Software Engineering

SE for Web Applications II: Frontend

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 29.11.2023

@andivogelsang

vogelsang@cs.uni-koeln.de

Learning Goals for Today

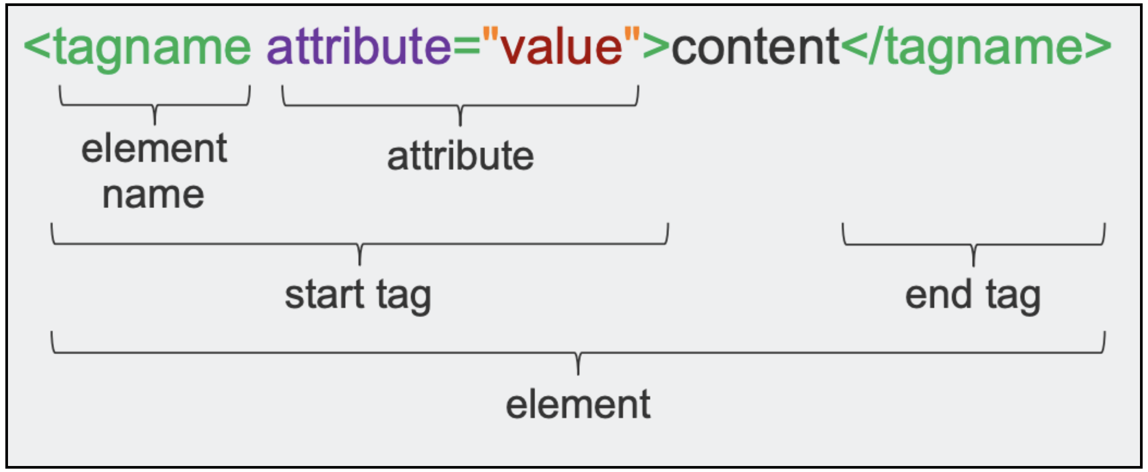
- Know how web content can be represented in a browser (via HTML, JavaScript, and DOM)
- Know how to manipulate content on web pages
- Know how to make asynchronous calls (to a backend)
- Understand how web frontend frameworks work
- Understand the Model-View-ViewModel pattern
- Understand the interplay between data and design/output and how binding can enable reactive frontends

HTML

HTML

Hyper Text Markup Language (HTML)

- Standardized by the W3C
- Describes structure and content of a document
- Human and non-human users
  - Browser parses the content and presents it to the end user
  - Crawler indexes the parsed content (machine-readability)



