

# AJAX & REST API

# JSON-SERVER

Vad är det egentligen?

## JSON-SERVER

*Get a full fake REST API with zero coding in less  
than 30 seconds (seriously)*

ReST API?

*REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.*

Roy Fielding

"Architectural Styles and the Design of Network-based Software  
Architectures"

Dissertation @ UNIVERSITY OF CALIFORNIA

There are six guiding constraints that define a RESTful system

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs

REST är varken en standard eller ett protokoll

*In simplest words, in the REST architectural style,  
data and functionality are considered resources  
and are accessed using Uniform Resource  
Identifiers (URIs).*

# STATELESS

*The client-server communication is constrained  
by no client context being stored on the server  
between requests.*

Alla **requests** sker isolerade från varandra.

Servern behöver inte ha någon information från dig: användaren.



# REPRESENTATIONAL

En resurs kan identifieras med en **URL**. Resursen som användaren ser behöver dock inte vara av samma format som lagras på servern.

På samma sätt som du hämtar informationen kan du även manipulera informationen och lagra ny information. All metadata för att göra detta har du redan.

*A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).*

Vi vet hur vi hämtar alla filmer

```
GET
```

```
http://localhost:3000/movies
```

Vi vet också nu hur vi får en film

```
GET
```

```
http://localhost:3000/movies/1
```

Med den informationen vet vi även hur vi skapar och uppdaterar filmer

POST

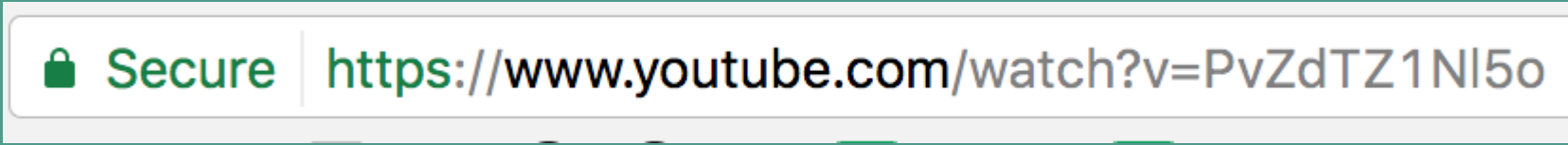
`http://localhost:3000/movies`

PATCH

`http://localhost:3000/movies/1`

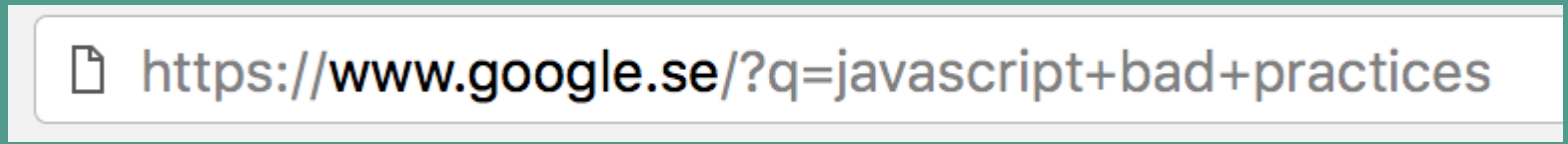
# URL QUERIES

Om APIt är kodat rätt så kan vi använda **URL** queries



A screenshot of a web browser's address bar. On the left, there is a green padlock icon followed by the word "Secure" in green. To the right of a vertical line, the URL "https://www.youtube.com/watch?v=PvZdTZ1NI5o" is displayed in a dark grey font.

Separeras från resten av URLn med frågetecknen **?**



A screenshot of a web browser's address bar. On the left, there is a document icon. To its right, the URL "https://www.google.se/?q=javascript+bad+practices" is displayed in a dark grey font.

```
GET /movies?title=seven
```

queries relaterade till **json-server**

```
GET /movies?_sort=rating&_order=DESC
```

Får tillbaka resultat sorterat på **rating** och ordnat från **hösta till lägsta**  
(om rating inte är en array)

## API KEY

Man behöver oftast en unik nyckel så man inte missbrukar API:t.

