Node & Firebase

Mötesplatsen is on fire

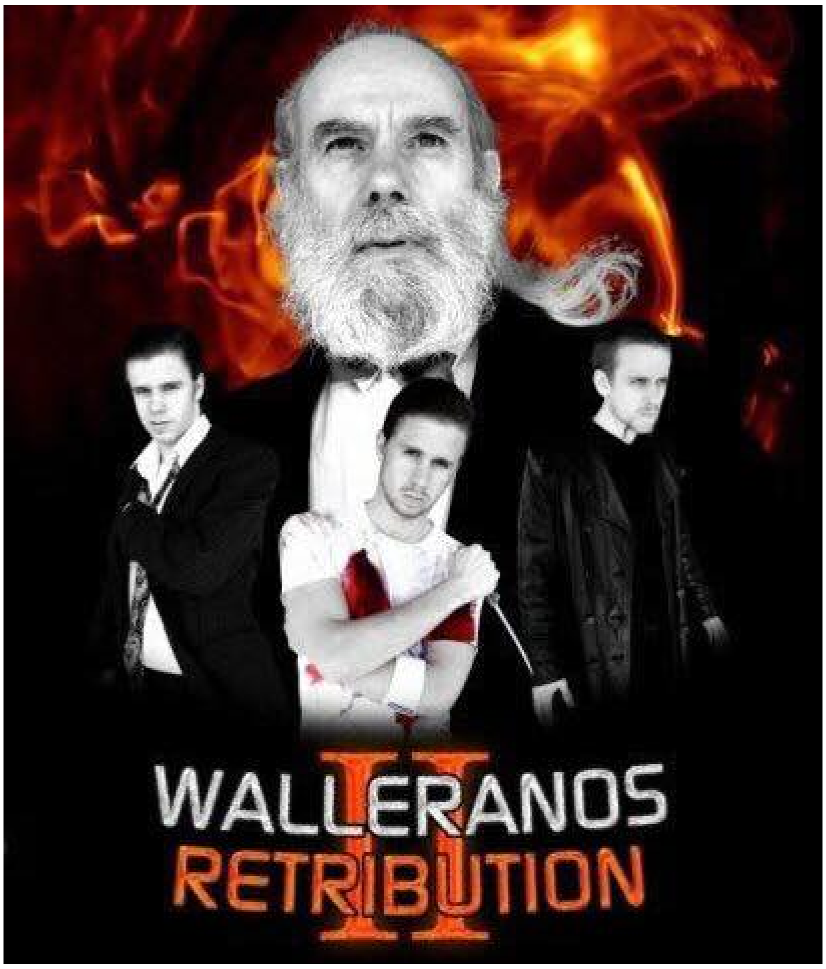
Course overview

chapter 1 / 8

Welcome

Setting the scene

Before we begin - **Alan** says hi and thanx! *1-0-1*



...and I want to thank you (Varbeg) for this:

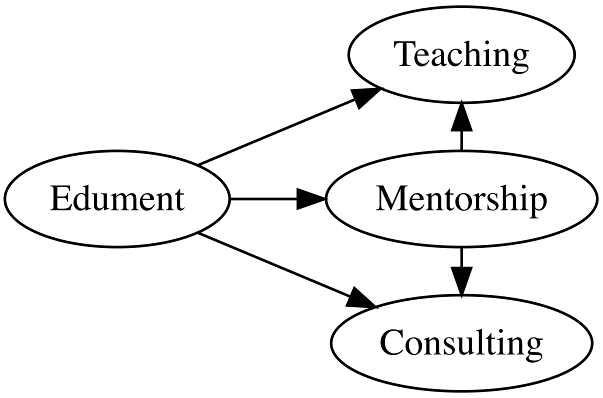
Sections in this chapter:

1-1. Welcome

...to the jungle?

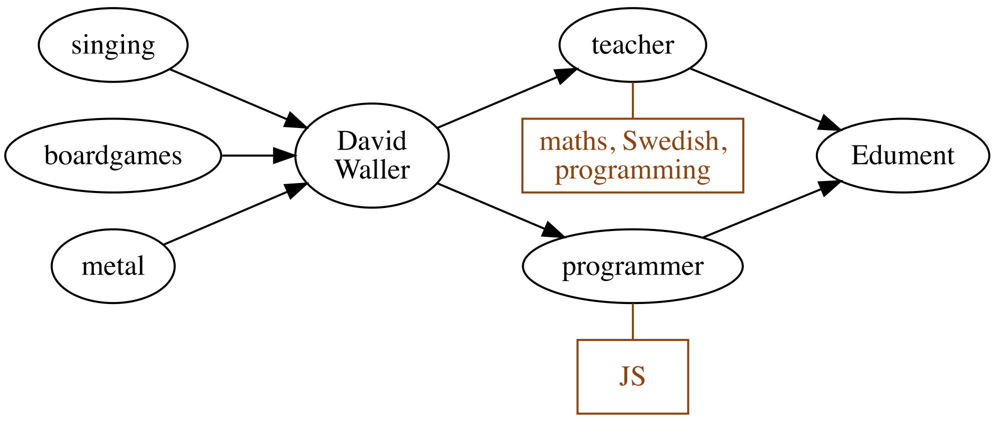
Hello Mötesplatsen! :)

Meet **Edument**!



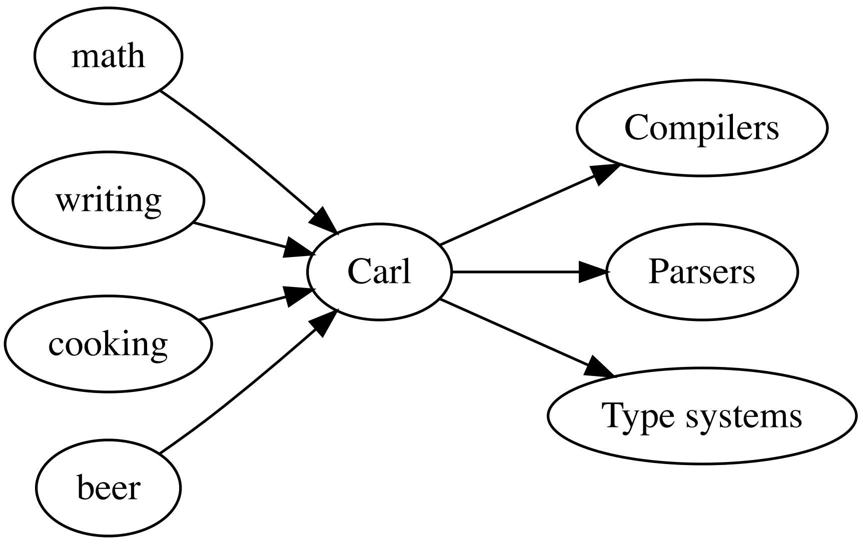
https://www.edument.se

Meet **David**!



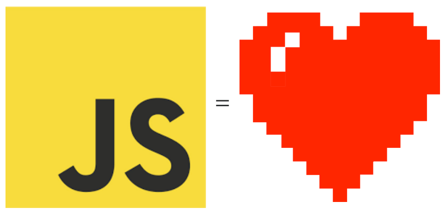
https://blog.krawaller.se david@edument.se

Meet **Carl**!



http://strangelyconsistent.org/ carl@edument.se

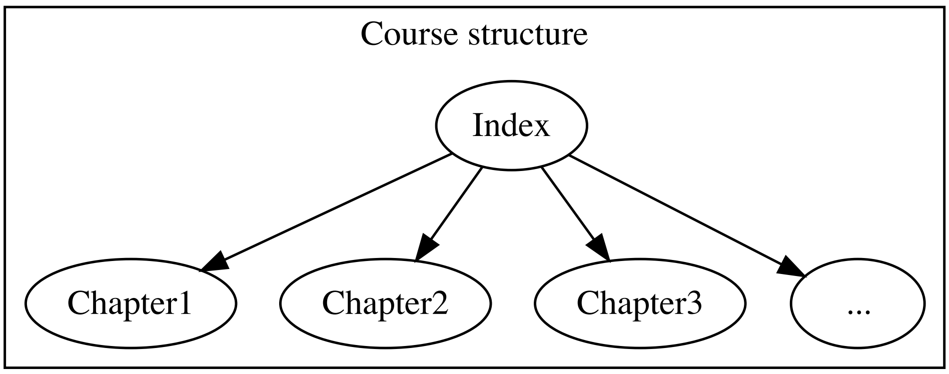
Also, **fair warning** - We spend our days singing this gospel:



1-2. Course structure

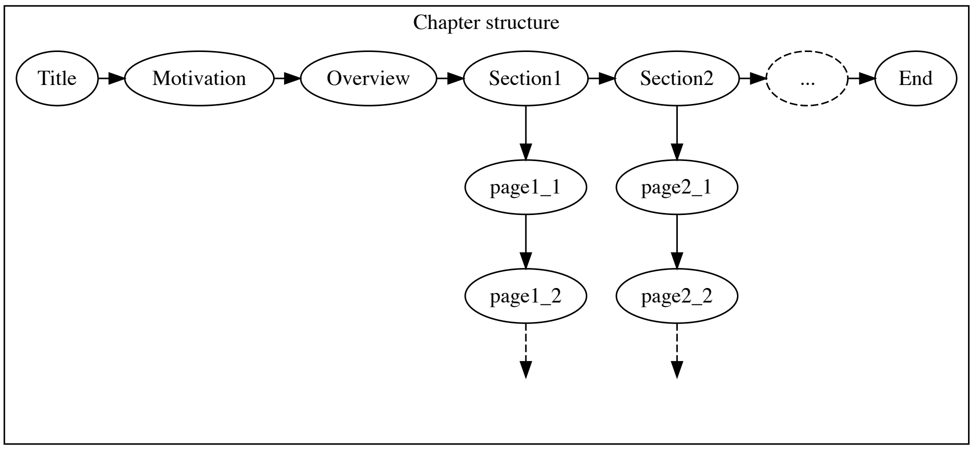
How we'll go about things

The material is divided into several **chapters**, where you're currently in the first. *1-2-1*



We access the chapters from an **index**, giving you a **birds-eye view** of the entire contents.

Each **chapter** has many **sections**: *1-2-2*

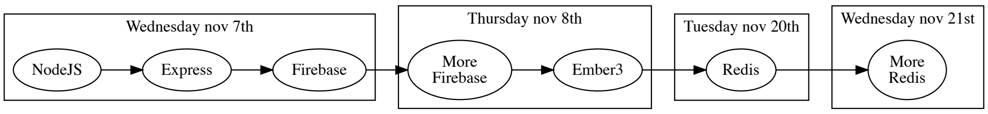


In the **printed material** (or PDF) and in the **presentation top-right corner**, the slides are numbered **`X-Y-Z`** where... *1-2-3*

- `X` is the number of the current **chapter**
- `Y` is the number of the section **section** within that chapter
- `Z` is the number of the **slide** within that section

1-3. Masterplan

Tajmad och klar



Also;

- Focus on **overview** and the hard-to-grasp
- **Practical moments** spliced in
- There are **no stupid questions**! ~~only stupid people~~

chapter 2 / 8

Baseline

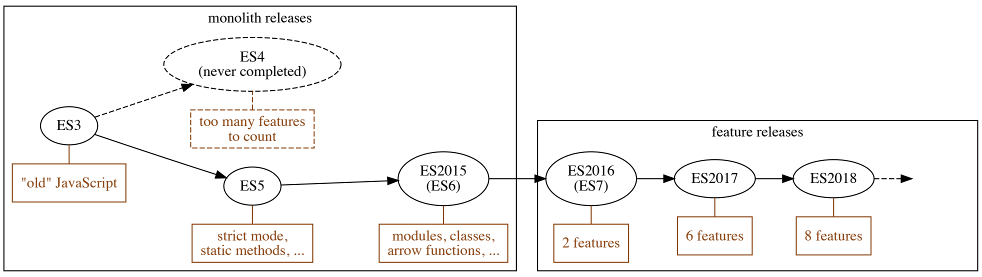
common ground

Sections in this chapter:

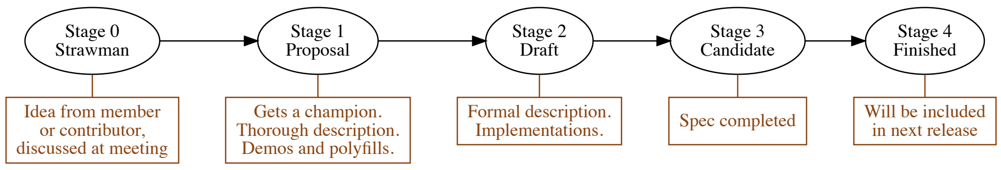
2-1. ECMAScript

JS vocabulary

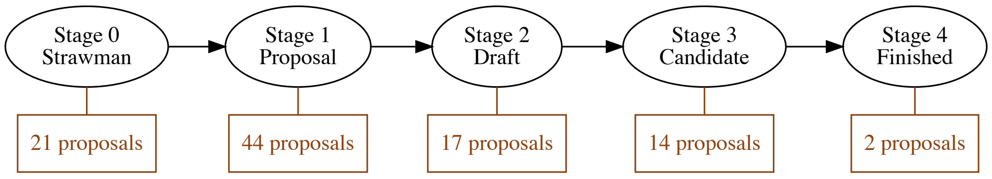
The **ECMASCript versions:** *2-1-1*



The **TC39 process:** *2-1-2*


Stage 0 Strawman
Idea from member or contributor, discussed at meeting
Stage 1 Proposal
Gets a champion. Thorough description. Demos and polyfills.
Stage 2 Draft
Formal description. Implementations.
Stage 3 Candidate
Spec completed
Stage 4 Finished
Will be included in next release

Current **proposal count** (as of 2018-11-07): 2-1-3



There's a (crude) **proposal overview** at https://es-overview.netlify.com/. 2-1-4

2-2. Functional programming

Jumping down the rabbit hole

JavaScript is (almost) a **functional language**, which modern JS developers make frequent use of. 2-2-1

We'll now **walk through a quick example** to make sure we **understand the power of this paradigm**!

So - since **functions are first class citizens**, we can **send them around just like any value**. 2-2-2

Which also means that **a function can take, and/or return, other functions**! Such a function is called a **higher order function**.

As a contrived **example**, say we have this function: 2-2-3

```
let spam = function() {
  console.log("SPAM!");
};
```

And then we have this **higher order function**: 2-2-4

```
function repeater(func, times) {
  for (let i = 0; i < times; i = i + 1) {
    func();
  }
}
```

If we **invoke `repeater`** like this: 2-2-5

```
repeater(spam, 3);
```

We would **see this in the console**:

```
"SPAM!"
"SPAM!"
"SPAM!"
```

To show we can also **return new functions**, take a look at this beauty: 2-2-6

```
function multiplier(func, times) {
  return function() {
    for (let i = 0; i < times; i = i + 1) {
      func();
    }
  };
}
```

Did you see the difference? `multiplier` doesn't execute the parameter function, but **returns a new function**! 2-2-7

```
let tripleSpam = multiplier(spam, 3);
```

If we **execute the returned function** we get the triple spam:

```
tripleSpam(); // SPAM! SPAM! SPAM!
```

Functional programming is a **really powerful tool**, and something that is likely to **get you hooked once you have learned it**. We warmly encourage you to explore the subject! 2-2-8

2-3. Map and reduce

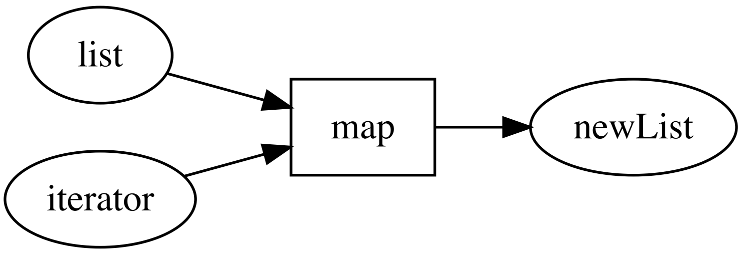
the functional cornerstones

For applied functional programming, there are **two list-operating tools** that have a very central position: 2-3-1

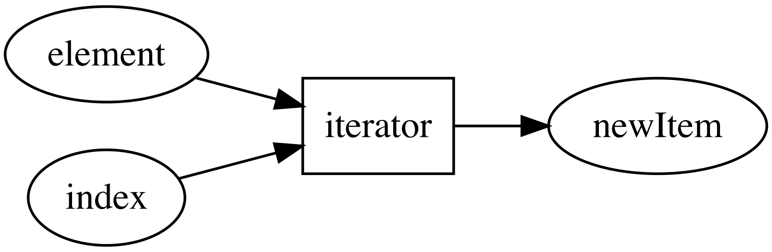
a The **`map`** method
b The **`reduce`** method

Let's **take a look at both of these**!

a First, **`map`**. It takes a **list and an iterator function** as arguments, and **returns a new list**. 2-3-2



The **iterator function** that we pass in is **called with each element and the index of that element**, and it should **return a new element** to be used instead: 2-3-3



Q Here's an example. What will this call return? 2-3-4

```
let list = ["David", "Carl", "Eric"];
let newList = list.map(function(elem) {
  return elem + "y Mc" + elem + "face";
});
```

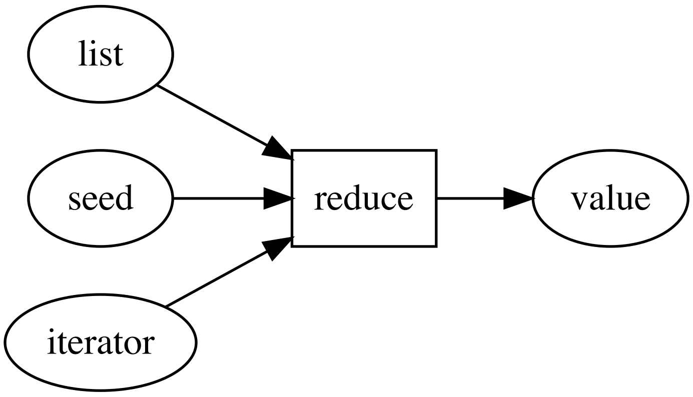
A Yup, `newList` will now equal this: 2-3-5

```
["Davidy McDavidface", "Carly McCarlFace", "Ericy McEricface"];
```

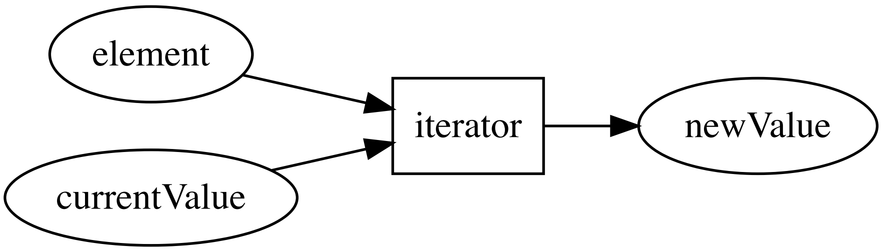
In essence, `.map` will **create a new list of the same length**, where **each element is individually transformed** by the iterator. 2-3-6

b Now for **.reduce**! It is more powerful, but also harder to grasp. 2-3-7

Similar to `.map` it **takes a list and an iterator**, but also a **seed value**. Unlike `.map` it doesn't necessarily return a list, instead it can **return any value**. 2-3-8



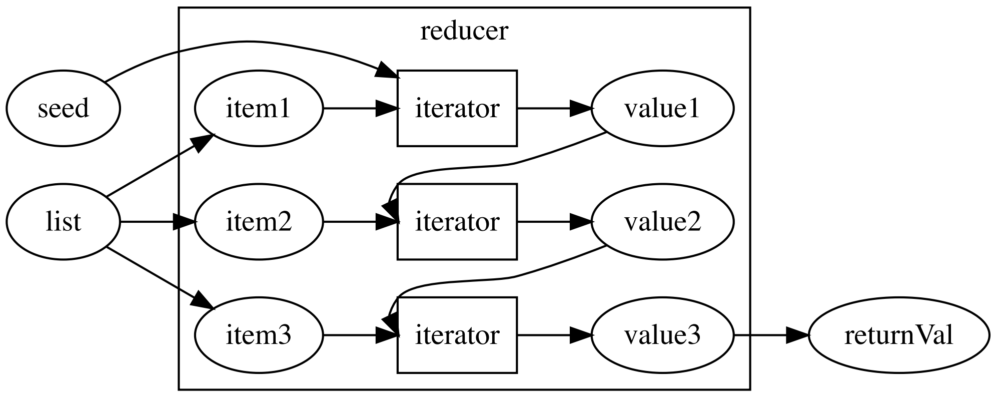
The **iterator function** is called with **each element** and the **`current value`** (sometimes called `memory` or `accumulator`), and **returns a new value**. 2-3-9



The **`seed`** that we passed in to `.reduce` is **used as the `current value` for the first iterator call**. 2-3-10

The **result of the `.reduce` call** is whatever is **returned from the last iterator call**.