<tagname attribute="value">content</tagname>

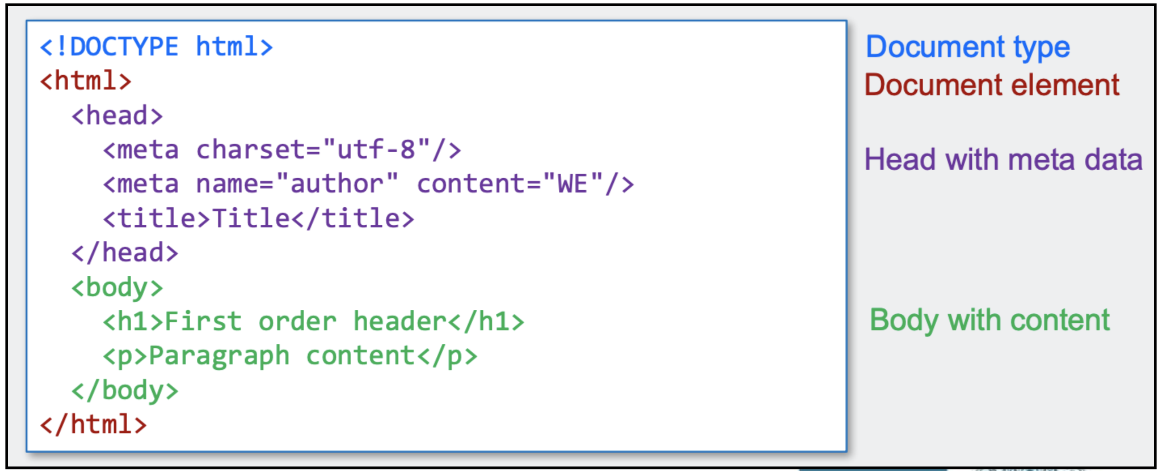
element name

attribute

start tag

end tag

element



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <meta name="author" content="WE"/>
    <title>Title</title>
  </head>
  <body>
    <h1>First order header</h1>
    <p>Paragraph content</p>
  </body>
</html>
```

Document type

Document element

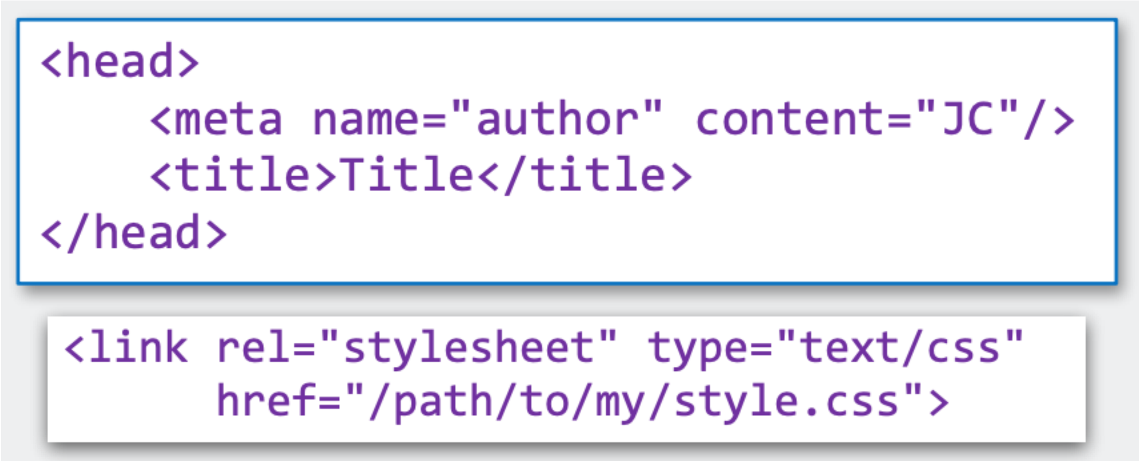
Head with meta data

Body with content

HTML Structure

Head with meta data

- Title
- Data from meta element
  - Author, Keywords, Date, …
- Linking to other resources
  - CSS, JavaScript, …



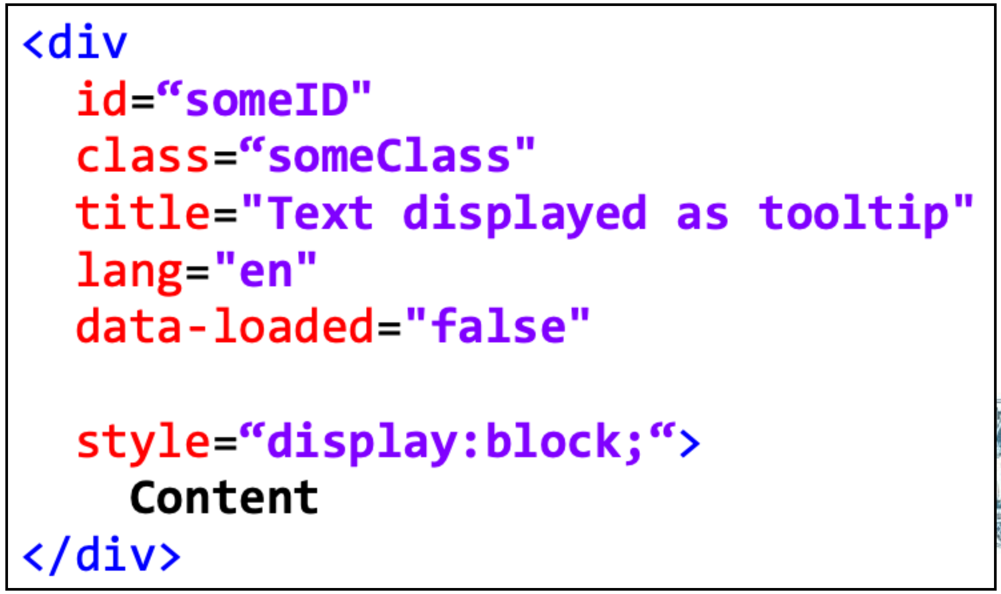
```
<head>
    <meta name="author" content="JC"/>
    <title>Title</title>
</head>
```

```
<link rel="stylesheet" type="text/css"
      href="/path/to/my/style.css">
```

Body containing content

Global attributes (excerpt)

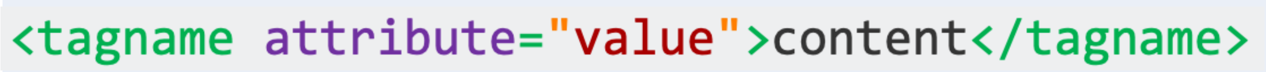
- **id**: Unique identifier
- **class**: Assigned class for CSS
- **title**: Description of an element
- **style**: Element-specific layout information
- **data-***: Invisible attached data (Custom data accessible through JavaScript)



```
<div
  id=“someID"
  class=“someClass"
  title="Text displayed as tooltip"
  lang="en"
  data-loaded="false"

  style=“display:block;“>
    Content
</div>
```

HTML Structure – Element Semantics

Syntax



