Rest Parameter, Spread Operator and Arrow Functions

Level 2 – Section 2

Issues With the arguments Object

The *arguments* object is a **built-in, Array-like** object that corresponds to the arguments of a function. Here's why relying on this object to read arguments is **not ideal:**

```
function displayTags(){
  for(let i in arguments){
    let tag = arguments[i];
    _addToTopic(tag);
  }
}
```

Hard to tell which parameters this function expects to be called with

Where did this come from?!

If we add an argument...

```
function displayTags(targetElement){

  let target = _findElement(targetElement);

  for(let i in arguments){
    let tag = arguments[i];
    _addToTopic(target, tag);
  }
}
```

...we'll break the loop, since the first argument is no longer a tag

Using Rest Parameters

ES2015

The new **rest parameter** syntax allows us to represent an indefinite number of arguments as an **Array.** This way, changes to function signature are **less likely to break code.**

```
function displayTags(...tags){
  for(let i in tags){
    let tag = tags[i];
    _addToTopic(tag);
  }
}
```

These 3 dots are part of the syntax

tags is an Array object

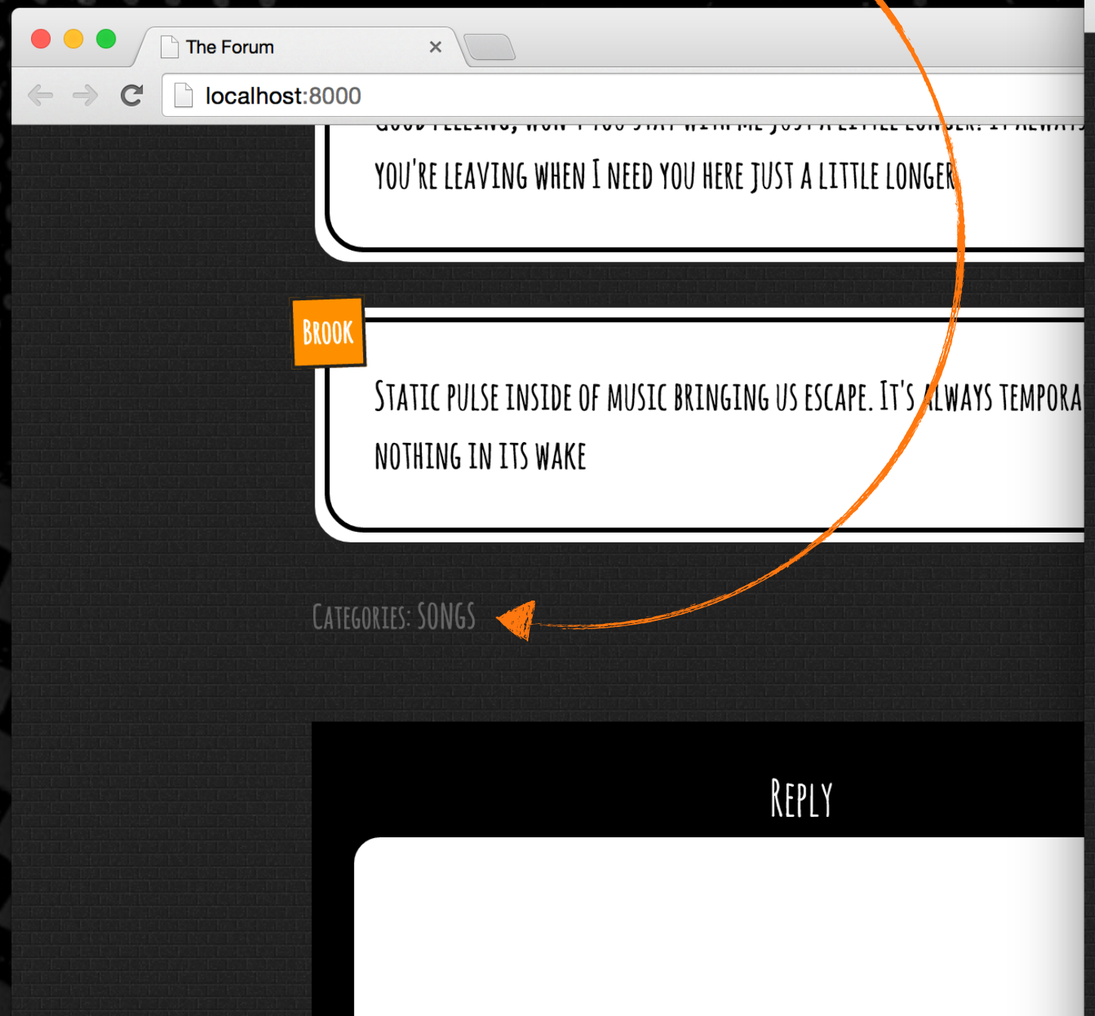
Must always go last

```
function displayTags(targetElement, ...tags){

  let target = _findElement(targetElement);

  for(let i in tags){
    let tag = tags[i];
    _addToTopic(target, tag);
  }
}
```

