

# EVENTS, EVENT EMITTERS, HTTP & LONG POLLING

---

*Building real-time software*

```
var userTweets = new EventEmitter();
```

```
// Elsewhere in the program . . .
```

```
userTweets.on('newTweet', function (tweet) {  
    console.log(tweet);  
});
```

```
// Elsewhere in the program . . .
```

```
userTweets.emit('newTweet', {  
    text: 'Check out this fruit I ate'  
});
```

```
var userTweets = new EventEmitter();
```

```
// Elsewhere in the program . . .
```

```
userTweets.on('newTweet', function (tweet) {  
    console.log(tweet);  
});
```

```
// Elsewhere in the program . . .
```

```
userTweets.emit('newTweet', {  
    text: 'Check out this fruit I ate'  
});
```

```
var userTweets = new EventEmitter();
```

```
// Elsewhere in the program . . .
```

```
userTweets.on( 'newTweet', function (tweet) {  
    console.log(tweet);  
})
```

```
// Elsewhere in the program . . .
```

```
userTweets.emit( 'newTweet', {  
    text: 'Check out this fruit I ate'  
})
```



```
var userTweets = new EventEmitter();
```

```
// Elsewhere in the program . . .
```

```
userTweets.on('newTweet', function (tweet) {  
    console.log(tweet);  
});
```

```
// Elsewhere in the program . . .
```

```
userTweets.emit('newTweet', {  
    text: 'Check out this fruit I ate'  
});
```

```
var userTweets = new EventEmitter();
```

```
// Elsewhere in the program . . .
```

```
userTweets.on('newTweet', function (tweet) {  
    console.log(tweet);  
});
```

```
// Elsewhere in the program . . .
```

```
userTweets.emit('newTweet', {  
    text: 'Check out this fruit I ate'  
});
```

# EVENT EMITTERS

- **Objects that can “emit” specific events with a payload to any amount of registered listeners**
- **An instance of the “observer/observable” a.k.a “pub/sub” pattern**
- **Feels at-home in an *event*-driven environment**