```
<tagname attribute="value">content</tagname>
```

Semantics

Not given by standard visual representation!

- <h1> is a first order header != the thickest printed text
- <b> prints text bold != <em> emphasizes the text
- <table> represents tabular data != layout mechanism

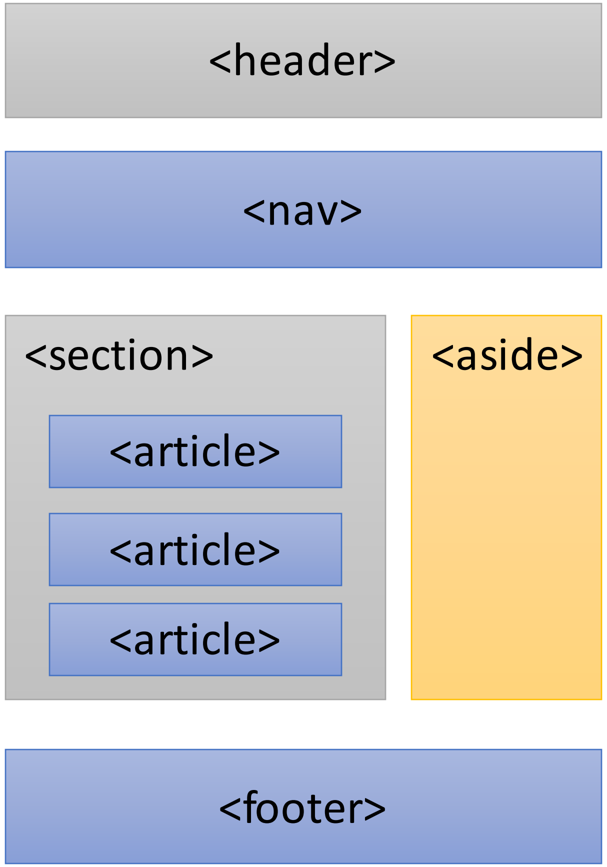
Why use syntactically and semantically correct elements?

- Browser compatibility, accessibility
- Easier processing for tools, e.g., transformations, indexing for search engines
- More efficient browsing (no interpretation of wrong HTML necessary)
- Shift towards better use of semantics enables
  - Ability for better interpretation for accessibility
  - Easier code understanding and maintainability

HTML Structure – Content Structure

Content Structure

- **<header>** defines header of document or section
- **<nav>** defines navigation region of page or section
- **<main>** main content of the page
- **<section>** thematic grouping of content
- **<h1-h6>** Heading from most to least important. Reflects structural depth, e.g. in sections. Exactly one <h1> per page
- **<article>** specifies complete, self-contained content
- **<aside>** defines content aside from main content
- **<footer>** defines footer of document or section



<header>

<nav>

<section>

<article>

<article>

<article>

<aside>

<footer>

Many of these elements can be nested and it's not always straightforward which element should be used!

HTML Elements

Generic elements

- **<div>** Generic block element
- **<span>** Generic inline element

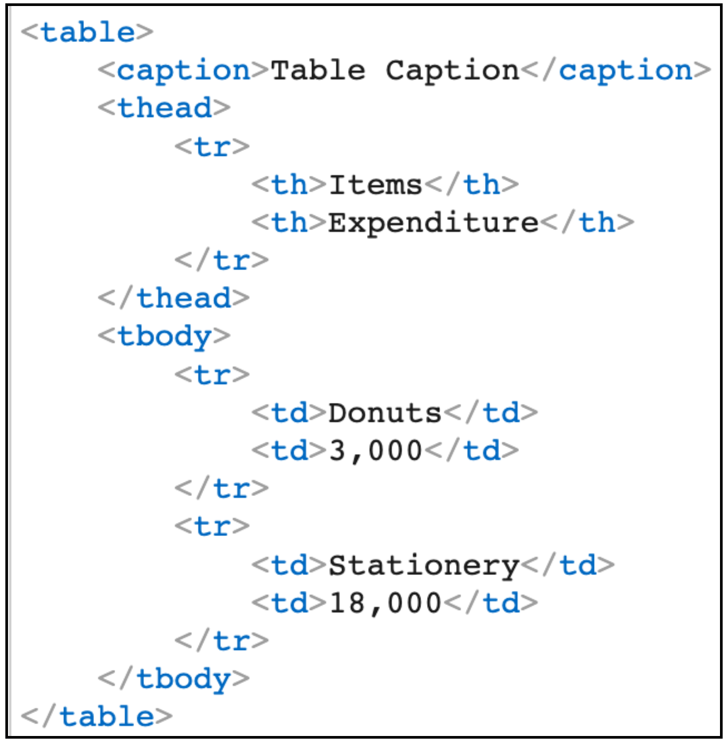
Use these when no other element with more appropriate semantics is left

Grouping elements

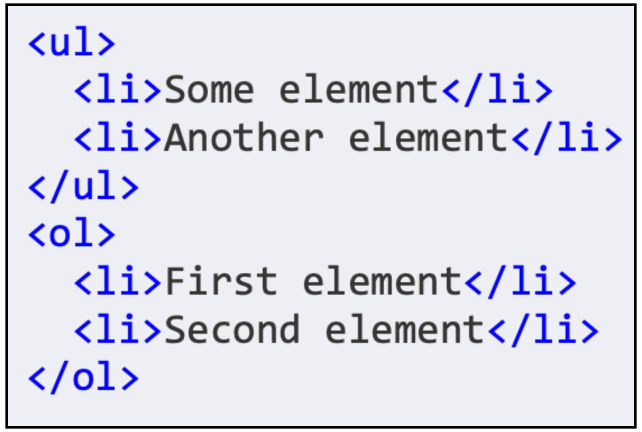
- **<p>** paragraphs
- **<ul>** unordered list
- **<ol>** ordered list
- **<table>** tabular data

Links and anchors

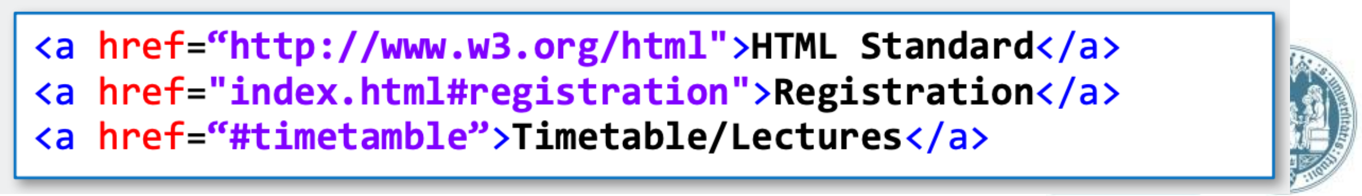
- **<a>** Link to another page or location



```
<table>
    <caption>Table Caption</caption>
    <thead>
        <tr>
            <th>Items</th>
            <th>Expenditure</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Donuts</td>
            <td>3,000</td>
        </tr>
        <tr>
            <td>Stationery</td>
            <td>18,000</td>
        </tr>
    </tbody>
</table>
```



```
<ul>
  <li>Some element</li>
  <li>Another element</li>
</ul>
<ol>
  <li>First element</li>
  <li>Second element</li>
</ol>
```

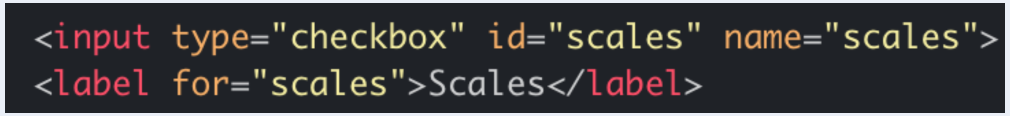


```
<a href="http://www.w3.org/html">HTML Standard</a>
<a href="index.html#registration">Registration</a>
<a href="#timetamble">Timetable/Lectures</a>
```

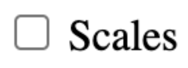
HTML Basic Forms

Input

- Checkboxes

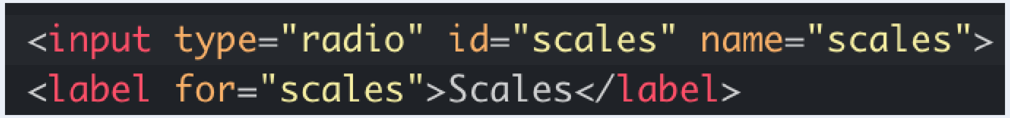


