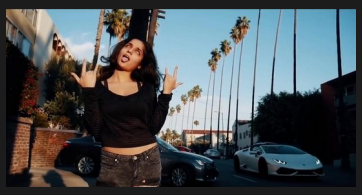
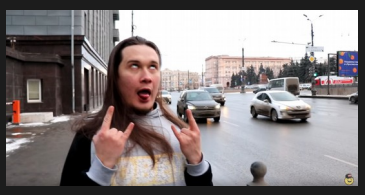
Contemporary Web Development
Lesson 6





https://www.youtube.com/watch?v=vUgyGVCDaYs

Fork async-spacehip

Step into the server side

Up until now we've been using the Webpack development server.

It can only serve files, we couldn't perform any asynchronous web requests.

express

...or

Express is a highly modular server/routing framework for Node JS. It's most powerful feature is the "middleware" system.

A standard for resource access: Some API history

SOAP (formerly known as Simple Object Access Protocol)
SOAP-over-UDP[3]
SOAP Message Transmission Optimization Mechanism
WS-Notification
  WS-BaseNotification
  WS-Topics
  WS-BrokeredNotification
WS-Addressing
WS-Transfer
WS-Eventing
WS-Enumeration
WS-MakeConnection

Metadata Exchange Specification

JSON-WSP
WS-Policy
WS-PolicyAssertions
WS-PolicyAttachment
WS-Discovery
  WS-Inspection
WS-MetadataExchange
Universal Description Discovery and Integration (UDDI)
WSDL 2.0 Core
WSDL 2.0 SOAP Binding
  Web Services Semantics (WSDL-S)
WS-Resource Framework (WSRF)

Security Specification

WS-Security
XML Signature
XML Encryption
XML Key Management (XKMS)
WS-SecureConversation
WS-SecurityPolicy
WS-Trust
WS-Federation
WS-Federation Active Requestor Profile
WS-Federation Passive Requestor Profile
Web Services Security Kerberos Binding
Web Single Sign-On Interoperability Profile
Web Single Sign-On Metadata Exchange Protocol
Security Assertion Markup Language (SAML)
XACML


```xml
<?xml version="1.0"?>
<SOAP:Envelope
    xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema/instance"
    xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
 <SOAP:Body>
     <calculateArea>
       <origin>
          <x xsd:type="float">10</x>
          <y xsd:type="float">20</y>
       </origin>
       <corner>
          <x xsd:type="float">100</x>
          <y xsd:type="float">200</y>
       </corner>
      </calculateArea>
   </SOAP:Body>
</SOAP:Envelope>
```

We needed a standard so software vendors could open up their services as APIs.

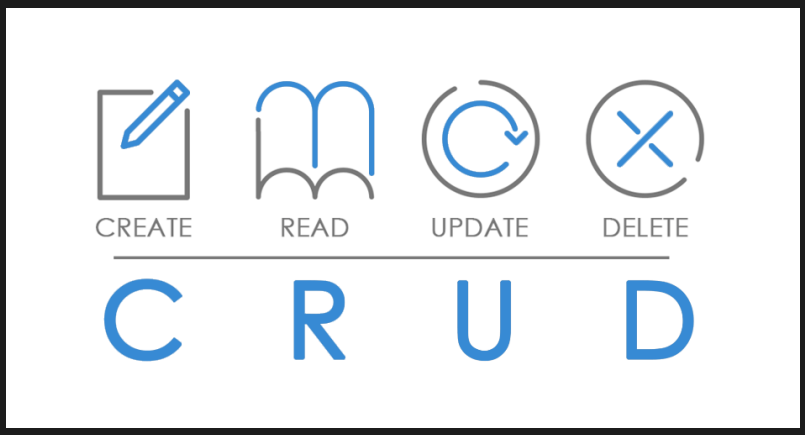
SOAP was overly bloated, required WSDL definition protocol, started branching into all kinds of enhancement specifications.

REST

GET	/movies	Get list of movies
GET	/movies/:id	Find a movie by its ID
POST	/movies	Create a new movie
PUT	/movies	Update an existing movie
DELETE	/movies	Delete an existing movie

Example: Twitter's API

REST stands for "Representational State Transfer"
From Roy Fielding's doctoral dissertation from 2000.
Slowly started getting adopted by tech companies such as Flickr and Amazon.
I saw it get popularized after DHH started pushing it.
Now is basically everywhere.
The idea is to use simple HTTP methods and a naming standard for accessing resources. Just like the simple HTTP GET I did in the first lesson, the GET can be replaced by PUT/POST/DELETE



REST is complemented by CRUD, a standard for updating resources. Any action could be defined as one of the following operations on a specified resource path.