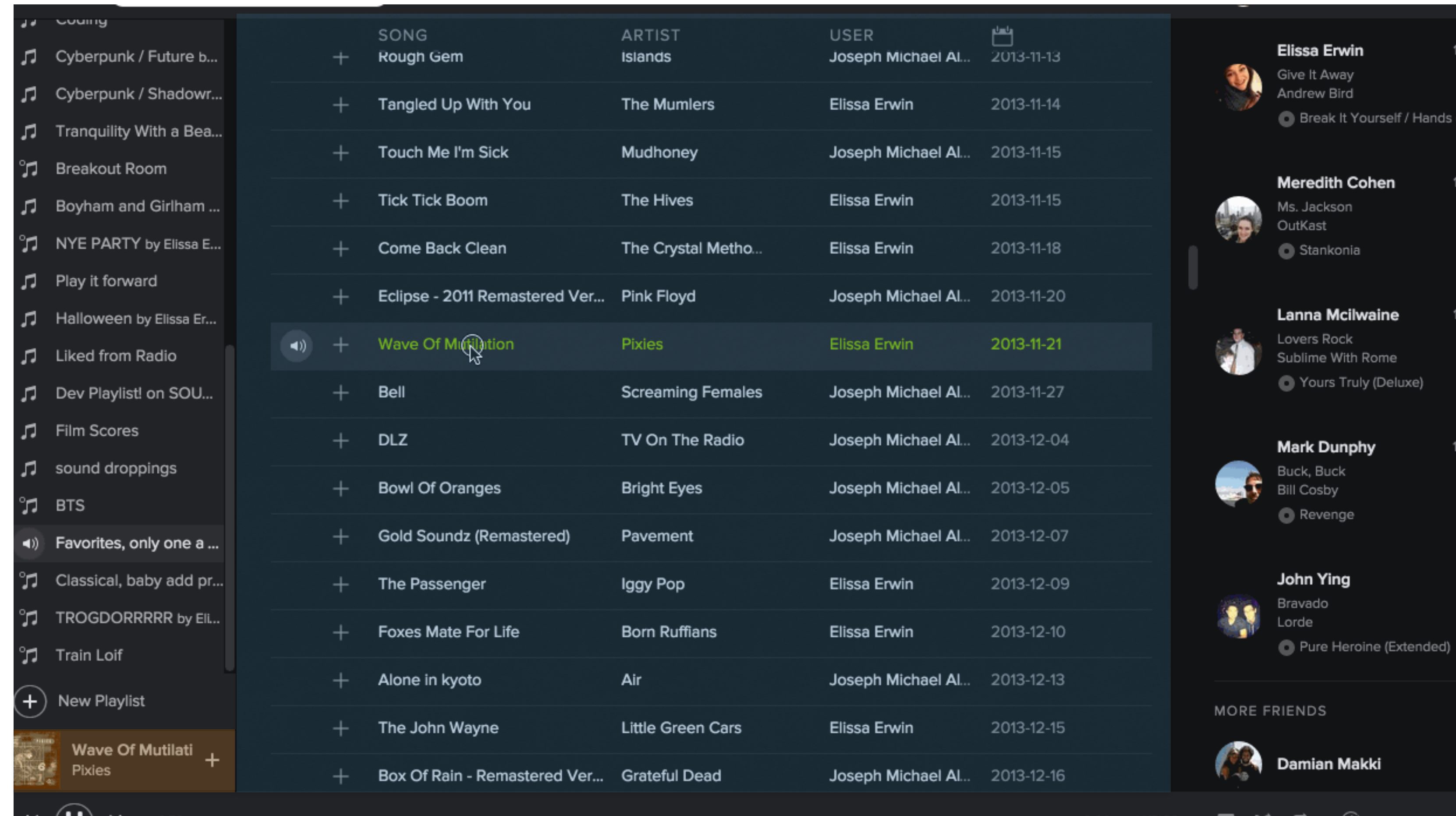
# PRACTICAL USES

- Connect two decoupled parts of an application

```
var currentTrack = new EventEmitter();
```

```
currentTrack.emit('changeTrack', newTrack);
```

```
currentTrack.on('changeTrack', function (newTrack) {  
  // Display new track!  
});
```





# PRACTICAL USES

- Represent multiple asynchronous events on a single entity.

```
var upload = uploadFile();
```

```
upload.on('error', function (e) {  
  e.message; // World exploded!  
});
```

```
upload.on('progress', function (percentage) {  
  setProgressOnBar(percentage);  
});
```

```
upload.on('complete', function (fileUrl, totalUploadTime) {  
  
});
```

# ALL OVER NODE

- **server.on('request')**
- **request.on('data') / request.on('end')**
- **process.stdin.on('data')**
- **db.on('connection')**
- **Streams**

# HTTP, PART 2

---

*Sequels are always worse than the original*



# WHAT WE KNOW ABOUT HTTP

- **A client makes a “request” to a server**
- **Server receives this “request” and generates a “response”**
- **One request, one response: them’s the rules**
- **Requests can include a body (payload)**
- **Responses can include a body (payload)**

# The New York Times



**FIFA WORLD CUP**  
**Brasil**

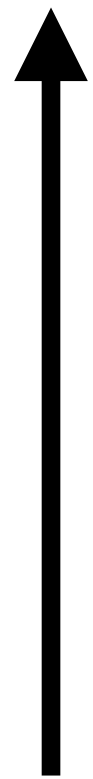
# LIVE WORLD CUP COVERAGE

- **A user visits a web page**
- **This web page has a live updating list of game coverage (“events”) provided by New York Times commentator (“Brazil receives yellow card”/“Germany scores goal”)**
- **When the event line is submitted by the commentator, it should immediately display to the user**



