# Internetværk og Web-programmering
# Web-services and Web-APIs (in Node.js)

Forelæsning 8
Michele Albano

Distributed, Embedded, Intelligent Systems

DEIS

## Agenda

- SSE and Websockets
- Service Oriented Applications
  - basic notions (service producer, consumer, etc)
  - Stateless server, HATEOAS, idempotent operations
- Representational State Transfer (REST)
- Frameworks, and API Code generation
- Testing with Postman

## Server-Sent Events

- In HTTP (and HTTPS), the client initiates communication
  - Web browser
  - `fetch()`
- Some web apps need to receive data from the server *when the server wants*
- Solution: Server-Sent Events:
  - The client makes a request to the server
  - Connection is kept open
  - When the server has data to send, it writes them to the connection
- From the client’s point of view, the server answers client’s request in a slow and bursty way with pauses
- Network connections will close automatically after some time
  - The client reopens the connection (repeats the initial request) whenever it detects the connection was closed

# How is SSE?

- Pros:
  - Quite efficient (low delays)
- Cons:
  - Consumes resources on the server (TCP port etc for the active connection)
- Practical usage: `EventSource` API
- Client-side:
  - The client creates a `EventSource` around the URL of the server
  - The client receives events through the `EventSource`

```
let ticker = new EventSource("stockprices.php");
ticker.addEventListener("bid", (event) => {
   displayNewBid(event.data);
}
```

String describing the “type” of the event, as set by the server

String sent by the server

# SSE: server-side

- It must have an endpoint to receive the `EventSource` objects and save them:

```
server.on("request", (request, response) => {
// Parse the requested URL
let pathname = url.parse(request.url).pathname;
if (pathname == "/chat" && request.method == "GET“) {
    clients.push(response);
}
```

- Whenever data must be sent back, use the `response` functions:

```
clients.forEach(client => client.write(”event: chat\ndata: hallo\n\n”));
```

event: chat
data: hallo
data: whatever
**Two newlines**

## SSE full example

- Let us now take a look at the example from the book

## Websockets

- Introduced with HTTP5 (created 2008, recommended 2014)
  - State-full, Full duplex peer to peer, Low latency, String and binary protocols.
- Keeps a connection open for two-way communication
- Allows the webserver to send content without first being requested

- **Lifecycle**
  - One peer (a client) initiates the connection by sending an HTTP handshake request.
  - The other peer (the server) replies with a handshake response.
  - The connection is established. From now on, the connection is completely symmetrical.
  - Both peers send and receive (binary) messages.
  - One of the peers closes the connection.

**State of the WebSocket:**

`WebSocket.CONNECTING`

This WebSocket is connecting.

`WebSocket.OPEN`

This WebSocket is ready for communication

`WebSocket.CLOSING`

This WebSocket connection is being closed

`WebSocket.CLOSED`

Either the WebSocket has been closed (no further communication), or initial connection attempt failed

## Practicality for websockets

Server-side, need to install the module:

```
npm install websocket
```

- The URL does not start with HTTP/HTTPS
  - It uses WS/WSS
  - Most browsers block WS for security reasons
- The browser first establishes an HTTP connection, then sends an Upgrade: websocket header requesting to switch to the WebSocket protocol
- Both client and server must now speak the WebSocket protocol

## Websockets small example

- Let us now take a look at some code

# Agenda

- SSE and Websockets
- Service Oriented Applications
  - basic notions (service producer, consumer, etc)
  - Stateless server, HATEOAS, idempotent operations
- Representational State Transfer (REST)
- Frameworks, and API Code generation
- Testing with Postman

## Service-oriented architectures 1/3

- *SOA* is a set of principles for the design, deployment and management of both applications and software infrastructure using sets of *loosely coupled* services that can be *dynamically discovered* and that allow a *producer* and a *consumer* to *communicate with each other* or are *coordinated* through *choreography* to provide *enhanced services*
- Four properties of a service:
  - It is a black box for its consumers
  - It is self-contained (loose-coupling between services)
  - It may consist of other underlying services
  - It should be stateless

