: wrap;
  align-content: space-between;
  background-color: DodgerBlue;
}
```




```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: center;
  background-color: DodgerBlue;
}
```

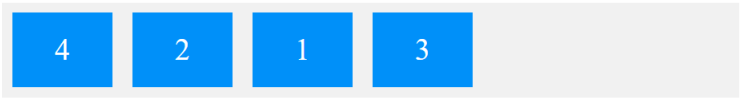


3.3 Własności flex items

3.3.1 order

Własność ta służy do określania kolejności w jakiej mają być wyświetlane flex items:

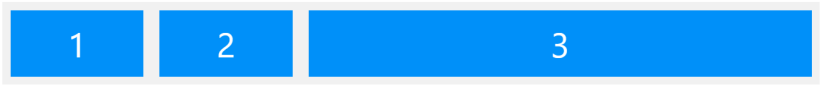
```
<div class="flex-container">
  <div style="order: 3">1</div>
  <div style="order: 2">2</div>
  <div style="order: 4">3</div>
  <div style="order: 1">4</div>
</div>
```



3.3.2 flex-grow

Określa jak bardzo element będzie się powiększał przy zwiększaniu kontenera relatywnie do innych elementów tego kontenera:

```
<div class="flex-container">
  <div style="flex-grow: 1">1</div>
  <div style="flex-grow: 1">2</div>
  <div style="flex-grow: 8">3</div>
</div>
```



Wartość musi być liczbą, defaultowa wartość to 0.

3.3.3 flex-shrink

Określa jak bardzo element będzie się zmniejszał przy zmniejszaniu kontenera relatywnie do innych elementów tego kontenera:

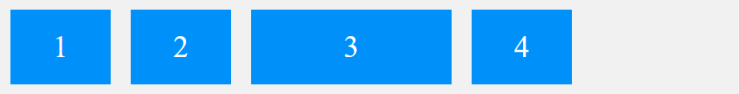
```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-shrink: 0">3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
  <div>10</div>
</div>
```

Wartość musi być liczbą, defaultowa to 1.

3.3.4 flex-basis

Określa początkową długość elementu.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-basis: 200px">3</div>
  <div>4</div>
</div>
```



3.3.5 flex

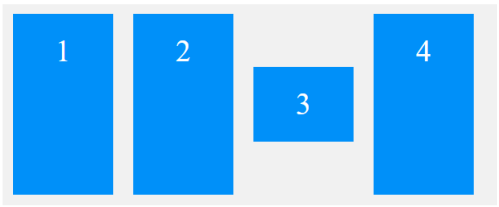
To skrót dla nadania wszystkich trzech własności naraz: *flex-grow*, *flex-shrink*, *flex-basis*.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex: 0 0 200px">3</div>
  <div>4</div>
</div>
```

3.3.6 align-self

Nadaje wyrównanie dla jednego wybranego flex item wewnątrz flex container. Własność ta nadpisuje wyrównanie nadawane przez własność kontenera *align-items*.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="align-self: center">3</div>
  <div>4</div>
</div>
```



3.3.7 Ważna uwaga

W stosowaniu powyższych własności należy zawsze uwzględnić *main axis* oraz *cross axis*, które są omówione dokładniej w kursie CSS (Udemy).

3.4 CSS Grid

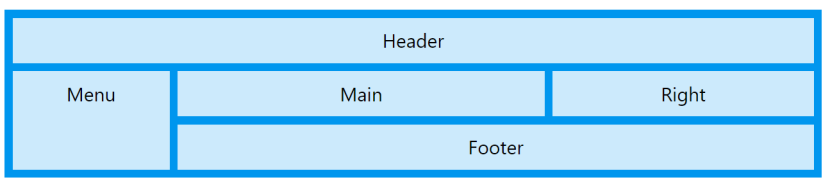
Grid to kolejne narzędzie w CSS, które pozwala nam na tworzenie layoutów dla naszej strony.

Podobnie jak przy flexbox, w Grid system mamy dwa rodzaje elementów: grid container oraz grid item. Również podobnie jak przy flexbox, aby utworzyć grid container musimy ustawić atrybut *display* na wartość grid lub inline-grid.

Wszystkie dzieci grid container automatycznie stają się grid items.

Egzamin

Utworzyć layout taki jak poniżej przy użyciu CSS grid.




```
<!DOCTYPE html>
<html>
<head>
<style>
.item1 { grid-area: header; }
.item2 { grid-area: menu; }
.item3 { grid-area: main; }
.item4 { grid-area: right; }
.item5 { grid-area: footer; }

.grid-container {
  display: grid;
  grid-template-areas:
    'header header header header header header'
    'menu main main main right right'
    'menu footer footer footer footer footer';
  grid-gap: 10px;
  background-color: #2196F3;
  padding: 10px;
}

.grid-container > div {
  background-color: rgba(255, 255, 255, 0.8);
  text-align: center;
  padding: 20px 0;
  font-size: 30px;
}
</style>
</head>
<body>

<h1>Grid Layout</h1>

<p>This grid layout contains six columns and three rows:</p>

<div class="grid-container">
  <div class="item1">Header</div>
  <div class="item2">Menu</div>
  <div class="item3">Main</div>
  <div class="item4">Right</div>
  <div class="item5">Footer</div>
</div>

</body>
</html>
```

4 JavaScript

Omówimy tu sobie tylko wybrane zagadnienia do egzaminu, głównie skupiamy się na nowościach w JavaScript.

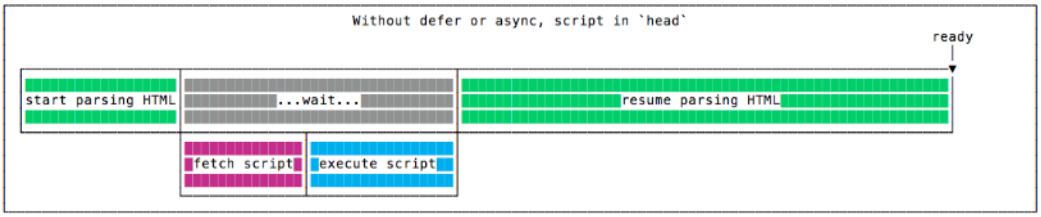
4.0.1 script, async i deffer

W HTML zawieramy fragmenty kodu JavaScript na stronie korzystając ze znacznika *script*. Znacznik ten może mieć atrybuty, przykładowo najbardziej powszechnym jest *src*, który służy do określania źródła kodu.

Jak wiemy znacznik ten może pojawić się w wielu miejscach w kodzie html:

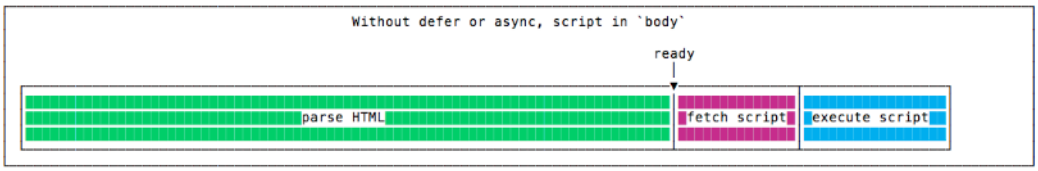
- w sekcji head
- w sekcji body
- poza sekcję body na dole dokumentu

Miejsce w którym umieścimy znacznik ma duże znaczenie wpływa to bowiem na sposób w jaki nasz kod html jest ładowany. Parser html skanuje kod od góry do dołu, w momencie napotkania znacznika *script* musi zaciągnąć kod JavaScript. Defaultowe zachowanie (script bez żadnych dodatkowych atrybutów) jest takie, że po napotkaniu script, kod jest zaciągany i wykonywany, może to jednak dość długo trwać jeśli plik jest duży:



Może to wpłynąć na czas w jakim nasza strona się załaduje i użytkownik zobaczy główną treść.

Częstym obejściem tego problemu jest wstawienie znacznika *script* na końcu dokumentu

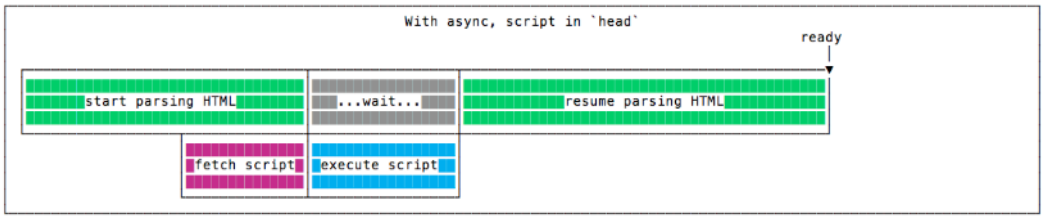


Takie rozwiązanie działa i jest w pełni akceptowalne.

Musimy być jednak świadomi, że znacznik *script* bez żadnych dodatkowych atrybutów powoduje tzw. zapytanie blokujące/synchroniczne. Aby umożliwić inne rodzaje zapytań wprowadzono dwa atrybuty: async oraz defer.

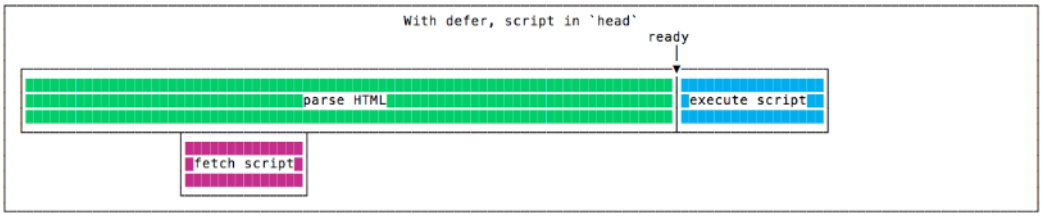
async pozwala na wykonanie typowego zapytania asynchronicznego, kiedy kod zostanie zaciągnięty jest wykonywany i blokowane jest renderowanie html:

```
1 <script async src="script.js"></script>
```



defer zaciąga kod asynchronicznie, ale jest on wykonywany dopiero po tym jak cały kod html zostanie załadowany:

```
1 <script defer src="script.js"></script>
```



Egzamin

Różnice pomiędzy *script*, *script async*, *script deffer*

4.0.2 Obsługa eventów - bubbling i capturing

JavaScript w przeglądarce korzysta z modelu programowania opartego na zdarzeniach. Funkcje rozpoczynają działanie po zajściu jakiegoś zdarzenia. Funkcja, która jest odpowiedzią na zdarzenie nazywana jest *EventHandler*. Może utworzyć wiele różnych handlerów dla tego samego zdarzenia.

W JavaScript mamy 3 możliwości do zarejestrowania handlera:

- inline

```
1 <a href="site.com" onclick="dosomething();">A link</a>
```

- zdarzenia typu on-event dla elementów DOM:

```
1 window.onload = () => {
2   //window loaded
3 }
```

- funkcja *addEventListener()*

```
1 window.addEventListener('load', () => {
2   //window loaded
3 })
```

Najbardziej nowoczesną metodą jest ostatnia z nich, umożliwia ona dodanie wielu handlerów dla tego samego eventu. Funkcję *addEventListener()* możemy dodawać do obiektu *window*, wtedy nasłuchujemy globalnych zdarzeń lub do pojedynczych elementów DOM, np. do dowolnego przycisku.

EventHandler dostaje jako argument obiekt zdarzenia, obiekt ten zawiera wiele użytecznych własności z których możemy korzystać, m.in.

