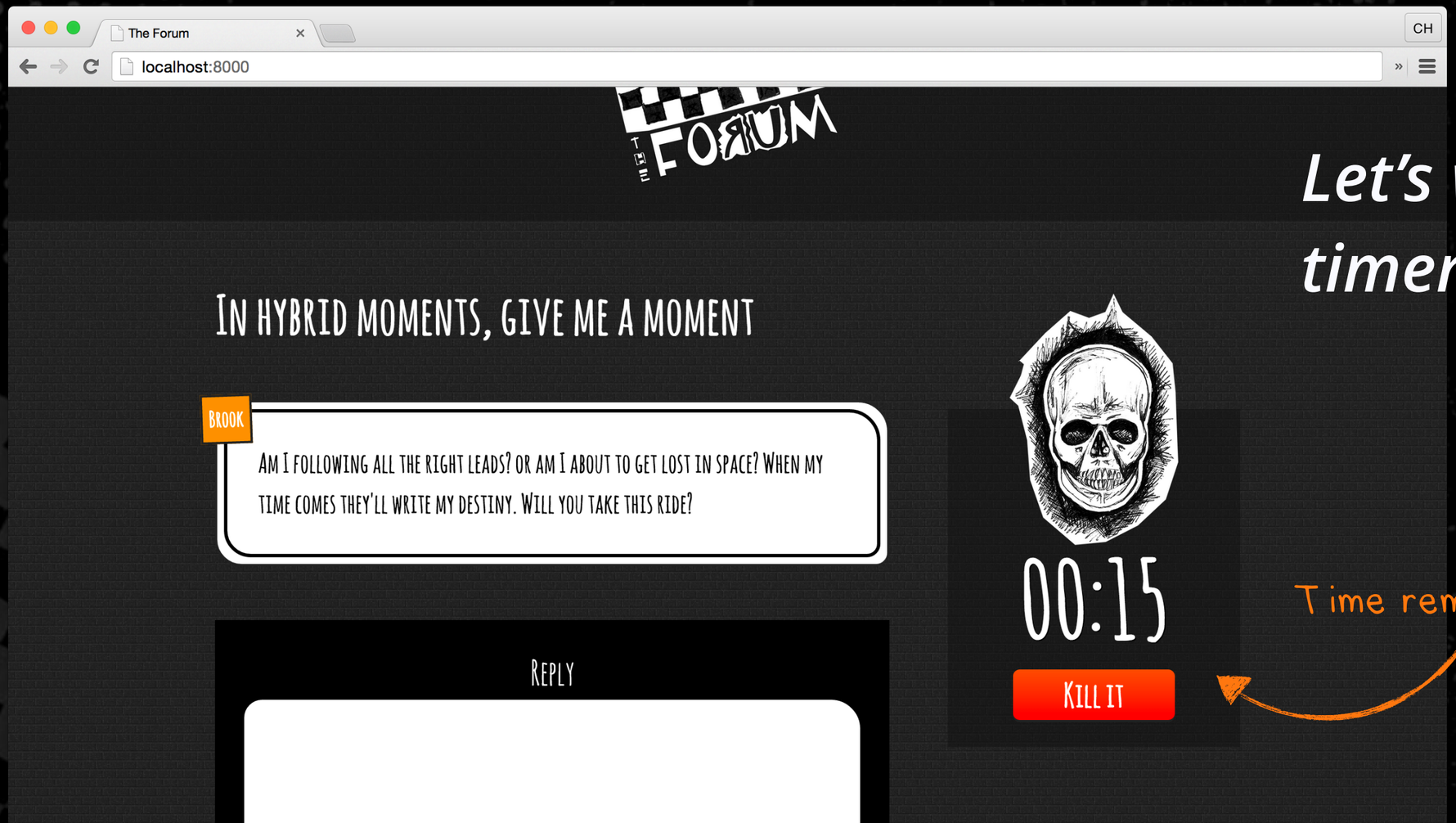
Object.assign

Level 3 – Section 2

Adding a Countdown Timer to Our Forum

The **countdown timer** displays the time left for users to undo their posts after they've been created. Once the time is up, they cannot undo it anymore.



Let's write a countdown timer function!

Time remaining to undo post

Writing More Flexible Functions

In order to cater to different applications and domains, our *countdownTimer* function needs to be called in **many different ways.**

As simple as this...