## Service-oriented architectures 2/3

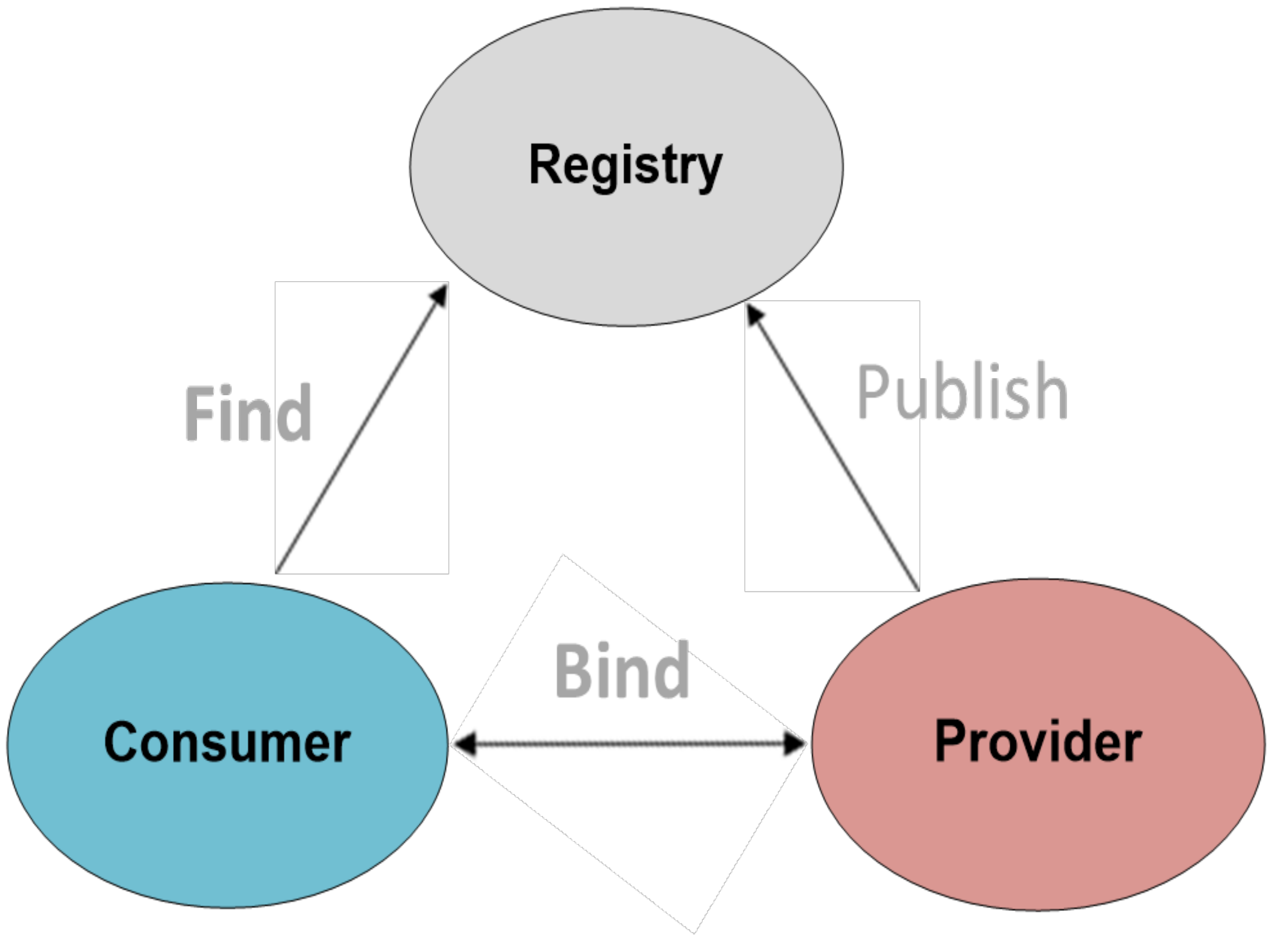
- It is a black box for its consumers
    - A service presents a simple interface articulated in endpoints to the requester that abstracts away the underlying complexity
    - The SOA infrastructure will provide standardized access mechanisms to *discover* services with service-level agreements
- It is self-contained (loose-coupling between services)
    - The consumer of the service is required to provide only the data stated on the interface definition, and to expect only the results specified on the interface definition
    - In the context of web services, loose coupling refers to minimizing the dependencies between services in order to have a flexible underlying architecture (reducing the risk that a change in one service will have a knock-on effect on other services)