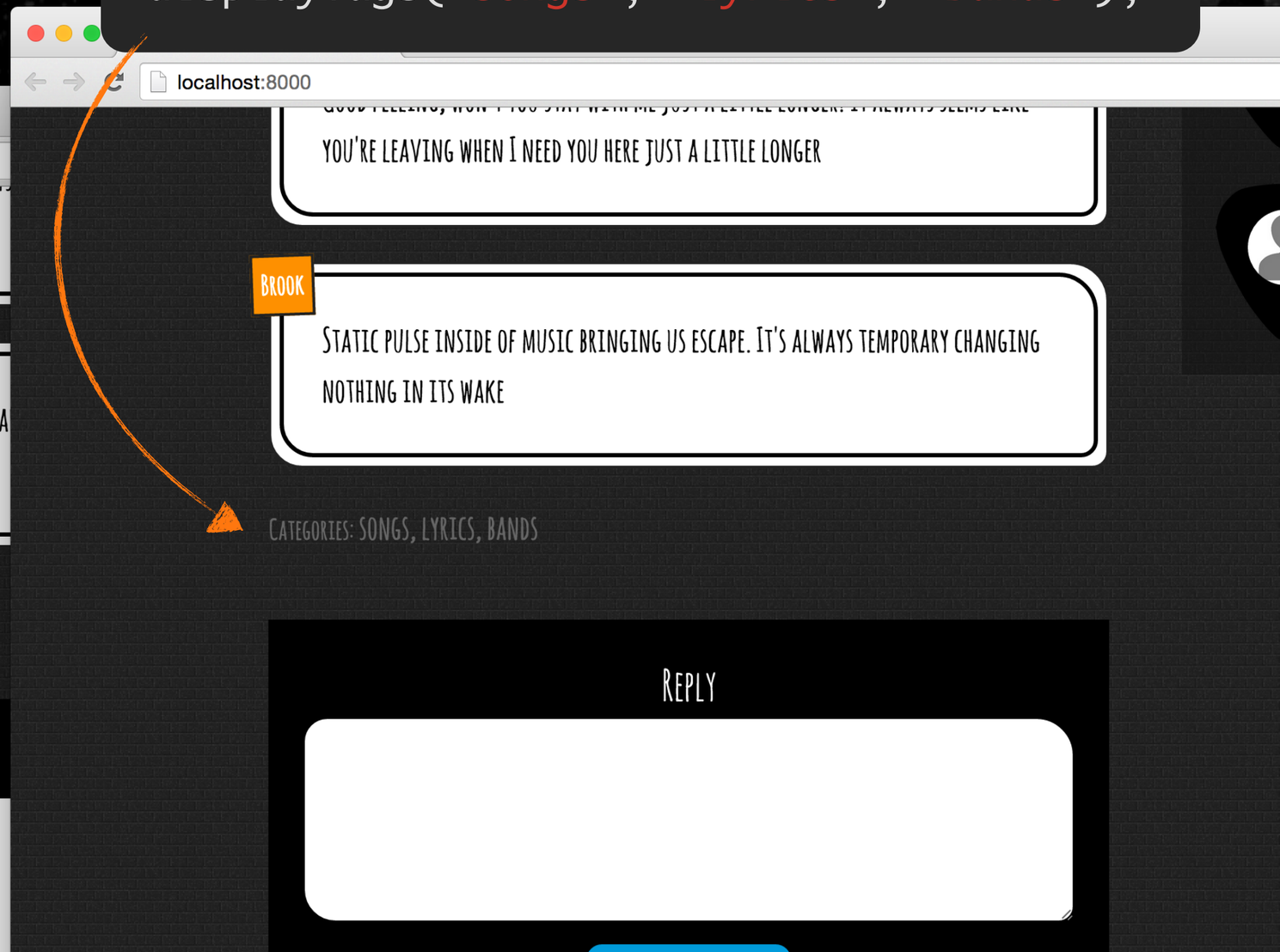
Not affected by changes to function signature