```
<input type="checkbox" id="scales" name="scales">
<label for="scales">Scales</label>
```

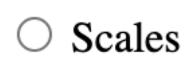


☐ Scales

- Radio Buttons

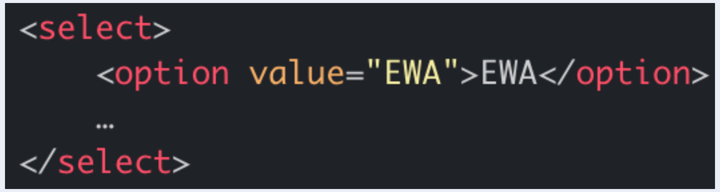


```
<input type="radio" id="scales" name="scales">
<label for="scales">Scales</label>
```

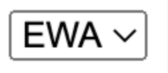


○ Scales

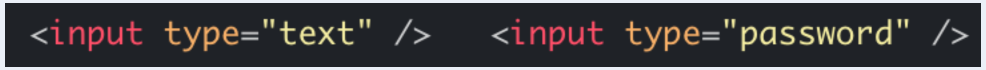
- Menus



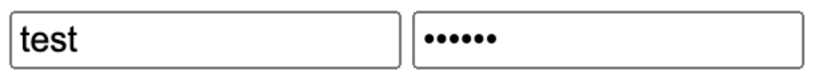
```
<select>
    <option value="EWA">EWA</option>
    …
</select>
```



- Text fields



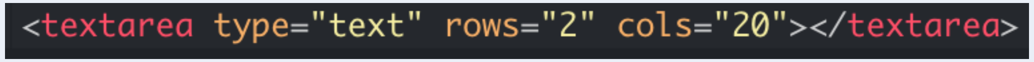
```
<input type="text" />    <input type="password" />
```



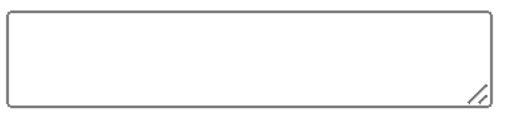
test

••••••

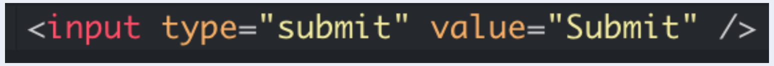
- Text area



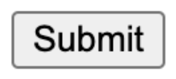
```
<textarea type="text" rows="2" cols="20"></textarea>
```



- Buttons

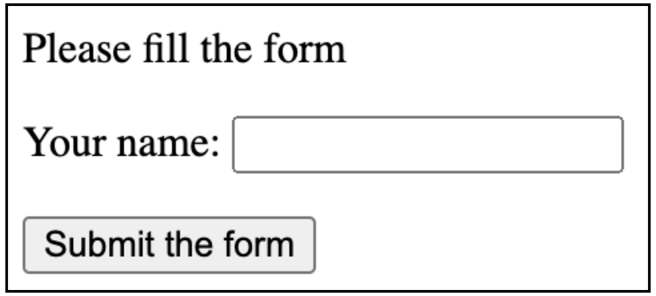


```
<input type="submit" value="Submit" />
```



What happens when I send a form?

```
<html>
  <head>
    <title>Simple Form</title>
  </head>
  <body>
    <p>Please fill the form</p>
    <form action=“/processForm” method=“post”>
      <p>
        <label for=“username”>Your name:</label>
        <input type=“text” id=“username” name=“username” />
      </p>
      <p><input type=“submit” value=“Submit the form” name=“action” />
      </p>
    </form>
  </body>
</html>
```

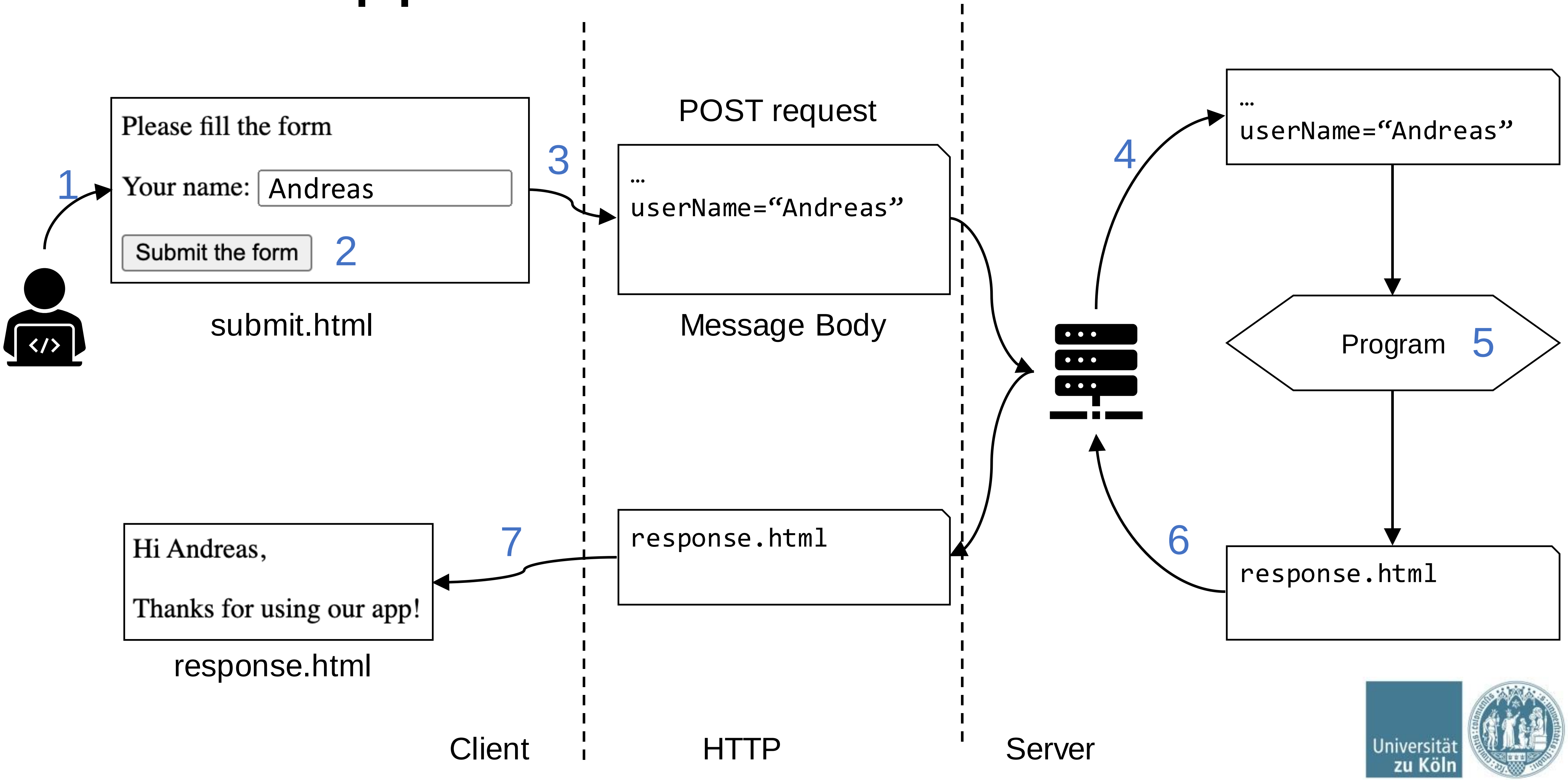


Please fill the form

Your name:

HTML forms only allow POST and GET requests

What happens when I send a form?



1

Please fill the form

Your name: Andreas

Submit the form

2

submit.html

3

POST request