## Service-oriented architectures 3/3

- It may consist of other underlying services
    - It allow users to combine and reuse them in the production of applications
    - It should be based on open standards.  Open standards ensure the broadest integration compatibility opportunities
- It should be stateless
    - The service does not maintain state between invocations
    - If a transaction is involved, the transaction is committed and the data is saved somewhere
        - The id of the transaction can provide context (sessions)

## Design patterns

- 3 main roles:
  - Service provider / publisher
    - Offers the service, registers it in the service broker for consumers to find
  - Service consumer
    - Retrieves service providers from service broker, then uses the services
  - Service broker / registry / repository



# SOLID principles (OOP, etc)

- Single-responsibility Principle: each component should do just one job
- Open/Closed Principle: your component should be open for extension but closed for modification
- Liskov Substitution Principle: if q(x) is a property provable about x of type T, q(y) should be provable for objects y of type S where S is a subtype of T
- Interface Segregation Principle: clients should not be forced to depend upon interfaces that they do not use
- Dependency Inversion Principle: an interface is an abstraction between a higher and a lower level component
    - Abstractions should not depend on details. Details should depend upon abstractions

## SOA and SOLID

- Single-responsibility Principle: each service should be specialized
- The Open/Closed Principle: service orchestration
- The Liskov Substitution Principle: clients consume services, which consume lower granularity services
- The Interface Segregation Principle: in place of using one fat service interface, we create multiple small services
- The Dependency Inversion Principle: you can replace interface implementations without changing the interface

## Microservice Architecture

- MSA is a modern interpretation of SOA
- Services are still processes that communicate with each other over the network
- This style emphasizes continuous deployment and other agile practices

## SOA vs Microservices: principles

- MSA stresses more on:
  - fine-grained interfaces (to independently deployable services),
  - business-driven development (e.g. domain-driven design),
  - services are autonomous,
  - polyglot programming and persistence,
  - lightweight container deployment,
  - decentralized continuous delivery.

## SOA vs MSA: more info 1/3

- Service Granularity:
  - MSA: generally single-purpose services that do one thing really, really well
  - SOA: service components can range in size. Coarse-grained to be useful to more applications
- Component Sharing:
  - one of the core tenets of SOA.
  - MSA tries to minimize on sharing through “bounded context” (coupling of a component and its data as a single unit with minimal dependencies)

## SOA vs MSA: more info 2/3

- Middleware vs API layer:
  - MSA provides an API layer
  - SOA has a messaging middleware component (provides mediation and routing, message enhancement, message, and protocol transformation)
- Remote services:
  - SOA architectures rely on messaging (AMQP, MSMQ)
  - Most MSAs rely on two protocols – REST and simple messaging (JMS, MSMQ), and the protocol found in MSA is usually homogeneous.

## SOA vs MSA: more info 3/3

- Heterogeneous interoperability:
  - SOA promotes the propagation of multiple heterogeneous protocols through its messaging middleware component: to integrate several systems using different protocols in a heterogeneous environment
  - MSA attempts to simplify the architecture pattern by reducing the number of choices for integration: all your services could be exposed and accessed through the same remote access protocol

## SOA vs MSA: bottom line

- SOA is better suited for large and complex business application environments that require integration with many heterogeneous applications using a middleware component
- Microservices are better suited for smaller and well-partitioned, web-based systems in which microservices give you much greater control as a developer

## Agenda

- SSE and Websockets
- Service Oriented Applications
  - basic notions (service producer, consumer, etc)
  - Stateless server, HATEOAS, idempotent operations
- Representational State Transfer (REST)
- Frameworks, and API Code generation
- Testing with Postman

# Representational State Transfer

- REST is an approach to services with a very constrained style of operation, applying “verbs” to “nouns”
  - Nouns are the URLs that identify web resources
  - Verbs are the HTTP operations *GET*, *PUT*, *DELETE* and *POST (and lately PATCH)* to manipulate resources