- target - obiekt DOM na których zaszło zdarzenie

- type - typ zdarzenia
- stopPropagation() - wywołanie, które ma na celu zatrzymanie propagacji eventu, (będzie nam bardzo przydatne przy bąbelkowaniu)

Kiedy zachodzi zdarzenia dla jakiegoś elementu DOM mamy do czynienia z jego propagacją w modelu DOM. Są dwa modele propagacji zdarzenia:

- bubbling (defaultowe zachowanie) - w tym modelu zdarzenie rozchodzi się od elementu DOM na którym zaszło w górę drzewa DOM
- capturing - w tym modelu zdarzenie rozchodzi się od korzenia modelu DOM do elementu na którym zaszło

Wyobraźmy sobie, że mamy taką sytuacje:

```
<div id="container">
  <button>Click me</button>
</div>
```

Mamy kontener w którym znajduje się przycisk. Zarówno dla kontenera jak i dla przycisku ustawiliśmy obsługę zdarzenia kliknięcia. Jeśli użytkownik kliknie w przycisk to klika również w kontener. W jakiej kolejności wykonają się handlery ? Model bąbelkowy mówi, że najpierw wykona się handler dla przycisku, a później dla kontenera, model *capturing* mówi, że najpierw będzie to kontener, a później przycisk.

Defaultowe zachowanie to model bąbelkowy, aby włączyć *capturing* musimy ustawić trzeci argument przy dodawaniu handlera:

```
document.getElementById('container').addEventListener(
  'click',
  () => {
    //window loaded
  },
  true
)
```

A co jeśli mamy kilka eventów w drzewie i niektóre z nich są w modelu *capturing* (ustawiliśmy trzeci parametr na true), a niektóre nie ?
Pełny obraz tego jak obsługiwane są eventy jest taki:

1. Zawsze zaczynamy od początku drzewa DOM
2. Przechodzimy w dół drzewa aż do najniżej położonego elementu na którym zaszło zdarzenie
3. Jeśli przechodząc w dół drzewa DOM napotkamy event, który jest w modelu *capturing*, to jest on wywoływany
4. Po dojściu do elementu najniższego wołany jest handler dla tego elementu
5. Wracamy od elementu do korzenia DOM i po drodze wywołujemy handlery, które są w modelu bąbelkowym

Event będzie propagowany do wszystkich elementów DOM, które są wyżej od niego w drzewie o ile jawnie nie zatrzymamy propagacji:

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // process the event
  // ...

  event.stopPropagation()
})
```

Egzamin

Funkcja fufu która wypisze jaki przycisk został wciśniety i następnie zatrzyma bąbelkowanie. Zarejestrować funkcje fufu do eventu wciśniecia przycisku myszy.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Test</title>
    </head>
    <body>
        <button id="button-test">Click Me !</button>
    </body>
    <script>
        document.getElementById("button-test").addEventListener('click', event => {
            console.log("Button: " + event.target.id);
            event.stopPropagation();
        });
    </script>
</html>
```

4.1 prototype

Każdy obiekt-funkcja (obiekt, który jest funkcją, a więc zwłaszcza każdy konstruktor) w JavaScript ma własność, która nazywa się *prototype*. Własność ta określa właściwości jakie posiada dany obiekt-funkcja (funkcje i pola).

Każdy dowolny obiekt w JavaScript posiada pole __proto__, które wskazuje na własność prototype pewnego obiektu-funkcji. Dzięki temu obiekt taki może używać pól i metod, które określa pole prototype tego obiektu-funkcji. W JavaScript w ten sposób realizowany jest mechanizm dziedziczenia.

Egzamin

Co to jest prototype ?

4.2 ES6 - nowe elementy języka

Nowe elementy języka wprowadzone z ES6 to:

- Arrow functions
- Promises
- Generators
- `let` and `const`
- Classes
- Modules
- Multiline strings
- Template literals
- Default parameters
- The spread operator
- Destructuring assignments
- Enhanced object literals
- The for..of loop
- Map and Set

Pytanie do opracowania

Jakie nowe elementy wprowadza ES6 do języka JavaScript ?

4.3 var, let i const

`let` and `const`

`var` is traditionally **function scoped**.

`let` is a new variable declaration which is **block scoped**.

This means that declaring `let` variables in a for loop, inside an if or in a plain block is not going to let that variable "escape" the block, while `var` s are hoisted up to the function definition.

`const` is just like `let`, but **immutable**.

In JavaScript moving forward, you'll see little to no `var` declarations any more, just `let` and `const`.

`const` in particular, maybe surprisingly, is **very widely used** nowadays with immutability being very popular.

```
function testVar() {
  var x = 5;
  if (x == 5) {
    var x = 8;        // ta sama zmienna o zasięgu funkcji
    console.log(x);   // 8
  }
  console.log(x);     // 8
}

function testLet() {
  let x = 5;
  if (x == 5) {
    let x = 8;  // nowa zmienna, lokalna dla bloku kodu
    console.log(x);   // 8
  }
  console.log(x);     // 5
}
```

Pytanie do opracowania

Czym różni się var, let i const ?

4.4 hoisting

Hositing (windowanie) to w JavaScript mechanizm, który pozwala na to żeby zmienna była używana przed tym jak została zadeklarowana:

```
1 x = 5; // Assign 5 to x
2
3 elem = document.getElementById("demo"); // Find an element
4 elem.innerHTML = x;                     // Display x in the element
5
6 var x; // Declare x
```

Taka możliwość istnieje jednak tylko dla deklaracji ze słowem kluczowym var, nie działa dla let oraz const. Ponadto hoisting działa tylko dla deklaracji bez inicjalizacji:

```
1 var x = 5; // Initialize x
2
3 elem = document.getElementById("demo"); // Find an element
```

```
elem.innerHTML = x + " " + y;            // Displays 5 undefined

var y = 7; // Initialize y
```

Pytanie do opracowania

Czym jest hoisting ? Kiedy działa a kiedy nie ?

4.5 arrow functions

Arrow functions to rozszerzenie języka o elementy programowania funkcyjnego. Zamiast pisać:

```
const foo = function foo() {
//...
}
```

możemy napisać zgrabniej:

```
const foo = () => {
//...
}
```

Pytanie do opracowania

Jak napisać arrow function ?

4.6 Parametry domyślne

```
function myFunc(x = 10) {
  return x;
}

console.log( myFunc() );

// 10 -- zadna wartosc nie jest podana, x otrzymuje wartosc domyslna, czyli 10
console.log( myFunc(5) )
// 5 -- wartosc podana, wiec x otrzymuje wartosc 5
console.log( myFunc(undefined) )
// 10 -- podana wartosc undefined, wiec x otrzymuje wartosc domyslna, czyli 10
console.log( myFunc(null) )
// null -- wartosc (null) jest podana
```

Pytanie do opracowania

Czym są parametry domyśle w JS ?

4.7 Wyrażenia interpolowane

To wyrażenie będące stringami w których możemy umieszczać zmienne , których wartość jest ustalana w momencie interpretacji kodu. Wyrażenia interpolowane umieszczamy w backticks (pod klawiszem esc na klawiaturze).

```
var imie = "Jan";
var witaj = `Czesc ${imie} !`;
consoel.log(witaj); \\ Czesc Jan !
```

W taki sposób możemy również wywoływać funkcje w wyrażeniach interpolowanych:

```
function upper(text){ return text.toUpperCase(); }

var imie = "Jan";
var info = `To jest pierwsze ${upper("wywolanie")} tej funckji przez ${upper(`${imie}
    a`)}`

console.log(info); // To jest pierwsze WYWOLANIE tej funkcji przez JANA
```

Pytanie do opracowania

Czym są wyrażenie interpolowane w JS ?

4.8 Destrukturyzacja obiektów

Jeśli mamy dany dany obiekt lub tablicę, to możemy wyciągnąć z nich określone wartości i zapisać je do zmiennych w następujący sposób:

```
const person = {
  firstName: 'Tom',
  lastName: 'Cruise',
  actor: true,
  age: 54, //made up
}
const {firstName: name, age} = person
```

```
const a = [1,2,3,4,5]
[first, second, , , fifth] = a
```

Pytanie do opracowania

Jak działa destrukturyzacja obiektów w JS ?

4.9 Operatory rest (rozproszenia/reszty) i spread

To dwa operatory, które zapisujemy tak samo (trzy kropki) ..., a które służą do dwóch odwrotnych względem siebie rzeczy.

- rest - scala kilka wartości w kontener (tablicę lub obiekt)

```
function add(...numbers) {
  return numbers.reduce((sum, elem) => sum + elem);
}

var sum = add(3,5,8);
```

- spread - rozwija kontener (tablicę lub obiekt) w pojedyncze wartości

```
const arr1 = ["a", "b", "c"];
const arr2 = [...arr1, "d", "e", "f"]; // ["a", "b", "c", "d", "e", "f"]
```

Egzamin

Różnica między spread operator a rest parameters.

Egzamin

Napisać funkcję w JavaScript, która usunie duplikaty z tablicy.

```
<script>
      let myArr = [3,4,5,3,4,7,8,3,6,10];

      function removeDuplicates(arr){
          return [...new Set(arr)];
      }

      console.log(removeDuplicates(myArr));
 </script>
```

4.10 Pętla for of oraz for in

W ES6 dodano pętle for of, która pozwala na iterowanie po wartościach dowolnego obiektu, który jest iterowalny (np. tablica, mapa itd.). Pętla for in iteruje natomiast po kluczach/indeksach.

```
var tab = ["a", "b", "c", "d", "e", "f"];

for (let i in tab) {
  console.log(i); // 0 1 2 3 4 5
}

for (let i of tab){
  console.log(i); // a b c d e f
}
```

Pytanie do opracowania

Czym różni się pętla for of od for in ?

4.11 map, filter, reduce

To metody tablicowe, zapożyczone z paradygmatu programowania funkcyjnego, które służą do wykonywania operacji na tablicach:

- *Array.prototype.map()* - przyjmuje jako argument tablicę, modyfikuje jej elementy i zwraca nową tablicę ze zmienionymi elementami.
- *Array.prototype.filter()* przyjmuje jako argument tablicę, decyduje, które elementy zatrzymać a które usunąć. Zwraca tablicę z zachowanymi elementami.
- *Array.prototype.reduce()* przyjmuje jako argument tablicę, wylicza z jej elementów jedną wspólną wartość, którą zwraca.

Najczęściej metody te używane są razem w celu obliczenia pewnej wartości:

```
const students = [
  { name: "Iza", grade: 10 },
  { name: "Ala", grade: 15 },
  { name: "Ula", grade: 19 },
  { name: "Ola", grade: 9 },
];

const sumaOcen =
  students
    .map(student => student.grade)
    // tworzymy tablice ocen z tablicy studentow za pomoca map
    .filter(grade => grade >= 10)
    // filtrujemy tablice ocen, by pozostawic wieksze lub rowne 10
    .reduce((prev, next) => prev + next, 0);
    // sumujemy wszystkie oceny powyzej 10
    console.log(sumaOcen) // 44 -- 10 (Iza) + 15 (Ala) + 19 (Ula), Ola ponizej 10 -
        ignorowana
```

Pytanie do opracowania

Jak działają funkcje map, filter, reduce ?

4.12 Przetwarzanie asynchroniczne

Sposoby implementacji asynchronicznego kodu w JS:

- Funkcje wywołań zwrotnych (callback) - funkcja wołana asynchronicznie otrzymuje jako argumenty funkcję, która będzie wywoływana po zakończeniu przetwarzania
- Generatory - od ES2015
- Obietnice - od ES2015
- Składnia *async/await* - bazuje na obietnicach i generatorach
- Programowanie reaktywne - ReactiveExtensions, RxJS

Pytanie do opracowania

Jakie znasz sposoby asynchronicznego przetwarzania w JavaScript ?