In other words, it looks like this (for a 3-item list):



Q Here's a classic example - what is `something`?

```
let list = [{ name: "shovel", price: 27 }, { name: "bucket", price: 14 }];

let something = list.reduce(function(mem, elem) {
  return mem + elem.price;
}, 0);
```

A

```
let something = list.reduce(function(mem, elem) {
  return mem + elem.price
}, 0);
```

We get the **total cost** of the items.

Q A somewhat tougher example, with more modern syntax:

```
let words = ["Bucket", "Pipe", "Flower"];
let something = words.reduce((mem, elem) => ({
  ...mem,
  [elem.length]: (mem[elem.length] || 0) + 1
}), {});
```

A It returns an object which counts how many words of each length the list contains:

```
{
  4: 1,
  6: 2
}
```

2-4. Promises

Give me your word!

In JS we usually deal with asynchronisity via **callbacks**: *2-4-1*

```
const bigRedBtn = document.getElementById("doomsdayBtn");
const clickHandler = e => alert("BOOM");

bigRedBtn.addEventListener("click", clickHandler); // <--- callback!
```

A callback is simply a function that will (potentially) **call back** at some point in the future. *2-4-2*

Sometimes we need to call back with **success or failure**. NodeJS popularised the following pattern: *2-4-3*

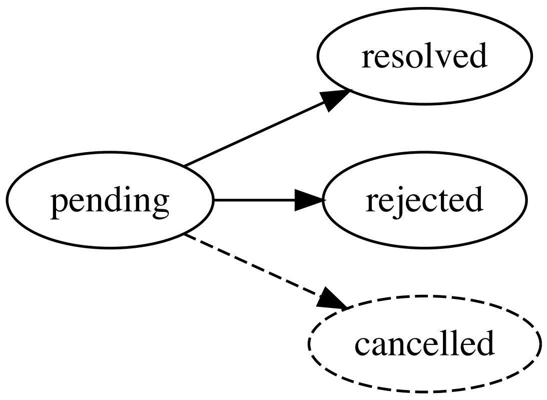
```
const fileHandler = (err, buffer) => {
  if (err) {
    throw new Error("OH NO :/ " + err);
  }
  console.log("file contents: ", buffer.toString());
};

fs.readFile(filepath, fileHandler);
```

A **Promises** is an alternative solution to handling future success/failure. *2-4-4*

Instead of taking a callback, an async operation can syncronously return a promise that will eventually **resolve or reject**.

This translates to the following **state transition**: 2-4-5



The user... 2-4-6

- synchronously gets the promise
- attaches callbacks for success and/or failure.

So instead of this... 2-4-7

```
readFile(path, (err, data) => {
  if (err) handleError(err);
  else handleData(data));
}
```

...we do this... 2-4-8

```
const promise = readFile(path);
promise.then(handleData);
promise.catch(handleError);
```

...or simply chain it like this: 2-4-9

```
readFile(path)
  .then(handleData)
  .catch(handleError);
```

Q So a whole **complicated abstraction** just to **save a few characters**?! 2-4-10

A Yes. 2-4-11

But more **complex asynchronous flow control** patterns can become much easier with promises.

For example, parallellising: 2-4-12

```
let data = {};

const maybeFinalize = (resp, which) => {
  data[which] = resp;
  if (data.A && data.B && data.C) {
    finalize(data.A, data, B, data.C);
  }
};

fetch(urlA, resp => maybeFinalize(resp, "A"));
fetch(urlB, resp => maybeFinalize(resp, "A"));
fetch(urlC, resp => maybeFinalize(resp, "A"));
```

With promises that could become: 2-4-13

```
Promise.all([fetch(urlA), fetch(urlB), fetch(urlC)]).then(finalize);
```

Promises are **created** like this: 2-4-14

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("3 seconds have passed!");
  }, 3000);
});
```

Note that you **rarely need to do that**. 2-4-15

Mostly we'll just use API:s that return promises.

2-4-16

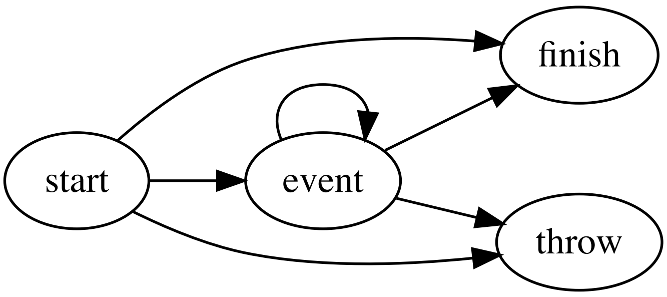
```
bigRedBtn.addEventListener("click", clickHandler);
```

Q But, when a callback can be called **more than once**, like a clickhandler... 2-4-17

...then **promises doesn't make sense**?

A Indeed not! The promise can only represent a **single asynchronous value**. 2-4-18

To represent **many async values** we can use a **stream**:


$$\begin{tikzcd} & & \text{finish} \\ \text{start} \arrow[r] \arrow[urr, bend left] \arrow[drr, bend right] & \text{event} \arrow[loop above] \arrow[ur] \arrow[dr] & \\ & & \text{throw} \end{tikzcd}$$

But that's a totally different story!

Finally, you should know that **promises are contested**. 2-4-19

Read more in the Broken promises blogpost.

2-5. Asynchronous functions

Async await

When we deal with **more than one promise**, things quickly get hairy: 2-5-1

```
function getInvocationOdds() {
  return fetch("/api/alignedPlanet").then(planetId => {
    return fetch(`/api/planetFavourability/${planetId}`).then(favourability => {
      return favourability * 100;
    });
  });
}
```

Using **async functions**, this can be rewritten as: 2-5-2

```
async function getInvocationOdds() {
  const planetId = await fetch("/api/alignedPlanet");
  const favourability = await fetch(`/api/planetFavourability/${planetId}`);
  return favourability * 100;
}
```

An **async function**... 2-5-3

- is **defined** with the **`async` prefix**
- lets you **wait for promises** with **`yield`**
- **returns a promise**

In other words, they let us **write asynch code more neatly**. 2-5-4

More newish features can be found in the **ES2015 appendix**. 2-5-5

chapter 3 / 8

Node, npm and Express

The three musketeers

Sections in this chapter:

3-1. NodeJS

It's JavaScript, Jim, but not as we know it

Node (or Node.js) is a **JavaScript runtime**. It can do two things: *3-1-1*

- REPL (type `node` in terminal)
- Execute file (type `node path/to/file`)

Node runs off of the **v8 runtime**, which also powers the Chrome family of browsers. *3-1-2*

When Node was first released, its biggest innovation was its **asynchronous API**: *3-1-3*

Almost any API function you called returned *immediately* and then did its work asynchronously.

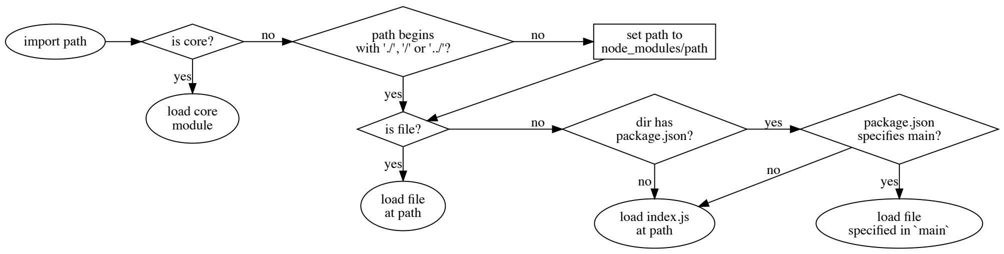
Over the past decade, three major ways to work with asynchronous operations have evolved, each better than the one before: 3-1-4

- Callback functions (used by Node's API)
- Promises (new API in ES2015)
- `async`/`await` (new syntax in ES2017)

In node, each file is a **module**. We can import... 3-1-5

- exports from **other files** in our source
- **build in Node modules** (`fs`, `path`, ...)
- **third party code** from npm

Here's the (almost) full **resolution** logic: 3-1-6



Just like the browser adds a bunch of stuff on top of JS... 3-1-7

```
document.someDOMrelatedMethod...
```

...so does Node!

You'll find the **documentation** for these here... 3-1-8

https://nodejs.org/api/

...and some more **readable guides** here:

https://nodejs.org/en/docs/guides/

Q What **ECMASCript version** runs in Node? 3-1-9

A Excellent question! It of course depends, but: 3-1-10

https://node.green/

3-2. npm

It literally stands for "npm"!

Bundled with node is the Node Package Manager, or **npm** for short. `npm` can be used to: *3-2-1*

1. Initialize projects
2. Install dependencies
3. Run scripts

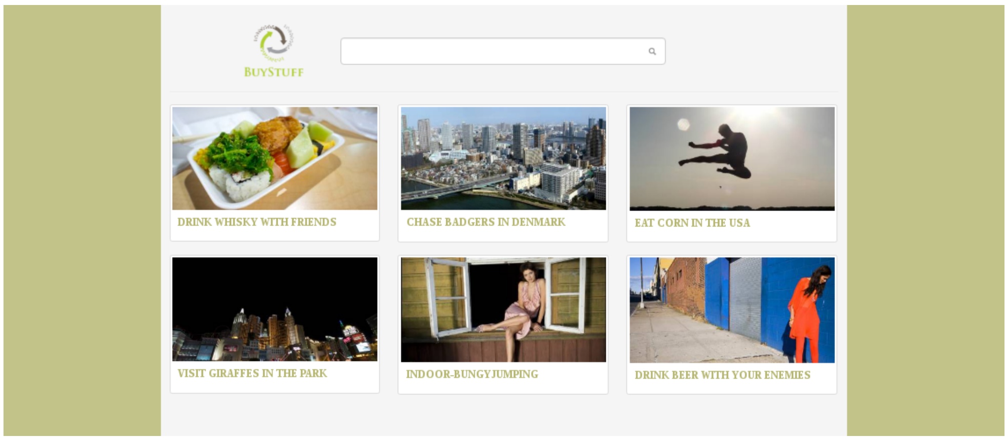
As of this writing, npm hosts over 800,000 modules. *3-2-2*

Let's build a project *3-2-3*

The problem domain *3-2-4*

To have something concrete to work with, we'll pick a **web application** to build.

We'll build a site that lists various offers and lets you buy one or more of them: *3-2-5*


BUYSTUFF
DRINK WHISKY WITH FRIENDS
CHASE BADGERS IN DENMARK
EAT CORN IN THE USA
VISIT GIRAFFES IN THE PARK
INDOOR-BUNGYJUMPING
DRINK BEER WITH YOUR ENEMIES

We won't dive into the details right now, but let's initialize a new project: 3-2-6

```
$ mkdir node-app && cd node-app
$ npm init

# ... Fill everything out
```

What just happened? 3-2-7

Turns out we now have a **package.json** in our folder, with the following content:

```
{
  "name": "BuyStuff",
  "version": "1.0.0",
  "description": "Project description",
  "main": "index.js",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "test": "echo \"We have no tests yet\""
  },
  "author": "David Waller",
  "license": "ISC"
}
```

We'll also need dependencies. While we COULD download them, put them somewhere and `require` them, we're going to install it automatically instead. 3-2-8

Let's install the `express` framework (which we'll deep-dive into today) 3-2-9

```
$ npm install express
```

Now `express` is added to our dependency list in `package.json`! 3-2-10

```
{
  "dependencies": {
      "express": "^4.15.3"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"We have no tests yet\""
  }
}
```

(in older versions of npm we had to install with a `--save` flag for this to happen)

Q Where are the downloaded dependencies stored? 3-2-11

A In a folder called `node_modules`. You'll want to ignore this one in your CVS, but keep `package.json` committed. 3-2-12

There's also the concept of **dev dependencies**, used during development only. 3-2-13

For instance, we'll use `babel` in the upcoming module to compile our code. Babel is not needed in deployment: 3-2-14

```
$ npm install --save-dev babel-cli
```

These are stored in the same folder as regular dependencies.

3-2-15

```
{
  "dependencies": {
      "express": "^4.15.3"
  },
  "devDependencies": {
    "babel-cli": "^6.24.1"
  },
  "scripts": {
    "test": "echo \"We have no tests yet\""
  }
}
```

In a production environment, you'd most likely want to run the following: 3-2-16

```
$ npm install --production
```

This grabs all **regular dependencies** (but not the dev ones) and downloads them.

sidenote - semantic versioning 3-2-17

npm uses **Semantic versioning** (also called **semver**)

A "version" is described as a string, such as `1.2.3`, where:

```
1 - MAJOR (incompatible API changes)
2 - MINOR (new functionality, backwards compatible)
3 - PATCH (patches and bugfixes)
```

In your `package.json` (and also when installing, you can specify a semver, together with one of several comparators. A few common examples include: 3-2-18

```
>=1.2.3     # Anything greater than or equal to this version

=1.2.3      # Exactly this version

^1.2.3      # Keep on the same major version
            # (1.3.0 OK, but not 2.0.0)

~1.2.3      # Keep on the same minor version
            # (1.2.4 OK, but not 1.3.0)
```

There's a giant heap of these. Check the following URL for the spec:

https://docs.npmjs.com/misc/semver

Adding a script 3-2-19

Runnable scripts can be added as a `json` property in `package.json`

```
{
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "printme": "echo \"This is a meaningless script\""
  }
}
```

Running scripts

npm uses a command called `run`:

```
npm run printme
```

This just runs our pointless command that we defined in `printme`

Let's make a more useful example

Adding a test script to `package.json` instead:

```
{
  "dependencies": {},
  "devDependencies": {
      "mocha": "^3.1.2",
      "babel-cli": "^6.18.0",
      "babel-preset-es2015": "^6.18.0",
  },
  "scripts": {
    "test": "mocha --compilers js:babel-core/register"
  }
}
```

This script will by convenience look for a folder named `test` and run all the test files.

Running the test script

```
npm run test
```

or

```
npm test
```

The later is a shortcut provided by npm. You can name a script to whatever you want but this shortcut only works for `test` and `start`.

3-3. File juggling

3 useful 3rd party modules

Let's make **three new friends**!

a More file methods with **fs-extra**
b Markdown support with **marked**
c YAML file metadata with **front-matter**

a First, **`fs-extra`**! It is...

- a drop-in **replacement for `fs`**
- with some **extra features** added

So instead of doing...

```
const fs = require("fs");
```

...you'd do this...

```
const fs = require("fs-extra");
```

...and now you can do this!