Installing Express and using with Babel/ES6

1) Can start by forking the *async-spaceship* project.
2) We need to separate client from server.
    1) # mv src client
    2) # mkdir server
    3) Update the webpack *entry*
3) # npm install --save express
4) Create server/index.js
5) # npm install --save-dev @babel/node
6) Change start command to "babel-node server/index.js" in pacakge.json
7) We need a .babelrc file where babel-node can read its configuration,

Regarding Stateless/Statefull

Webpack is best suited for client environments.
Client programs are statefull, refresh resets the state and we don't want that.
We should keep Server APIs stateless! Then restarting is not a problem.

Adding Client Webpack support And server Nodemon support

1) # npm install --save-dev webpack-dev-middleware
2) # npm install --save-dev webpack-hot-middleware
3) Add obscure entry to webpack.config.js
4) Update server/index.js

---

1) #npm install --save-dev nodemon
2) Update the package.json start script to:
   nodemon server/index.js --exec babel-node
3) And we need to configure nodemon in package.json to not restart on client changes:

```
"nodemonConfig": {
  "ignore": ["client/*"]
}
```

{JSON}

JavaScript Object Notation

```
{
        "spaceship" : {
                "engines": [
                        {
                                "model": "NYAN5K"
                                "power" : 5000,
                        },
                        {
                                "model" : "NYAN6K"
                                "power": 6000
                        }
                ]
        }
}
```

- [ ] - Array
- { } - Object/Dictionary.
- " " - String.
- No quotes - number.
- "xxx": yyyy – Key: value
- Other data types such as *date* are normally parsed by the application from string.

Adding a server-side spaceship log

./file-logger.js

```
import fs from 'fs'

const LOG_FILE = 'log.txt'
export function writeLog(text) {
  return new Promise((resolve, reject) => {
    fs.appendFile(LOG_FILE, text, (err) => {
      if (err) {
        reject(err);
      }
      else {
        resolve();
      }
    });
  });
}
```

./routes.js

```
app.post('/log', async function (req, res) {
  try {
    await FileLog.writeLog(
      new Date().toLocaleString() + ": "  +
      req.body.text + "\n"
    );
    res.send({status: "success"})
  }
  catch(err) {
    res.status(500).send({ error: err.toString() });
  }
});
```