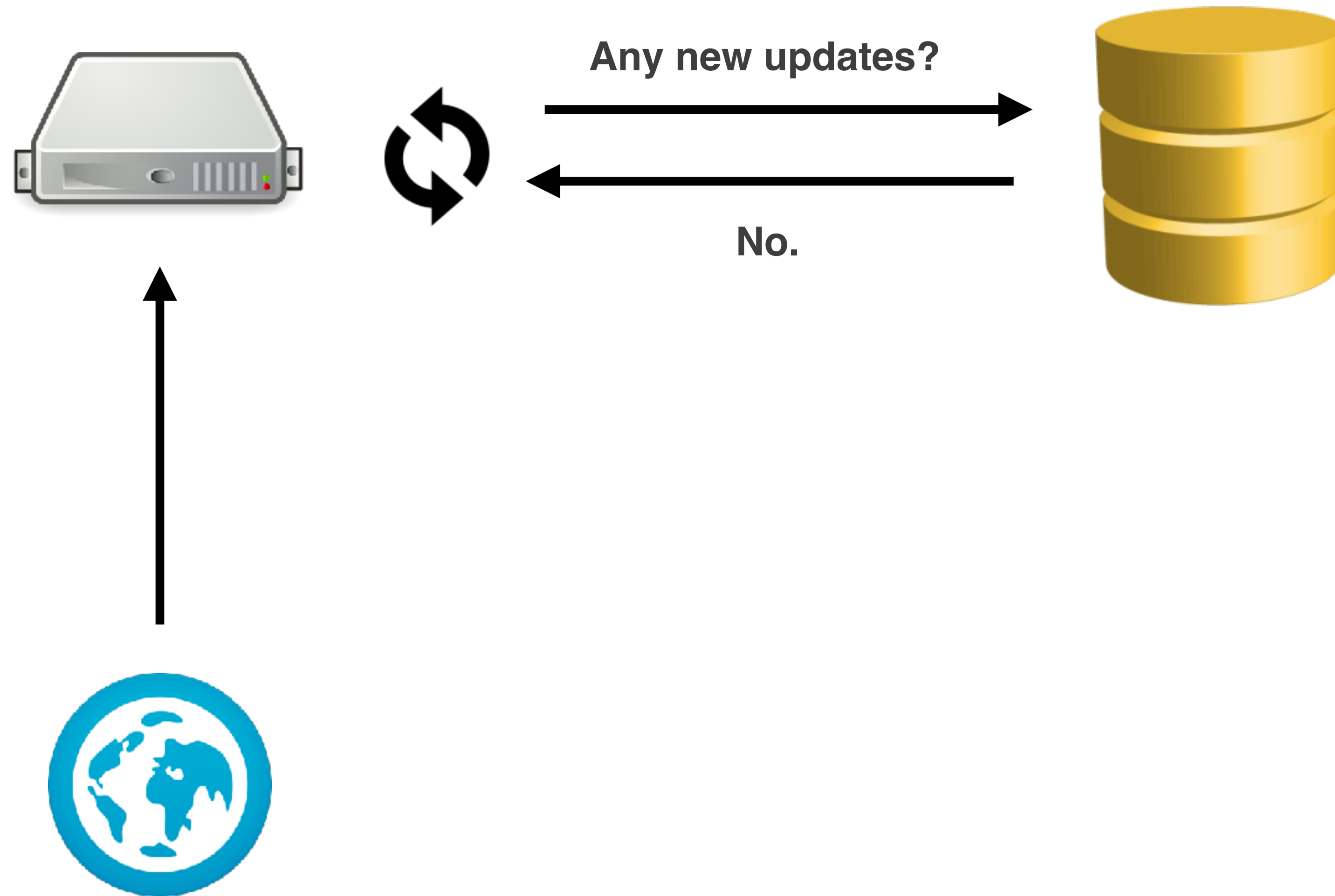


# HTTP LONG POLLING



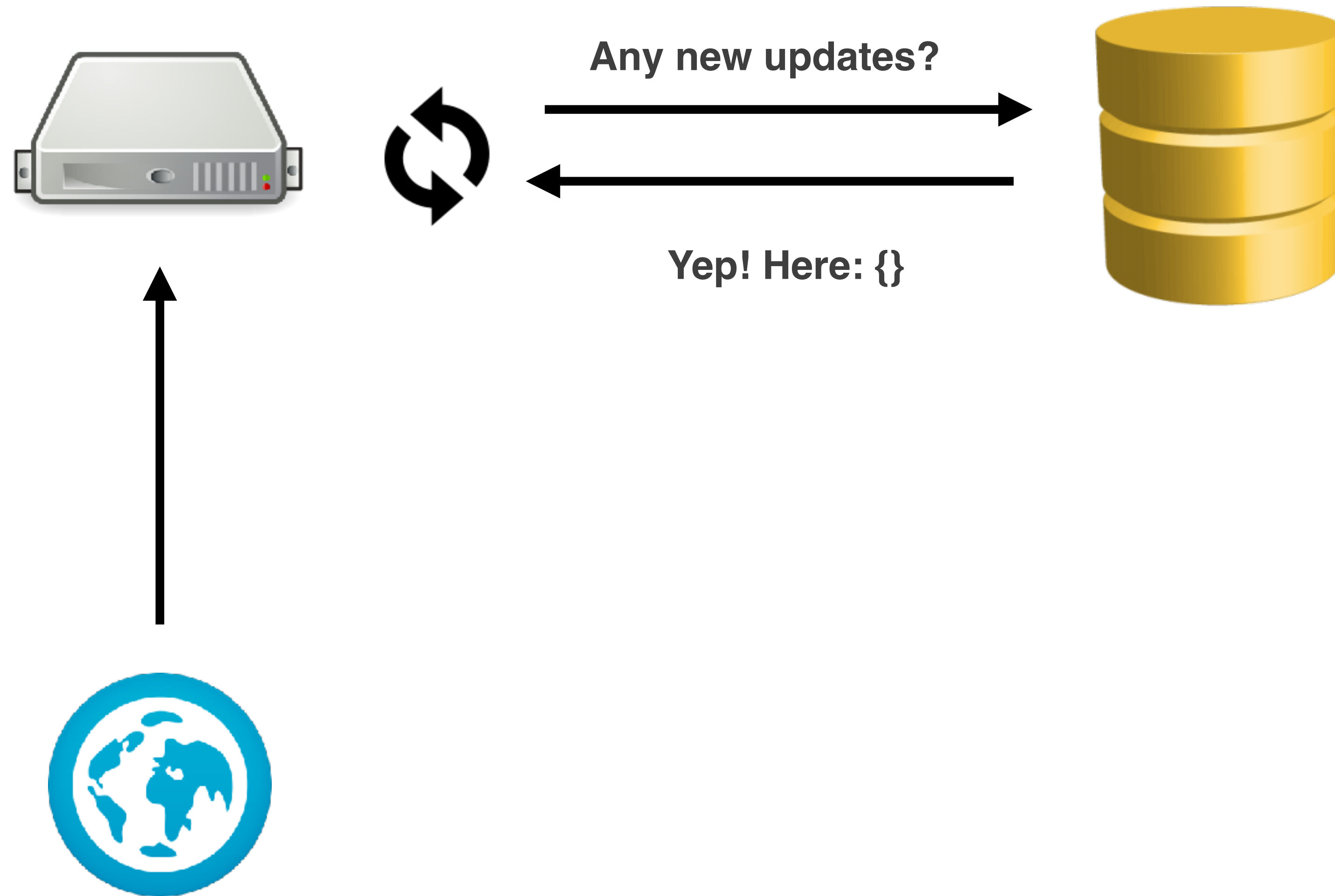


# HTTP LONG POLLING