- GET to retrieve the representation of a resource
- POST to add a new resource
- PUT to update the representation of a resource using a new one
- PATCH to change part of the representation of a resource
- DELETE to discard a resource

## The tenets of REST

- Resources are identified by uniform resource identifiers (URIs)
- Resources are manipulated through their representations
- Messages are self-descriptive and stateless
- Multiple representations are accepted or sent
- Hypertext is the engine of application state

## Statelessness

- Two kinds of state
- **Application state** is the information necessary to understand the context of an interaction (e.g.: authentication, session)
  - In REST, all messages must include all *application* state as part of the content transferred from client to server back to client
- Changes in **resource state** are unavoidable
  - Someone has to POST new resources before others can GET them
  - REST is about avoiding implicit or unnamed state; resource state is named by URIs
- No application state means:
  - It prevents partial failures
  - It allows for substrate independence
    - Load-balancing
    - Service interruptions

## Hypermedia as the Engine of Application State

- Web application as a state machine:
  - State machines fit into REST when the states are expressed as resources with links indicating transitions
- HATEOAS:
  - hypermedia links in the response contents so that the client can dynamically navigate to the appropriate resource by traversing the hypermedia links
- A REST client hits an initial API URI and uses the server-provided links to dynamically discover available actions and access the resources it needs
  - The client need not have prior knowledge of the service or the different steps involved in a workflow
  - Thus, HATEOAS allows the server to make URI changes as the API evolves without breaking the clients