Nyckeln är en del av query, kan ibland skickas som header.



```
https://api.github.com/orgs/FEND16/repos?sort=pushed
```

Hämtar alla repos sorterar efter senast pushad

```
http://api.openweathermap.org/data/2.5/group?  
appid=b3d5b9f4fc8a&id=2666199
```

**ID** i det här fallet är det unika idt för staden Uppsala

**&** separerar de olika argumenten

DET FINNS INGEN STANDARD FÖR ATT STRUKTURERA SITT API MEN  
DE FLESTA FÖLJER ÄNDÅ SAMMA **DESIGN PRINCIPLES**

Ni borde kunna använda de flesta APIer

**SYNC VS. ASYNC**

**JAVASCRIPT**

Du står i kö.

Din polare bestämmer sig för att springa iväg  
och köpa något att äta i kön.

Du behåller platsen i kön.

Polaren kommer tillbaka med käket och höjer stämningen rejält.

**Så funkar asynkront JavaScript**

men...

Du vet inte om du kommer att stå kvar i kön

Kön kanske plötsligt rör sig snabbt och du är inne.

Polaren kanske chansar ändå

**I JavaScript chansar vi inte för då går det garanterat åt helvete**

# BLOCKING CODE

Koden läses typ uppifrån och ner

```
//Can take forever  
for(let i = 0; i < 10000000000000000; i++){}
```

Har du en fet loop kommer den att blockera resterande kod

Vi kan garantera att kod körs som vi vill men vi får långsammare kod.

# CALL STACK

```
function foo(){  
    return "Pffft!"; //remove from stack  
}  
function bar(){  
    foo();  
}  
  
bar();
```

call stack

bar();

foo();

# STACK OVERFLOW

```
foo();
```

```
foo();
```

```
foo();
```



# NON BLOCKING CODE

`AJAX` & `setTimeout()`

Funktioner eller kodblock som inte blockerar `call stack`

I princip placeras dessa funktioner i en annan stack som hanteras av `event loop`

```
//non blocking
console.log('Starting Sequence!');

setTimeout(function(){
    console.log('Hello from timeout!');
}, 250);

console.log('Ending Sequence!');
```

```
//non blocking
console.log('Starting Sequence!');

setTimeout(function(){
    console.log('Hello from timeout!');
}, 0); //No timeout, 0 milliseconds!

console.log('Ending Sequence!');
```

Vad kommer att skrivas ut?

```
//loop 6 times
for(var i = 1; i <= 5; i++) {
    //Increase the timeout for each loop
    //and print the value
    setTimeout(function() {
        console.log(i);
    },i);
}
```

```
6 6 6 6 6 6 //-\_(ツ)_/-
```

```
for(var i = 1; i <= 5; i++) {  
    (function(i) {  
        setTimeout(function() {  
            console.log(i);  
        }, i);  
    })(i);  
}
```

```
//-\_(ツ)_/-
```

Eller strunta i att skapa funktioner i loopar

```
function setDelay(i) {  
  setTimeout(function() {  
    console.log(i);  
  }, i);  
}
```

```
for (var i = 0; i <= 5; ++i) {  
  setDelay(i);  
}
```

```
for(let i = 0; i <=5; i++){  
  setTimeout(function(){  
    console.log(i);  
  }, i)  
}
```

let skapar **block scope** och då behålls värdet för varje iteration och ökas

## Call stack

```
foo();  
bar();
```

Event loopen kollar först om call stack  
är tom eller inte

Händelserna i högra kolumnen pushas i  
det här fallet efter att loopen har kört  
klart då `i == 6`

## Event loop

```
setTimeout();  
setTimeout();  
setTimeout();  
setTimeout();
```



```
setTimeout(function(){  
  console.log('Hello from timeout!');  
}, 0); //No timeout, 0 milliseconds!
```