```
fs.removeSync(pathTodirOrFile); // example of added method
```

b **Markdown** is a popular **shorthand** format for writing **HTML**.

For example, this...

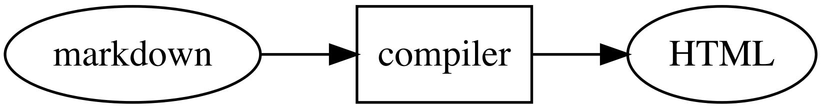
```
## Lego set 6086

The [6086 castle](https://brickset.com/sets/6086-1) is ** totally awesome**!
```

...is equivalent to this:

```
<h2>Lego set 6086</h2>
<p>
The <a href="https://brickset.com/sets/6086-1">6086 castle</a> is
<strong>totally awesome</strong>!
</p>
```

In other words we need a **Markdown compiler**... 3-3-7



...and a very good one is marked.

We can use Marked as a command-line tool... 3-3-8

```
marked -i source.md -o dest.html
```

...or from node: 3-3-9

```
const fs = require('fs-extra');
const mdParser = require('marked').setOptions(...);
const input = fs.readFileSync('./source.md').toString();
const html = mdParser(input);
fs.writeFile('./dest.html', html);
```

c Finally, **front-matter**! 3-3-10

So. When dealing with **content files**, it is common to want to **associate metadata**.

For example, in a **blog post file**, we might want to list... 3-3-11

- author id
- post slug (html path)
- associated tags

...etc.

Using YAML syntax, this might look like this: *3-3-12*

```
---
authorId: david
title: "Using Firebase in React"
tags: JS, Firebase, React
summary: "Showing off Firebase in a React app"
---

# How to use Firebase in React

Hello! Today we'll take a look at how to bla bla
...etc etc, full post content follows here
```

When dealing with such files, we need a method that **separates** the YAML data from the body: *3-3-13*

$$\begin{tikzcd}
 & & \text{metadata} \\
\text{file} \arrow[r] & \boxed{\text{method}} \arrow[ur] \arrow[dr] & \\
 & & \text{body}
\end{tikzcd}$$

...and this, of course, is exactly what the **`front-matter`** module lets us do: *3-3-14*

```
const fs = require("fs-extra");
const rawPostFile = fs.readFileSync("./post.md").toString();
const fm = require("front-matter");
const blogPost = fm(rawPostFile);

const content = blogPost.body; // everything below the YAML
const attributes = blogPost.attributes; // obj with all medata
```

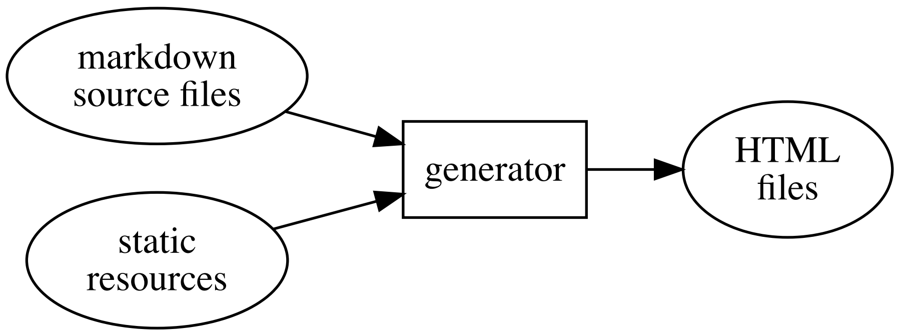
Of course, together with **marked**, a natural next step would be to process the body: *3-3-15*

```
const html = mdParser(blogPost.body);
```

3-4. Exercise - SSG part A

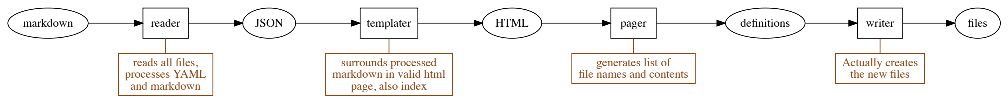
Who needs a DB anyway?

A hot trend today is to replace DB-driven sites by **static site generators**. 3-4-1

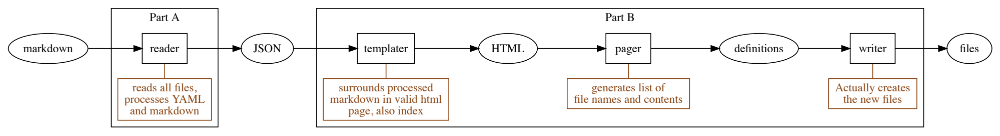


Every time a new source file is added, we regenerate the site!

There are **many steps** to the generating: 3-4-2


markdown
reader
reads all files, processes YAML and markdown
JSON
templater
surrounds processed markdown in valid html page, also index
HTML
pager
generates list of file names and contents
definitions
writer
Actually creates the new files
files

So many, that we start with **just the reader**: 3-4-3


Part A
Part B
markdown
reader
reads all files, processes YAML and markdown
JSON
templater
surrounds processed markdown in valid html page, also index
HTML
pager
generates list of file names and contents
definitions
writer
Actually creates the new files
files

Here's the plan: 3-4-4

- a Set up project
- b Set up static resources
- c Create post source files
- d Create data reader

It is quite a long journey, but for this one you'll get **detailed instructions**. 3-4-5

Try to take **one step at a time**!

a First we **set up the project**! 3-4-6

- create a **new folder** called `ssg`
- navigate to it in a terminal
- execute `npm init` and go through the questions
- add our dependencies:

```
npm install --save-dev front-matter marked fs-extra
```

b Now for our **static resources**! 3-4-7

```
/static
  styles.css
  /images
    nicepic.png
    anotherpic.jpg
```

For now just add in an **empty style sheet** and some **images** that you want to include in your blog posts. 3-4-8

c Now create a **directory `/posts`** with two or three **markdown files**. 3-4-9

```
/posts
  my_first_post.md
  another_post.md
  a_third_post.md
```

These are our post source files, that we eventually want to turn to HTML.

Each post file should have **YAML frontmatter** containing... 3-4-10

- slug (will be the URL)
- title
- summary

...and a markdown body.

Something like this: 3-4-11

```
---
slug: hello_dear_world
title: My first post
summary: "The inaugural hello world post!"
---

Greetings **world**! My name yada blah ...
```

If you want to **reference an image** in a post, the markdown shorthand is: 3-4-12

```
![some title](images/nicepic.png)
```

In other words, use a **relative path** as if you were inside the `static` folder.

d Add a folder `generate` with a `getData.js` file: 3-4-13

```
/generate
  getData.js
```

This file will house logic for **reading the post source files**.

It should export a single method that returns the data: 3-4-14

```
// inside of /generate/getData.js

module.exports = function() {
  // Loop through all files inside "../posts",
  // and return an array of objects, one per
  // post. These objects should contain
  // processed YAML and markdown.
  return posts;
};
```

The exact shape of the post objects is up to you! 3-4-15

You will need to use the **file API**...

```
const fs = require("fs"); // build in file methods
```

...to **find all file names in a folder**...

```
fs.readdirSync(sourcePath); // array of file names
```

...and **read a single file**:

```
fs.readFileSync(filePath).toString();
```

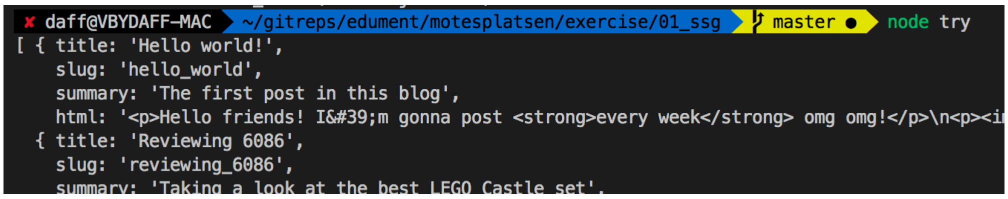
How do we **test** the getData function? One way:

- Create a `try.js` file in the root
- Inside that file, add

```
const getData = require("./generate/getData");
const posts = getData();
console.log("Parsed posts", posts);
```

- Execute `node try` in the terminal

Once you see the correct data logged out, you're done!

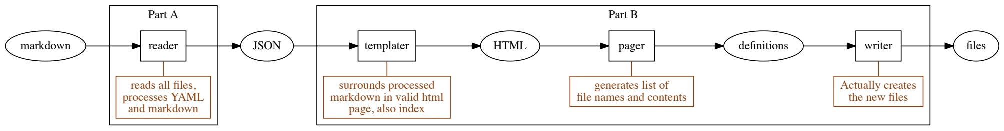


```
✘ daff@VBYDAFF-MAC  ~/gitreps/edument/motesplatsen/exercise/01_ssg  master ●  node try
[ { title: 'Hello world!',
    slug: 'hello_world',
    summary: 'The first post in this blog',
    html: '<p>Hello friends! I&#39;m gonna post <strong>every week</strong> omg omg!</p>\n<p><i
  { title: 'Reviewing 6086',
    slug: 'reviewing_6086',
    summary: 'Taking a look at the best LEGO Castle set',
```

3-5. Exercise - SSG part B

Almost there!

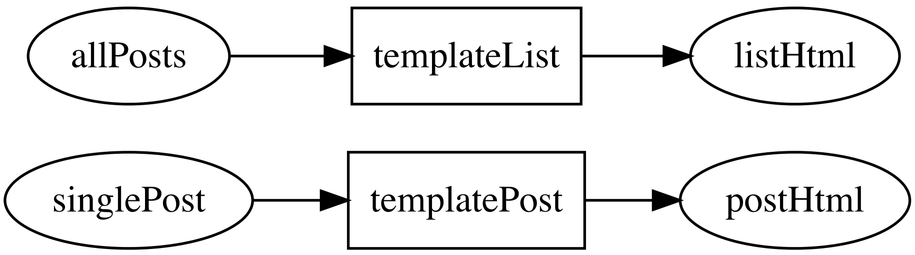
Time for step B!


markdown
Part A
reader
reads all files, processes YAML and markdown
JSON
Part B
templater
surrounds processed markdown in valid html page, also index
HTML
pager
generates list of file names and contents
definitions
writer
Actually creates the new files
files

In other words, we need to 3-5-2

- a Make templates
- b Make a page getter
- c Make a writer

a First, some **templating** functionality! We want to make **blog pages** and also a **list** to use as index: 3-5-3



So we'll need **two different templates**.

Make that **three**, since we also want a master template! 3-5-4

We'll implement them using ES2015 template literals - see the "misc" section in the ES2015 Appendix if they're not familiar!

Add a **`/templates`** dir inside `/generate`, containing **three files**: 3-5-5

```
/generate
  /templates
    master.js
    post.js
    list.js
```

The **`master.js`** should be something like this: 3-5-6

```
module.exports = function(title, content) {
  return `
<!DOCTYPE html>
<html lang="en">
<head>
  ...
</head>
<body>
  ...
</body>
</html>`;
};
```

The `<head>` part should **set the title** and **reference the stylesheet**: *3-5-7*

```
<meta charset="UTF-8">
<title>${title}</title>
<link rel="stylesheet" href="styles.css">
```

And in the `<body>` we simply **show the content**: *3-5-8*

```
<div>
  <h1>My awesome blog</h1>
  <hr/>
  <h2>${title}</h2>
  <main>${content}</main>
</div>
```

The **`post.js`** file should template the page for a single post: *3-5-9*

```
const master = require("./master");

module.exports = function(post) {
  return master(post.title, post.html);
};
```

It will simply pass the processed markdown content to the master template.

In **`list.js`** we template the front page listing all posts: *3-5-10*

```
const master = require("./master");
module.exports = function(posts) {
  let listOfPosts = ...; // build a <ul> with a <li> per post
  return master("My awesome blog", listOfPosts);
}
```

Each `<li>` should be something like this: *3-5-11*

```
let htmlForPost = `
  <li>
    <a href="${post.slug}.html">
      ${post.title}
    </a>
    <p>${post.summary}</p>
  </li>
`;
```

To **test** your templates, simply **edit `try.js`**: 3-5-12

```
const getData = require("./generate/getData");
const posts = getData(); // <--- that we make in part A

const list = require("./generate/templates/list");
const post = require("./generate/templates/post");

console.log("A templated post", post(posts[0]));
console.log("The index", list(posts));
```

b Now we need to make the **`getPages`** function! 3-5-13

```
/generate
  getPages.js
```

It should use `getData` and the templates to return an array of page definitions like this: 3-5-14

```
[
  ["index", "<!DOCTYPE hml><html><head><title>My awesome blog..."],
  ["hello_world", "<!DOCTYPE hml><html><head><title>My first post..."],
  ["review_6086", "<!DOCTYPE hml><html><head><title>Reviewing 6086..."],
  ...
]
```

In other words, each page definition is `[slug,templatedPage]`.

To **test** it, **edit `try.js`** again: 3-5-15

```
const getPages = require("./generate/getPages");
const pageDefinitions = getPages();

console.log("Page definitions", pageDefinitions);
```

c Finally, time to add a **writer** file that actually does some work! 3-5-16

```
/generate
  index.js
```

This file will **generate the blog** into `/output`!

It needs to: 3-5-17

- **remove previous `/output`**
- **copy `/static` to `/output`**
- **write an html file** per entry in `getPages()`

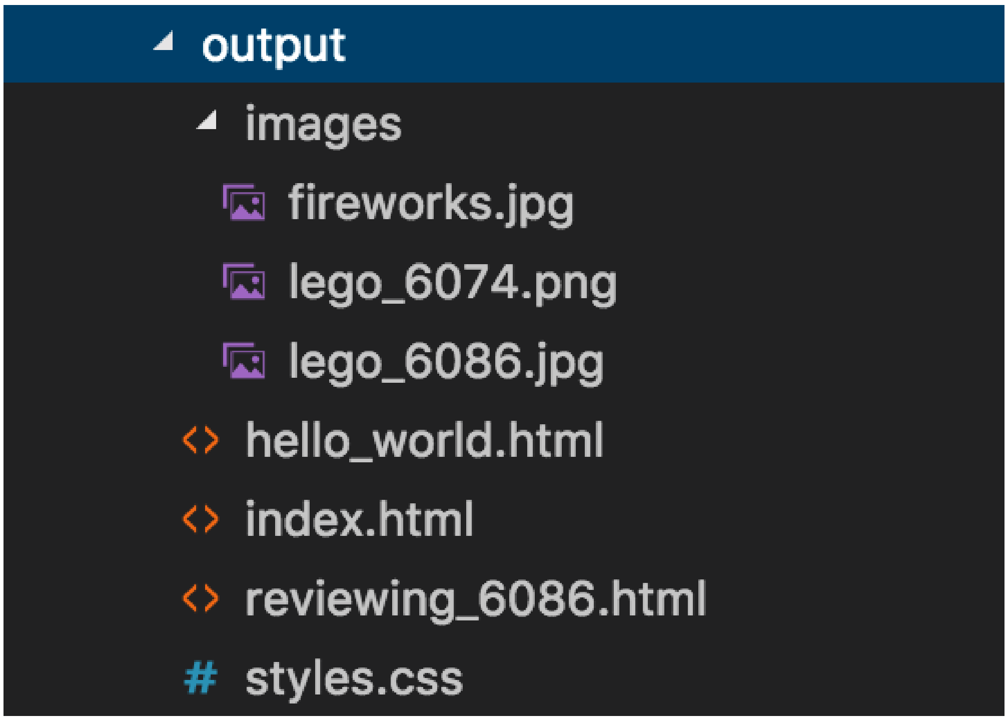
Finally, we're done! 3-5-18

You can now **remove `try.js`**, and instead test the command by doing

```
node generate
```

(this will execute `./generate/index.js`)

...and an `output` folder with static files and html files should appear! 3-5-19



```
output
  images
    fireworks.jpg
    lego_6074.png
    lego_6086.jpg
  hello_world.html
  index.html
  reviewing_6086.html
  styles.css
```

However, it is customary to add "app-level commands" such as the writer file to **npm scripts**, so let's do that! 3-5-20

In **`package.json`**:

```
{
  // other stuff truncated
  "scripts": {
    "generate": "node generate"
  }
}
```

Now the user can create the site by typing 3-5-21

```
npm run generate
```

To **see the generated site in a browser**, either.. 3-5-22

- double-click a .html-file to open it over the `file:///` protocol
- ...or serve the `/output` folder to localhost via a web server

(for the latter, try the **Live Server** extension to VSC)

3-6. Express

Of the expressionist school

Express allows us to work with **routes** as well as **static content**. In this first module, we'll start by looking at serving static content, and then move onto setting up a router. 3-6-1

We have a simple front-end, that we currently serve as static files from Express - let's take a look at the source. 3-6-2

Our server can be started with `npm start`, and the application can be reached on `localhost:8080` 3-6-3

First of all, we're initiating the application as such: 3-6-4

```
import express from 'express';
let app = express();
```

The imported `express` module returns a function that you can call, which returns a new Express.js application.

In this case, we set up a static route, using: 3-6-5

```
import express from 'express';
let app = express();

app.use(express.static('frontend'));

app.listen(8080, function () {
  console.log('Listening on port 8080')
});
```

Static files are mapped as a **route**. In fact, the following: 3-6-6

```
app.use(express.static('frontend'));
```

... Is short for:

```
app.use('/', express.static('frontend'));
```

Another common pattern is to map static files under a certain route. This also means that you can map several routes for different folders: 3-6-7

```
app.use('/static/images', express.static('frontend/images'));
app.use('/static/docs', express.static('lib/docs'));
```

Or more commonly (and safer), using the absolute path: 3-6-8

```
app.use('/static/images', path.join(__dirname, 'frontend', 'images'));
app.use('/static/docs', path.join(__dirname, 'lib', 'docs'));
```

Static routes are great for static content - but what about dynamic pages? 3-6-9

Routes are at the heart of an Express application, and we can set up more than just static ones. 3-6-10

In this course, we'll start by looking at some simple templating using **Handlebars**.

Handlebars is an extension to the Mustache templating language. It lets you write nifty templates such as: 3-6-11

```
<h1>{{title}}</h1>
<div>{{description}}</div>
```

In order to install Handlebars: 3-6-12

```
$ npm install --save handlebars
```

Before we start using Handlebars, let's setup a new Express route. A typical `HTTP GET` route looks like this: 3-6-13

```
app.get('/my-route', (req, res) => {
    // ...
});
```

The name of the function (`get`) is the HTTP verb. Similarly, there are functions for `POST`, `PUT`, `DELETE` and so on. 3-6-14

The first parameter is the route mapping, and the second parameter is the handler: 3-6-15

```
app.get('/my-route', (req, res) => {
    // ...
});
```

The handler has parameters for **request** and **response**.

In order to actually return something, we can call `send`: 3-6-16

```
app.get('/my-route', (req, res) => {
    res.send('Hello world');
});
```

We can easily switch this to return a compiled template using Handlebars instead: 3-6-17

```
import handlebars from 'handlebars';

// ...

app.get('/my-route', (req, res) => {
    // Compile a template from a string:
    let template = handlebars.compile("<h1>{{greeting}}</h1>");

    // Link the template to a context and send it back:
    res.send(template({greeting: "HELLO WORLD"}));
});
```

While our last example did in fact work, we'd like to read our templates from files, rather than using hard-coded strings. 3-6-18

What we'll do in this module is: 3-6-19

- Write a new module to allow for simple parsing from files
- Setup routes for a couple of templates for a frontend and link them with some data

Let's start by building a simple get route again: 3-6-20

```
app.get('/', (req, res) => {
    // ...
});
```

Let's assume that we have a module that allows us to compile and link Handlebars templates, like this: 3-6-21

```
import templates from './templates';

// ...

app.get('/', (req, res) => {
  templates.link('src/templates/index.hbt', {
    some: 'value'
  }, (err, data) => {
    if (err) {
      res.status(500).send(err);
    }

    res.send(data);
  });
});
```

Note that this needs to be asynchronous since we'd be blocking on file reads otherwise!

In this module, we'll stub the `link` call in the following way: 3-6-22

```
// templates.js

export default {
    link: function (template, context, callback) {
        fs.readFile(template, 'utf-8', (err, data) => {
            // File is HOPEFULLY read here
        });
    }
};
```

Remember, the Node.js convention is based on callbacks: 3-6-23

```
export default {
    link: function (template, context, callback) {
        fs.readFile(template, 'utf-8', (err, data) => {
            if (err) {
                /* Something went wrong */
            } else {
                /* We're good */
            }
        });
    }
};
```

Handling potential errors and invoking the callback gives us: 3-6-24