4.13 Generatory

Generatory to funkcje, które mogą kontrolować iteratory. Generatory tworzymy tak jak normalne funkcje, dodajemy jedynie gwiazdkę przed nazwą funkcji. W ciele funkcji używamy słowa kluczowego *yield*, napotkanie tego słowa powoduje, że generator zwraca wartość, która znajduje się po *yield*, następne wywołanie generatora zaczyna działanie od ostatniego *yield* i kontynuuje aż do napotkania następnego.

```
1  function * withYield(a) {
2    let b = 5;
3    yield a + b;
4    b = 6; // it will be re-assigned after first execution
5    yield a * b;
6  }
7
8  const calcSix = withYield(6);
9
10 calcSix.next().value; // 11
11 calcSix.next().value; // 36
```

Wywołanie metody *next* na generatorze zawsze zwraca obiekt, który ma dwa pola: *value* oraz *done*. Drugie z nich określa czy generator zakończył działanie (ma wartość true jeśli generator nie może zwrócić więcej wartości).

Pytanie do opracowania

Omów czym są generatory w JavaScript.

4.14 Promise

Promise to nowy element języka w ES6, który pozwala zapobiec zjawisku tzw. callback hell przy wywołaniach asynchronicznych.

Promise to obiekt, który przyjmuje jako parametr konstruktora funkcję z dwoma parametrami: resolve i reject.

W funkcji tej wykonujemy nasze asynchroniczne zapytania. Kod tej funkcji jest wykonywany natychmiast po tym jak obiekt zostanie stworzony.

Jeśli nasze zapytanie się powiedzie zwracamy rezultat przez resolve, a jeśli nie, to przez reject:

```
const promise1 = new Promise(function(resolve, reject) {

  // asynchronous task
  setTimeout(function() {
    resolve('foo');
  }, 300);
});

promise1.then(function(value) {
  console.log(value);
  // expected output: "foo"
});
```

Następnie dla tak utworzonego obiektu możemy wywołać metodę then, która jest wołana po tym jak obietnica zwróci wartość przez resolve oraz catch, która służy do obsługi odrzuconej obietnicy. W tych metodach możemy np. zrobić update dla naszego UI.

Możemy również tworzyć łańcuchy obietnic:

```
/_ ES6 _/
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);

const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};

// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};

askMom();
```

Pytanie do opracowania

Omów działanie obietnic, jak przy ich pomocy realizować zapytania asynchroniczne ?

4.15 async await

W ES7 wprowadzono kolejny sposób na wykonywanie zapytań asynchronicznych. Jeśli nasza funkcja wykonuje jakąś asynchroniczną pracę to poprzedzamy ją słowem *async*, a jeśli wywołujemy taką funkcję

i chcemy poczekać na jej rezultat, to wstawiamy przed taim wywołaniem słowo *await*. Przepiszmy powyższy kod z użyciem async i await:

```
/_ ES7 _/
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => {
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);

// 2nd promise
async function showOff(phone) {
    return new Promise(
        (resolve, reject) => {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};

// call our promise
async function askMom() {
    try {
        console.log('before asking Mom');

        let phone = await willIGetNewPhone;
        let message = await showOff(phone);

        console.log(message);
        console.log('after asking mom');
    }
    catch (error) {
        console.log(error.message);
    }
}

(async () => {
    await askMom();
})();
```

Pytanie do opracowania

Do czego służą słowa async, await dodane w ES7 ? Jak realizować przy ich pomocy zapytania asynchroniczne ?

5 NodeJS

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  const url = req.url;

  if (url === '/getjson') {
    res.setHeader('Content-Type', 'application/json');

    let rawdata = fs.readFileSync('Country.json');
    let countryJSON = JSON.parse(rawdata);

    res.end(JSON.stringify(countryJSON));
  }

});

server.listen(3000);
```

Egzamin

W czystym Node.js napisać serwer który będzie serwował Country.json

6 Angular

6.1 Elementy dodane do JavaScript

TypeScript jest językiem w którym tworzymy aplikacje angularowe. Typescript jest swego rodzaju nakładką na JavaScript, wszystkie konstrukcje poprawne w JavaScript ES6 są nadal poprawne w TypeScript, ale język TypeScript dodaje do języka kilka nowych elementów:

- deklaracje typów
- sprawdzanie typów
- generyczność
- moduły
- interfejsy
- dekoratory

```
let x: number;
let name: string;
let empty: boolean;

let tab1: number[];
let tab2: Array<number>;
```

```
interface Person {
  name: string;
  age?: number;

  order(): void;
}
```

```
interface KeyValuePair<K, T> {
key: K;
value: T;
}
```

Pytanie do opracowania

Czym jest TypeScript ? Jakie nowe elementy wprowadza ?

6.2 npm

Node Package Manager (npm) to menadżer pakietów dla node.js. Przy jego pomocy zarządzamy naszą aplikacją angularową.

```
npm install <Nazwa Modulu> \\ instalacja modulu
```

Pytanie do opracowania

Jak zainstalować moduł przy użyciu npm ?

6.3 Angular CLI

Angular Command Line Interface to narzędzie używane w formie linii poleceń, które pozwala na inicjalizowanie, rozwijanie, generowanie i utrzymywanie kodu angulara.

```
ng new NowyProjekt \\ utworzenie nowego projektu angulara
```

```
ng serve \\ uruchomienie serwera
```

```
ng generate component ComponentName  \\ wygenerowanie komponentu
```

```
ng generate service ServiceName  \\ wygenerowanie serwisu
```

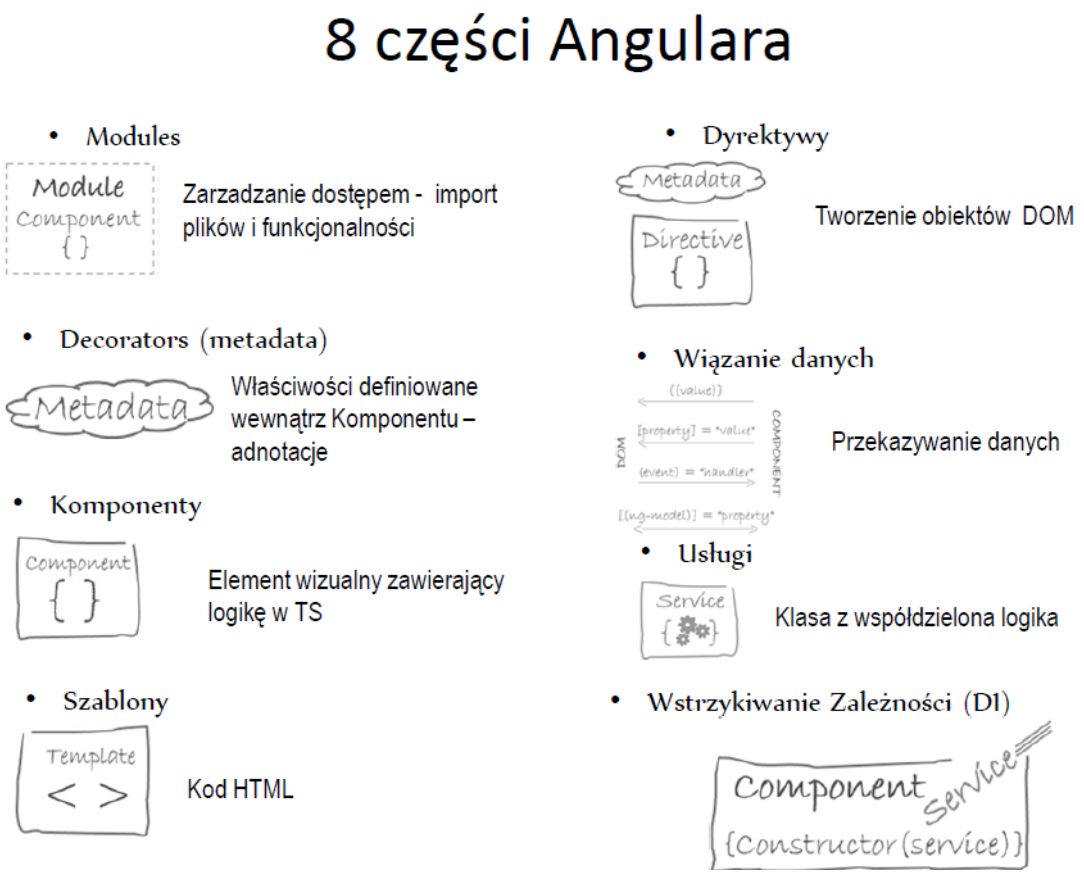
Pytanie do opracowania

Jak przy użyciu angular CLI:

- utworzyć nowy projekt
- uruchomić serwer
- wygenerować nowy komponent
- wygenerować nowy serwis

6.4 Główne składowe angulara

8 głównych bloków konstrukcyjnych angulara to:



Omówimy sobie krótko każdy z tych bloków.

Pytanie do opracowania

Wymień 8 głównych bloków konstrukcyjnych angulara.

6.4.1 Moduły

@NgModule

- Narzędzie do organizacji struktury aplikacji
- Zawierają powiązane ze sobą komponenty, dyrektywy i pipes
- Wszystkie podstawowe funkcjonalności Angulara są zawarte w modułach (**`FormsModule`**, **`HttpModule`**, **`RouterModule`**)
- Mogą być *lazy*- lub *eager loaded*
- Aplikacja zawiera jeden główny moduł, który służy do jej wystartowania (bootstrap) i importuje pozostałe moduły
- Deklarujemy jako klasę z dekoratorem **`@NgModule`**

Przykładowy moduł do routingu z mojego projektu:

```
const routes = [
  {path: 'login', component: LoginComponent},
  {path: 'signup', component: SignupComponent},
  {path: 'trips', component: TripsComponent, canActivate: [AuthGuard]},
  {path: 'trip/:id', component: TripDetailedViewComponent, canActivate: [AuthGuard]},
  {path: 'basket', component: UserBasketComponent, canActivate: [AuthGuard]},
  {path: 'history', component: UserHistoryComponent, canActivate: [AuthGuard]},
  {path: 'admin', component: AdminComponent, canActivate: [AuthGuard, AdminGuard]},
  {path: '**', redirectTo: 'login'}
];

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    RouterModule.forRoot(routes),
  ],
  exports: [
    RouterModule
  ],
  providers: [
    AuthGuard,
    AdminGuard
  ]
})
export class AppRoutingModule { }
```

Główny moduł z mojego projektu (importuje moduł routingu):

```
@NgModule({
  declarations: [
    AppViewComponent,
    TripsComponent,
    HeaderComponent,
    UserBasketComponent,
    TripcarddetailsComponent,
    AdminComponent,
    AdminAddTripComponent,
    AdminRemoveTripComponent,
```

```
    TripsFilterCriteriaComponent,
    TripDetailedViewComponent,
    LoginComponent,
    SignupComponent,
    FooterComponent,
    UserHistoryComponent
  ],
  imports: [
    BrowserModule,
    NgbModule,
    FormsModule,
    ReactiveFormsModule,
    AppRoutingModule,
    AngularFireModule.initializeApp(environment.firebaseConfig),
    AngularFirestoreModule,
    AngularFireAuthModule
  ],
  providers: [TripsService, BasketService, AuthService, CommentsService,
      UserHistoryService],
  bootstrap: [AppViewComponent]
})
export class AppModule { }
```

Aplikację angularową uruchamiany przez odpalenie jej głównego modułu:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- Angular jest dostarczony jako kolekcja modułów JavaScript. Można o nich myśleć jako o bibliotece modułów.
- Nazwa każdej biblioteki Angulara zaczyna się od przedrostka `@angular`.
- Biblioteki instalujemy narzędziem **npm** i importujemy jakieś ich części poleceniem `import` języka JavaScript.
- Przykład importowania dekoratora `Component` Angulara z biblioteki `@angular/core`:

```
import { Component } from '@angular/core';
```

Moduły Angular i moduły JavaScript

- Moduł Angular - klasa zmodyfikowana dekoratorem `@NgModule`.
- JavaScript (TypeScript) ma swój własny system modułów do zarządzania kolekcjami obiektów JavaScript. To jest **kompletnie inny** i **niezwiązany z Angularem** system modułów.
- W JavaScript każdy plik jest modułem i wszystkie obiekty zdefiniowane w tym pliku należą do tego modułu. Moduł deklaruje niektóre obiekty jako publiczne oznaczając je słowem kluczowym `export`. Inne moduły JavaScript używają polecenia `import` aby dostać się do publicznych obiektów innych modułów.