Two arguments only

```
countdownTimer($('.btn-undo'), 60);
```

...and as complicated as this

```
countdownTimer($('.btn-undo'), 60,  {  container: '.new-post-options' });

countdownTimer($('.btn-undo'), 3,   {  container: '.new-post-options',
                                       timeUnit:  'minutes',
                                       timeoutClass: '.time-is-up' });
```

Custom options will vary according to each application

Using Too Many Arguments Is Bad

For functions that need to be used across different applications, it's okay to accept an **options object** instead of using named parameters

```
function countdownTimer(target, timeLeft,
  { container, timeUnit, clonedDataAttribute,
    timeoutClass, timeoutSoonClass, timeoutSoonSeconds
  } = {}){

  //...
}
```

Too many named arguments make this function harder to read

(Named arguments are okay, but for a different purpose)

```
function countdownTimer(target, timeLeft, options = {}){
  //...
}
```

Easier to customize to different applications

Using Local Values and || Is Bad for Defaults

Some options might not be specified by the caller, so we need to have **default values**.

```
function countdownTimer(target, timeLeft, options = {}){

  let container = options.container || ".timer-display";
  let timeUnit = options.timeUnit || "seconds";
  let clonedDataAttribute = options.clonedDataAttribute || "cloned";
  let timeoutClass = options.timeoutClass || ".is-timeout";
  let timeoutSoonClass = options.timeoutSoonClass || ".is-timeout-soon";
  let timeoutSoonTime = options.timeoutSoonSeconds || 10;

  //...
}
```

Default strings and numbers are all over the place... Yikes!

Using a Local Object to Group Defaults

Using a local object to group **default** values for user options is a common practice and can help write more **idiomatic JavaScript**.

```
function countdownTimer(target, timeLeft, options = {}){

  let defaults = {
    container:              ".timer-display",
    timeUnit:               "seconds",
    clonedDataAttribute:    "cloned",
    timeoutClass:           ".is-timeout",
    timeoutSoonClass:       ".is-timeout-soon",
    timeoutSoonTime:        10
  };

  //...
}
```

Looks much better

Merging Values Into a Combined Variable

We want to merge *options* and *defaults.* Upon duplicate properties, those from *options* **must override properties** from *defaults*.

3. ....and the result stored here.

```
let settings
{
  container: ...,
  timeUnit: ...,
  clonedDataAttribute: ...,
  timeoutClass: ...,
  ...
}
```

=

1. Default properties declared here...

```
let defaults
{
  container: ...,
  timeUnit: ...,
  clonedDataAttribute: ...,
  timeoutClass: ...,
  ...
}
```

+

2. ...will be merged with these...

```
options
{
  timeUnit: ...,
  timeoutClass: ...,
}
```

Values passed for timeUnit and timeoutClass will override properties on defaults object

Merging Values With Object.assign

ES2015

The *Object.assign* method copies properties from one or more **source objects** to a **target object** specified as the very first argument.

```
function countdownTimer(target, timeLeft, options = {}){

  let defaults = {
    //...
  };

  let settings = Object.assign({}, defaults, options);
  //...
}
```

Merged properties from defaults and options

Target object is modified and used as return value

Source objects remain unchanged

Merging Objects With Duplicate Properties

In case of **duplicate properties** on source objects, the value from the **last object** on the chain always prevails.

```
function countdownTimer(target, timeLeft, options = {}){

  let defaults = {
    //...
  };

  let settings = Object.assign({}, defaults, options, options2, options3);

}
```

Duplicate properties from options3 override those on options2, which override those on options, etc.

Using Object.assign

There are a couple incorrect ways we might see *Object.assign* being used.

```
Object.assign(defaults, options);
```

defaults is mutated, so we can't go back and access original default values after the merge

```
let settings = Object.assign({}, defaults, options);
```

Can access original default values and looks functional

```
let settings = {};
Object.assign(settings, defaults, options);
console.log( settings.user );
```

Not reading return value

Default values are not changed, but settings is passed as a reference

Reading Initial Default Values

Preserving the original default values gives us the ability to compare them with the options passed and act accordingly when necessary.

```
function countdownTimer(target, timeLeft, options = {}){

  let defaults = {
    //...
  };

  let settings = Object.assign({}, defaults, options);

  if(settings.timeUnit !== defaults.timeUnit){
    _conversionFunction(timeLeft, settings.timeUnit)
  }
}
```

Runs when value passed as argument for timeUnit is different than the original value

Object.assign in Practice

Let's run *countdownTimer()* passing the value for *container* as argument...

```
countdownTimer($('.btn-undo'), 60, { container: '.new-post-options' });
```

...and using the default value for everything else.

```
function countdownTimer(target, timeLeft, options = {}){
  let defaults = {
    container: ".timer-display",
    timeUnit:  "seconds",
    //...
  };

  let settings = Object.assign({}, defaults, options);

  console.log( settings.container );
  console.log( settings.timeUnit );
}
```

```
> .new-post-options
> seconds
```

C