```
export default {
    link: function (template, context, callback) {
        fs.readFile(template, 'utf-8', (err, data) => {
            if (err) {
                callback(err, undefined)
            } else {
                try {
                    let compiled = handlebars.compile(data);
                    callback(undefined, compiled(context));
                } catch (e) {
                    callback(e, undefined);
                }
            }
        });
    }
};
```

We should be able to use our module, and link some static data: 3-6-25

```
app.get('/', (req, res) => {
  templates.link('src/templates/index.hbt', {
        items: [
          {
            image: "http://lorempixel.com/400/200/",
            title: "Drink whisky with friends",
            tags: ["whisky", "friends", "drink"]
          },
          {
              image: "http://lorempixel.com/400/203/",
              title: "Visit giraffes in the park",
              tags: ["giraffes", "park", "animals"]
          }
        ]
  }, (err, data) => { /* Abbreviated */ });
});
```

What about the `index.hbt` file? 3-6-26

The interesting template pieces: 3-6-27

```
{{#each items}}
    <div class="col-sm-6 col-md-4">
        <div class="thumbnail item">
            <a href="#"><img src="{{this.image}}" width="400"
               height="200" alt="..."></a>
            <div class="caption">
                <span><a href="#">{{this.title}}</a></span>
            </div>
        </div>
    </div>
{{/each}}
```

This works, but we don't exactly have a **clean interface** between application data and logic right now. 3-6-28

In order to make some logical **separation of concerns**, we'll move the static data into a **module** 3-6-29

We'll hook this up to a real database later in the course

We *know* that database access is going to be asynchronous, so we'll keep this in mind for the new module: 3-6-30

```
// store.js
export default class Store {
    allOffers(callback) {
        callback(undefined, [
            {
                image: "http://lorempixel.com/400/200/",
                title: "Drink whisky with friends",
                tags: ["whisky", "friends", "drink"]
            },
            // ... More static data here
        ])
    }
}
```

Using it is easy (albeit a bit ugly): 3-6-31

```
import Store from './store';

app.get('/', (req, res) => {
  let s = new Store();
  s.allOffers((err, items) => {
    if (err) { res.status(500).send(err); }
    template.link('src/templates/index.hbt', { items },
        (err, data) => {
          if (err) {
            res.status(500).send(err);
          }
          res.send(data);
        });
  });
});
```

The problem with asynchronous code structures in Node.js is already becoming apparent. 3-6-32

3-7. Exercise - server

Express delivery!

Let's serve our generated site from an **Express backend**! 3-7-1

(this won't give us any advantages, we do it just to **try out express**) 3-7-2

(And don't worry - this is **nowhere near as complex** as the last exercise) 3-7-3

(in fact, nothing else will be) 3-7-4

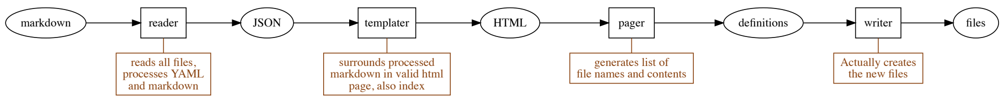
Add a folder `/server` and a file `index.js`: 3-7-5

```
/server
  index.js
```

In that file we need to **create an express app** that... 3-7-6

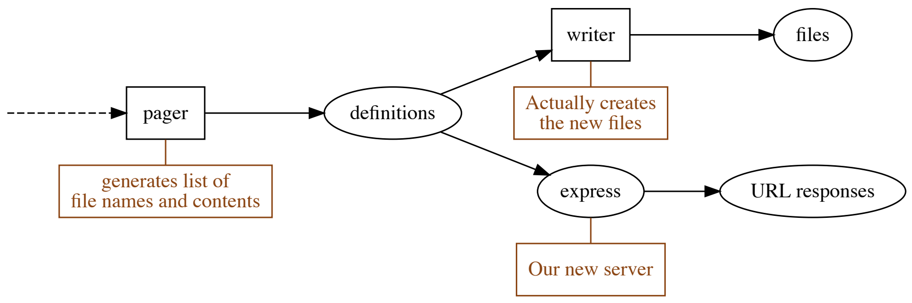
- a has a **route per page definition**
- b **serves the files in `static`**...
- c is available on a localhost port

a Remember how you mumbled to your friend that the last exercise was split into **too many steps**? 3-7-7



markdown
reader
reads all files, processes YAML and markdown
JSON
templater
surrounds processed markdown in valid html page, also index
HTML
pager
generates list of file names and contents
definitions
writer
Actually creates the new files
files

Well, we can now **repurpose the `getPages` function**! 3-7-8



pager
generates list of file names and contents
definitions
writer
Actually creates the new files
files
express
Our new server
URL responses

In other words we'll be **using the `pages` definitions** again... 3-7-9

```
const getPages = require("../generate/getPages");
const pages = getPages();
```

...and for **each of them** we must **set a route**: 3-7-10

```
app.get("/" + fileName, (req, res) => res.send(fileContent));
```

(you'll also want to set an additional route for `/`, serving the same thing as `/index.html`) 3-7-11

b Regarding serving the `/static` files - snoop at the previous section on how to use `express.static`! 3-7-12 c Finally, make your server **`.listen` to a port** of your choosing! 3-7-13

You should also **log out a message** in the console with the port number

Now you can try your server by doing... 3-7-14

```
node server
```

...which runs `/server/index.js`!

But, just like with `generate`, you should also **set up an npm script** to start the server! 3-7-15

chapter 4 / 8

Firebase

Fight fire with fire

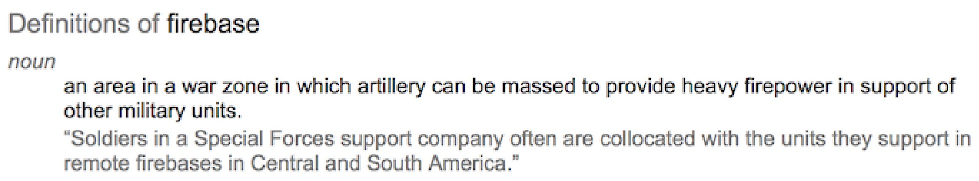
Sections in this chapter:

4-1. Firebase

The platform

Q What is Firebase? *4-1-1*

4-1-2



Definitions of firebase

noun

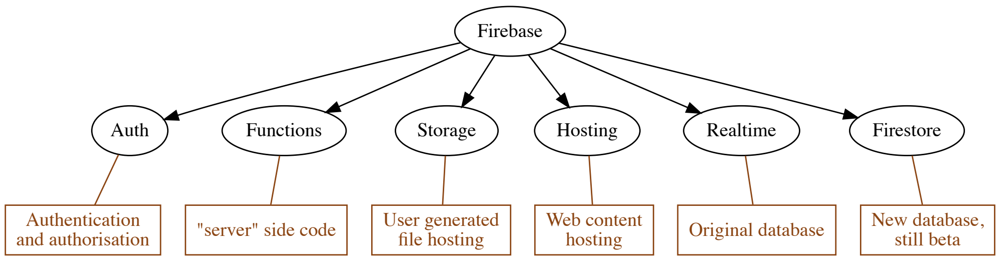
an area in a war zone in which artillery can be massed to provide heavy firepower in support of other military units.

"Soldiers in a Special Forces support company often are collocated with the units they support in remote firebases in Central and South America."

A

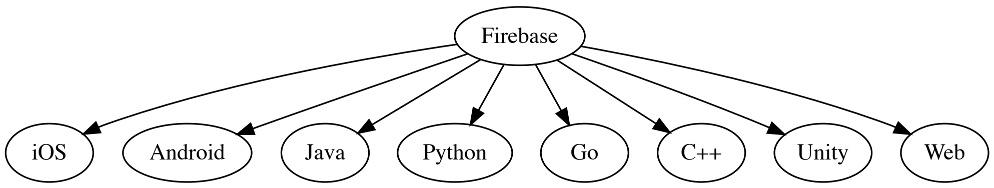
Firebase is a **platform**:

4-1-3



It is even a **platform that supports platforms**:

4-1-4



They even have their own **conference**!

4-1-5

The official homepage is...

4-1-6

https://firebase.google.com/

...because they were bought by Google in 2014.

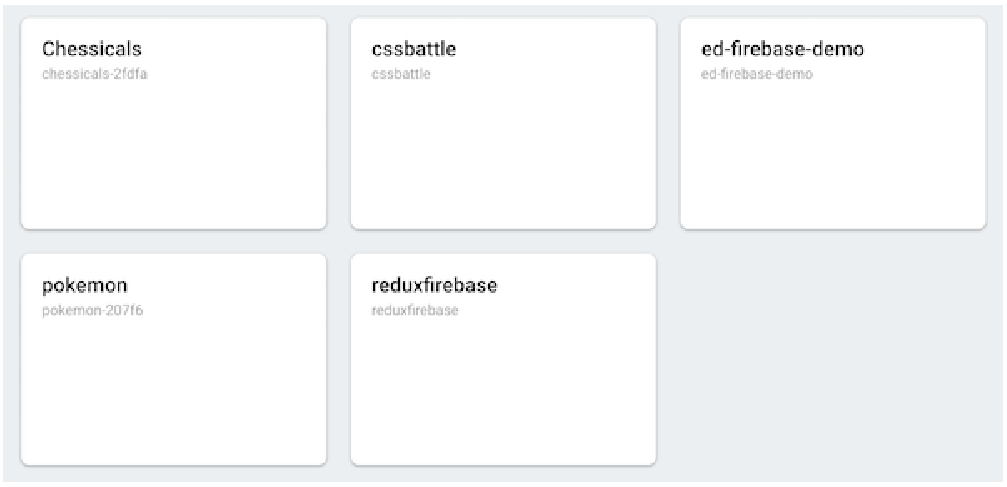
4-2. Workflow

The moving parts

There are a bunch of moving parts when working with Firebase: *4-2-1*

- a The online console
- b The CLI
- c `firebase.json`
- d `.firebaserc`
- e `firebase-debug.log`
- f `.firebase`

a First off you will create **Firebase projects** at https://console.firebase.google.com. *4-2-2*



Chessicals
chessicals-2fdfa

cssbattle
cssbattle

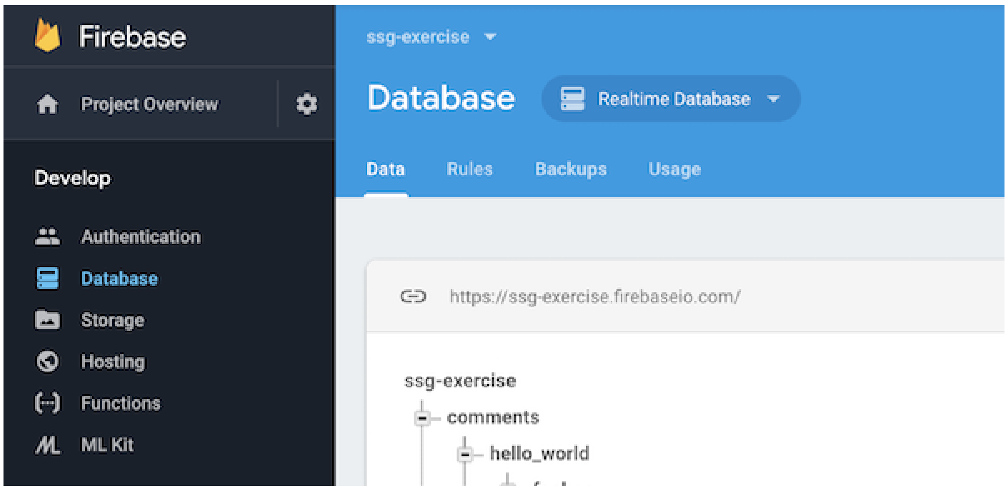
ed-firebase-demo
ed-firebase-demo

pokemon
pokemon-207f6

reduxfirebase
reduxfirebase

Each of them might use any and all Firebase products.

And for each of them we can **config the products** in the console: *4-2-3*



b Locally on our machine we'll use the **Firebase tools**. We install them from npm... *4-2-4*

```
npm install -g firebase-tools
```

...and can then use the `firebase` command:

```
firebase --help
```

c In the source codes for projects intended to use Firebase projects, there is **`firebase.json`**. It holds **config** for various platform parts: *4-2-5*

```
{
  "database": ..., // database config
  "functions": ..., // functions config
  "hosting": ..., // hosting config
  // ...and more...
}
```

We can **create the file by hand**, or **generate it with the CLI**. Same for updates. *4-2-6* d Weirdly there's also another config file, **`.firebaserc`**. Commonly it looks something like this: 4-2-7

```
{
  "projects": {
    "default": "ssg-exercise" // <-- ssg-exercise is a project id
  }
}
```

The job of `.firebaserc` is to connect the code to a **specific firebase project instance**. 4-2-8

Q So, should we **version control `.firebaserc`** along with `firebase.json`? 4-2-9

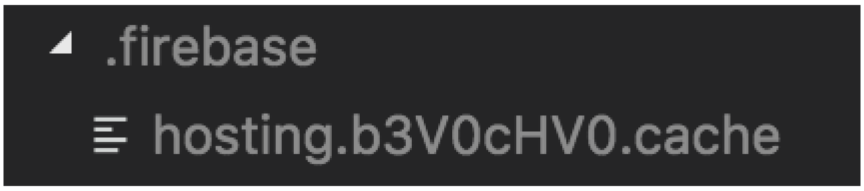
A **Sometimes**, but **probably not**. 4-2-10

Here's a good writeup by a Firebase employee: https://stackoverflow.com/a/43528761

e Next, **`firebase-debug.log`**. 4-2-11

This file will be **generated by some CLI commands**. As the name implies it is a simple log file, and should be gitignored.

f Finally, **`.firebase`**. This is a folder with nonsense files that might turn up for some CLI commands. 4-2-12



It should be ignored and gitignored.

4-3. Hosting

host host

Time to take a look at some actual Firebase functionality! 4-3-1

We'll start by exploring the **hosting** product, which **serves static files**.

It can be **configured** in `firebase.json` thusly: 4-3-2

```
{
  "hosting": {
    "public": ..., // path to folder to publish
    "redirects": [...], // array of redirect rules
    "rewrites": [...], // array of rewrite rules
    "headers": [...], // array of header rules
  }
}
```

The **`public`** is the most important.

A **redirect** rule can look like this... 4-3-3

```
{
  "source": "/foo", // path to catch
  "destination": "/bar", // path to redirect to
  "type": 301 // what code to send
}
```

...or like this, redirecting to an HTTP function: 4-3-4

```
{
  "source": "/foo", // path to catch
  "function": "myFunc", // name of HTTP function
  "type": 301 // what code to send
}
```

We can also **rewrite**, which likely more appropriate when delegating to functions: 4-3-5

```
{
  "source": "**",
  "function": "app"
}
```

Here we route **all traffic** to a single function.

Q Functions? 4-3-6

A Another Firebase product! More on that very soon! 4-3-7

To **publish** our static files, run: 4-3-8

```
firebase deploy --only hosting
```

This will push the contents of the public dir to

```
https://<YOUR-APP-NAME>.firebaseapp.com
```

You can also **serve it locally**: 4-3-9

```
firebase serve --only hosting
```

Find out more about hosting here: 4-3-10

https://firebase.google.com/docs/hosting/

4-4. Exercise - hosting, take I

Hosting the generated files

Time to finally start using Firebase! 4-4-1

And, of course, we do it by adding **hosting** to our blog.

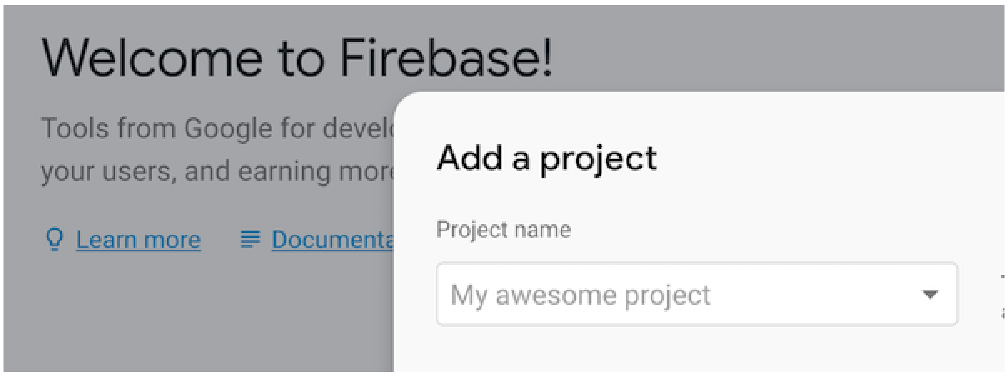
More specifically, we want to **host the generated files in `/output`**! 4-4-2

Here're the steps you need to take: 4-4-3

- a create firebase project
- b connect google account
- c init firebase locally
- d connect firebase project
- e configure hosting
- f test locally
- g publish files online

a First we need to **create a firebase project** in the online console! 4-4-4

Navigate to https://console.firebase.google.com, **log in** with a google account, then **create a project**: 4-4-5



For now you don't need to do anything else.

b Next you need to **connect the google account** to Firebase on your local machine by... 4-4-6

```
firebase login
```

...and then enter your credentials.

We can tell it worked by doing... 4-4-7

```
firebase list
```

...which will show the available Firebase projects for the current user (which should be the one you just made).

c Now we need to **init firebase** locally in our code folder! 4-4-8

In the root folder of our project, execute...

```
firebase init
```

...and answer the questions.

This will generate a `firebase.json`. 4-4-9

We could also have written it by hand.

d Now we have to **connect the firebase project** to the local files. 4-4-10

Execute...

```
firebase use
```

...and select your project in the list.

This will generate a `.firebaserc` linking in the project id. 4-4-11

Again, we could have created that file by hand too.

e Time to **configure the hosting**! 4-4-12

Make sure `firebase.json` contains hosting configuration setting the public folder as `/output`.

As a bonus, add a **`404.html`** file to `/static` and regenerate! 4-4-13

This is a magical file name that will be used by Firebase when the URL doesn't match anything.

f Now everything is set up, and we can **test it locally**! 4-4-14

Run...

```
firebase serve --only hosting
```

...and open `localhost:5000` in a browser!

g Cooler still of course is to **publish it online**! 4-4-15

Execute...