```
import { NgModule }     from '@angular/core';
import { AppComponent } from './app.component';
```

```
export class AppModule { }
```

- To są dwa inne ale komplementarne systemy modułów.
  Obu używamy do tworzenia aplikacji.

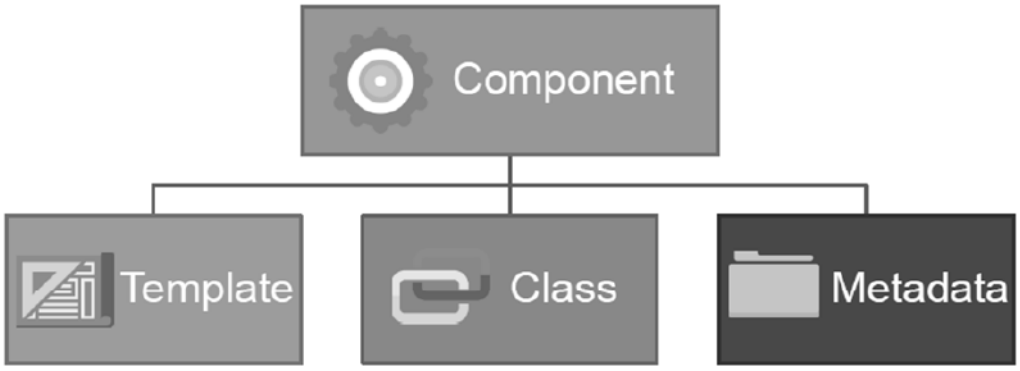
Pytanie do opracowania

Omów moduły w Angularze, do czego służą, jak je tworzymy. Czy moduły angulara są tym samym co moduły JavaScript ?

6.4.2 Komponenty

Czym jest komponent w Angular

- Komponenty są podstawowymi blokami konstrukcyjnymi aplikacji Angular.
- Kontrolują jakiś obszar ekranu - widok - poprzez związany z nimi szablon.
- Wewnątrz klasy komponentu definiujemy logikę aplikacji - określamy jak komponent obsługuje widok.
- Klasa komponentu komunikuje sie z widokiem poprzez API pól i metod.



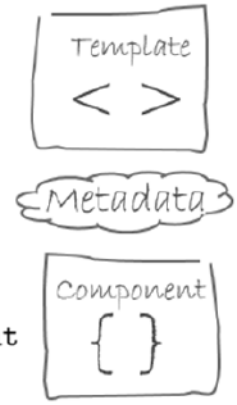
Pytanie do opracowania

Czym są komponenty w Angular ?

6.4.3 Metadane

Metadane

- Metadane w dekoratorze `@Component` informują Angular, gdzie uzyskać główne bloki konstrukcyjne, z których będzie korzystał komponent.
- Szablon, metadane i komponent **razem definiują widok**.
- Stosujemy inne dekoratory z ich metadanymi w podobny sposób, żeby kierować zachowaniem Angulara.
- Inne popularne dekoratory: `@Injectable`, `@Input`, `@Output`



Wniosek:

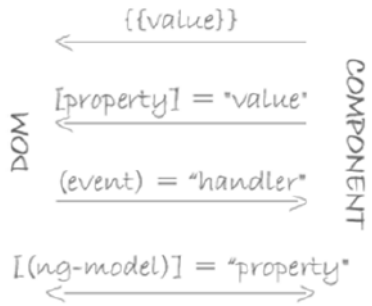
musimy dodać metadane do naszego kodu, żeby Angular wiedział co ma robić.

Pytanie do opracowania

Do czego służą metadane w Angular ?

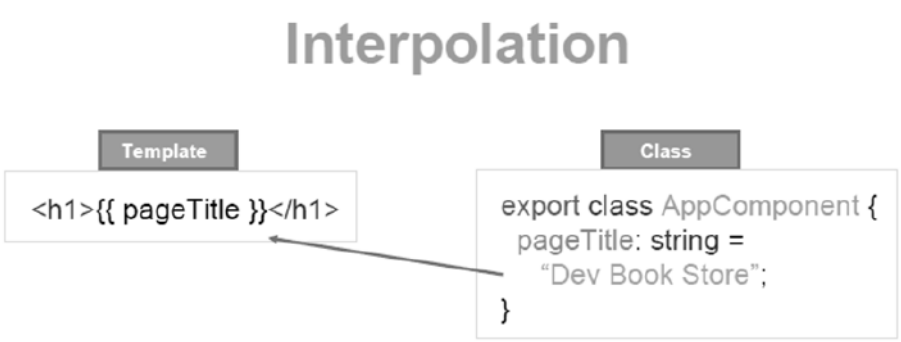
6.4.4 Wiązanie danych

To mechanizmy, które pozwalają wymieniać dane pomiędzy klasą komponentu, a odpowiadającym mu szablonem. Są cztery takie możliwe mechanizmy:



Nazwy tych mechanizmów to kolejno:

- interpolation



- property binding

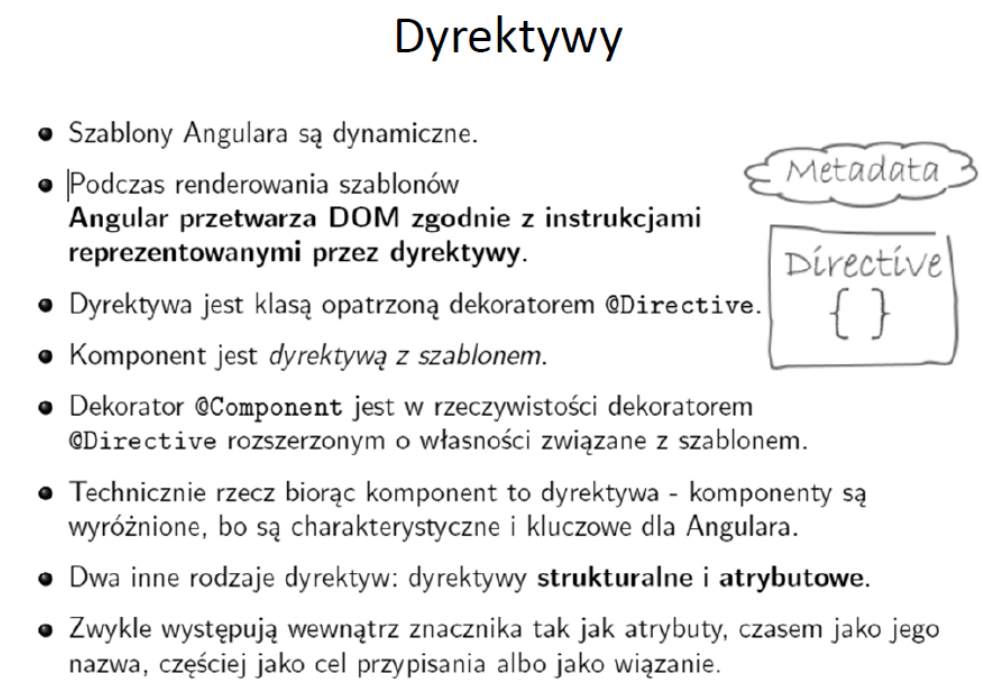
- event binding

- two way binding

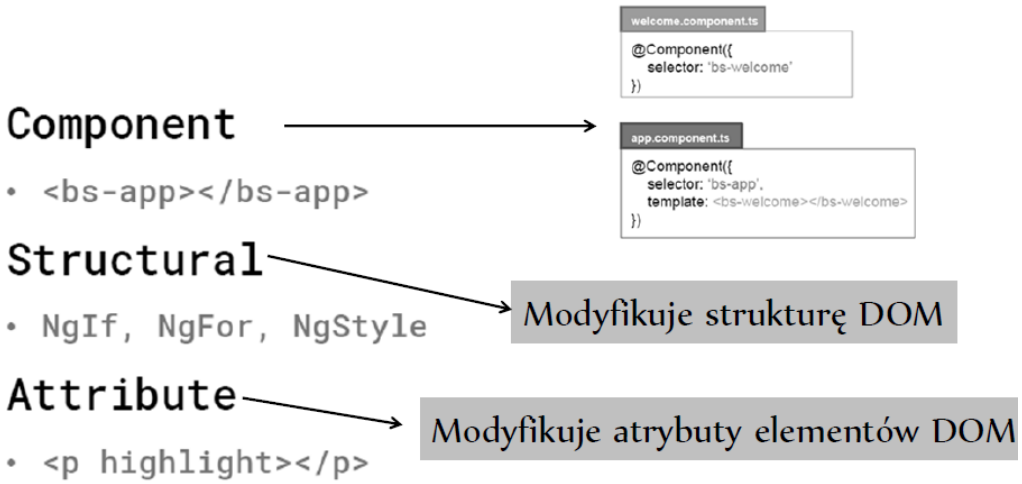
Pytanie do opracowania

Czym jest wiązanie danych w Angular ? Wymień i opisz cztery mechanizmy stosowane w wiązaniu danych.

6.4.5 Dyrektywy



Typy Dyrektyw



Dyrektywy strukturalne

Dyrektywy strukturalne zmieniają układ graficzy poprzez dodawanie, usuwanie i zamianę elementów w drzewie DOM.

Przykładowy szablon używa dwóch wbudowanych dyrektyw strukturalnych:

src/app/hero-list.component.html (structural)

```
<li *ngFor="let hero of heroes"></li>
<hero-detail *ngIf="selectedHero"></hero-detail>
```

- `*ngFor` nakazuje Angularowi wygenerować po jednym elemencie `<li>` na każdego bohatera z listy.
- `*ngIf` dodaje komponent `HeroDetail` do widoku tylko, jeśli wybrany bohater istnieje.

Dyrektywy atrybutowe

Dyrektywy atrybutowe:

- zmieniają wygląd lub zachowanie istniejących elementów,
- w szablonie wyglądają jak zwykłe atrybuty znaczników HTML - stąd ich nazwa,
- dyrektywa `ngModel` (dwukierunkowe wiązanie danych) jest przykładem dyrektywy atrybutowej,
- `ngModel` modyfikuje zachowanie istniejącego elementu (zwykle `<input>`) poprzez ustawianie jego właściwości `value` i reagowanie na zdarzenia zmiany.

src/app/hero-detail.component.html (ngModel)