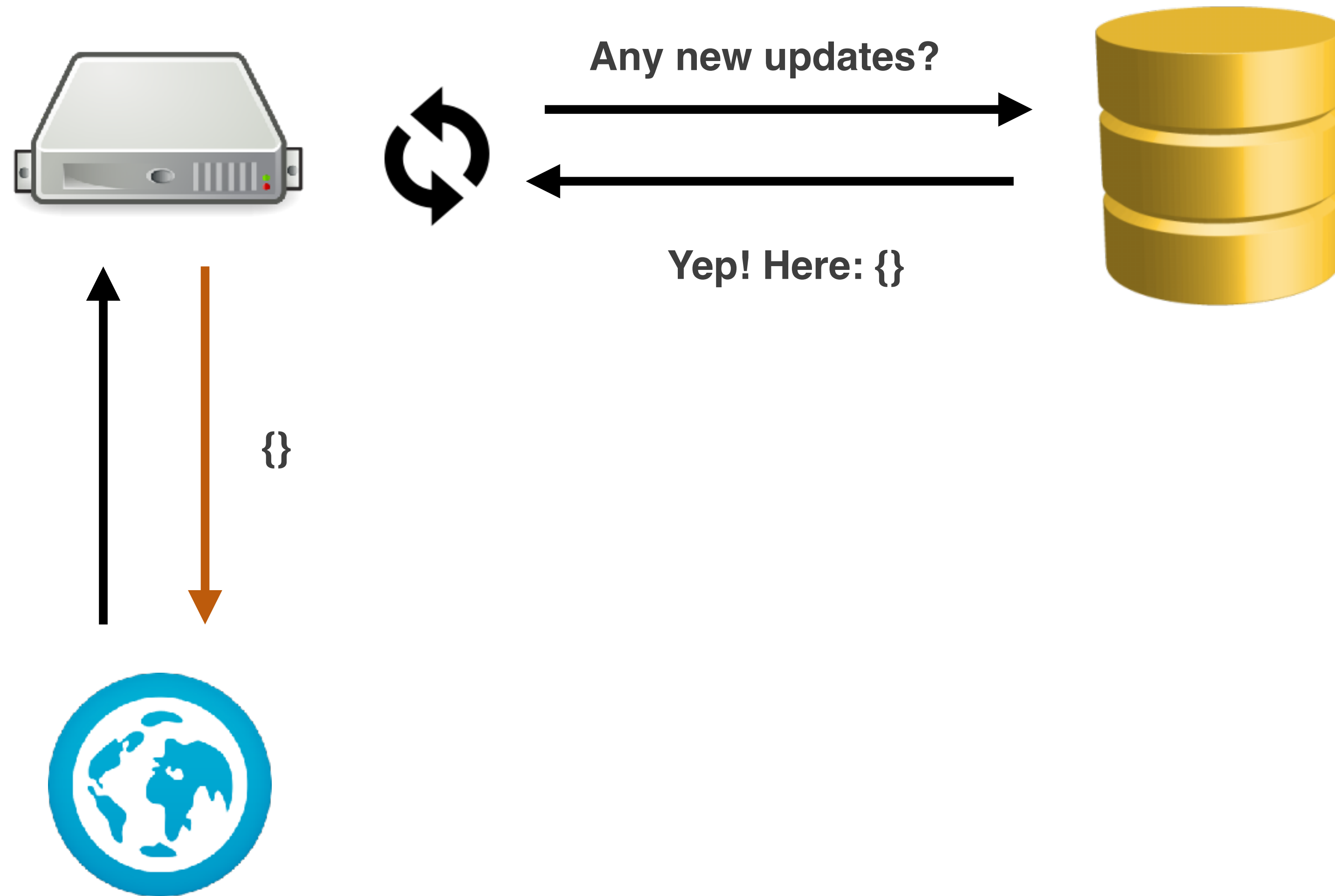
# HTTP LONG POLLING





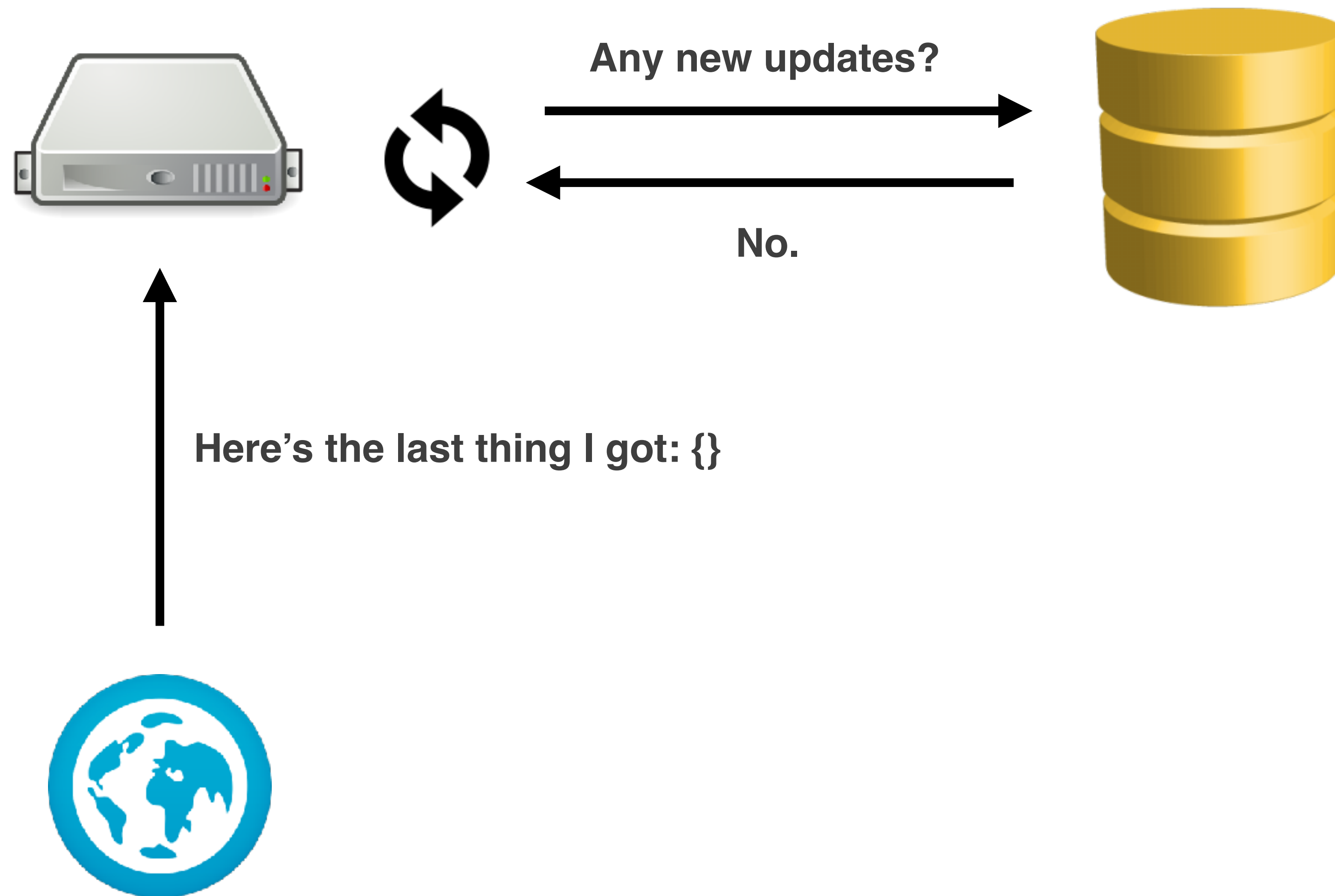


# HTTP LONG POLLING





# HTTP LONG POLLING



# HTTP IS A REQUEST/RESPONSE PROTOCOL

- Clients must send a *request* before the server can issue a *response*
- There is no way for the server to *push* data to the client without an outstanding request
- No live updates without long polling 😓