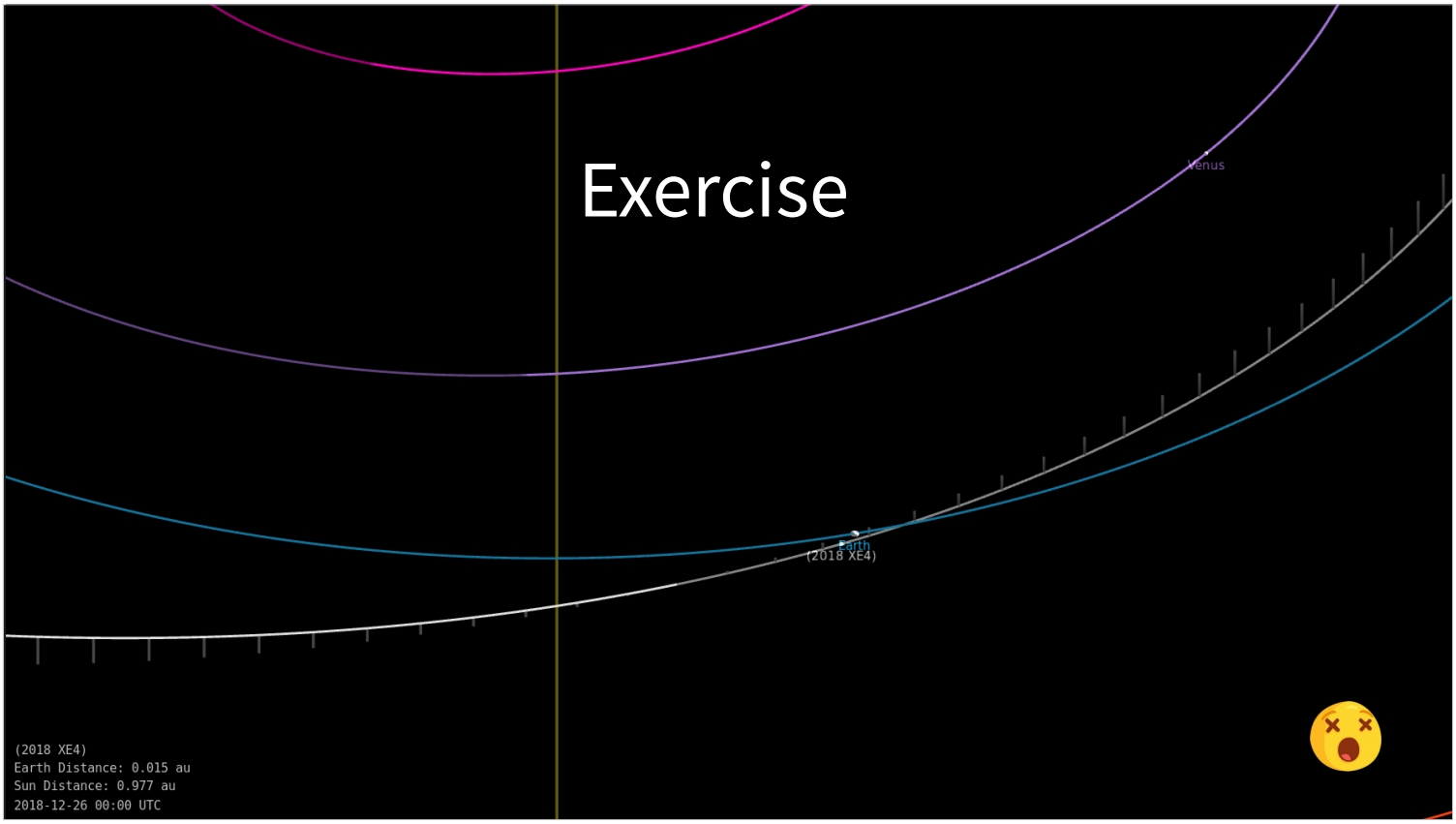
Using *Fetch*

./index.js

```
async function serverLog(text) {
  try {
    let data = {text: text};
    let response = await fetch(BASE_URL + 'log', {
      method: 'POST',
       body: JSON.stringify(data),
      headers:{
        'Content-Type': 'application/json'
      }
    })
    if (!response.ok) {
      let body = await response.json();
      throw new SpaceError(body.error);
    }
  }
  catch (err) {
    console.warn("Error posting log to server", err)
  }
}
```

- We are creating a new log entry, so we use POST.
- Data is sent and received back as JSON.
- We catch an HTTP error and display it on the console.
- The transaction can be viewed on the "Network" tab in the dev tools.

Exercise


Venus
Earth
(2018 XE4)
(2018 XE4)
Earth Distance: 0.015 au
Sun Distance: 0.977 au
2018-12-26 00:00 UTC

1) Add a */nasa/asteroids* route to the server which GETS the "list of Asteroids based on their closest approach date" from Nasa's OpenAPI – **for the current day only**.
2) Add an *async asteroidCheck()* function to *spaceship.js* that queries the server and fails the launch in case any of the asteroids coming today are within less than 0.1 astronomical units away. Run the check at the beginning of the launch sequence.
3) Add */log/filename* route to the server that UPDATES the name of the log file. Test it from the client.

Regarding API access

- Getting an API key to access NASA's OpenAPI is pretty easy, however we must **never store API keys in the client**, because they can easily be picked up by any user.
- Therefore we "proxy" the request from the server, and that way we can also return data that is already formatted for the client's usage.
- However, we must also **never store API keys in git**, so instead of writing them in the source file, we define them only locally as an environment variable.
- To define an API key in the server environment, before running the server we run:
  export NASA_API_KEY=123456789 (In windows use *set* instead of *export*)
- Then, in the Node JS server code we can access that key using the code:
  process.env.NASA_API_KEY

Use ES6 Template Strings for constructing the NASA URL

- ES6 Template Strings allow us to easily construct a string that includes variables. They are recognized by the ` ` quotes and variables are injected using ${var}.
  For example:

```
let name = "avner";
let key = "123456";
let url = `http://avner.js.org?name=${name}&key=${key}`
```

  - The url would appear as:
    http://avner.js.org?name=avner&key=123456

Tips

- In order to use *fetch* in Node, you will have to:
  - npm install –save-dev node-fetch
  - import fetch from 'node-fetch'
- Getting today's date is done using:
  new Date();
- But you will have to format it for NASA's requirements. You can do it with the built-in functions or use a library like *moment.js.*
- Don't forget to always *try* {} and *catch* {}! If you want to *catch,* print something and then have the error propagate up, you can always *throw* it again!
- Use async/await for every async operation!
- Don't forget that when using *fetch*, after let *response = await fetch* you also need to *let result = await response.json();*
- To easily iterate an array in ES6, use *for (let object of array) { }*
- Contact me if you have any questions.