```
<input [(ngModel)]="hero.name">
```

Angular posiada więcej dyrektyw, które:

- albo zmieniają strukturę układu (na przykład `ngSwitch`),
- albo modyfikują pewne aspekty elementów drzewa DOM i komponentów (na przykład `ngStyle` i `ngClass`).

Możemy pisać nasze własne dyrektywy. Komponenty, np. `HeroListComponent`, są jednym z rodzajów własnych dyrektyw.

Egzamin (???) Chyba chodzi o to aby użyć dyrektyw i wypisać w szablonie

W angularze wypisać tablicę krajów ({country: "POland", population: "38mlln",...), z tym że kazdy kraj w osobnym wierszu, poprzedzonym indexem kraju i jeśli populacja jest mniejsza niż 10mln to wypisać "Mały Kraj"

```
@Component({
  selector: 'my-app',
  template: `
    <div *ngFor="let country of countries; index as i;">
       <div *ngIf="country.population < 10; else elseBlock">
        {{i}}. "Maly kraj"
       </div>
      <ng-template #elseBlock>{{i}}. {{country.country}}, {{country.population}}</
          ng-template>
    </div>
  `,
})
class HomeComponent {

    countries = [
      {country: "Poland", population: 34},
      {country: "UK", population: 86},
      {country: "Maldives", population: 2},
      {country: "Germany", population: 23}
    ];

}
```

W Angular mamy również możliwość tworzenia własnych dyrektyw, nową dyrektywę tworzymy przy użyciu *@Directive*.

Przykładowo stwórzmy sobie dyrektywę, która będzie dyrektywą atrybutową dołączaną do dowolnego elementu i dla tego elementu będzie zmieniać kolor tekstu i ramki jeśli ktoś naciśnie klawisz na tym elemencie:

```
import {
  Directive,
  HostBinding,
```

```
  HostListener } from '@angular/core';

@Directive({
  selector: '[appRainbow]'
})
export class RainbowDirective {
  possibleColors = [
    'darksalmon', 'hotpink', 'lightskyblue', 'goldenrod', 'peachpuff',
    'mediumspringgreen', 'cornflowerblue', 'blanchedalmond', 'lightslategrey'
  ];

  @HostBinding('style.color') color: string;
  @HostBinding('style.border-color') borderColor: string;

  @HostListener('keydown') newColor() {
    const colorPick = Math.floor(Math.random() * this.possibleColors.length);

    this.color = this.borderColor = this.possibleColors[colorPick];
  }
}
```

Takie dyrektywy używalibyśmy w następujący sposób:

```
<input type="text" appRainbow>
```

Zwróćmy uwagę na użycie wewnątrz definicji dwóch nowych dyrektyw

- *@HostBinding* - umożliwia połączenie zmiennej z własnością elementu, który hostuje daną dyrektywę np.

  ```
  @HostBinding('style.color') color: string;
  ```

  przypisuje do zmiennej kolor atrybut stylu color dla elementu hostującego. Dzięki temu możemy przypisywać mu wartość.

- *@HostListener* - umożliwia reagowanie na zdarzenia pojawiające się na elemencie hostującym, np.

  ```
  @HostListener('keydown') newColor() {
      const colorPick = Math.floor(Math.random() * this.possibleColors.length);

      this.color = this.borderColor = this.possibleColors[colorPick];
    }
  ```

  Taki zapis definiuje funkcję, która będzie wołana gdy na elemencie hostującym ktoś naciśnie klawisz.

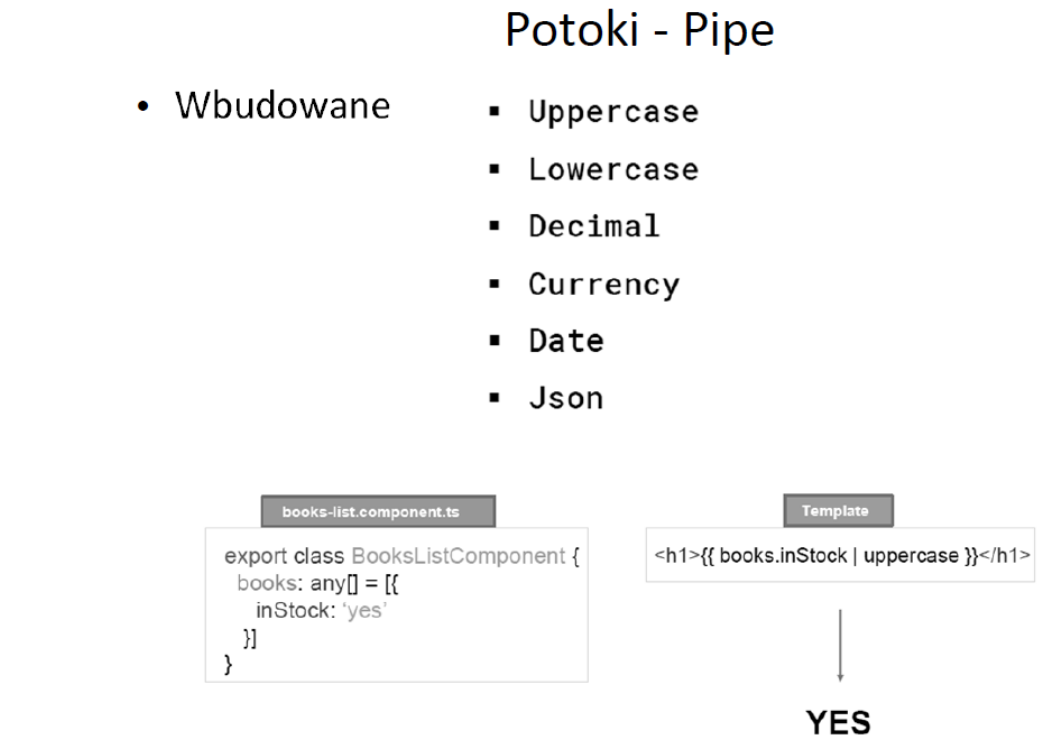
Pytanie do opracowania

Czym są dyrektywy ? Do czego służą ? Wymień rodzaje dyrektyw i podaj przykłady.

Egzamin

@HostBinding i *@HostListner* - opisać

6.4.6 Potoki



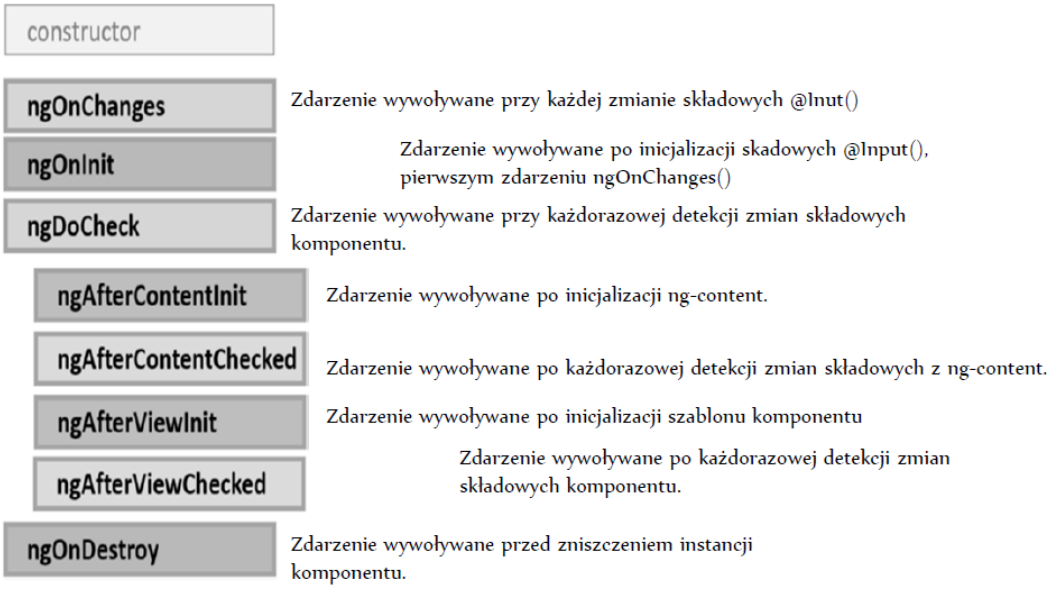
Pytanie do opracowania

Czym są potoki w Angular ? Do czego służą ?

6.5 Cykl życia komponentu (Lifecycle Hooks)

Każdy komponent posiada swój cykl życia. Oznacza to, że może zostać m.in, zainicjalizowany, wyrenderowany czy zniszczony, a my możemy na te zdarzenia zareagować (np. zwolnić zasoby).

Przykładowe hooki z wykładu (uruchamiane w takiej kolejności jak zapisana):



Wszystkich hooków jest znacznie więcej, ale rzadko używamy wszystkich z nich w naszych komponentach.

Hook *ngOnDestroy* jest typowo używany do odłączenia od subskrypcji. Robimy to w celu uniknięcia wycieków pamięci.

Pytanie do opracowania

Czym jest cykl życia komponentu ?

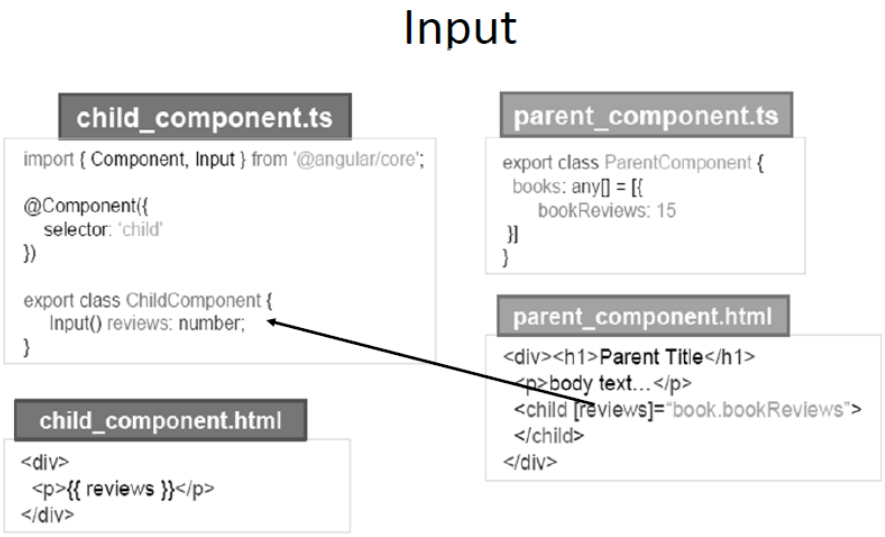
Egzamin

Hooki komponentu Angulara. Do którego hooka podpiąłbyś odłączenie od subskrypcji ?

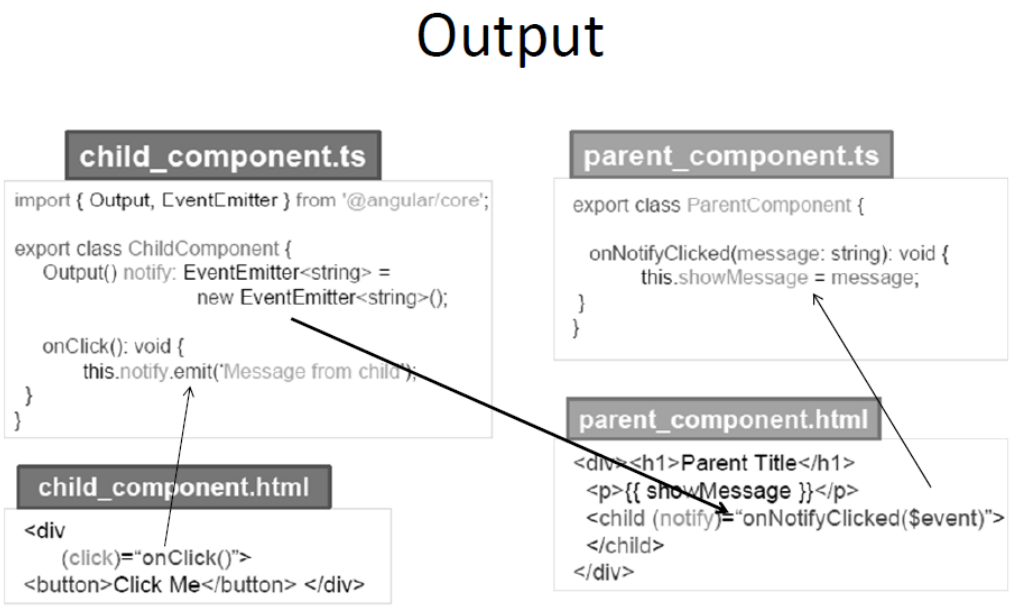
6.6 Komunikacja pomiędzy komponentami

W angularze istnieją trzy sposoby na komunikację pomiędzy komponentami:

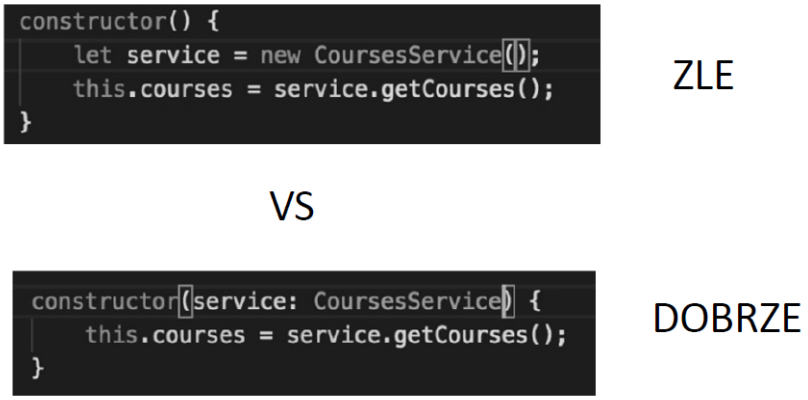
- Input - rodzic do dziecka



- Output - dziecko do rodzica



- Serwisy - dowolne komponenty pomiędzy sobą Serwisy to klasy oznaczane dekoratorem *@Injectable*, które mogą być wstrzykiwane do innych serwisów lub komponentów przez mechanizm Dependency Injection. Usługi należy wstrzykiwać w konstruktorze w następujący sposób:



Usługi mogą być w aplikacji tworzone jako singletony lub jako pojedyncze instancje, w zależności od naszych potrzeb. Aby utworzyć usługę jako singleton mamy dwie możliwości:

– ustawić *providedIn* na wartość *'root'* w dekoratorze *@Injectable* serwisu (od Angular 6.0)

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

– dodać serwis do tablicy *providers* głównego modułu aplikacji:

```
@NgModule({
  ...
  providers: [UserService],
  ...
})
```

Postępując w taki sposób każde wstrzyknięcie będzie sprawdzało czy instancja nie została już stworzona, a jeśli taki, zostanie użyta istniejąca instancja serwisu.

Jeśli chcemy natomiast tworzyć nową instancję dla komponentu wtedy dodajemy serwis do tablicy *providers*, ale w dekoratorze *@Component* wybranego komponentu:

```
1 @Component({
2 /* . . . */
3   providers: [UserService]
4 })
```

Pytanie do opracowania

Jakie znasz sposoby komunikacji pomiędzy komponentami w angular ?

Egzamin

Jak utworzyć usługę singleton a jak instancję usługi dla komponentu w Angular ?

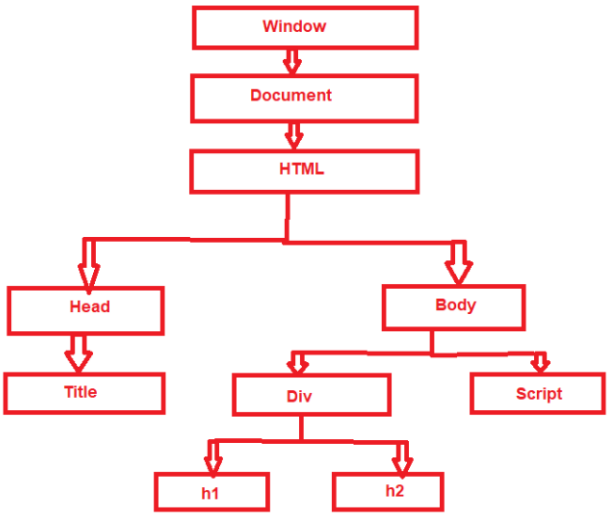
6.7 Property binding czy attribute binding ?!

Zacznijmy od tego, że atrybuty języka HTML, to nie to samo co atrybuty modelu DOM, ponieważ język HTML nie jest równoważny co do nazw z modelem DOM.

Chodzi o to, że jeśli napiszemy sobie fragment kodu HTML, np. taki:

```
<html>
<head>
    <title>This is Title</title>
</head>
<body>
    <script src="Scripts/jquery-1.10.2.js"></script>
    <div>
        <h1>This is Browser DOM</h1>
        <h2>This is Inside H2</h2>
    </div>
</body>
</html>
```

To przeglądarka tworzy na jego podstawie model DOM, gdzie każdy z elementów jest reprezentowany przez obiekt i wyświetlany w przeglądarce:



To co wyświetla przeglądarka opiera się właśnie na tym modelu DOM. Dzięki temu modelowi możemy następnie od strony programu manipulować wyglądem HTML na naszej stronie właśnie poprzez tak naprawdę manipulację wygenerowanym modelem DOM.

W kodzie HTML możemy umieszczać *atrybuty znaczników HTML*, np.

```
<input id="test1" type="text" value="Dominik" >
```

W powyższym kodzie *type* oraz *value* to atrybuty html. Następnie po translacji na model DOM są one zamieniane na początkowe wartości własności modelu DOM. To właśnie jest główna rola atrybutów HTML, stanowią one wartość początkową dla własności modelu DOM.

My jako programiści jeśli chcemy wprowadzić jakąś widoczną zmianę na stronie musimy korzystać z własności modelu DOM, a nie atrybutów HTML (to co jest wyświetlanie to przecież obiekty z modelu DOM). Często zdarza się, że własności modelu DOM mają takie same nazwy jak odpowiadające im atrybuty HTML, ale nie zawsze tak jest.

Korzystając z property binding tak naprawdę podłączamy się pod własność modelu DOM i to bardzo dobrze, bo dzięki temu możemy wprowadzić jakieś widoczne zmiany na stronie.

Atrybuty HTML są natomiast niezmienne, mają cały czas wartość taką jaką im nadano na początku, do nich też możemy mieć dostęp z poziomu programu, ale nie jest to nam za bardzo do niczego przydatne.

Podsumowując musimy pamiętać, że property binding w angularze zawsze dotyczy własności modelu DOM, a nie atrybutów HTML. Jeśli więc podłączamy się pod jakąś własność, musimy się upewnić, że taka własność istnieje w modelu DOM, a nie jako atrybut HTML.

Pytanie do opracowania

Jaka jest różnica pomiędzy atrybutami HTML, a własnościami modelu DOM ? Czego dotyczy property binding w angularze - atrybutów HTML czy własności modelu DOM ?

6.8 Style komponentów - metody enkapsulacji

W tym zagadnieniu chodzi o to gdzie i skąd dostępne są style dla naszych komponentów. Istnieją trzy rodzaje trybów:

- *Emulated*
- *Native (Shadow DOM)*
- *None*

Tryb dla komponentu ustawimy w dekoratorze tego komponentu:

```
@Component({
  templateUrl: 'card.html',
  styles: [`
    .card {
      height: 70px;
      width: 100px;
    }
  `],
  encapsulation: ViewEncapsulation.Native
  // encapsulation: ViewEncapsulation.None
  // encapsulation: ViewEncapsulation.Emulated is default
})
```

Dokładna dyskusja tego jak działa każdy z tych trybów dla różnych sposobów zawierania css znajduje się tutaj `https://scotch.io/tutorials/all-the-ways-to-add-css-to-angular-2-components`. My jednak podajmy sobie tylko najważniejsze informacje o każdym z tych trybów:

- Emulated (default) - style z głównego HTML (head) przechodzą do komponentu, ale style definiowane bezpośrednio w komponencie poprzez dekorator *@Component* są dostępne tylko dla komponentu
- Native (shadow DOM) - style z głównego HTML (head) nie są dostępne w komponencie, a style zdefiniowane bezpośrednio w komponencie przez dekorator *@Component* są dostępne tylko dla komponentu (analogia do shadow DOM, który omawialiśmy na początku)
- None - style z komponentu są dostępne również w innych częściach projektu (propagacja wsteczna), możemy ich używać w innych komponentach na stronie

Egzamin

Shadow DOM dla stylów CSS.

6.9 Usługi asynchroniczne

Usługa asynchroniczna w angularze to serwis, który wykonuje jakąś pracę asynchronicznie, np. pobiera dane z serwera w sposób asynchroniczny, wówczas funkcje takiego serwisu zwracają obiekty Promise lub Observable. Przykładowo:

```
import { Observable } from 'rxjs/Observable';
import { of } from 'rxjs/observable/of';

getProducts(): Observable<Product[]> {
  return of(this.products);
}
```

I w kodzie komponentu:

```
// zamiast pisac:
ngOnInit() {
  this.products = this.productService.getProducts();
}

// piszemy:
ngOnInit() {
  this.productService.getProducts().subscribe(
    products => this.products = products
  );
}
```

Pytanie do opracowania

Czym są usługi asynchroniczne w Angular ? Co zwracają ?

6.10 Programowanie reaktywne

Mamy też na wykładzie omówienie od podstaw programowania reaktywnego, jednak pewną podstawową wiedzę przyjąłem tutaj a priori więc zakładam, że pojęcia typu Observable, Observer, Subscription, Operators, Subject są znane i rozumiane.

6.10.1 Tworzenie Observable

Generalnie Observable można utworzyć na mnóstwo różnych sposobów i z wielu różnych obiektów i struktur danych, musimy o prostu skorzystać z wybranego operatora konstruowania Observable.

Jednym z prostszych sposobów tworzenia Observable jest operator *.from*, który może utworzyć Observable z tablicy, obietnicy lub obiektu dowolnego iterable.