```
…
userName="Andreas"
```

Message Body

4

```
…
userName="Andreas"
```

Program 5

6

```
response.html
```

```
response.html
```

7

Hi Andreas,

Thanks for using our app!

response.html

Client

HTTP

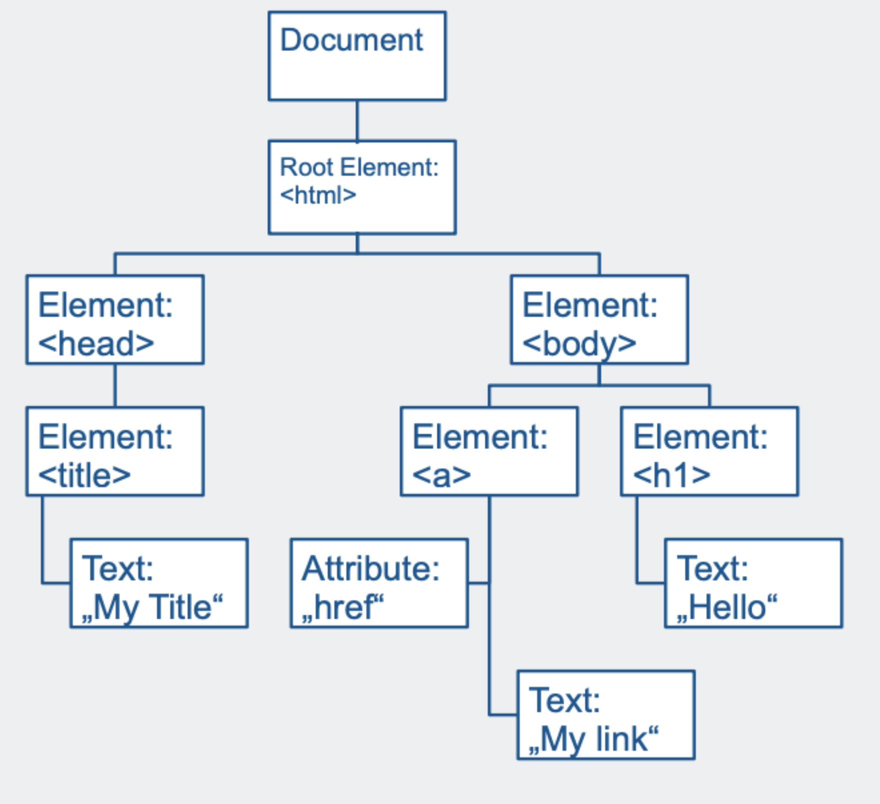
Server

DOM and Asynchronous Requests

Document Object Model

Document Object Model (DOM)

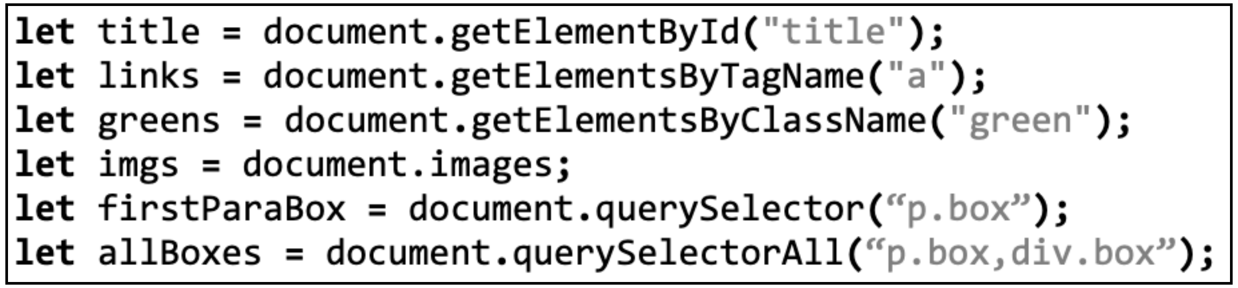
- Tree structure for interacting with (X)HTML and XML documents
  - HTML elements as objects with properties, methods and events
- Standardized by the W3C
  - Platform- and language-independent



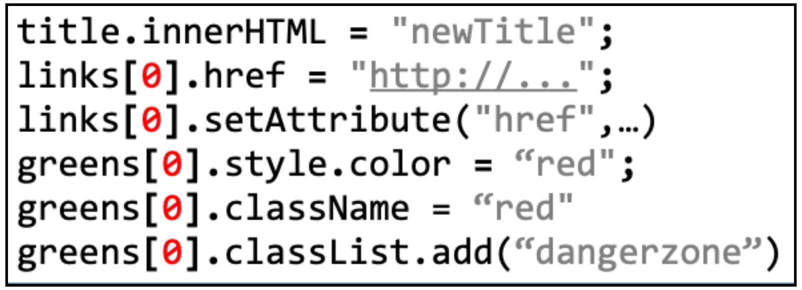
Document Object Model

DOM Operators

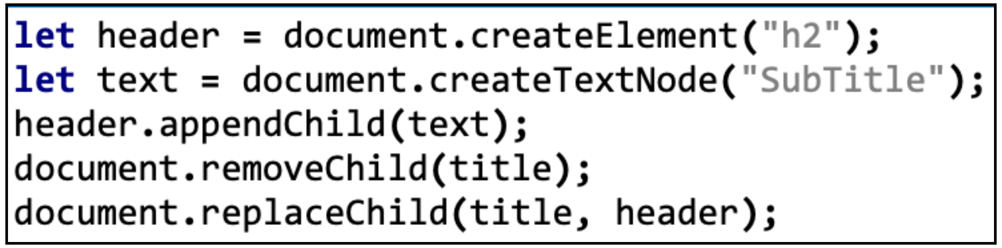
- Retrieve Elements
- Change Elements
  - Content, attributes, style, class
- Manipulating DOM nodes
  - Create, append, remove
- DOM traversal on elements
  - `parentElement, nextElementSibling, previousElementSibling, childNodes`