Seeing displayTags in Action

```
displayTags("songs");
```




```
displayTags("songs", "lyrics", "bands");
```



Splitting Arrays Into Individual Arguments

We need a way to convert an **Array** into individual arguments upon a **function call**.

```
getRequest("/topics/17/tags", function(data){
  let tags = data.tags;
  displayTags(tags);
})
```

tags is an Array, e.g., ["programming", "web", "HTML"] ...

...but displayTags expects to be called with individual arguments, like this:

```
displayTags("programming");
```

```
displayTags("programming", "javascript");
```

How can we convert* Arrays *into individual elements on a function call ?

Using the Spread Operator

ES2015

The spread operator allows us to **split an Array** argument into **individual elements**.

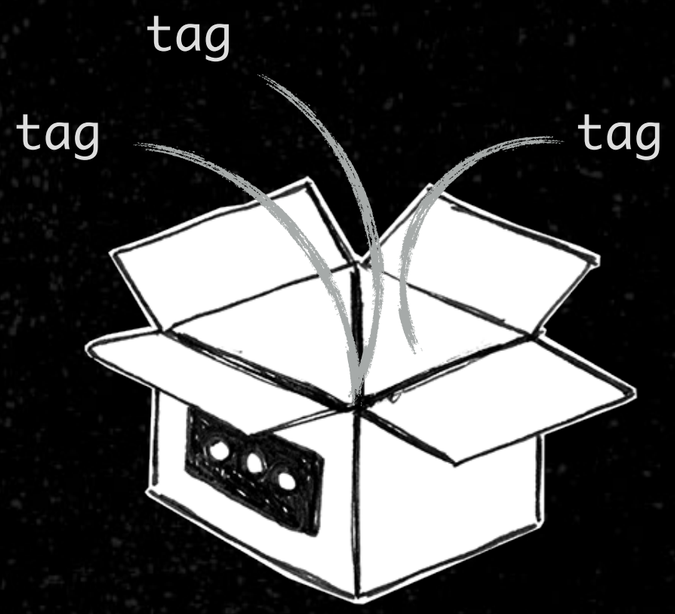
```
getRequest("/topics/17/tags", function(data){
  let tags = data.tags;
  displayTags(...tags);
})
```

The displayTags function is now receiving individual arguments, not an Array

Three dots are part of the syntax

Same as doing this

```
displayTags(tag, tag, tag);
```



Rest and Spread look the same

ES2015

Rest parameters and the spread operator **look the same**, but the former is used in function **definitions** and the later in function **invocations**.

Rest Parameters

```
function displaytags(...tags){
  for(let i in tags){
    let tag = tags[i];
    _addToTopic(target, tag);
  }
}
```

Function definition

vs.

Function invocation

Spread Operator

```
getRequest("/topics/17/tags", function(data){
  let tags = data.tags;
  displayTags(...tags);
})
```

From Functions to Objects

JavaScript objects can help us with the **encapsulation, organization**, and **testability** of our code.

```
getRequest("/topics/17/tags", function(data){
  let tags = data.tags;
  displayTags(...tags);
})
```

Functions like getRequest and displayTags should not be exposed to caller code

We want to convert code like this...

