A HTML document forms a DOM tree, where everything, even attributes and text content are nodes.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <span id="f" class="a"></span>
    <span id="g">
      <div class="a">
        Text Content
      </div>
    </span>
  </body>
</html>
```
Web Applications Selecting Parts of a HTML document

- JavaScript offers methods to select certain parts of a DOM tree

```html
1  <span id="f" class="a"></span>
2  <span id="g">
3     <div class="a">
4         Text Content
5     </div>
6  </span>
```

```javascript
document.getElementById('f')
=> 1

document.getElementsByClassName('a')
=> 1, 3

document.getElementsByTagName('div')
=> 3

document.querySelector('.a')
=> 1

document.querySelectorAll('.a')
=> 1, 3

document.querySelectorAll('span .a')
=> 3
```
Web Applications  Changing a HTML document

- Each node offers methods for modifying it
  - appending children
    ```javascript
    const fNode = document.getElementById('f');
    fNode.appendChild(document.createTextNode('Content'));
    fNode.innerHTML = 'Content';
    ```
  - changing classes
    ```javascript
    fNode.classList.add('b');
    fNode.classList.remove('a');
    ```
  - changing style
    ```javascript
    fNode.style.border = '1px solid black';
    ```
Web Applications  Letting users interact with a HTML document

- Major part of every (user-facing) application is interaction
  - different kinds of events can be attached to a node in a DOM tree
  - type of events that can be attached depending on node type
  - click, mouseover, mousemove, blur, focus, ...

- `<element>.addEventListener(type, listener)`

  ```javascript
  const fNode = document.getElementById('f');
  fNode.addEventListener('click', e => console.log('Clicked on f.'));
  ```

- `<element>.removeEventListener(type, listener)`

  ```javascript
  const fNode = document.getElementById('f');
  const whenHovering = () => console.log('Hovering over f.');
  
  fNode.addEventListener('mouseover', whenHovering);
  fNode.removeEventListener('mouseover', whenHovering);
  ```
Web Applications Building a To-do List

- popular example for evaluating JavaScript frameworks: Building a To-do list application
  - adding to-dos
  - marking to-dos as completed
const addButton = document.getElementById('addButton');
const todoInput = document.getElementById('todoInput');
const errorMessage = document.getElementById('errorMessage');

addButton.addEventListener('click', () => {
    if (todoInput.value !== '') {
        // Logic for adding a new Todo
    } else {
        errorMessage.style.display = 'block';
    }
});

todoInput.addEventListener('focus', () => {
    errorMessage.style.display = 'none';
});
Web Applications  Building a To-do List

```javascript
addButton.addEventListener('click', () => {
    if (todoInput.value !== '') {
        todos.push({
            title: todoInput.value,
            completed: false
        });
        displayTodos(todos);
    } else {
        errorMessage.style.display = 'block';
    }
});
```
displayTodos = todos => {
    todoList.innerHTML = ''; 
    todos.forEach(todo => {
        const todoNode = document.createElement('div');
        const todoTitle = document.createElement('span');
        const completedCheckbox = document.createElement('input');

        todoTitle.appendChild(document.createTextNode(todo.title));

        completedCheckbox.setAttribute('type', 'checkbox');
        if(todo.completed) {
            completedCheckbox.setAttribute('checked', '');
        }

        todoNode.appendChild(todoTitle);
        todoNode.appendChild(completedCheckbox);
        todoList.appendChild(todoNode);
    });
}
displayTodos = (todos) => {
  ...
  completedCheckbox.setAttribute('type', 'checkbox');
  if(todo.completed) {
    completedCheckbox.setAttribute('checked', '');
  }
  completedCheckbox.addEventListener('change', () => {
    todo.completed = !todo.completed;
    displayTodos(todos);
  });
  todoNode.appendChild(todoTitle);
  todoNode.appendChild(completedCheckbox);
  ...
};
Web Applications  Interacting with the outside world

- Communication with a server is required
- JavaScript basically offers two ways to communicate with the outside world
  - REST
    - use a combination of HTTP verbs and resource-specific URLs to tell the server what you want to do
    - always initiated by the client, i.e., the server has no means of sending something to the client, exception being the response to a request
  - WebSockets
    - bidirectional communication between client and server
    - one’s own (top-level) protocol for communication has to be implemented
Web Applications REST

- GETting something from a location