```
let title = document.getElementById("title");
let links = document.getElementsByTagName("a");
let greens = document.getElementsByClassName("green");
let imgs = document.images;
let firstParaBox = document.querySelector(“p.box”);
let allBoxes = document.querySelectorAll(“p.box,div.box”);
```



```
title.innerHTML = "newTitle";
links[0].href = "http://...";
links[0].setAttribute("href",…)
greens[0].style.color = “red";
greens[0].className = “red"
greens[0].classList.add(“dangerzone”)
```

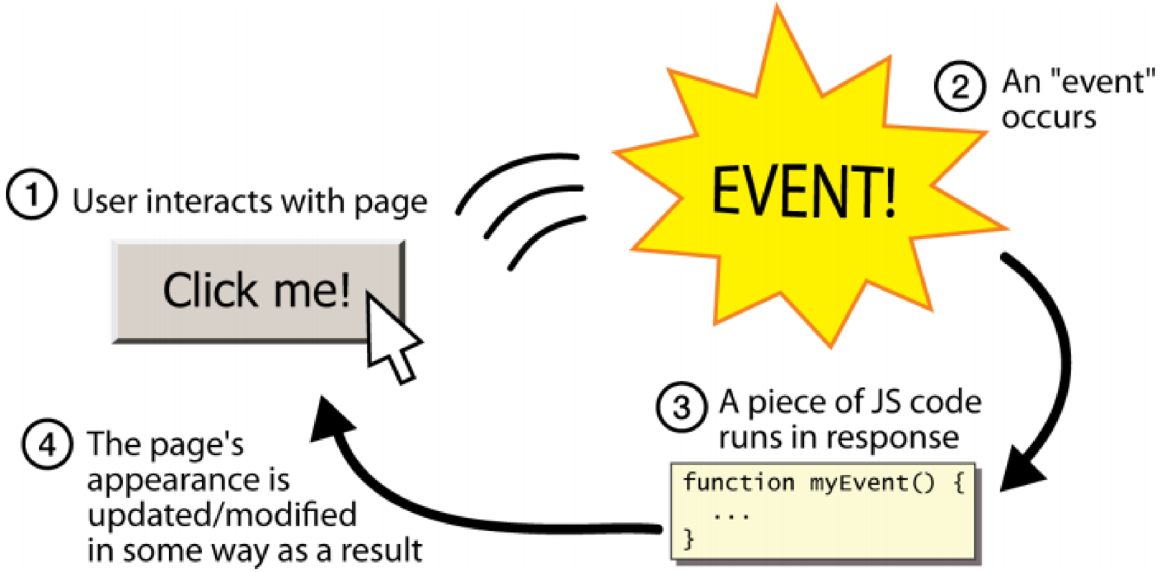


```
let header = document.createElement("h2");
let text = document.createTextNode("SubTitle");
header.appendChild(text);
document.removeChild(title);
document.replaceChild(title, header);
```

Event-driven and Asynchronous Programming

Event-driven Programming

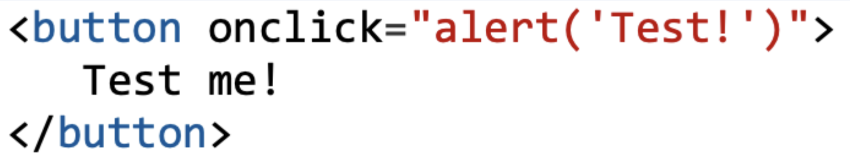
- Flow of the program is determined by responding to user actions called events
- Writing programs driven by user events



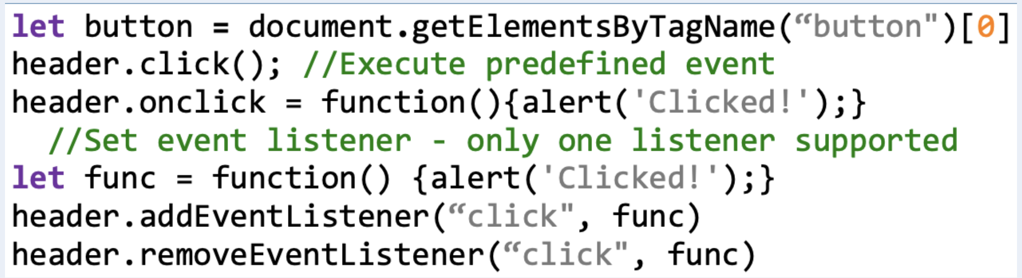
1 User interacts with page
Click me!
EVENT!
2 An "event" occurs
3 A piece of JS code runs in response
function myEvent() {
...
}
4 The page's appearance is updated/modified in some way as a result

DOM Events

Event callback attached to HTML elements