```
// RxJS v6+
import { from } from 'rxjs';

//emit array as a sequence of values
const arraySource = from([1, 2, 3, 4, 5]);
//output: 1,2,3,4,5
const subscribe = arraySource.subscribe(val => console.log(val));
```

Innym ze sposobów jest tworzenie Observable przez funkcję *create*, do której podajemy metodą, która ma być wołana podczas wywołania *subscribe*:

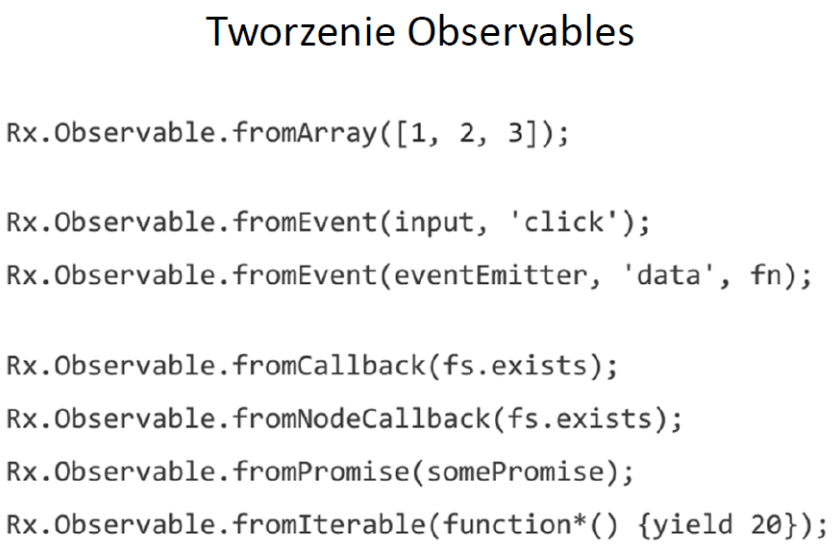
```
// RxJS v6+
import { Observable } from 'rxjs';
/*
  Create an observable that emits 'Hello' and 'World' on
  subscription.
*/
const hello = Observable.create(function(observer) {
  observer.next('Hello');
  observer.next('World');
  observer.complete();
});

//output: 'Hello'...'World'
const subscribe = hello.subscribe(val => console.log(val));
```

Jeszcze inaczej, możemy utworzyć Observable ze zmiennej liczby parametrów i wyemitować każdy z nich po kolei:

```
// RxJS v6+
import { of } from 'rxjs';
//emits any number of provided values in sequence
const source = of(1, 2, 3, 4, 5);
//output: 1,2,3,4,5
const subscribe = source.subscribe(val => console.log(val));
```

Inne:

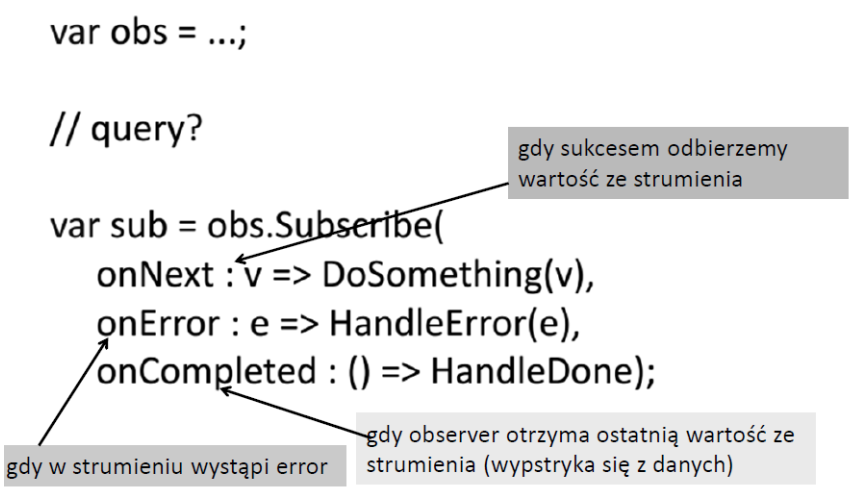


Egzamin

Jak z obiektu tablicy utworzyć Observable ?

6.10.2 Subskrypcja

Do zasubskrybowania się do Observable służy funkcja *subscribe*, która jako parametry przyjmuje funkcje, które są wołane, gdy Observable wyemituje wartość, wystąpi błąd lub na zakończenie emisji:



Częściej jedna korzystamy z arrow functions, przykład z projektu:

```
1    // subscribe to service trips array
2    this.tripsSubscription = this.productsService.tripsObservable.subscribe(
3      res => {
4        this.tripsArray = res;
5        this.updateMinMaxPriceAndAllBookedTrips();
6        // this.onFilteringCriteriaApplied(this.displayCriteria);
7      },
8      error => console.log('Can not load trips array from service')
9    );
```

6.10.3 Hot and cold observable

Cold observable:

- są leniwe - nie emitują zdarzeń jeśli ktoś się do nich nie zasubskrybował
- każda subskrypcja powoduje wywołanie konstruktora i wyemitowanie wartości
- wartości, które emitują zawierają się zazwyczaj w strukturze Observable
- emituje wszystkie możliwe wartości za jednym razem
- każda subskrypcja tworzy nowy strumień
- Przykład: emitowanie zahardkodowanej tablicy

Hot observable:

- nie są leniwe
- emitują wartości nawet jeśli nikt nie utworzył subskrypcji
- wywołanie subscribe nie powoduje za każdym razem wywołanie konstruktora
- zazwyczaj emitują dane, które nie są zawarte w ich strukturze
- przykład: event kliknięcia na stronie

Egzamin

Różnica między hot i cold Observable.

6.10.4 Operatory w RxJS

Wiemy czym są i co robią, przypomnijmy więc tylko jakieś podstawowe przykłady:

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
const doubleNumbers$ = numbers$.map(num => num * 2);
doubleNumbers$.subscribe(num => console.log(num));
// Output:
// 2
// 4
// 6
// 8
// 10
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
const smallNumbers$ = numbers$.filter(num => num < 4);
smallNumbers$.subscribe(num => console.log(num));
// Output:
// 1
// 2
// 3
```

```
const numbers$ = Observable.from([1, 2, 3, 4, 5]);
numbers$
  .filter(num => num > 2)
  .map(num => num * 2)
  .subscribe(num => console.log(num));
// Output:
// 6
// 8
// 10
```

Pytanie do opracowania

Jak zastosować operatory do Observable, podaj przykłady.

6.10.5 Tworzenie własnych operatorów

Możemy również tworzyć własne operatory, operator jest zwykłą funkcją, która przyjmuje jako wejście Observable, subskrybuje się do niego i zwraca również Observable. Przykładowo:

```
const echo = (input$) => {
  return new Observable(observer => {
    input$.subscribe({
      next: val => {
        observer.next(val);
        observer.next(val);
    },
    error: err => observer.error(err),
    complete: () => observer.complete()
  });
  });
}
const numbers$ = Observable.from([1, 2, 3, 4, 5]);

const echoNumbers$ = echo(numbers$);

echoNumbers$.subscribe(num => console.log(num));
```

Pytanie do opracowania

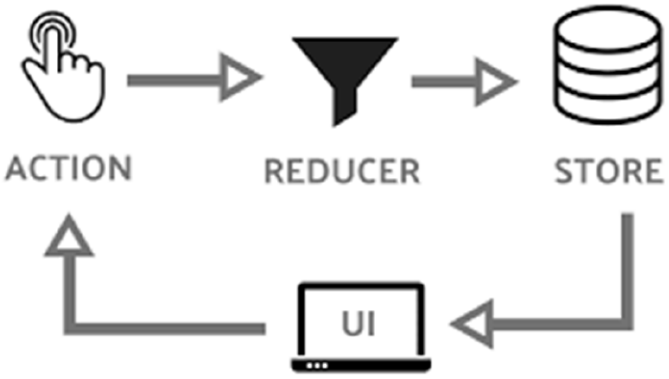
Jak stworzyć własny operator w RxJS ?

6.11 Zarządzanie stanem w angular

Redux jest to wzorzec projektowy/architektura projektowa, który umożliwia łatwe zarządzanie stanem w projekcie. Redux jest w pewnym sensie implementacją architektury Flux chociaż nie jest to w 100% ta sama architektura.

Założenia Redux:

- Aplikacja posiada jeden centralny *Store*, który przechowuje cały stan aplikacji
- *Widok* otrzymuje dane tylko ze *Store*
- *Akcje* to zdarzenia, które powodują, że w aplikacji zmienia się stan, np. kliknięcie w przycisk
- *Akcje* są przekazywane do tzw. *Reducer*, który zamienia je na format odpowiedni do przetrzymywania w *Store*
- Cały obieg danych w aplikacji odbywa się tylko w jedną stronę:



ngRx to biblioteka, która implementuje wzorzec Redux w angularze korzystając przy tym z programowania reaktywnego RxJS. Biblioteka ta zapewnia nam klasy i funkcje do obsługi wzorca Redux.

Egzamin

Co to jest Redux i @*ngRx* ?

6.12 Routing

Routing

- Zadaniem routera w ramach frameworka Angular jest nawigacja między widokami
  - Jest on wydzielonym, opcjonalnym modułem
  - Pojedyncza instancja usługi routera w aplikacji
- Router Angulara bazuje na modelu nawigacji przeglądarki
  - URL wprowadzony w pasku adresu prowadzi do wskazanego widoku
  - Kliknięcie linku w aktualnym widoku powoduje przejście do innego
  - Adres URL może zawierać parametry dla widoku
  - Przyciski Back i Forward w przeglądarce nawigują po historii
- Obszar na stronie, w którym wyświetlane mają być różne komponenty zależnie od stanu routera, wskazuje się znacznikiem <router-outlet></router-outlet>

Konfiguracja routera:

```
import { NgModule }            from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { StudentsComponent }      from './students/students.component';
import { StudentDetailComponent } from './student-detail/student-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/students', pathMatch: 'full' },
  { path: 'students', component: StudentsComponent },
  { path: 'detail/:id', component: StudentDetailComponent }
];

@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

Pytanie do opracowania

Czym jest Router ? Jak go skonfigurować ?

Tak tworzymy w kodzie linki dla routera:

Linki prowadzące do komponentów (we wzorcach HTML komponentów)

```
<nav>
  <a routerLink="/students">Students</a>
  <a routerLink="/about">About</a>
</nav>
```

```
<a routerLink="/detail/{{student.id}}">
   <span>{{student.index}}</span>
</a>
```

A tak możemy wyciągnąć ze ścieżki routera nasze dane:

Dostęp do trasy, która aktywowała bieżący komponent w elemencie <router-outlet>, np. w celu odczytania parametrów zawartych w URL

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
...
@Component({
  selector: 'app-student-detail',
  templateUrl: './student-detail.component.html'
})
export class StudentDetailComponent implements OnInit {

  student: Student;

  constructor( private route: ActivatedRoute, ...) {}

  getStudent(): void {
    const id = +this.route.snapshot.paramMap.get('id');
    this.studentService.getStudent(id)
      .subscribe(student => this.student = student);
  }
```

Egzamin (???) - nie wiem o co chodzi w tym pytaniu

Jak przekazać dane do widoku przed routem ?

7 Ruby on Rails

7.1 Scaffolding (Rusztowanie)

Scaffolding pozwala wygenerować wiele plików na podstawie modelu, który tworzymy. Jako wejście podajemy model danych. Scaffolding sam generuje dla nas:

- plik modelu
- schemat w bazie danych (który musimy jedynie wykonać)
- widoki dla podstawowych operacji CRUD
- routing
- kontroler z akcjami i podłączonymi widokami

Przykładowo:

```
rails generate scaffold author name:string surname:string
```

Pytanie do opracowania

Czym jest rusztowanie ? Jak je utworzyć ? Co zmienia w strukturze projektu ?

7.2 Migracje

Migracje to pliki, które zawierają instrukcje dotyczące zmian w bazie danych (np. dodanie do bazy nowego modelu, dodanie pola do modelu, powiązanie kluczem obcym itp.). Instrukcje takie zawierane są w metodzie *change*.