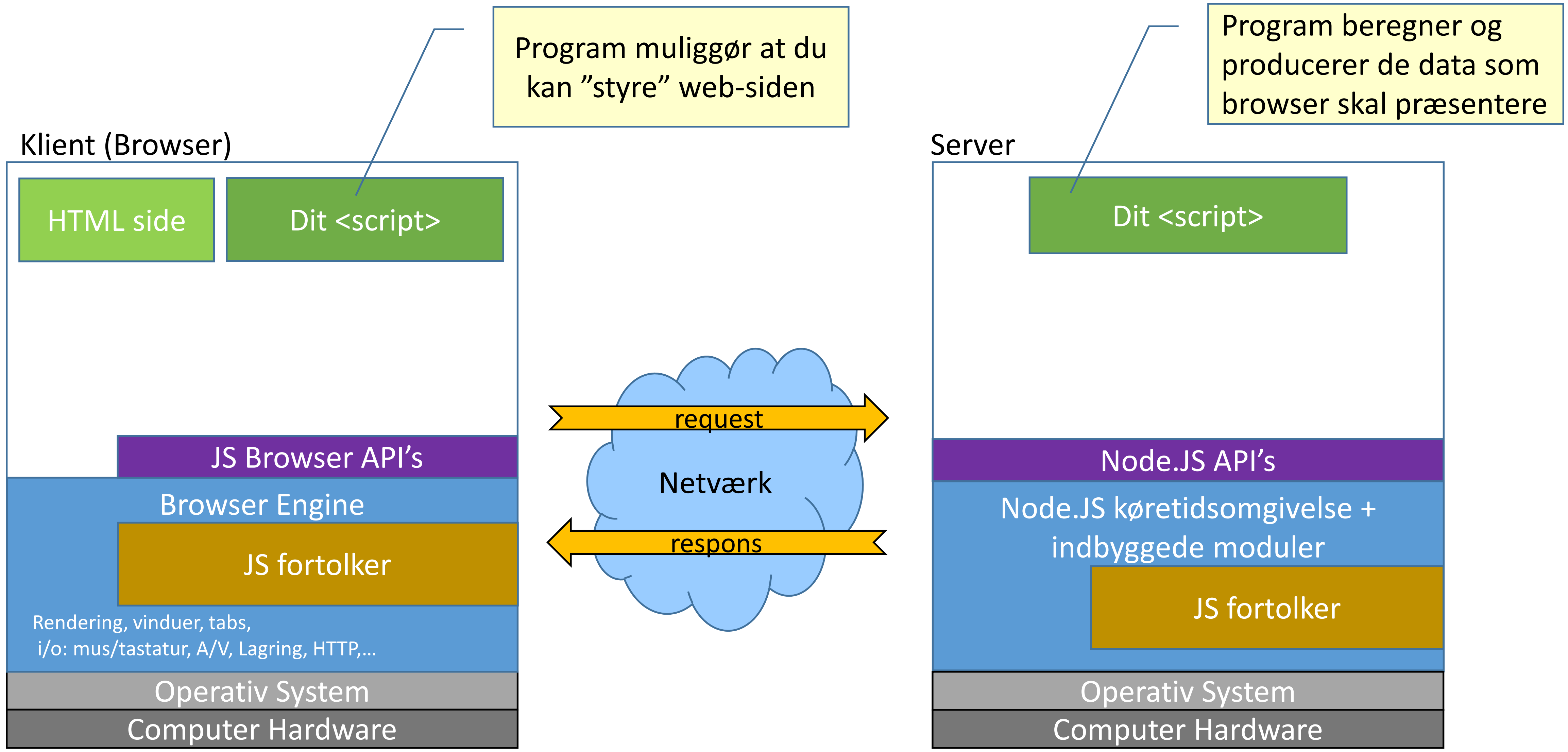
# Idempotent operations

- A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request.
- By this specification, GET, PUT and DELETE methods are idempotent.
- Do multiple requests return the same response every time they are called?
  - GET with Authentication: returns 401 (invalid authentication) vs with valid authentication (e.g.: 200).
  - PUT: either a 201 (resource is created) or 200/204 if the resource is updated
  - DELETE: 204 if the resource is deleted or a 404 if the resource was already deleted
- Due to this, idempotency in REST does not mean that consecutive calls to the same method and resource must return the same response, but rather that consecutive calls to the same method and resource MUST have the same intended effect on the server.

## Agenda

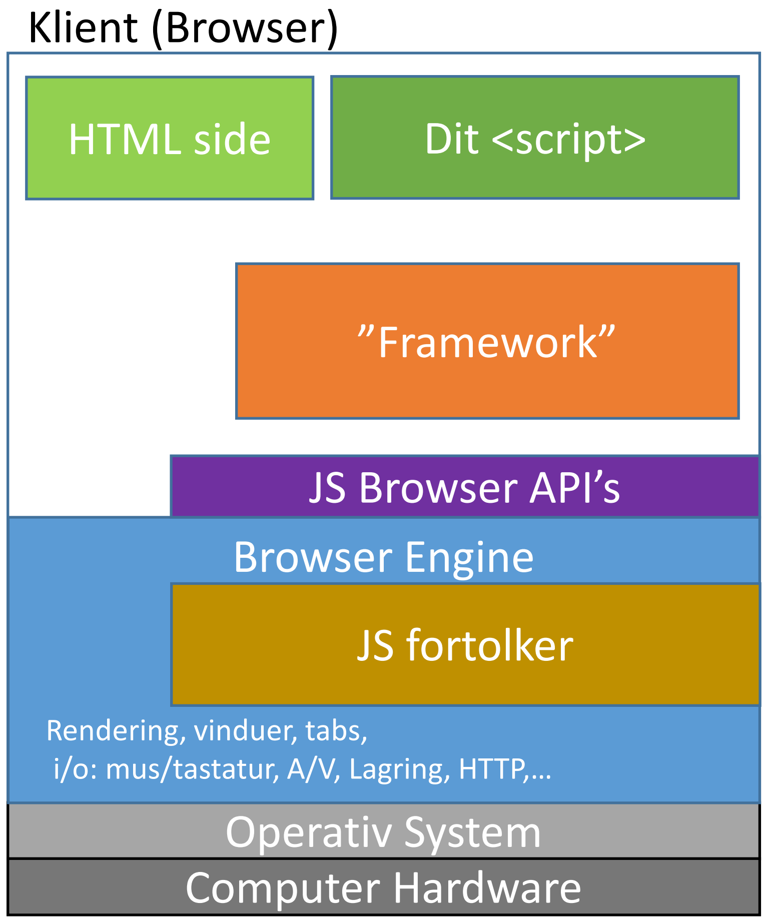
- SSE and Websockets
- Service Oriented Applications
    - basic notions (service producer, consumer, etc)
    - Stateless server, HATEOAS, idempotent operations
- Representational State Transfer (REST)
- Frameworks, and API Code generation
- Testing with Postman