```
<button onclick="alert('Test!')">
   Test me!
</button>
```



```
let button = document.getElementsByTagName(“button")[0]
header.click(); //Execute predefined event
header.onclick = function(){alert('Clicked!');}
  //Set event listener - only one listener supported
let func = function() {alert('Clicked!');}
header.addEventListener(“click", func)
header.removeEventListener(“click", func)
```

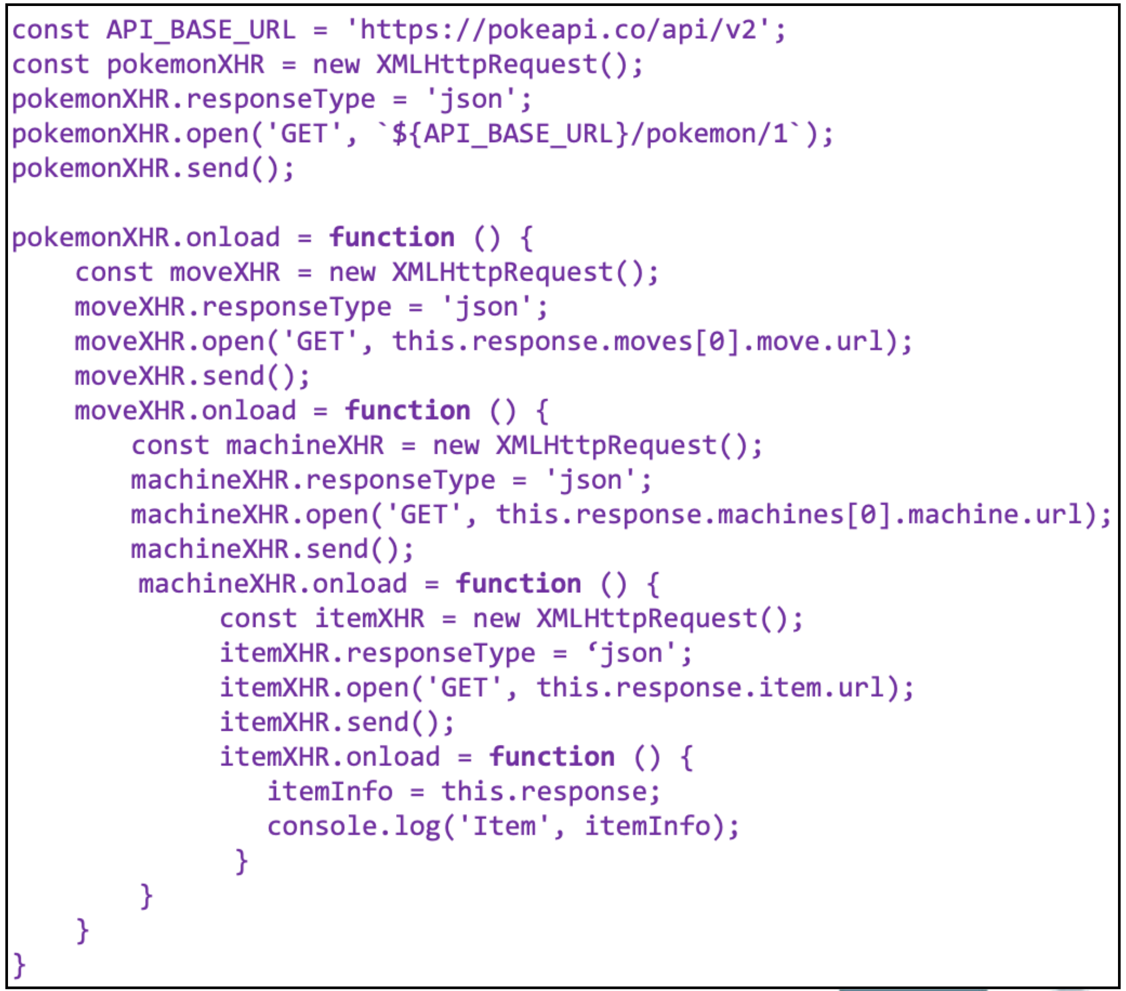
Event types (selection)

- `load/unload:` User enters/leaves a page
- `change:` Form input field changes
- `Focus/blur:` User focuses/unfocuses an input field
- `submit:` Form is submitted
- `mouseover/mouseout:` Mouse enters/leaves region
- `mousedown/mouseup/click:` Mouse click events
- `Keydown/keyup/keypress:` Keyboard events
- `drag:` User drags an elements

Sending Asynchronous Requests (Callbacks)

Classic network request API (XMLHttpRequest)

- Used callbacks - a mechanism to provide a function that gets called once you receive a response from HTTP
- Callbacks resulted in increasingly nested callback chains dubbed “callback hell”: http://callbackhell.com/



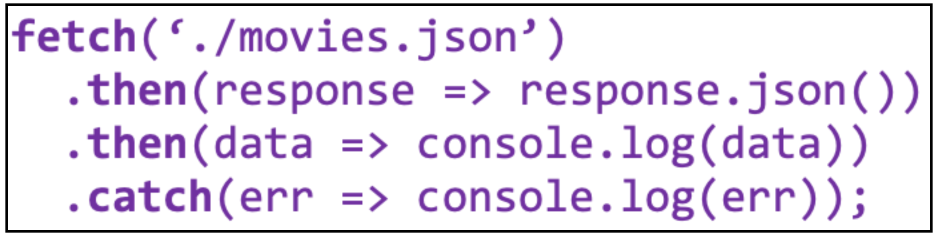
```
const API_BASE_URL = 'https://pokeapi.co/api/v2';
const pokemonXHR = new XMLHttpRequest();
pokemonXHR.responseType = 'json';
pokemonXHR.open('GET', `${API_BASE_URL}/pokemon/1`);
pokemonXHR.send();