Przykład migracji, która dodaje pole do modelu:

```
class AddPhoneToTickets < ActiveRecord::Migration[6.0]
  def change
    add_column :tickets, :phone, :string
  end
end
```

Lub takiej, która tworzy relacje pomiędzy modelami (dodajemy do modelu biletu referencję do zdarzenia):

```
class AddEventToTickets < ActiveRecord::Migration[6.0]
  def change
     add_reference :tickets, :event, foreign_key: true
  end
end
```

Pustą migracje tworzymy poleceniem:

```
rails generate migration migration\_name
```

Migrację uruchamiamy poleceniem:

```
rake db:migrate
```

Metody migracji

1. tworzenie tabel - create_table;
2. usuwanie tabel - drop_table;
3. ustawianie indeksów - add_index;
4. usuwanie indeksów - remove_index;
5. dodawanie kolumn w tabelach - add_ccolumn;
6. edycja kolumn w tabelach - change_column;
7. usuwanie kolumn z tabel - remove_column;

Pytanie do opracowania

Czym są migracje ? Napisz migrację dodającą kolumnę dla modelu biletów oraz taką, która wprowadzi relację pomiędzy eventami i biletami (doda dla każdego biletu referencję do eventu). Jak zastosować migracje ?

7.3 Działania na bazie danych

W rails operacje na bazie danych wykonujemy przez wywołania funkcji na klasach modelu, przykładowo:

```
# zwraca pierwszego uzytkownika o imieniu Grzegorz
users = User.find_by(name: 'Grzegorz')
```

```
# zwraca kolekcje wszystkich uzytkownikow
users = User.all
```

```
# lista wszystkich uzytkownikow o imieniu Grzegorz posortowana malejaco pod katem
    daty utworzenia
users = User.where(name: 'Grzegorz').order(created_at: :desc)
```

Pytanie do opracowania

Jak w rails wykonujemy operacje na bazie danych ?

7.4 Asocjacje

Asocjacje dodawane są w klasach modelu, przykładowo:

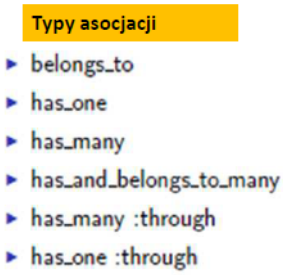
```
class Event < ApplicationRecord
    validates :artist, presence: true
    validates :price_low, presence: true, numericality: true
    validates :price_high, presence: true, numericality: true
    validates :event_date, presence: true
    has_many :tickets

    validate :event_date_not_from_past, :price_low_not_higher_than_price_high

    def event_date_not_from_past
        if event_date < Date.today
            errors.add('Event date', 'can not be date from the past.')
        end
    end

    def price_low_not_higher_than_price_high
        if price_low > price_high
            errors.add('Low price', 'has to be lower than high price.')
```

```
19          end
20      end
21
22  end
```



- *belongs_to* - jeden do jeden z innym modelem, każda instancja należy do jednej instancji innego modelu
- *has_one* - relacja typu jeden do jeden pomiędzy dwoma modelami, każda instancja jednego modelu posiada jedną instancję drugiego
- *has_many* - relacja jeden do wielu, często po drugiej stronie występuje *belongs_to*. Relacja wskazuje, że każda instancja modelu posiada zero i więcej instancji innego modelu
- *has_many :through* - często używana jest do stworzenia relacji wiele-do-wiele z innym modelem. Asocjacja wskazuje, że zadeklarowany model może być powiązany z zerem i większą ilością instancji innego modelu poprzez trzeci model.
- *has_one :through* - ustanawia relację typu jeden-dojeden z innym modelem. Asocjacja ta wskazuje, że deklaracja jednego modelu może być dopasowana do instancji innego modelu poprzez model trzeci.
- *has_and_belongs_to_many* - tworzy bezpośrednią relację typu wiele-do-wiele z innym modelem bez ingerencji modelu pośredniego (jest wykorzystywna stosowna tabela wiążąca)

Egzamin

Jakie są typy powiązań w rails ? Zaprojektować modele dla: Book, Auhtor, Category.

```ruby
class Person < ActiveRecord::Base
end

class Author < Person
  has_many :books
end

class Book < ActiveRecord::Base
  belongs_to :author
  has_and_belongs_to_many :categories
end

class Category < ActiveRecord::Base
  has_and_belongs_to_many :books
end
```

7.5 Walidacja

W rails możemy w łatwy sposób walidować nasze dane. Walidatory dodajemy do naszego modelu. Możemy korzystać z wbudowanych walidatorów jak również pisać swoje własne. Walidacja danych

jest przeprowadzana przy próbie zapisu do bazy danych.

Przykład

```ruby
class Event < ApplicationRecord
    validates :artist, presence: true
    validates :price_low, presence: true, numericality: true
    validates :price_high, presence: true, numericality: true
    validates :event_date, presence: true
    has_many :tickets

    validate :event_date_not_from_past, :price_low_not_higher_than_price_high

    def event_date_not_from_past
        if event_date < Date.today
            errors.add('Event date', 'can not be date from the past.')
        end
    end

    def price_low_not_higher_than_price_high
        if price_low > price_high
            errors.add('Low price', 'has to be lower than high price.')
        end
    end

end
```

Pytanie do opracowania

Jak realizowane są walidatory w Ruby on Rails ? Napisz swój własny walidator.

7.6 Zadanie z tablicami

Egzamin

Napisac kod w rails który stworzy obiekt klasy Student (janek, 24 lat) i doda go do tablicy, a nastepnie kod który pobierze wszystkich studentów majacych wiecej niz 25 lat, posortowanych malejąco.

```ruby
@students = []
@student = Student.new
@student.name = "Janek"
@student.age = 24
@students << student
@under = @students
  .select { |student| student.age > 25 }
  .sort_by { |student| student.age }
```

8 Indeks pytań

8.1 Egzamin 2019

Egzamin

Nowe znaczniki i opisać szczegółowo dwa z nich.

Egzamin

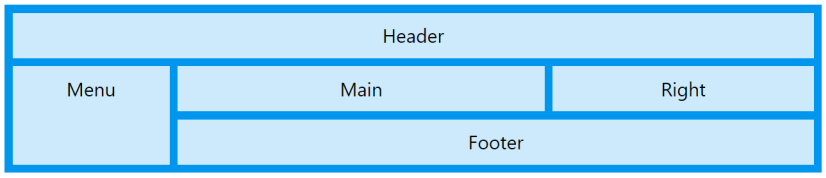
Shadow DOM

Egzamin

Co to jest prototype ?

Egzamin

Utworzyć layout taki jak poniżej przy użyciu CSS grid.



Egzamin

Różnice pomiędzy *script*, *script async*, *script deffer*

Egzamin

Funkcja fufu która wypisze jaki przycisk został wciśniety i następnie zatrzyma bąbelkowanie. Zarejestrować funkcje fufu do eventu wciśniecia przycisku myszy.

Egzamin

Różnica między spread operator a rest parameters.

Egzamin

Napisać funkcję w JavaScript, która usunie duplikaty z tablicy.

Egzamin

W czystym Node.js napisać serwer który będzie serwował Country.json

Egzamin

@HostBinding i *@HostListner* - opisać

Egzamin (???) Chyba chodzi o to aby użyć dyrektyw i wypisać w szablonie

W angularze wypisać tablicę krajów ({country: "POland", population: "38mlln",...), z tym że kazdy kraj w osobnym wierszu, poprzedzonym indexem kraju i jeśli populacja jest mniejsza niż 10mln to wypisać "Mały Kraj"

Egzamin

Hooki komponentu Angulara. Do którego hooka podpiąłbyś odłączenie od subskrypcji ?

Egzamin

Jak utworzyć usługę singleton a jak instancję usługi dla komponentu w Angular ?

Egzamin

Shadow DOM dla stylów CSS.

Egzamin

Jak z obiektu tablicy utworzyć Observable ?

Egzamin

Różnica między hot i cold Observable.

Egzamin

Co to jest Redux i $@ngRx$?

Egzamin

Jakie są typy powiązań w rails ? Zaprojektować modele dla: Book, Auhtor, Category.

Egzamin

Napisac kod w rails który stworzy obiekt klasy Student (janek, 24 lat) i doda go do tablicy, a nastepnie kod który pobierze wszystkich studentów majacych wiecej niz 25 lat, posortowanych malejąco.

Egzamin (???) - nie wiem o co chodzi w tym pytaniu

Jak przekazać dane do widoku przed routem ?

8.2 Opracowanie

Pytanie do opracowania

Jakie nowe elementy wprowadza ES6 do języka JavaScript ?

Pytanie do opracowania

Czym różni się var, let i const ?

Pytanie do opracowania

Czym jest hoisting ? Kiedy działa a kiedy nie ?

Pytanie do opracowania

Jak napisać arrow function ?

Pytanie do opracowania

Czym są parametry domyśle w JS ?

Pytanie do opracowania

Czym są wyrażenie interpolowane w JS ?

Pytanie do opracowania

Jak działa destrukturyzacja obiektów w JS ?

Pytanie do opracowania

Czym różni się pętla for of od for in ?

Pytanie do opracowania

Jak działają funkcje map, filter, reduce ?

Pytanie do opracowania

Jakie znasz sposoby asynchronicznego przetwarzania w JavaScript ?

Pytanie do opracowania

Omów czym są generatory w JavaScript.

Pytanie do opracowania

Omów działanie obietnic, jak przy ich pomocy realizować zapytania asynchroniczne ?

Pytanie do opracowania

Do czego służą słowa async, await dodane w ES7 ? Jak realizować przy ich pomocy zapytania asynchroniczne ?

Pytanie do opracowania

Czym jest TypeScript ? Jakie nowe elementy wprowadza ?

Pytanie do opracowania

Jak zainstalować moduł przy użyciu npm ?

Pytanie do opracowania

Jak przy użyciu angular CLI:

- utworzyć nowy projekt
- uruchomić serwer
- wygenerować nowy komponent
- wygenerować nowy serwis

Pytanie do opracowania

Wymień 8 głównych bloków konstrukcyjnych angulara.

Pytanie do opracowania

Omów moduły w Angularze, do czego służą, jak je tworzymy. Czy moduły angulara są tym samym co moduły JavaScript ?

Pytanie do opracowania

Czym są komponenty w Angular ?

Pytanie do opracowania

Do czego służą metadane w Angular ?

Pytanie do opracowania

Czym jest wiązanie danych w Angular ? Wymień i opisz cztery mechanizmy stosowane w wiązaniu danych.

Pytanie do opracowania

Czym są dyrektywy ? Do czego służą ? Wymień rodzaje dyrektyw i podaj przykłady.

Pytanie do opracowania

Czym są potoki w Angular ? Do czego służą ?

Pytanie do opracowania

Czym jest cykl życia komponentu ?

Pytanie do opracowania

Jakie znasz sposoby komunikacji pomiędzy komponentami w angular ?

Pytanie do opracowania

Jaka jest różnica pomiędzy atrybutami HTML, a własnościami modelu DOM ? Czego dotyczy property binding w angularze - atrybutów HTML czy własności modelu DOM ?

Pytanie do opracowania

Czym są usługi asynchroniczne w Angular ? Co zwracają ?

Pytanie do opracowania

Jak zastosować operatory do Observable, podaj przykłady.

Pytanie do opracowania

Jak stworzyć własny operator w RxJS ?

Pytanie do opracowania

Czym jest Router ? Jak go skonfigurować ?

Pytanie do opracowania

Czym jest rusztowanie ? Jak je utworzyć ? Co zmienia w strukturze projektu ?

Pytanie do opracowania

Czym są migracje ? Napisz migrację dodającą kolumnę dla modelu biletów oraz taką, która wprowadzi relację pomiędzy eventami i biletami (doda dla każdego biletu referencję do eventu). Jak zastosować migracje ?

Pytanie do opracowania

Jak w rails wykonujemy operacje na bazie danych ?

Pytanie do opracowania

Jak realizowane są walidatory w Ruby on Rails ? Napisz swój własny walidator.