```javascript
const xhr = new XMLHttpRequest();
xhr.open('GET', <location>);
xhr.addEventListener('load', () => console.log(xhr.response));
xhr.send();
```

- POSTing something against a location (normally associated with creating something)

```javascript
const xhr = new XMLHttpRequest();
xhr.open('POST', <location>);
xhr.addEventListener('load', () => console.log(xhr.status));
xhr.send({title: 'Save on server', completed: false});
```
jQuery is a library that offers easier ways to do many of the things we did today

$ as general entry point into library

```javascript
$('#f').
  append('Hallo')

$('.a')
  .css({ color: 'green' });

$('#f')
  .on('click', () => console.log('Clicked on #f'));

$('#f')
  .off('click');
```

- other functionality: animations ($ .animate), REST ($ .ajax), ...
displayTodos = todos => {
    $('#todos').empty();
    todos.forEach(todo => {
        const todoTitle = $('span>${todo.title}</span>');
        const todoCheckbox = $('input type="checkbox" />'
            .prop('checked', todo.completed)
            .on('change', () => {
                todo.completed = !todo.completed;
                displayTodos(todos);
            });
    });
    $('#todos').append(
        $('<div />'
            .append(todoTitle, todoCheckbox)
        );
    });
}
$.ajax({
    url: url,
    method: 'POST' | 'GET' | 'PUT' | ...,
    data: data,
    success: (data, status, jqXHR) => {
        // Called if request was executed successfully
    },
    error: (jqXHR, textStatus, errorThrown) => {
        // Called if something went wrong while executing the request
    },
    dataType: 'xml' | 'json' | 'text' | ...
});
Web Applications JavaScript on the server-side

- "a JavaScript runtime built on Chrome’s V8 JavaScript engine"
  - basically, a program that executes JavaScript that is not a browser
  - includes modules that allow doing things that cannot be done using JavaScript in a browser
    - building servers
    - communication with nearly any protocol
    - accessing the file system
    - executing commands

- npm is a package manager for JavaScript and Node.js
  - is not a part of Node.js, but nearly always included in a Node.js installation
  - can be used to install packages
  - manages dependencies between packages
Web Applications  Getting started with Node.js

- install Node.js
- create a new directory, navigate into the directory
- execute `npm run init` and follow the instructions
  - this creates a `package.json` in the current directory, where the project’s dependencies are saved
- using `npm install --save <packagename>` installs a package locally in the current directory
  - parameter `--save` automatically adds an entry as dependency to `package.json`
  - visit `npmjs.com` for packages
  - can be used for frontend packages as well (see next time)

- for now: we need the package express for building a REST API
  - `npm install --save express`
  - `npm install --save body-parser`
Web Applications Libraries & package.json

- package.json keeps track of a project’s dependencies (and other metadata)
  - semantic versioning is used (recommended) for package versions
    - increasing first number: breaking changes
    - increasing second number: new functionality, but no breaking changes
    - increasing third number: only bug fixes
  - for each dependency, a rule is defined to what extent this package can be updated (without breaking the application)
    - ^major.minor.fix means that all increases in minor and fix are allowed
    - ~major.minor.fix means that only increases in fix are allowed
    - major.minor.fix means that exactly the given version has to be installed

```json
{
  ...
  "dependencies": {
    "body-parser": "^1.18.2",
    "express": "^4.16.1"
  }
}
```
**Web Applications**

- **package.json**
  - can be used to update dependencies
    - `npm outdated` - checks for which packages updates are available
    - `npm update` - updates packages which are allowed to be updated
  - to install the whole project anew or on another machine
    - `npm install`
Web Applications

- Node.js framework for creating REST APIs

```javascript
const express = require('express');
const app = express();

const todos = [
  { id: 1, title: 'Show this in the browser', completed: false },
  { id: 2, title: 'Save something on the server', completed: false }
];

app.get('/todos', function (req, res) {
  res.send(todos);
});

app.listen(3000, () => console.log('Listening on port 3000.'));
```
Web Applications  Middleware

- `app.use(...)` is used to attach middlewares to all routes
  - A middleware is a function `(req, res, next) => { ... next()}`
  - An incoming request is passed through all middlewares before it arrives at the actual route handler
  - `body-parser.json()`, a middleware that takes the body of the request and parses the body into an object