Oberoende av tid som sätts så kommer koden att  
först hanteras av **event loop** sedan läggas till i **call stack**

**Async alltid**

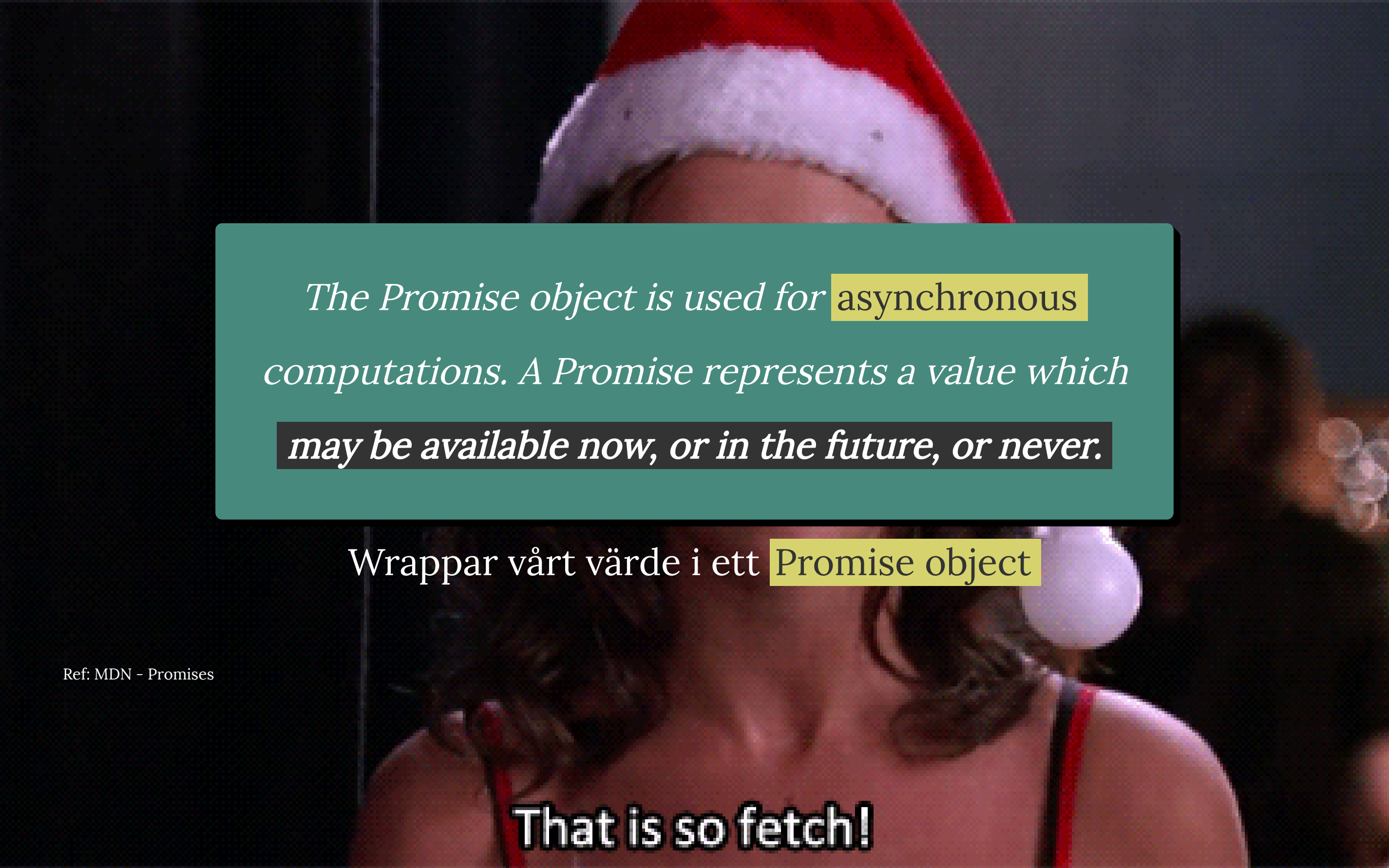
Philip Roberts: What the heck is the event loop anyway