# Overordnet arkitektur: ”frameworks”


Program muliggør at du kan ”styre” web-siden
Program beregner og producerer de data som browser skal præsentere
Klient (Browser)
HTML side
Dit <script>
JS Browser API’s
Browser Engine
JS fortolker
Rendering, vinduer, tabs,
i/o: mus/tastatur, A/V, Lagring, HTTP,…
Operativ System
Computer Hardware
request
Netværk
respons
Server
Dit <script>
Node.JS API’s
Node.JS køretidsomgivelse + indbyggede moduler
JS fortolker
Operativ System
Computer Hardware

# Overordnet arkitektur: ”klient side frameworks”

- ***Framework:*** 3 parts JS biblioteker  udvidelser, som gør visse avancerede ting nemmere og mere automatisk



Velkendte klient-side frameworks:

- React,
- Vue,
- Angular,
- QueryJS,
- Bootstrap,
- …

Opfordring: UNDLAD at bruge dem!

https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/

https://www.academind.com/learn/javascript/jquery-future-angular-react-vue/

# Overordnet arkitektur: ”server side frameworks”

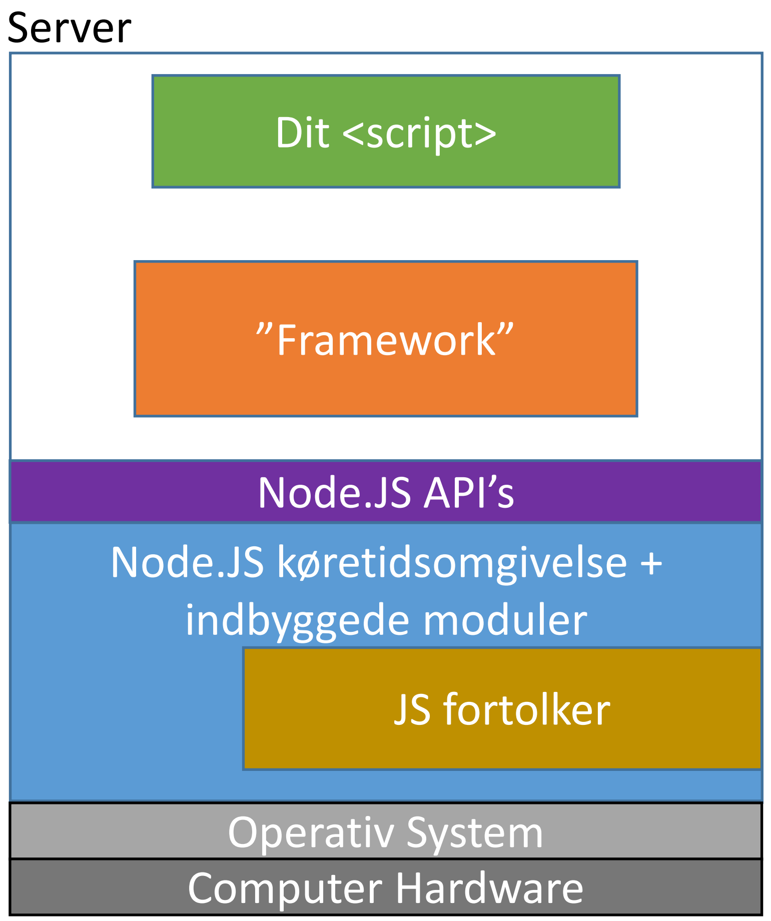
- ***Framework:*** 3 parts JS biblioteker  udvidelser, som gør visse avancerede ting nemmere og mere automatisk

Velkendte server-side frameworks:

- Express.js
- Meteor
- Loopback
- Backbone
- Koa
- …

Opfordring: UNDLAD at bruge dem!

https://scotch.io/bar-talk/10-node-frameworks-to-use-in-2019



## Express.js

- Express.js https://en.wikipedia.org/wiki/Express.js
  - A very popular web application framework built to create Node.js Web based applications.
- Core features of Express framework:
  - Allows to set up middleware to respond to HTTP Requests.
  - Defines a routing table which is used to perform different action based on HTTP method and URL. See https://expressjs.com/en/starter/basic-routing.html
  - Allows to dynamically render HTML Pages based on passing arguments to templates.