```
firebase deploy --only hosting
```

...and visit `https://<YOUR-APP-NAME>.firebaseapp.com` in a browser!

4-5. Functions

Fun fun functions

Next product to explore: **functions**! 4-5-1

Hosting was a vehicle for files to be consumed by the **client**. 4-5-2

Functions is the opposite - it is a host for files to be consumed by the **server**.

In other words, functions are **server-side code**! It splits broadly into **three categories**: 4-5-3

- **HTTP functions**, which are called via requests
- **database functions**, triggered by DB events
- **misc functions**, triggered by other outside events

Main available `firebase.json` configs: 4-5-4

```
{
  "functions": {
    "predeploy": ..., // likely a build script
    "source": ..., // defaults to './functions'

  }
}
```

In the `index.js` file in the functions source folder, all **named exports become registered functions**: 4-5-5

```
exports.myFunc = ...
exports.myOtherFunc = ...
```

The functions are created by using the `firebase-functions` module from npm... 4-5-6

```
const functions = require("firebase-functions");
```

...which then has **different create methods** for different categories.

For example, a HTTP function that catches **all requests** is created like this: 4-5-7

```
exports.app = functions.https.onRequest(funcToRegister);
```

To **publish** the functions we have defined, run: 4-5-8

```
firebase deploy --only functions
```

To **serve** HTTP functions locally, run: 4-5-9

```
firebase serve --only functions
```

Normally we also want the static assets:

```
firebase serve --only functions,hosting
```

We'll come back to how to play with non-HTTP functions locally later! 4-5-10

Finally - the **documentation for functions** can be found here: 4-5-11

https://firebase.google.com/docs/functions/

4-6. Demo - hosting, take II

Hosting the server

We'll now do a **quick teacher demo** of how to use HTTP functions to host our Express server! 4-6-1

chapter 5 / 8

More Firebase

Return fire

Sections in this chapter:

5-1. Database

Realtime shenanigans

Firebase's claim to fame is of course the **database**. 5-1-1

Let's dig into how it works!

...except we first have to decide **which** database to use, since we have two! 5-1-2

- **Realtime database** (the original)
- **Firestore** (in beta)

The **differences** are explained here: 5-1-3

https://firebase.google.com/docs/database/rtdb-vs-firestore

We will focus on the original **realtime database**.

The basic concept is **ridiculously simple** - our data is just a **big object**: *5-1-4*

```
{
  "comments": {
    "hfue3824": {
      "authorId": "jlkh32u432",
      "content": "I dreamt i lost my teeth"
    }
    // ...
  },
  "users": {
    // ...
  }
}
```

In our JavaScript code we get a **reference** to a certain point in the tree: *5-1-5*

```
let ref = firebase.database().ref("/comments");
```

On that reference we can then **subscribe to changes**: *5-1-6*

```
ref.on("value", function(snapshot) {
  let comments = snapshot.val();
});
```

This will be called immediately and for all changes.

We can **set the tree** at that point: *5-1-7*

```
ref.set({ haha: "all comments now gone" });
```

Or we can **push** a new child: *5-1-8*

```
ref.push({
  name: user.name,
  content: field.value
});
```

This is now at `/comments/someNewGUID/`.

The **intro guides** can be found here: *5-1-9*

https://firebase.google.com/docs/database/web/start

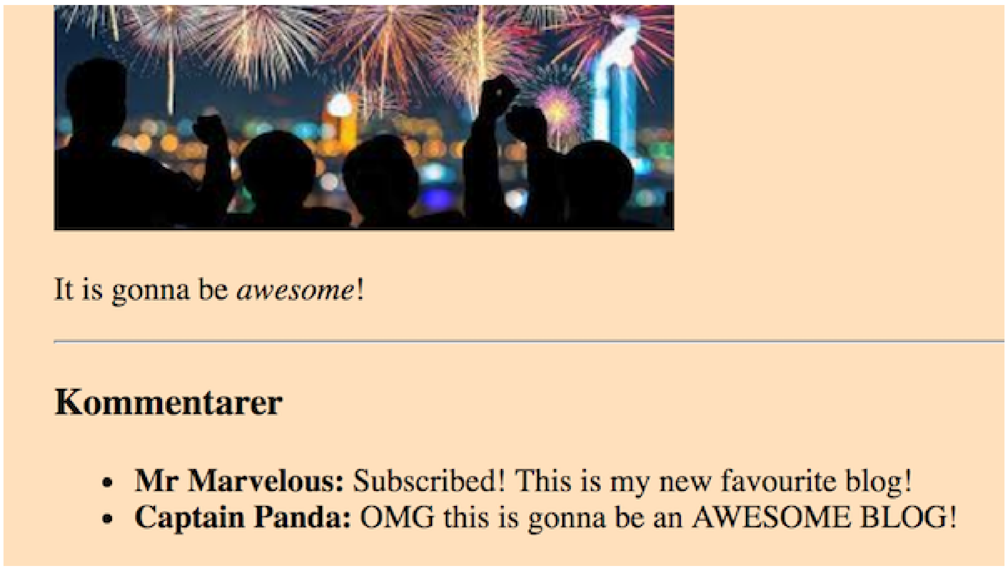
Some **selling points**: *5-1-10*

- Database in the cloud
- objects all the way
- free "push" functionality
- dead simple to integrate

5-2. Exercise - comments I

Read

We will now **show comments from a database** below each post! *5-2-1*



It is gonna be *awesome*!

Kommentarer

- **Mr Marvelous:** Subscribed! This is my new favourite blog!
- **Captain Panda:** OMG this is gonna be an AWESOME BLOG!

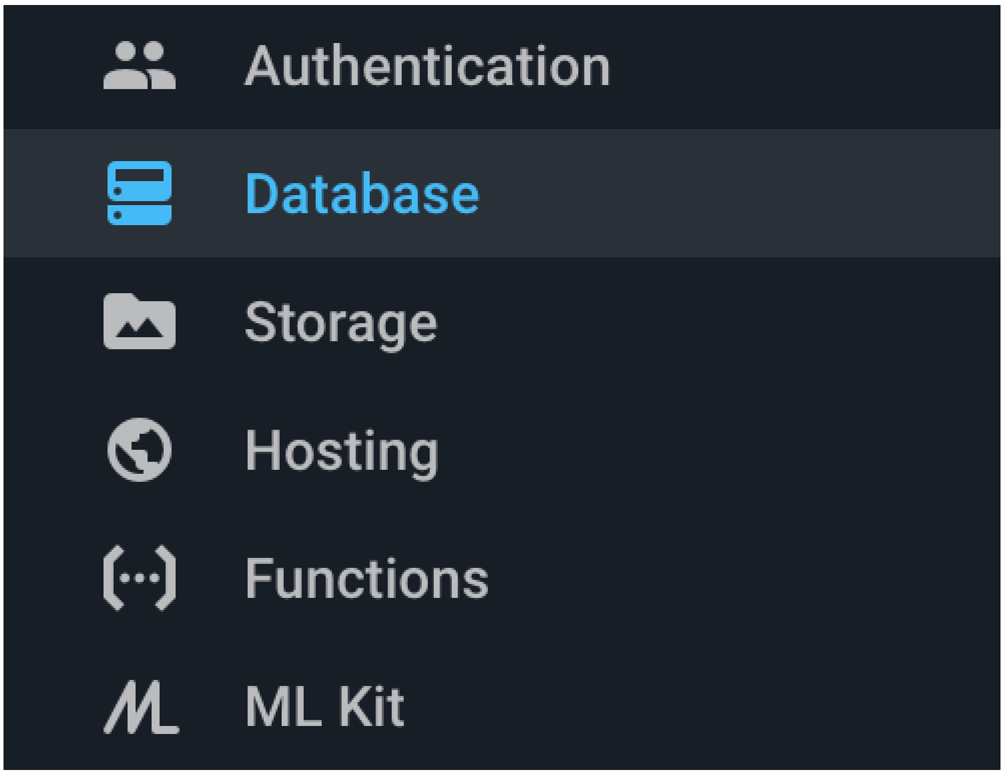
Here's the plan: *5-2-2*

a Create database with fake data
b Initialize DB clientside
c Show data on post pages

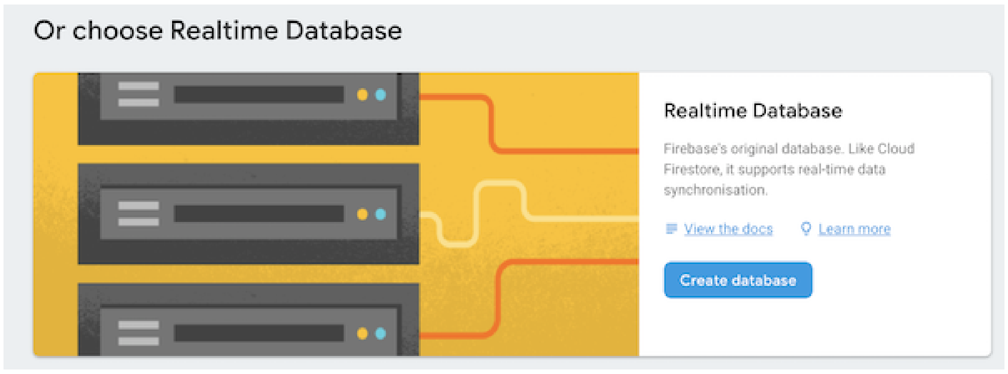
a First, create a database! 5-2-3

Go to https://console.firebase.google.com/ and click on your project

Now select **Database** in the left-hand menu. 5-2-4



You'll get a spammy option screen - scroll down to find **realtime database**. 5-2-5



Or choose Realtime Database

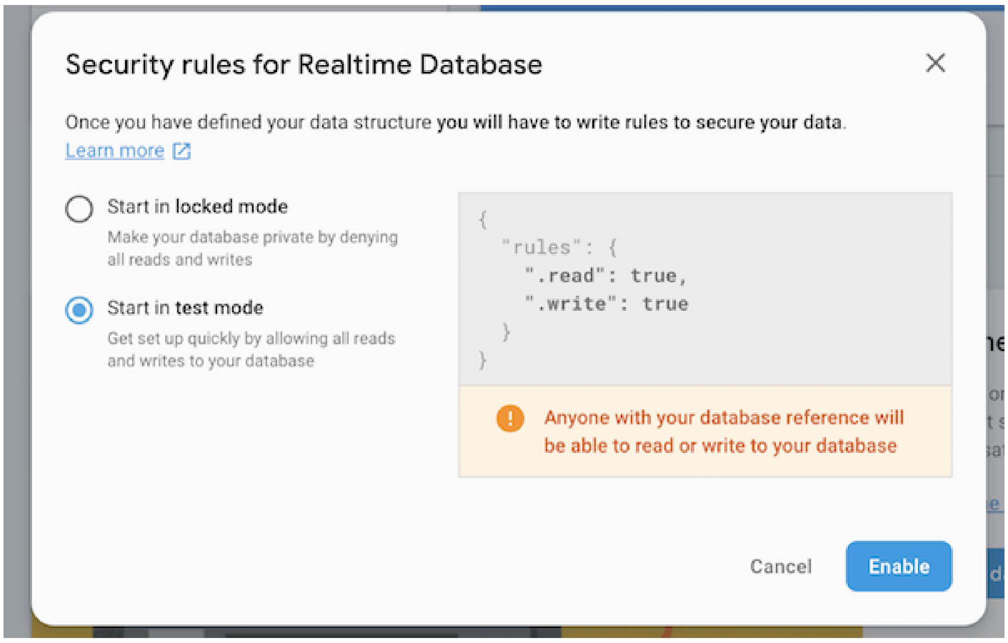
Realtime Database

Firebase's original database. Like Cloud Firestore, it supports real-time data synchronisation.

View the docs Learn more

Create database

In the next popup, select **test mode**: 5-2-6



Security rules for Realtime Database

Once you have defined your data structure you will have to write rules to secure your data.
Learn more

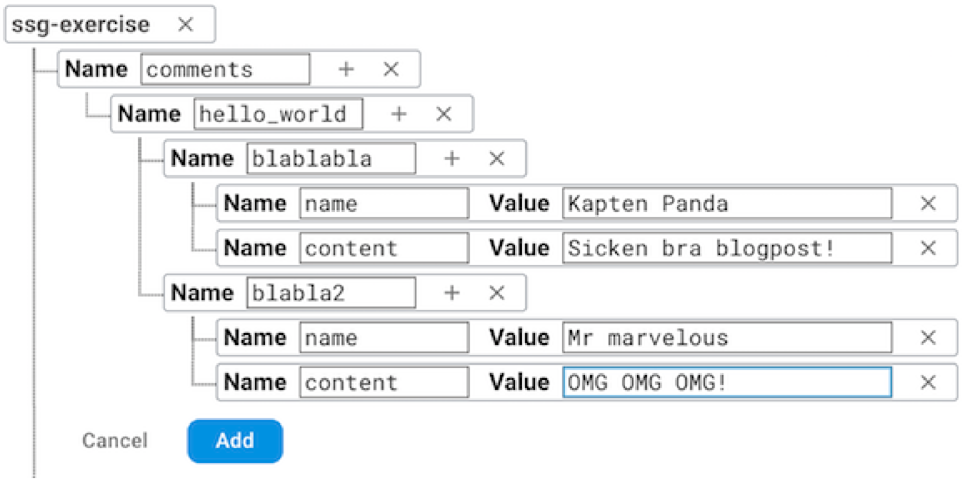
Start in locked mode
Make your database private by denying all reads and writes

[✓ Start in test mode]
Get set up quickly by allowing all reads and writes to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

Anyone with your database reference will be able to read or write to your database

In the data tab, fill in some **fake data**! 5-2-7



```
ssg-exercise
  Name comments
    Name hello_world
      Name blablabla
        Name name     Value Kapten Panda
        Name content  Value Sicken bra blogpost!
      Name blabla2
        Name name     Value Mr marvelous
        Name content  Value OMG OMG OMG!
```

In other words we want this structure: 5-2-8

```
comments // <-- just as a high-level namespace
  hello_world // <--- the slug for a blog
    blablabla // <--- just a made-up guid
      name: NameOfCommenter
      content: CommentContent
```

Test it locally on your machine by executing: 5-2-9

```
firebase database:get "/comments"
```

b Before we get into loading the comments, we need to do some **general firebase initialising** in the client code! *5-2-10*

Add a new file `init` in a `scripts` folder in `static`: *5-2-11*

```
/static
  /scripts
    init.js
```

In that file we need to **initialise firebase**: *5-2-12*

```
firebase.initializeApp({
  // your config
});
```

To see what goes in the config, check
https://firebase.google.com/docs/database/web/start

(for now you just need `apiKey` and `databaseURL`)

Now add that file, along with some firebase source files, to the head in the **master template**: *5-2-13*

```
  <script src="https://www.gstatic.com/firebasejs/5.5.7/firebase-app.js"></scrip
  <script src="https://www.gstatic.com/firebasejs/5.5.7/firebase-database.js"></
  <script src="scripts/init.js"></script>
```

c Time to **show the comments**! First, add some HTML to the bottom of the **post template**: *5-2-14*

```
  <hr/>
  <h3>Kommentarer</h3>
  <ul id="comments">...laddar...</ul>
```

Add a new script file `comments` to house the loading code! *5-2-15*

```
/static
  /scripts
    comments.js
```

That file should **update the DOM** whenever we get new comments: *5-2-16*

```
database.ref("/comments/" + window.postSlug).on("value", function(snapshot) {
  // object with generated keys,
  // each value is object with { name, content }
  // (or we get null if there are now comments)
  var comments = snapshot.val();

  // ...build HTML from comments here...

  document.getElementById("comments").innerHTML = html;
});
```

Q Wait - `window.postSlug`, where did that come from? *5-2-17*

A We set it in the **post template**, where we also link in the `comments` file: *5-2-18*

```
  <script>window.postId = "${post.slug}";</script>
  <script src="scripts/comments.js"></script>
```

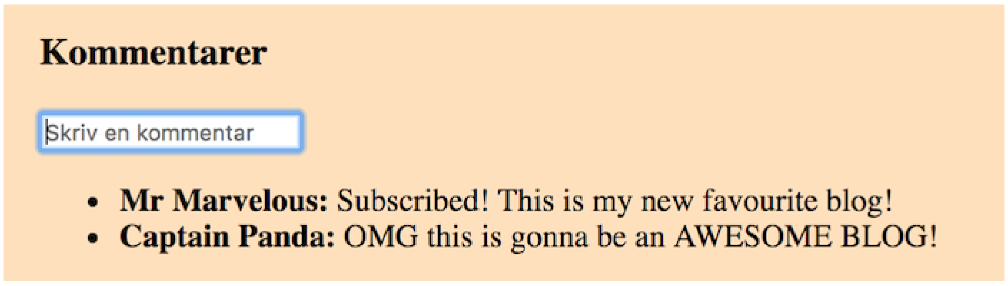
Now, *5-2-19*

- **regenerate** the blog
- **serve** it locally
- navigate to **post with slugId from DB**
- enjoy the (static) discussion!

5-3. Exercise - comments II

Write

We *should* probably allow our users to **write** comments too! *5-3-1*



Kommentarer

Skriv en kommentar

- **Mr Marvelous:** Subscribed! This is my new favourite blog!
- **Captain Panda:** OMG this is gonna be an AWESOME BLOG!

This means... 5-3-2

- adding an **input** to the post template
- adding a **listener** in `comments.js`

That listener should trigger when we hit Enter, and update the DB: 5-3-3

```
database.ref("/comments/" + window.postId).push({
  name: "Anonymous",
  content: field.value
});
```

...wait, that's it? 5-3-4

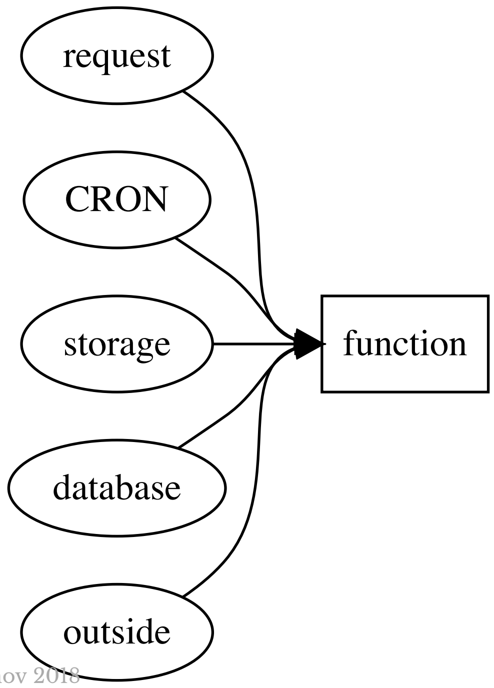
Yep, off you go! :)

5-4. More functions

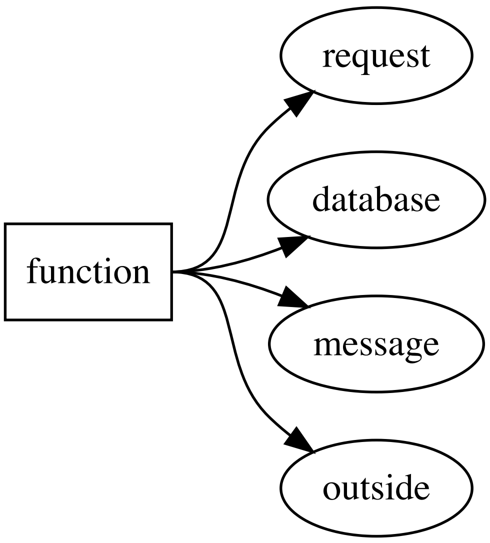
I can't function without you

Let's talk some more about **functions**! 5-4-1

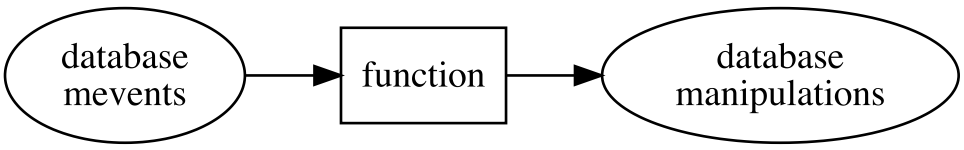
They can be **triggered** by a bunch of things... 5-4-2



...and **do** a lot of things!



Let's now focus on **this scenario**:



By using the **functions** package...