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

**DETTA GÄLLER ALLA AJAX REQUESTS**

**PROMISES**

*`fetch()` allows you to make network requests similar to `XMLHttpRequest`. The main difference is that the Fetch API uses `Promises`, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of `XMLHttpRequest`.*



*The Promise object is used for asynchronous computations. A Promise represents a value which may be available now, or in the future, or never.*

Wrappar vårt värde i ett Promise object

Ref: MDN - Promises

**That is so fetch!**

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.

```
//Create new promise that will resolve after 250ms
new Promise(function(resolve, reject){
  //Make async "request"
  setTimeout(function(){
    //Resolve promise after 250ms
    resolve("Success!");
  }, 250);
}
```

**async** men fortf. inget AJAX inblandat



```
//Create new promise that will reject after 250ms
new Promise(function(resolve, reject){
  setTimeout(function(){
    //reject promise after 250ms
    reject("Rejection your onion");
  }, 250);
})
```

```
var promise = new Promise(function(resolve, reject) {  
    // do async stuff, ajax?  
  
    if (status == 200) {  
        resolve(response);  
    }  
    else {  
        reject(Error("So error! So much!"));  
    }  
});
```

```
promise.then(function(response) {  
    console.log(response)  
}, function(error) {  
    console.log(error);  
})
```

# FETCH()

fetch returnerar ett promise object

```
fetch('get.com').then(res => res.json());
```

"thenable" om vi ska använda värdet måste vi plocka värdet från objektet

```
var p = fetch('get.com').then(res => res.json());
```

```
p.then(data => console.log(data));
```

```
$.ajax({  
  url: "get.com",  
  success: (response) => {response},  
  error: (err) => {error}  
})
```

Hanterar både lyckad request samt misslyckad

## Alternativ error-hantering

```
$.ajax("get.com")  
  .done(function() {  
    console.log("OKELDOKELI");  
  }).fail(function() {  
    console.log("WRONG");  
  });
```

Error handling är rätt lik jQuery

```
fetch( 'get.com' )  
  .then(res => res.json())  
  .catch(error => error);
```

Om `.then()` inte lyckas så går koden vidare till `.catch()`

LÄS MER

<https://www.tjvantoll.com/2015/09/13/fetch-and-errors/>



# PROMISE.ALL()

```
//AJAX then convert to json
```

```
var a = fetch('get.com/1').then(res => res.json());
```

```
var a = fetch('get.com/2').then(res => res.json());
```

```
//Resolve all promises at once
```

```
Promise.all([a,b]).then(data => console.log(data));
```

# LÄS MER

PonyFoo: Promises in Depth

David Walsh: Promises

Promisees: Promise Visualisation

ExploringJS - Promises

MDN example: XMLHttpRequest with Promise

# THE NEXT BIG THING: ASYNC/AWAIT

CODE BARBARIAN: GUIDE TO ASYNC/AWAIT

YOUTUBE: CHROME DEV SUMMIT (12 MIN IN)

PONYFOO: ASYNC/AWAIT

```
async load() {  
    let response = await fetch('get.com');  
    return await response.text();  
}  
  
load().then(data => console.log(data));
```

Heal funktionen returnerar ett Promise

# INLÄMNINGSUPPGIFTEN

# ARBETSPROCESS

- Ha ett syfte, kan din applikation lösa något problem du har?
- Vilka APIer ska jag använda för att lösa problemet?
- Sätt upp en plan: grundläggande funktionalitet
- Skriv upp en lista på vad applikationen ska kunna göra
- Lägg upp en enklare wireframe
- Skriv upp allting kring arbetet eftersom det ska sedan presenteras, dels i **README** och för mig