...into code like this

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```

Let's see how we can implement our* TagComponent *function!

Creating TagComponent

The *TagComponent* object **encapsulates** the code for fetching tags and adding them to a page.

```
function TagComponent(target, urlPath){
  this.targetElement = target;
  this.urlPath       = urlPath;
}

TagComponent.prototype.render = function(){

  getRequest(this.urlPath, function(data){
    //...
  });
}
```

Properties set on the constructor function...

...can be accessed from other instance methods

Passing target element and the URL path as arguments

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```

Issues With Scope in Callback Functions

Anonymous functions passed as callbacks to other functions create **their own scope**.

```
function TagComponent(target, urlPath){
  this.targetElement = target;
  this.urlPath       = urlPath;
}

TagComponent.prototype.render = function(){

  getRequest(this.urlPath, function(data){
    let tags = data.tags;
    displayTags(this.targetElement, ...tags);
  });
}
```

The scope of the TagComponent object...

...is not the same as...

...the scope of the anonymous function

Returns undefined

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```

Using Arrow Functions to Preserve Scope

ES2015

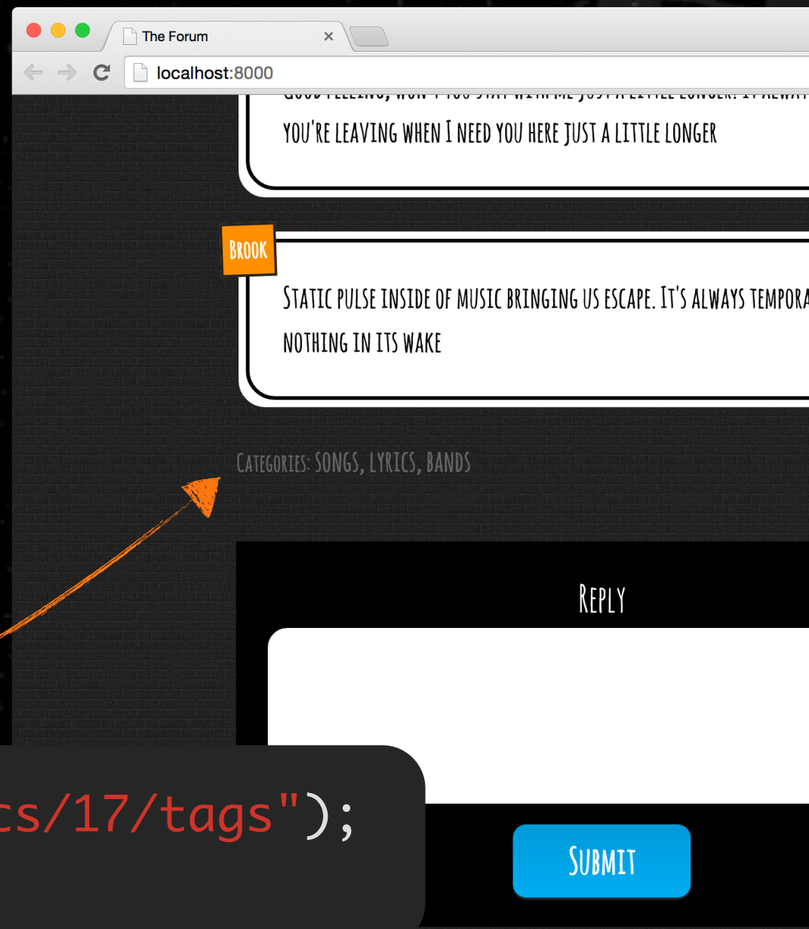
Arrow functions bind to the scope of where they are **defined**, not where they are used.

(also known as **lexical binding)**

```
function TagComponent(target, urlPath){
  this.targetElement = target;
  this.urlPath       = urlPath;
}

TagComponent.prototype.render = function(){

  getRequest(this.urlPath, (data) => {
    let tags = data.tags;
    displayTags(this.targetElement, ...tags);
  });
}
```

Arrow functions bind to the lexical scope

this now properly refers to the TagComponent object




```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```