```
const funcs = require("firebase-functions");
```

...we can get a **reference** to a point in the tree...

```
const ref = funcs.database.ref(path);
```

...where we **register a listener** on events

```
ref.onWrite(handler);
```

The **path** we pass in often wants to **filter** some parts:

```
const path = "/comments/{postId}/{commentId}";
```

The **handler** we use gets two things passed in: *5-4-7*

```
function handler(change, context) {
  // ...
}
```

- `change` has the **new data**, and methods to manipulate the DB
- `context` has **metadata** such as the `postId` and `commentId` from the path

In other words there are **many parallels** with the client-side code! *5-4-8*

Q How do we **test** this code? Must we... *5-4-9*

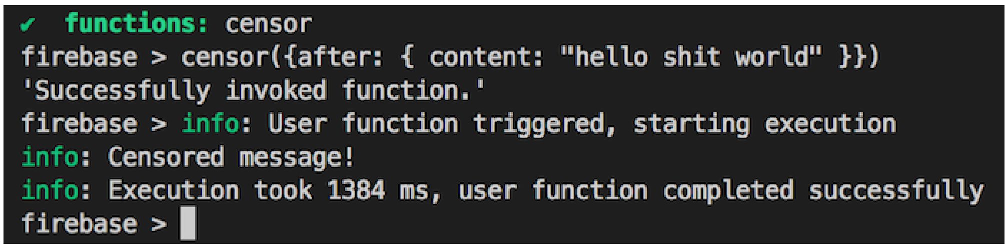
1. deploy them
2. trigger them
3. check the online logs

...because that seems like a big hassle?

A Fortunately, **no**! This is where the **shell** comes in. *5-4-10*

```
firebase functions:shell
```

That command will open a **REPL**, where **all functions are available**. Simply **call your function with some test data**, and see what happens! *5-4-11*



```
✔  functions: censor
firebase > censor({after: { content: "hello shit world" }})
'Successfully invoked function.'
firebase > info: User function triggered, starting execution
info: Censored message!
info: Execution took 1384 ms, user function completed successfully
firebase > 
```

The exact **format of the input data** (and options) vary with **trigger type** - see full docs here: *5-4-12*

https://firebase.google.com/docs/functions/local-emulator

If you don't want to type the test code inline in the shell, you can pipe it in from a file: 5-4-13

```
firebase functions:shell < myFileWithTestCode.js
```

While the shell is for testing during development, there's also a **unit test** tool: 5-4-14

```
npm install --save-dev firebase-functions-test
```

It gives you **similar test capabilities**, allowing you to express unit tests in a convenient form.

More details here: https://firebase.google.com/docs/functions/unit-testing

As a final note - the firebase team have a repo full of **function examples** here: 5-4-15

https://github.com/firebase/functions-samples

5-5. Exercise - censoring

Function gymnastics

Let's keep the peace by **moderating messages**! 5-5-1

(idea stolen from firebase function samples, sorry)

5-5-2

- a set up functions in project
- b add dependencies
- c add censor file
- d publish functions to server
- e try it out!

a First we **set up project** to use functions! 5-5-3

Add this to the `firebase.json`:

```
  "functions": {
    "source": "functions"
  }
```

And, of course, **create the corresponding folder** with an `index.js` file: 5-5-4

```
/functions
  index.js
```

b Next, **set up the dependencies**! Add a `package.json` file next to `index.js`... 5-5-5

```
/function
  package.json
```

...and make it valid JSON by adding this inside it:

```
{}
```

Add the following **dependencies**: 5-5-6

```
npm install firebase-functions firebase-admin bad-words
```

Observe that we do this **in the `functions` dir**, and *not* for the root `package.json`!

Q Wait - **`firebase-admin`**? 5-5-7

A This is for giving server code **admin priviliges**. Sometimes we need to interact with it, but for this function it is enough to just include it. 5-5-8

More info here:

https://firebase.google.com/docs/admin/setup

C Now for the actual coding part - let's make our **censoring function** inside `index.js`! 5-5-9

It should

- trigger on every comment write, and
- rewrite the comment if it has bad words!

We **register an onWrite function** like this: 5-5-10

```
const functions = require("firebase-functions");

exports.censor = functions.database
  .ref("/comments/{postId}/{commentId}")
  .onWrite((change, context) => {
    // implementation goes here
  });
```

We can **get the new message** object like this: 5-5-11

```
change.after.val();
```

And **update it** like this:

```
return change.after.ref.update(cleanedMessage);
```

To make Firebase really happy, your function should `return`... 5-5-12

- `null` if it didn't do anything
- the `.update` return value (hence `return` on the previous slide)

You should also use `console.log` to show what you're doing, since that will show up in the online function logs. 5-5-13

So how do we **determine if a message contains swearing**? And how do we clean it? 5-5-14

We use the `bad-words` package!

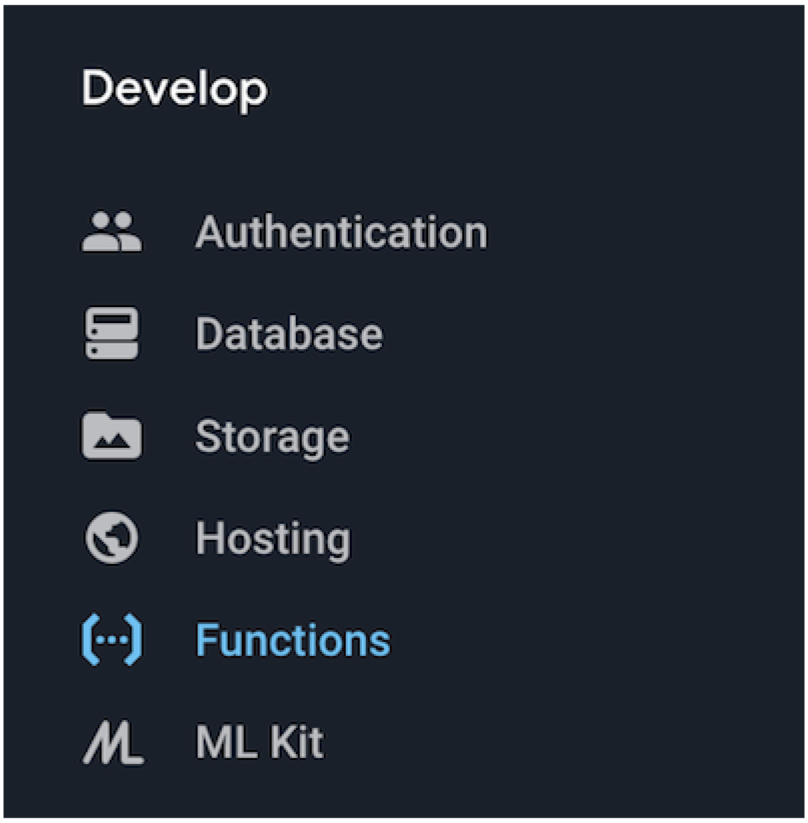
See instructions here: https://www.npmjs.com/package/bad-words d Once you think you have something that might work, do... *5-5-15*

```
firebase deploy --only functions
```

This will *5-5-16*

- **push** the contents of the **function source folder** to the firebase server
- **register all exports** from the `index.js` file as functions and **hook them up**

Go to the **Functions** part in the console... *5-5-17*



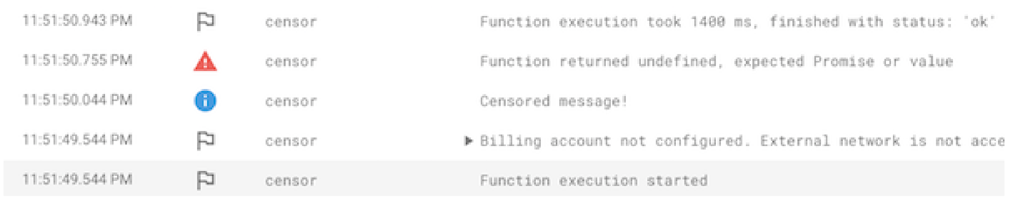
...and you should see your function in the dasboard list!

e Time to **try it out**! *5-5-18*

Visit your blog and **add some new vulgar comments**.

If the censor function is working, the comment should change after a few seconds!

How do we debug this if that doesn't happen? Visit the **log** tab in the dashboard! 5-5-19



```
11:51:50.943 PM   censor   Function execution took 1400 ms, finished with status: 'ok'
11:51:50.755 PM   censor   Function returned undefined, expected Promise or value
11:51:50.044 PM   censor   Censored message!
11:51:49.544 PM   censor   ▶ Billing account not configured. External network is not acce
11:51:49.544 PM   censor   Function execution started
```

You can also download the logs to the terminal by: 5-5-20

```
firebase functions:log
```

And, of course, you can also **use the shell** to develop and debug your function! 5-5-21

chapter 6 / 8

Even more Firebase

Hearts on fire

Sections in this chapter:

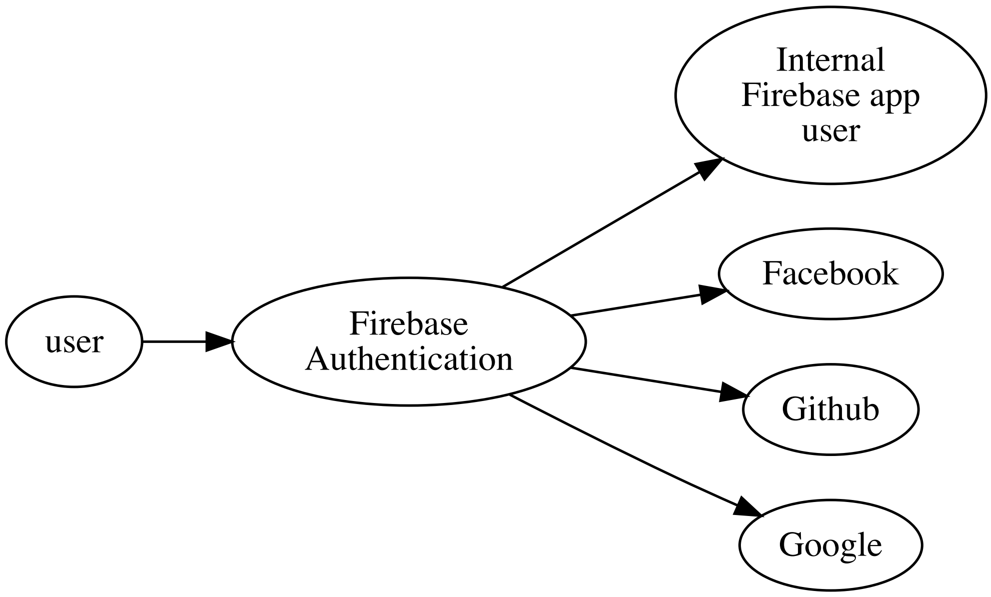
6-1. Auth

Who goes there

The Firebase platform has its own **authentication** product. *6-1-1*

In other words, a direct competitor of for example Auth0.

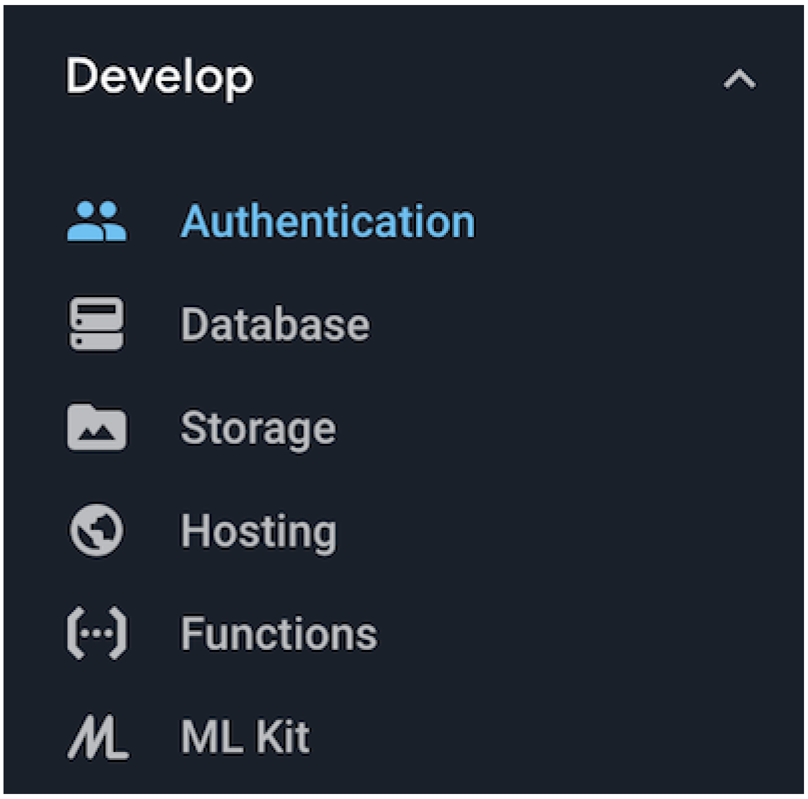
Like many such services, Firebase authentication can both **handle accounts** and/or **integrate with other account givers**: *6-1-2*



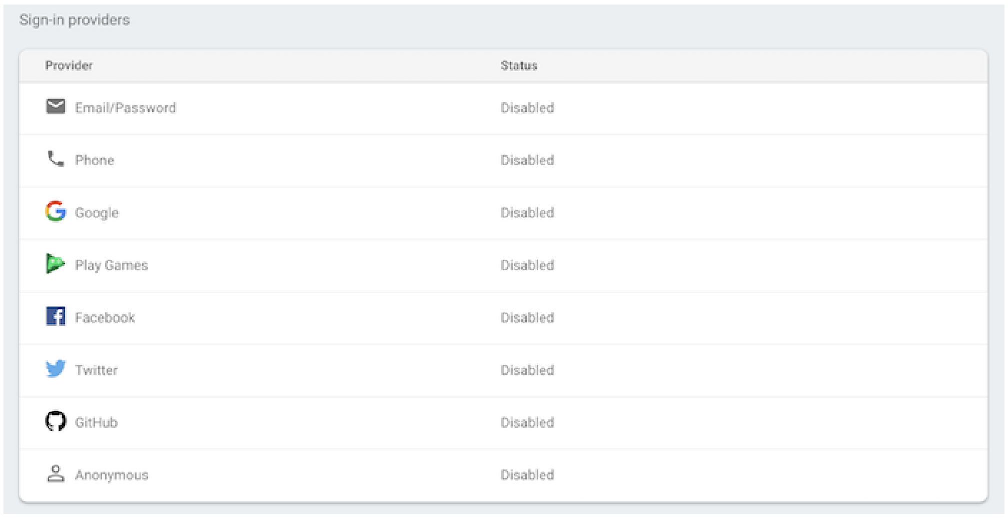
Since Firebase authentication is both **simple and convenient**, there are many apps who use it **without using a Firebase database** or any other Firebase product. *6-1-3*

But most commonly we use it to **protect some part of our Firebase databse data**. *6-1-4*

Authentication is set up per project in the **Firebase console**: *6-1-5*



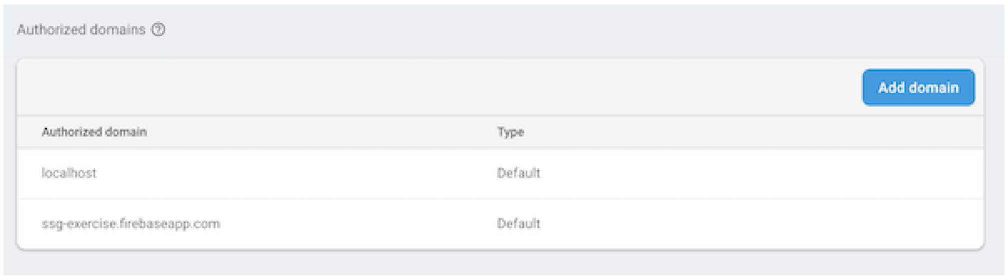
There is a whole range of **different providers**: *6-1-6*


Sign-in providers

Provider	Status
Email/Password	Disabled
Phone	Disabled
Google	Disabled
Play Games	Disabled
Facebook	Disabled
Twitter	Disabled
GitHub	Disabled
Anonymous	Disabled

Once configured and users have logged in, we can also **manage users and see usage** of our app here in the online console. 6-1-7

We can also control what **domains** we can log in from: 6-1-8


Authorized domains

Authorized domain	Type
localhost	Default
ssg-exercise.firebaseapp.com	Default

We want to **remove localhost** for production!

On our local machines we can also **export all user data**: 6-1-9

```
firebase auth:export out.json --format=json
```

In our client code we can **listen to auth changes**... 6-1-10

```
firebase.auth().onAuthStateChanged(authHandler);
```

...which will be called with `null` or a `user` everytime auth state changes.

The exact shape of the user varies with provider; here is a **Github user**: 6-1-11