```javascript
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

let currentId = 2;
const nextId = () => ++currentId
...`
Path parameters

- Dynamic Information can be passed directly in the URL
  - in most cases the identifier of the resource to modify or retrieve

```javascript
app.put('/todos/:todoId', (req, res) => {
  const todoId = req.params.todoId;
  const body = req.body;

  const todo = todos.filter(todo => todo.id === todoId)[0];

  if (todo) {
    todo.completed = body.completed;
  }

  res.sendStatus(200)
});
```
Web Applications Payload

- POST/PUT can contain payload (i.e., the thing that is passed xhr.send(…))
  - specification does not prohibit payload for GET/DELETE, but most libraries don’t support payload for those HTTP methods

```javascript
app.put('/:todoId', (req, res) => {
  const todoId = req.params.todoId;
  const body = req.body;

  const todo = todos.filter(todo => todo.id === todoId)[0];

  if (todo) {
    todo.completed = body.completed;
  }

  res.sendStatus(200);
});
```
Web Applications  Checking for null and undefined

- **undefined**
  - not initialized, e.g., variables without an assignment, “out of bounds” array indices

- **null**
  - invalid reference, explicitly set by the programmer

```javascript
app.put('/todos/:todoId', (req, res) => {
  const todoId = req.params.todoId;
  const body = req.body;

  const todo = todos.filter(todo => todo.id === todoId)[0];

  if (todo) {
    typeof todo !== "undefined" && todo !== null
    todo.completed = body.completed;
  }

  res.sendStatus(200)
});
```
app.post('/todos', (req, res) => {
    const todo = req.body;

    if(todo && todo.title) {
        const newTodo = {
            id: nextId().toString(),
            title: todo.title,
            completed: false
        };
        todos.push(newTodo);
        res.send({serverId: newTodo.id});
    }
    res.sendStatus(400);
});
When loading the application, all to-dos are fetched from the server

```javascript
let todos = [];
$.ajax({
    url: 'http://localhost:3000/todos',
    method: 'GET',
    dataType: 'json',
    success: function(data) {
        todos = data;
        displayTodos(todos);
    }
});
```
Web Applications Creating new To-Dos

- Optimistic UI is used for creating (and updating) to-dos
  - changes are immediately reflected on the client, with server confirmation pending

```javascript
if (todoInput.value !== '') {
  <update Client>
  <update Server>
  <reconcile Client and Server>
} else {
  ...
```

- changes are immediately reflected on the client
- afterwards the server is told about the change
- after server response, client and server state are reconciled
Optimistic UI is used for creating (and updating) to-dos
  - changes are immediately reflected on the client, with server confirmation pending

```javascript
if (todoInput.value !== '') {
    const newTodo = {
        id: nextId().toString(),
        title: todoInput.value,
        completed: false
    };
    todos.push(newTodo);
    displayTodos(todos);
    <talk to Server, update according to response>
} else {
    ...
```
### Optimistic UI: Updating the Server and Reconciliation

```javascript
if (todoInput.value !== '') {
  <update Client>
  $.ajax({
    url: 'http://localhost:3000/todos',
    method: 'POST',
    contentType: 'application/json',
    data: JSON.stringify(newTodo),
    success: data => {
      newTodo.id = data.serverId;
    }
  });
}
else {
  ...
}
```

- Request to server is only dispatched after to-do is already displayed on the client.
- Server responds with the id on the server, client updates the object.
Web Applications Updating To-Dos

- Optimistic UI as well for updating completed of a to-do
  - no reconciliation, fails silently if something goes wrong on the server

```javascript
completedCheckbox.addEventListener('change', () => {
  todo.completed = !todo.completed;
displayTodos(todos);

$.ajax(
  url: `http://localhost:3000/todos/${todo.id}`,
  method: 'PUT',
  contentType: 'application/json',
  data: JSON.stringify({ completed: todo.completed })
});
```

update the client
update the server
Optimistic UI Caveats

- very incomplete implementation of optimistic UI
  - error handling completely omitted
  - How to handle consecutive user actions before a server identifier is known?