## Express.js Hello World

- Create a file named app.js and add the following codes:

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
       res.send('Hello World!');
});
app.listen(3000, function () {
       console.log('app.js listening to http://localhost:3000/');
});
```

- Run the app.js in the server:

```
node app.js
```

- Then, load http://localhost:3000/ in a browser to see the output.

## Basic Routing

- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:

  app.METHOD(PATH, HANDLER);
- Where:
  - app is an instance of express.
  - METHOD is an HTTP request method, in lower case.
  - PATH is a path on the server
  - HANDLER is the function executed when the route is matched.
- Example: GET method route:

```
app.get('/', function (req, res) {
    res.send('Get request to the homepage');
});
```

# Response Methods

- Quite similar to “normal” Javascript
- You call methods on the response object (res)

| Method | Description |
|---|---|
| res.download() | Prompt a file to be downloaded. |
| res.end() | End the response process. |
| res.json() | Send a JSON response. |
| res.jsonp() | Send a JSON response with JSONP support. |
| res.redirect() | Redirect a request. |
| res.render() | Render a review template. |
| res.send() | Send a response of various types. |
| res.sendFile() | Send a file as an octet stream. |
| res.sendStatus() | Set the response status code and send its string representation as the response body. |

## Designing and generating a REST API

- The service is provided at an URL
- The service allow access through a set of endpoints
- Each endpoint allows for REST verbs/operations (GET, POST, PUT, PATCH, DELETE)
- The input data can be JSON or XML
- The output is a HTTP status code, and sometimes data is JSON or XML

## OpenAPI

- OpenAPI Specification is an API description format for REST APIs, to describe:
  - Available endpoints and operations on each endpoint
  - Operation parameters Input and Output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
- An OpenAPI document itself is a JSON object, which may be represented and written either in JSON or YAML format

# OpenAPI 3.0

- We will use OpenAPI version 3 (the latest one)

  https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md

- The OpenAPI document is composed by a number of parts
- Most important parts (except for the "headers" `openapi`, `info` and `servers`):
  - Paths
  - Components

## Paths

- It is a series of `path` objects
- It is the “path” part of the resource URL, followed by the operations it accepts
- For each operation:
  - The specific of the input data
  - A list of HTTP status code, and optionally the specific of return data for each case

## Components

- It contains a set of reusable objects
  - `schemas`
  - `responses`
  - `parameters`
  - etc
- All objects defined within the `components` object will have no effect on the API unless they are explicitly referenced

# Example