```
{
  "localId": "pYMyFF133UV9cN0HHl4xa6KAt042",
  "email": "david@krawaller.se",
  "emailVerified": false,
  "displayName": "David Waller",
  "photoUrl": "https://avatars2.githubusercontent.com/u/249764?v=4",
  "lastSignedInAt": "1541439524000",
  "createdAt": "1541431458000",
  "providerUserInfo": [
    {
      "providerId": "github.com",
      "rawId": "249764",
      "email": "david@krawaller.se",
      "displayName": "David Waller",
      "photoUrl": "https://avatars2.githubusercontent.com/u/249764?v=4"
    }
  ]
}
```

6-2. Exercise - Login

Time to try it - let's make it so **only logged-in people can comment** on our blog! 6-2-1

We'll actually **only add client-side protection** for now. 6-2-2

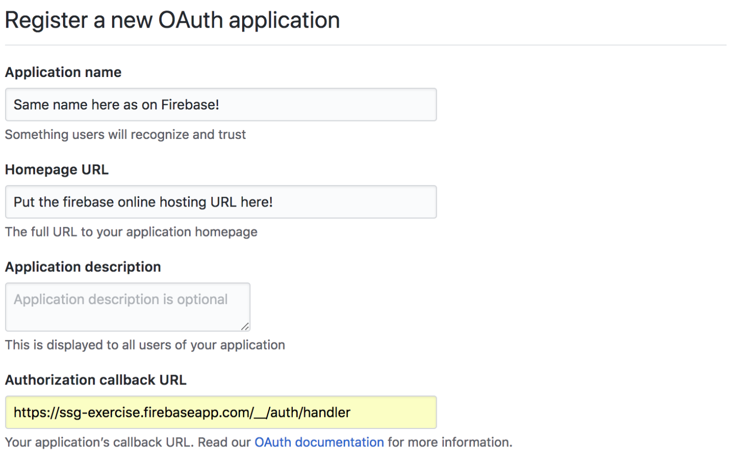
Server-side protection means dealing with the **Security** rules, which will come later!

6-2-3

- a Register a Github app
- b Set up Github Authentication
- c Setup local login
- d Ditch anonymous
- e Hook up to commenting

a Go to https://github.com/settings/applications/new to **register a Github OAuth app**! 6-2-4

Here's what you should fill in: 6-2-5



Register a new OAuth application

Application name

Same name here as on Firebase!

Something users will recognize and trust

Homepage URL

Put the firebase online hosting URL here!

The full URL to your application homepage

Application description

Application description is optional

This is displayed to all users of your application

Authorization callback URL

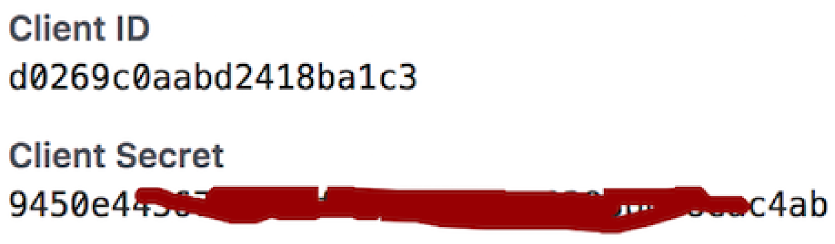
https://ssg-exercise.firebaseapp.com/__/auth/handler

Your application's callback URL. Read our OAuth documentation for more information.

Note the field **`Authorization callback URL`** at the bottom - there you should put... 6-2-6

```
https://<YOUR-APP-ID>.firebaseapp.com/__/auth/handler
```

Click create, and make note of the **Client ID** and **Client Secret**: 6-2-7



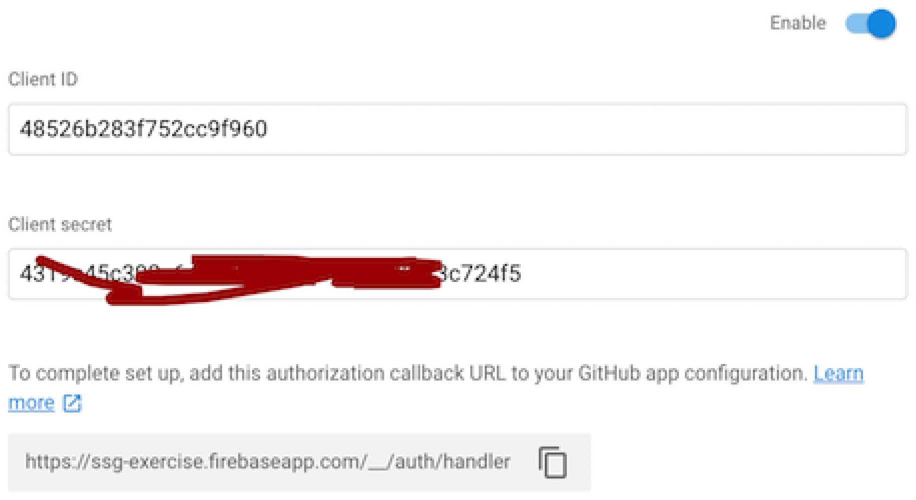
Client ID
d0269c0aabd2418ba1c3

Client Secret
9450e44[redacted]c4ab

b Now time to **connect the Github app to Firebase**! 6-2-8

- Go to **Authentication** in the online console
- Click the **sign-in method** tab
- Click **enable** for Github

In the appearing window, **fill in the Github Client info**: 6-2-9



Enable: on

Client ID
48526b283f752cc9f960

Client secret
431[redacted]3c724f5

To complete set up, add this authorization callback URL to your GitHub app configuration. Learn more

https://ssg-exercise.firebaseapp.com/__/auth/handler

Hit **`Save`**, and you're done!

c To be able to **use the `auth` namespace** we have to **add a reference** in our master template: 6-2-10

```
<script src="https://www.gstatic.com/firebasejs/5.5.7/firebase-auth.js"></script
```

Also in the master template, we want to add **UI for logging in**... *6-2-11*

```
<span id="loggedOut"><button id="logInBtn">Log in</button></span>
```

...and one for **logging out**:

```
<span id="loggedIn">
  Logged in as <span id="userName"></span>
  <button id="logOutBtn">Log out</button>
</span>
```

Something like this, somewhere on the page!

Add a **`login.js` file** to house the logic... *6-2-12*

```
/scripts
  login.js
```

...and include it too in the master template.

In there we want to **set an auth handler**... *6-2-13*

```
firebase.auth().onAuthStateChanged(authHandler);
```

...and **open login popup** for login button clicks...

```
var githubProvider = new firebase.auth.GithubAuthProvider();
firebase.auth().signInWithPopup(githubProvider);
```

...and **log out** for logout buton clicks:

```
firebase.auth().signOut();
```

The **`authHandler`** is called with a `user` or `null`; 6-2-14

```
function authHandler(user) {
  // show or hide the login/logout UI

  window.user = user; // for convenience
}
```

Note how we also **store the user onto `window`, in case other code will need it.

d We **shouldn't be anonymous** anymore! Tweak the code in `comments.js` to use... 6-2-15

```
window.user.email;
```

...instead of `"Anonymous"`.

e Finally we should also set a **auth listener in `comments.js`** that enables/disables the comment field! 6-2-16

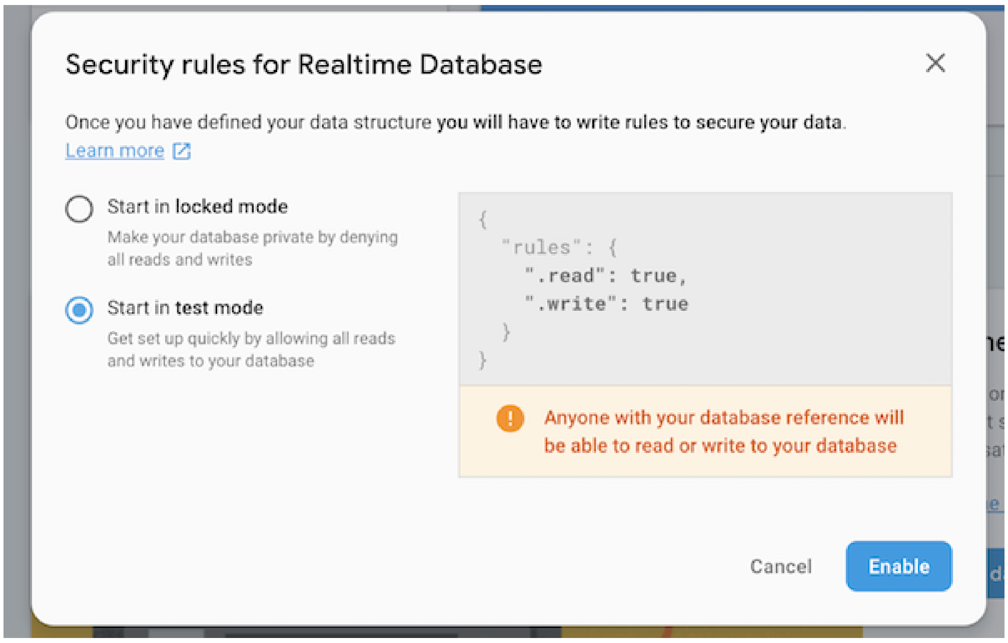
Again, this is of course **only client side protection**. 6-2-17

Proper measures need to be taken for **server-side security** too.

6-3. Security

Breaking the law

Remember when we selected **test mode**? *6-3-1*



Security rules for Realtime Database

Once you have defined your data structure you will have to write rules to secure your data.
Learn more

Start in locked mode
Make your database private by denying all reads and writes

Start in test mode
Get set up quickly by allowing all reads and writes to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

Anyone with your database reference will be able to read or write to your database

It selected this **rule set** for our data: *6-3-2*

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

Click the **rules** tab in the **Database** part of the online console, and you'll see exactly this JSON.

We can (and should) make these rules **part of our source code**. *6-3-3*

If we create a file `database.rules.json` containing the rules, we can **register it in `firebase.json`**...

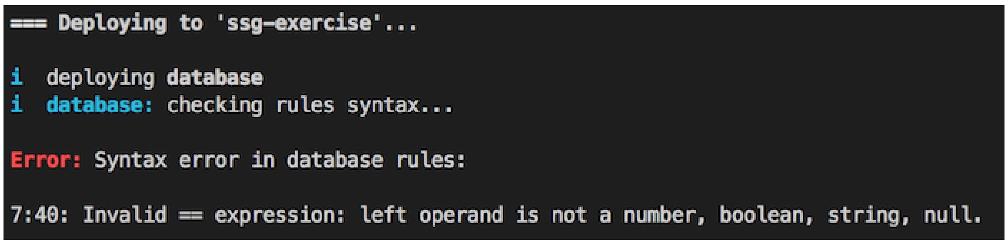
```
{
  "database": {
    "rules": "database.rules.json"
  }
  // other config like hosting etc
}
```

...and then **deploy** that file to the online console: 6-3-4

```
firebase deploy --only database
```

(you'll need to **reload** the console to see it)

When we deploy we'll be **notified of syntax errors**: 6-3-5



```
=== Deploying to 'ssg-exercise'...

i  deploying database
i  database: checking rules syntax...

Error: Syntax error in database rules:

7:40: Invalid == expression: left operand is not a number, boolean, string, null.
```

The rules structure **matches the data**. By saying... 6-3-6

```
{
  ".write": true,
  ".read": true
}
```

...we say reads and writes are allowed from the **top of the pyramid and down**. In other words, everywhere.

But we can set **specific rules for specific parts**: 6-3-7

```
{
  ".read": true,
  "comments": {
    ".write": true
  }
}
```

Now **only `/comments` is writeable**.

We're not limited to `true` and `false` - we can also write **expressions** in a string, like this... 6-3-8

```
"newData.child('content').isString"
```

...or this...

```
"auth != null"
```

Or, if we include **wild cards in our rules tree**... 6-3-9

```
{
  "users": {
    "$uid": {
      ".write": "auth.uid === $uid"
    }
  }
}
```

In other words, the **rule expressions** can contain 6-3-10

- variables referring to the tree path
- special built-in variables and methods

Learn more here: https://firebase.google.com/docs/database/security/

6-4. Exercise - server security

'Cause we don't trust the client

You guessed it - time to **secure our data**! 6-4-1

Your mission - allow writes to `/comments/someRandomGUID` only if... 6-4-2

- there isn't already a comment there
- the `name` child matches `auth.token.email`

Test it by tweaking `comments.js` to...

- allowing non-logged-in users to comment
- putting some nonsense for `name` instead of using `user.email`

For either of these, it should **no longer work**!

To see the needed **rules syntax** for these to things, check:

- https://firebase.google.com/docs/database/security/securing-data
- https://firebase.google.com/docs/database/security/user-security

chapter 7 / 8

Ember

The future is already here — it's just not evenly distributed

Sections in this chapter:

7-1. Ember

It takes a village

First off: **Ember 3**. *7-1-1*

- Ember 3.0 (2018-02-14)
- Ember 3.1 (2018-04-13)
- Ember 3.2 (2018-07-02)
- Ember 3.3 (2018-07-16)
- Ember 3.4 (2018-10-07)
- Ember 3.5 (2018-10-15)

Q Anything in particular we should pay attention to with Ember 3.x? *7-1-2*

A Nah. 🙃 *7-1-3*

The **elephant in the room**: *7-1-4*

Ember is clearly interesting, but it's not a big player.

It's a "third option", dwelling in the shadows of giants like **React** and **Angular**...

...and **Vue**. 7-1-5

(How did *Vue* manage to sneak past *Ember* in terms of popularity? I, what... how?)

At least Ember is still more popular than **Aurelia**. 7-1-6

("Aure-who?" - Exactly.)

Regardless, Ember has been "leading from behind". A lot of the things we like in React/Angular were first innovated in Ember: 7-1-7

- Client-side **router** (+ URLs as application state)
- Opinionated **directory structure**
- Super-helpful **command-line tool**
- **Compilation** as a problem-solver
- **Async** app/tests
- Community **process**

On the "library/framework" scale, Ember has always been "most opinionated". 7-1-8

- **React** and **Vue** are like modern kitchens in a hostel: cook your own food (or heat some ready-made food).
- **Angular** is like a buffet: just pick what you want, but it's already cooked.
- **Ember** is like an à la carte restaurant: you point to a dish, and it'll be prepared for you.

Ember is the **Python** of the web frameworks world. 7-1-9

"Coherent top-to-bottom story, and a rich community aggressively seeking out shared solutions."

"Stability without stagnation." 7-1-10

Since Ember 1.0, the evolution of the framework is already *striking*. (Cf. single index.html, metamorph tags.)

But always guarantees about backwards compatibility + automatic migrations.

Early 2018, there was a drive to write blog posts and opinion pieces about how Ember can reclaim its place in the world. *7-1-11*

`#Emberjs2018`

The story of Ember 3 needs to be about how *so many* good things are in the pipeline. *7-1-12*

The conclusion of the blogging drive was "we need to **ship what we have been working on**".

Stay tuned for **Ember Octane**. *7-1-13*

Note: Ember Octane is not a new *product*, don't worry. It's just a code name, an *edition*, the target version the Ember team is targeting.

"Emphasizing its *modern productivity and performance*."

When? When will this beautiful edition be available. *7-1-14*

In... 2018? Um...

⌚ 📅

Yeah...

Glimmer *7-1-15*

Is not *new*, exactly. It shipped in 1.13.

Using it fully from inside Ember is still rolling out, slowly.

Glimmer is a templating engine architected as a VM. Data gets sent to the client as bytecode. *7-1-16*

It's the only thing that's cooler than React's "virtual DOM".

- Yehuda Katz shows Glimmer compilation
- Tom Dale explains Glimmer at ReactiveConf 2017

Ember Data

7-1-17

Data-persistence *abstraction* layer.

It's the same kind of highly opinionated framework as Ember, but for your "backend" data storage.

Ember Data's richness is in its adapters.

The following is based on two `#Emberjs2018` blog posts:

7-1-18

- My Safari into the future
- Further adventures into the future

New file structure/module unification

7-1-19

Robert Martin's comment about taking in a blueprint at a glance.

Old structure:

7-1-20

```
app/
  components/
    expense-entry.js
    income-entry.js
  templates/
    components/
      expense-entry.hbs
      income-entry.hbs
tests/
  integration/
    components/
      expense-entry-test.js
      income-entry-test.js
```

New structure:

7-1-21

```
src/
  ui/
    expense-entry/
      component-test.js
      component.js
      template.hbs
    income-entry/
      component-test.js
      component.js
      template.hbs
```

Co-location of related files. Makes perfect sense. 7-1-22

The "legacy" layout bunches all views together, all models, all controllers, all services...

👎

Let's try this out! 7-1-23

```
$ npm install -g https://github.com/ember-cli/ember-cli.git
$ MODULE_UNIFICATION=true EMBER_CLI_MODULE_UNIFICATION=true \
    ember new my-cool-app
$ cd my-cool-app
$ EMBER_CLI_MODULE_UNIFICATION=true ember g component my-awesome-component
```

Native JavaScript classes/decorators 7-1-24

Ember started out with its own home-brewed object model.

More and more, projects are leaning on the new ES6+ classes. Ember is getting there too.

Old way: 7-1-25

```
const Foo = Component.extend({ ... });
```

New way:

```
class Foo extends Component { ... }
```

Angle bracket components (landed in 3.4) 7-1-26

A borrow-back from Glimmer. Components now look different from Handlebars helpers.

This is a small but important step towards converging with Glimmer.

Old style: 7-1-27

```
{{site-header user=this.user class=(if this.user.isAdmin "admin")}}
```

New style:

```
<SiteHeader @user={{this.user}} class={{if this.user.isAdmin "admin"}} />
```

Note! Classic invocation syntax has not been deprecated! 7-1-28

You're free to use both. The release announcement suggests you start using the new component syntax, since it is *the future*. The guides will be updated with the new syntax... at some point.

Template-only glimmer components (new in 3.1) 7-1-29

For components that are so simple that they only have a template, no corresponding JavaScript logic.

```
src/
  ui/
    simple-component/
      template.hbs
```

Q Can we use this today? 7-1-30

A Yes! You just need to install the optional-features add-on in order to switch on the feature. 7-1-31

```
$ ember install @ember/optional-features
$ ember feature:enable template-only-glimmer-components
```

Full Glimmer components

7-1-32

Still very experimental, but in the works.

```
@classNames('static-class')
class Tomster extends Component {
  @computed('@filename', '@filetype')
  get tomsterPath() {
    return `/img/${this.filename}.${this.filetype}`
  }
}
```

Asynchronous code

7-1-33

Tasks and microtasks:

```
setTimeout(() => console.log("setTimeout fired"), 0);
Promise.resolve().then(() => console.log("promise resolved"));
```

Which line will be printed first?

Read more about microtask scheduling.

7-1-34

Good news! With Ember 3.0, all supported browsers have reliable, correct microtask scheduling.

7-1-35

Ember can now start to lean on browser microtasks. This means that the dreaded runloop can go away.

Q So… when will it go away?

7-1-36

A Good news! Using `async/await` in your code has been working since Ember 3.2!

7-1-37

Testing

7-1-38

Ember has among the best asynchronous testing stories out there.

The tests are amazingly clean, both unit tests and integration tests.

New-style render test:

```
test('it renders', async function(assert) {
  this.model = {
    filename: "tomster",
    filetype: "png"
  };

  await render(hbs`
    <TomsterLogo
      @filename={{model.filename}}
      @filetype={{model.filetype}}
    />
  `);

  assert.equal(this.element.querySelector('img').alt, 'Tomster Logo');
}
```

Old-style acceptance test:

```
test('should add new post', function(assert) {
  visit('/posts/new');
  fillIn('input.title', 'My new post');
  click('button.submit');
  andThen(() => assert.equal(
    find('ul.posts li:first').text(),
    'My new post')
  );
});
```

New-style acceptance test:

```
test('should add new post', async function(assert) {
  await visit('/posts/new');
  await fillIn('input.title', 'My new post');
  await click('button.submit');

  assert.equal(
    this.element.querySelectorAll('ul.posts li')[0].textContent,
    'My new post'
  );
});
```

Goodbye jQuery

7-1-42

jQuery has been slowly eaten from below (by browsers) and from above (by frameworks).

This has generally been considered a good thing, in the sense that jQuery was only ever a stopgap.

Ember does its part by moving away from jQuery.

7-1-43

Using `ember-fetch` is a good way to do AJAX requests from Ember.

If you want to do the same from Ember Data, there's an `AdapterFetch` in the `ember-fetch` module.

Bonus: TypeScript

7-1-44

Ember will never require TypeScript, but a stated goal is to make Ember+TypeScript just as usable and pleasant as Ember+JavaScript.

Glimmer is already using TypeScript. (And implemented in TypeScript.)

Bonus: Fastboot

7-1-45

Server-side rendering. Has reached 1.0 now, at long last.

See the web page.

Bonus: ember-animated

7-1-46

Really cool library, combining Ember's model data and route information with declaratively expressed animations.

Watch the EmberConf 2018 video with Edward Faulkner explaining the concepts.

7-1-47

In sum,

- Ember is here to stay.
- The vision is certainly there.
- Ember is *ahead* of the curve in some aspects.
- Now the goal is largely to deliver on the started projects.
- Looking forward to Ember Octane!

chapter 8 / 8

Appendix: ES2015

The new shinies

In this appendix chapter we will more fully **explore the new features in ES2015**! *8-0-1*

Sections in this chapter:

8-1. Versatile object definitions

defining objects like a boss

In ES2015 we got five small but nice features for **defining objects in a smoother way**: *8-1-1*

a dynamic keys
b automatic same-key-value
c method shorthand
d getters
e setters

a If we wanted to create an **object with a dynamic key** we had to go about it in a roundabout way before: *8-1-2*

```
let obj = {};
let obj[dynamicKey] = someValue;
```

Now, instead, we can use the **dynamic key syntax** by wrapping it in brackets: *8-1-3*

```
let obj = { [dynamicKey]: someValue };
```

b Also, if our value is in a **variable with the same name as the intended key**, like here: *8-1-4*

```
let person = {
  name: name,
  age: age
};
```

...ES2015 introduces a **shorthand syntax**: *8-1-5*

```
let person = { name, age };
```

c And if we define an **object with a method**: *8-1-6*

```
let obj = {
  method: function(arg1, arg2) {
    // do advanced stuff
  }
};
```

...ES2015 lets us be less verbose by using the method shorthand syntax: *8-1-7*

```
let obj = {
  method(arg1, arg2) {
    // do advanced stuff
  }
};
```

This can also be **combined with the dynamic key syntax**: 8-1-8

```
let obj = {
  [methodName](arg1, arg2) {
    // do advanced stuff
  }
};
```

d Finally, ES2015 also introduced **getters and setters**. 8-1-9

Let's look at **getters** first. They are very useful for dealing with **computed properties**.

Say we're working with **user objects** like this: 8-1-10

```
let user = {
  firstName: "John",
  lastName: "Doe"
};
```

Now we want to implement a **computed property `fullName`**.

Here's an **ES3 solution** doing it as a **method**: 8-1-11

```
let user = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

user.fullName(); // John Doe
```

By using an **ES2015 getter** we can access the computed property normally instead: *8-1-12*

```
let user = {
  firstName: "John",
  lastName: "Doe",
  get fullName() {
    return this.firstName + " " + this.lastName;
  }
};

user.fullName; // John Doe, without invocation!
```

Cf. uniform access principle.

e A **setter** let's you **act upon prop mutation**, for example **logging**... *8-1-13*

```
let user = {
  set userName(str) {
    log(this._userName + " changed name to " + str);
    this._userName = str;
  }
};

user.userName = "Steve"; // Bob changed name to Steve
```

...or **validation**: *8-1-14*

```
let user = {
  set userName(str) {
    if (str.match(/[^a-z]/)) {
      throw "Name can only contain lowercase letters!";
    }
    this._userName = str;
  }
};

user.userName = "Bob the 1 and only"; // Name can only contain..
```

```
question)

Did you note that we used a **different property name** inside the setter? The s

Why do you think that is?
```

answer)

If we mutated the same property inside the setter then that would trigger the setter to be called, which would mutate the property, which would trigger the setter, etc. We would end up in an **infinite loop**.

8-2. Destructuring and rest

cherry-picking the raisins from the cookie

Destructuring is a way to pick values from nested structures without having to do the manual digging. *8-2-1*

Let's say we have an **array of `contenders`**, each represented by an object. *8-2-2*

```
let contenders = [
  { name: "David", age: 39 },
  { name: "Carl", age: 37 }
  /* and a few others */
];
```

They are **sorted by position** so the first contender won, etc.

If we wanted the **name of the winner** we would do something like this in ES5: *8-2-3*

```
let winnersName = contenders[0].name;
```

With **destructuring**, we can instead do this: *8-2-4*

```
let [{ name: winnersName }] = contenders;
```

Or, combined with the **same-key-value shorthand**:

```
let [{ name }] = contenders;
```

Destructuring also allows us to use the powerful **rest** element which can **lump up many array elements into one**, making for some very succinct code: *8-2-5*

```
let [winner, ...losers] = contenders;
```

Note that the rest element **has to be the last one in the array**, so this wouldn't work: *8-2-6*

```
let [...others, superloser] = contenders; // syntax error
```

Q Wait.. Theoretically, **the rest could be placed *anywhere***, as long as there's just one. The parser should still be able to figure out what's what! *8-2-7*

Right?

A True. But that would **require lookahead**, which is **complex and more taxing**. And so the choice was made to only allow the rest element in the last position. *8-2-8*

8-3. Versatile function definitions

defining function like a boss

ES2015 provides **several neat features for defining functions**: *8-3-1*

a default parameter values
b rest parameters
c destructuring parameters
d arrow functions

a **Default parameter values** exist in many languages, and was popularised in JS through CoffeeScript. *8-3-2*

The idea is to **handle optional parameters** in a smoother way.

Creating a **function with an optional parameter** in ES3 meant we had to do a sometimes tedious dance of initialization: *8-3-3*

```
function makePerson(name, age) {
  let age = age || "unknown";
  // do complex stuff
}
```

This may or may not do what you want. (Hint: is `0` a reasonable value for `age`?)

With **default parameter values** we can instead do this: *8-3-4*

```
function makePerson(name, age = "unknown") {
  // do complex stuff
}
```

b The second new feature, **rest parameters**, is a way of capturing multiple arguments into a single variable like a rest element in a destructuring. *8-3-5*

This can often save us from having to do awkward stuff with the not-quite-an-array `arguments` object.

Imagine a `competition` function that is called with all contenders one by one: *8-3-6*

```
function competition() {
  let contenders = Array.prototype.slice.call(arguments);
  let winner = contenders[0];
  let losers = contenders.slice(1);
  // do something with winner and losers
}
```

Using rest parameters, this function simply becomes: *8-3-7*

```
function competition(winner, ...losers) {
  // do something with winner and losers
}
```

Note that the rest parameter **has to be the last parameter**, just like the rest element, and for the same reason. *8-3-8* c Remember **destructuring**? We can **use that in signatures**: 8-3-9

```
function introduce({ name, age }) {
  console.log(name, "is", age, "years old");
}
let me = { name: "David", age: 39 };
introduce(me); // David is 39 years old
```

d Finally - know how **defining anonymous functions** in JS is rather verbose? 8-3-10

```
let mcboatify = function(arg) {
  return arg + "y Mc" + arg + "Face";
};
```

With **arrow functions** things feel less heavy: 8-3-11

```
let mcboatify = arg => {
  return arg + "y Mc" + arg + "Face";
};
```

They can become smaller still - if we have **exactly one parameter**, we can **omit the parenthesis** in the signature: 8-3-12

```
let mcboatify = arg => {
  return arg + "y Mc" + arg + "Face";
};
```

Finally, if you **just want to return an expression**, we can **skip brackets and the return keyword**: 8-3-13

```
let mcboatify = arg => arg + "y Mc" + arg + "Face";
```

Now the function body consists of a single expression, which will be implicitly returned.

Note however that if you want to use the **single expression form with an object literal**, we have to **wrap it in parenthesis** to distinguish it from a regular function block: 8-3-14

```
let createUser = (name, age) => ({ name, age });
```

Arrow functions are not only less heavy to write, they are also lighter for the interpreter since they **don't get an implicit context parameter**. 8-3-15

Which means that if you refer to `this` inside an arrow function, it is the **same `this` as on the outside**. 8-3-16

```
let me = this;
setTimeout(() => {
  console.log(this === me); // true
}, 10);
setTimeout(function() {
  console.log(this === me); // false
}, 10);
```

As a final note; arrow functions can beautifully describe the flow for **nested higher order callbacks**. For example, this... 8-3-17

```
function multiplier(func, times) {
  return function() {
    for (let i = 0; i < times; i = i + 1) {
      func();
    }
  };
}
```

...could be written as this: 8-3-18

```
let multiplier = (func, times) => () => {
  for (let i = 0; i < times; i = i + 1) {
    func();
  }
};
```

8-4. Spreads

the dark side of rests

You have already seen how we use **rest** element/parameter to capture several array elements into a single variable: 8-4-1

```
let [winner, ...losers] = competitors;
```

Now imagine **the opposite scenario** - we have the `winner` and `losers` variables, and want to define `competitors`. In ES3 this is done like this: 8-4-2

```
let competitors = [winner].concat(losers);
```

ES2015 gives us a new options - **spreads**! It looks exactly like rest, but we use it on the *right side* instead (or when we *call* a function as opposed to when we define it): 8-4-3

```
let competitors = [winner, ...losers];
```

We say that we *spread* the contents of the expression into the outer array.

Spreads gives us a less verbose way to **copy an object and add properties to it**, which is otherwise done like this: 8-4-4

```
let augmentedObj = Object.assign({}, oldObj, newProps);
```

With spreads we can instead do this: 8-4-5

```
let augmentedObj = { ...oldObj, ...newProps };
```

Note that while spreads and rests *with arrays* are in the spec for ES2015, **object spread came in ES2018**. 8-4-6

8-5. Modules

getting into the import/export business

As we saw earlier, **Node gave us modules** through the `require` and `module.exports` globals it provides. *8-5-1*

But with ES2015, we got **native modules** for the very first time!

~~~
While **Node modules** followed the **CommonJS module standard**, what was imple

But the **concepts are the same**. While you would do this in **CommonJS**...

```javascript
// file1.js
module.exports = {..};

//file2.js
let lib = require("./file1.js");
```
~~~

...you would do this with **ES modules**:

```
// file1.js
export const lib = {..};

//file2.js
import lib from './file1.js'
```

We have to **name our exports** here, otherwise things are **pretty similar**.

```
There are **other differences too**, so for the full scope you should **check th
```

Note that even though this is now **part of the language**, there are **no browsers that implement the functionality yet**.

This is mainly because it **wouldn't be practical** - we'd get a **gazillion http requests for small files**.

```
And since we **likely have a build step anyway** to do minification and transpil

But, with the advent of HTTP2, **who knows what the future will hold**!
```

8-6. Classes

Waiter, there are classes in my JS!

Before ES2015, JavaScript used to famously **lack classes**. *8-6-1*

This was **not an oversight**. Consider what **classes are normally used for**:

- **resusing functionality** and
- setting up **hierarchies**

In **JavaScript** this is addressed by *8-6-2*

- simply **grabbing methods** and/or **mixing objects**
- **prototypal "inheritance"**, which should really be called delegation

This means that **classes didn't really serve a purpose**. Yet they were **still frequently used**, through the weird, bolted-on **`new`** syntax which **makes functions behave like constructors**: *8-6-3*

```
let user = new User("David", 1979);
```

But to really make this **behave like normal classes**... *8-6-4*

```
let lucas = new Dog("Lucas");
lucas instanceOf Dog; // true
lucas instanceOf Animal; // true
lucas.bark(); // Lucas goes woof!
```

...then lots of **jumping through hoops** had to be done: *8-6-5*

```
Dog.prototype = new Animal();
Dog.prototype.constructor = Animal;
Dog.prototype.bark = function() {
  console.log(this.name, "goes woof!");
};
```

To **facilitate "class" use** in JavaScript, **ES2015 introduced the `class`** syntax: *8-6-6*

```
class Dog extends Animal {
  bark() {
    console.log(this.name, "goes woof!");
  }
}
```

Note how **method shorthands** are available in class declarations too!

But it is important to note that this does **not mean that JavaScript has actual classes.** *8-6-7*

Under the hood the same weird `prototype` and `constructor` dance happens.

Still, since the **syntax hides the mismatch**, it can be a **convenient way to package functionality**, so let's **check out some details**! Specifically: *8-6-8*

- a constructor
- b methods
- c properties

a First off, what **used to go in the fake constructor**... *8-6-9*

```
function Animal(name) {
  this.name = name;
}
```

...is now placed in a literal **`constructor` method** in the class declaration: *8-6-10*

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}
```

If you want **the inherited constructor to be invoked too**, you must **do so yourself** with the new **`super` keyword**: 8-6-11

```
class Dog extends Animal {
  constructor(name) {
    super(name);
    this.nickname = name + "y boy";
  }
}
```

b And you've **already seen methods**: 8-6-12

```
class Dog extends Animal {
  constructor() { ... }
  bark() {
    console.log(this.name,"goes woof!");
  }
}
```

Similar to object methods, **`this`** (normally) **points to the instance**.

c Finally, as you saw, **properties are normally initialised in the constructor**: 8-6-13

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}
```

...but when we use **ES2018** (or TypeScript) we can also **initialise properties directly on the class declaration**: 8-6-14

```
class Dog {
  numberOfLegs = 4;
}
```

So, to **recap**: 8-6-15

- classes are just a **light syntactic sugar** introduced in ES2015
- we normally **don't need them in JavaScript**
- but they are a **convenient way to bundle related functionality**

8-7. Miscellaneous

odds and ends

There's three more things worth mentioning: *8-7-1*

- a declaring variables with **`let`**
- b declaring variables with **`const`**
- c **template strings**

a Variables in JavaScript have **functional scope**. *8-7-2*

Even if you declare them inside an **if-block in the middle of a function**, the variable is still **visible throughout the entire function**.

So when you write this... *8-7-3*

```
function myFunc(arg, lib) {
  if (arg === 42) {
    var ret = lib.method() + 7;
    return ret;
  }
  // do sth else
}
```

...this is what (conceptually) happens: *8-7-4*

```
function myFunc(arg, lib) {
  var ret;
  if (arg === 42) {
    ret = lib.method() + 7;
    return ret;
  }
  // do sth else
}
```

In other words, the **declaration is hoisted to the top**.

This is generally considered a **design mistake**, and can give rise to **weird bugs**. 8-7-5

ES6 therefore introduces **`let` as an alternative to `var`** for declaring variables, and the **only difference** is that **`let` has block scope**.

b In most languages there's a way to **define constants**, meaning a **variable that cannot change**. 8-7-6

This is **missing from JavaScript**.

A common "hack" is to **name constants in all capitals**: 8-7-7

```
var SOME_CONST = 42;
```

But this has **no technical significance**, it is just a hint.

ES6 therefore introduces **`const` as another alternative to `var`**, and the **only difference** is that you **cannot reassign the value**. 8-7-8

```
const answer = 42;
answer = 43; // throws an error
```

c Finally, **template strings**! 8-7-9

```
let userTempl = `
  First name: ${user.fname}
  Last name: ${user.lname}
`;
```

As you saw, template strings... 8-7-10

- are **defined inside two backticks**
- can **contain linebreaks**
- allow **interpolation inside ${}**

There's also a **semi-secret way to invoke functions with templates**. Here's an example from Choo:

```
html`
  <main class="app">
    Count: ${state.counter.count}
    <button onclick=${e => send("counter:increment")}>+</button>
  </main>`;
```

The **`html` function is invoked** with the templates and interpolated values.

External links

2-4-19 Broken promises blogpost: https://medium.com/@avaq/broken-promises-2ae92780f33

3-3-1 front-matter: https://www.npmjs.com/package/front-matter

3-3-1 fs-extra: https://www.npmjs.com/package/fs-extra

3-3-1 marked: https://www.npmjs.com/package/marked

3-3-4 Markdown: https://daringfireball.net/projects/markdown/

3-3-7 marked: https://www.npmjs.com/package/marked

3-3-10 front-matter: https://www.npmjs.com/package/front-matter

3-3-12 YAML: http://yaml.org/

3-5-4 template literals: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

6-1-1 Auth0: https://auth0.com/

7-1-10 single index.html, metamorph tags: https://www.emberjs.com/blog/2017/04/05/emberconf-2017-state-of-the-union.html

7-1-16 Tom Dale explains Glimmer at ReactiveConf 2017: https://www.youtube.com/watch?v=nXCSloXZ-wc

7-1-16 Yehuda Katz shows Glimmer compilation: https://www.youtube.com/watch?v=vL8sCi1Bv6E

7-1-17 adapters: https://emberobserver.com/categories/ember-data-adapters

7-1-18 Further adventures into the future: https://medium.com/@chrisdmasters/emberjs2018-further-adventures-into-the-future-b0854cab9155

7-1-18 My Safari into the future: https://blog.usejournal.com/emberjs2018-my-safari-into-the-future-e4f31a4902ea

7-1-19 taking in a blueprint at a glance: https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

7-1-31 optional-features: https://github.com/emberjs/ember-optional-features

7-1-34 microtask scheduling: https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/

7-1-35 dreaded runloop: https://guides.emberjs.com/v1.10.0/understanding-ember/run-loop/

7-1-45 web page: https://ember-fastboot.com/

7-1-46 the EmberConf 2018 video: https://www.youtube.com/watch?v=4JofVQ3nGrw

8-1-7 method shorthand syntax: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions

8-1-12 uniform access principle: https://en.wikipedia.org/wiki/Uniform_access_principle

8-2-1 Destructuring: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

8-7-11 Choo: https://github.com/yoshuawuyts/choo