```
• openapi: 3.0.3
• info:
•   title: ping test
•   version: '1.0‘
• servers:
•   - url: 'http://localhost:8000/‘
• paths:
•   /ping:
•     get:
•       operationId: pingGet
•       responses:
•         '201':
•           description: OK
```

## Try OpenAPI out

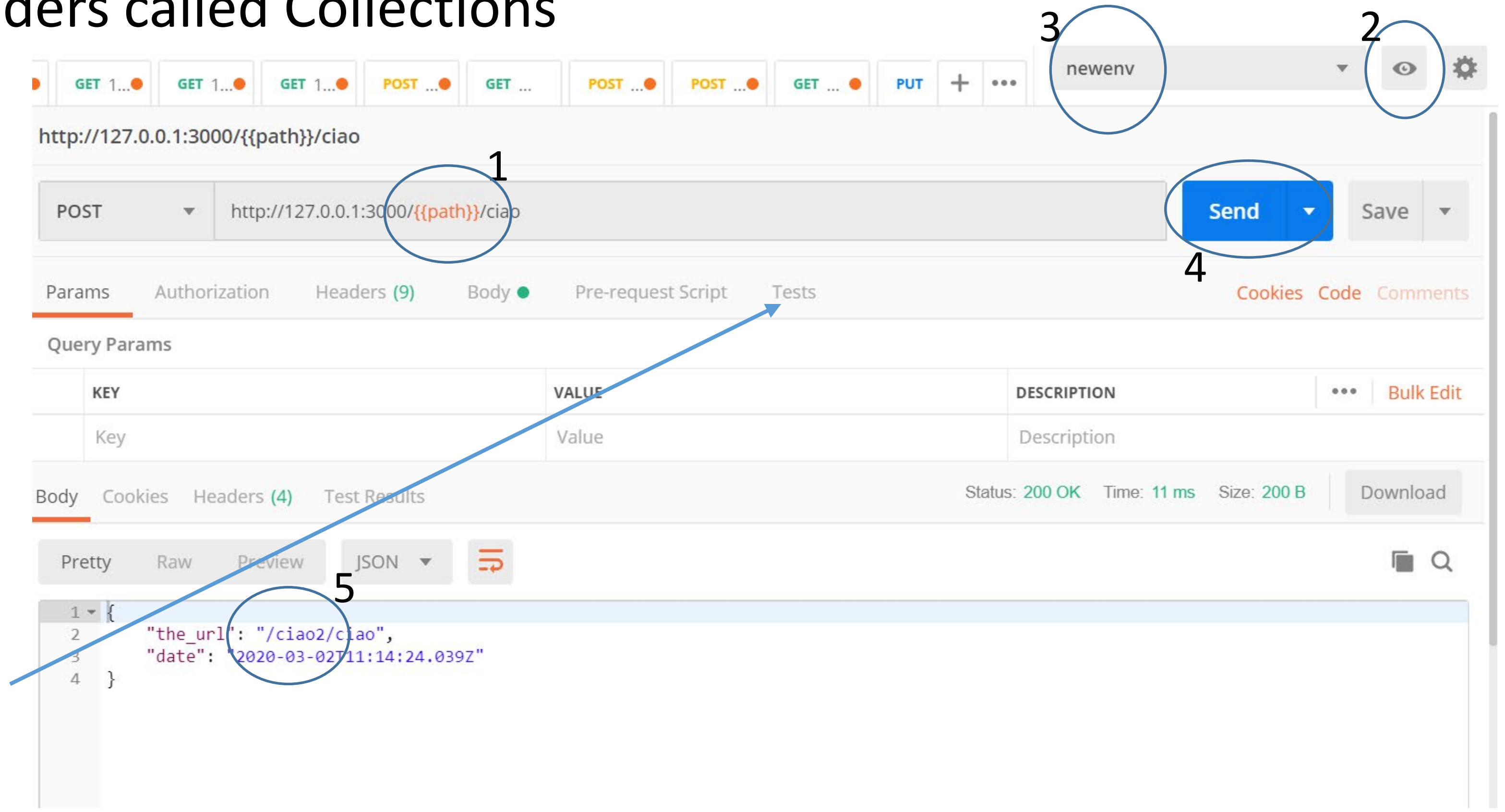
- Write an OpenAPI specification
- Use one of the generators to generate a server:
  - https://editor.swagger.io/
  - http://api.openapi-generator.tech/index.html
- The first one is easier to use, the second one must be installed but provides many more  languages
- Download the generated code for server and client
  - Or you can use Postman as client
- Implement the business logic
- Try it out: Compile, execute, etc

## Agenda

- SSE and Websockets
- Service Oriented Applications
  - basic notions (service producer, consumer, etc)
  - Stateless server, HATEOAS, idempotent operations
- Representational State Transfer (REST)
- Frameworks, and API Code generation
- Testing with Postman

# Postman

- Tool useful to test APIs
  - Set up GET/PUT/POST/DELETE/etc requests and see responses
  - Save requests to repeat later (History)
  - Organize requests into folders called Collections
- Parametrization of requests ➔ ➔ ➔ ➔ ➔
  1. Parametrize using {{..}}
  2. Set up an environment
  3. Apply it
  4. Execute the request
  5. See the result
- Possible to create “tests”
- Automate with CLI



http://127.0.0.1:3000/{{path}}/ciao

POST http://127.0.0.1:3000/{{path}}/ciao

Status: 200 OK Time: 11 ms Size: 200 B

```
{
    "the_url": "/ciao2/ciao",
    "date": "2020-03-02T11:14:24.039Z"
}
```

END