pokemonXHR.onload = function () {
    const moveXHR = new XMLHttpRequest();
    moveXHR.responseType = 'json';
    moveXHR.open('GET', this.response.moves[0].move.url);
    moveXHR.send();
    moveXHR.onload = function () {
       const machineXHR = new XMLHttpRequest();
       machineXHR.responseType = 'json';
       machineXHR.open('GET', this.response.machines[0].machine.url);
       machineXHR.send();
        machineXHR.onload = function () {
             const itemXHR = new XMLHttpRequest();
             itemXHR.responseType = ‘json';
             itemXHR.open('GET', this.response.item.url);
             itemXHR.send();
             itemXHR.onload = function () {
                itemInfo = this.response;
                console.log('Item', itemInfo);
              }
        }
    }
}
```

Sending Asynchronous Requests (Promises)

fetch API

`fetch` API allows processing HTTP requests/ responses using **promises**:

- Promises are a general wrapper around asynchronous computations and callbacks
- They represent how to get a value - you tell it what to do as soon as it receives the value
- A promise is a proxy object for a value that is not yet known. It is modeled with the following states
  - Pending (initial state)
  - Fulfilled (execution successful)
  - Rejected (operation failed)



```
fetch(‘./movies.json’)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

Sending Asynchronous Requests (async/wait)

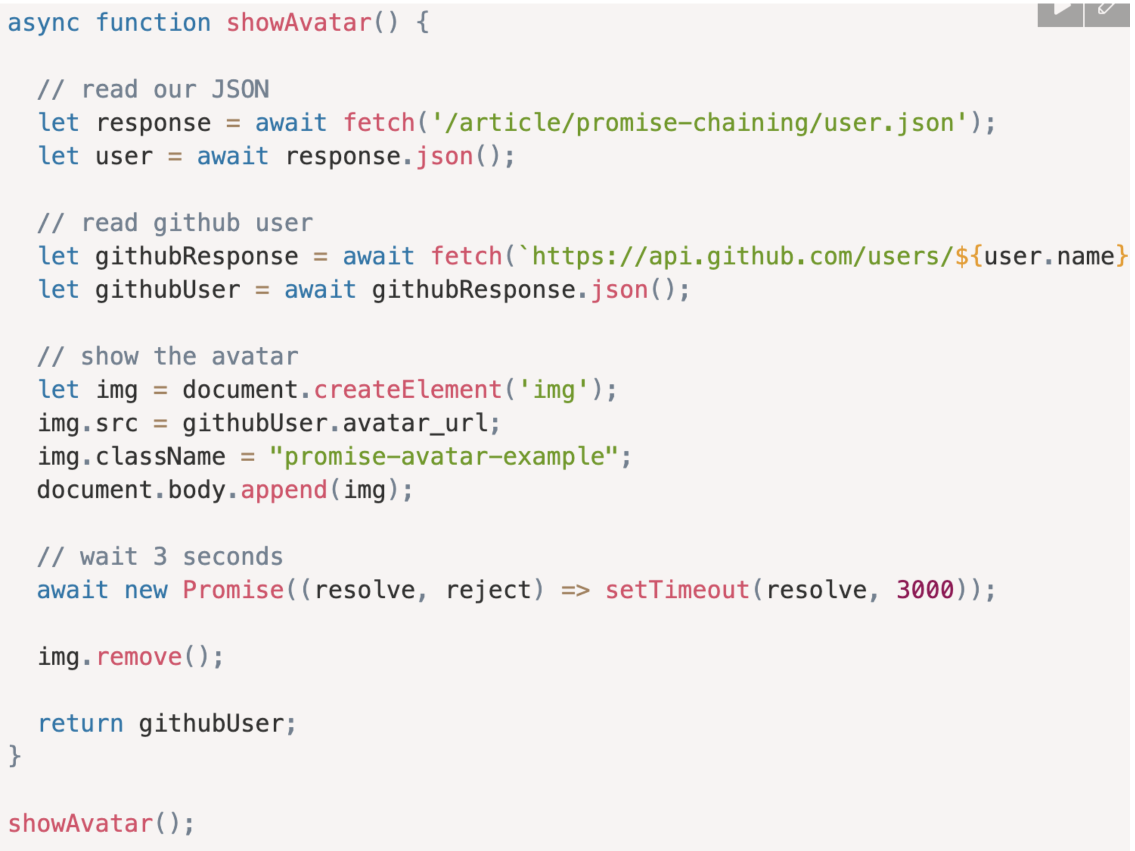
`async/await` is a special syntax to work with promises

- `async` is a keyword around a function that wraps a promise around its return value.

```
async function f() { return 1; }

f().then(alert); //requires then to resolve result
```

- `await` is a keyword that makes JavaScript wait until the promise is resolved and can then return the value (only works within async functions!)

```
let response = await fetch(“./movies.json”)
```



```
async function showAvatar() {

  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

https://javascript.info/async-await

Sending Asynchronous Requests (Observables)

`Observables` are an extension to promises

- Offered by the RxJS library; heavily used e.g., in Angular
- Promises deal with one asynchronous event at a time, while observables handle a sequence of asynchronous events over a period of time

Promises	Observables
Emit a single value at a time.	Emit multiple values over a period of time.
Are not lazy: execute immediately after creation.	Are lazy: they're not executed until we subscribe to them using the `subscribe()` method.
Are not cancellable.	Have subscriptions that are cancellable using the `unsubscribe()` method, which stops the listener from receiving further values.
Don't provide any operations.	Provide the map for `forEach`, `filter`, `reduce`, `retry`, and `retryWhen` operators.
Push errors to the child promises.	Deliver errors to the subscribers.

Sending Asynchronous Requests (Observables)

Operations	Promises	Observables

Creation	`const promise = new Promise(() => {`
	`  resolve(10);`
	`});`
		`const obs = new Observable((observer) => {`
		`  observer.next(10);`
		`}) ;`

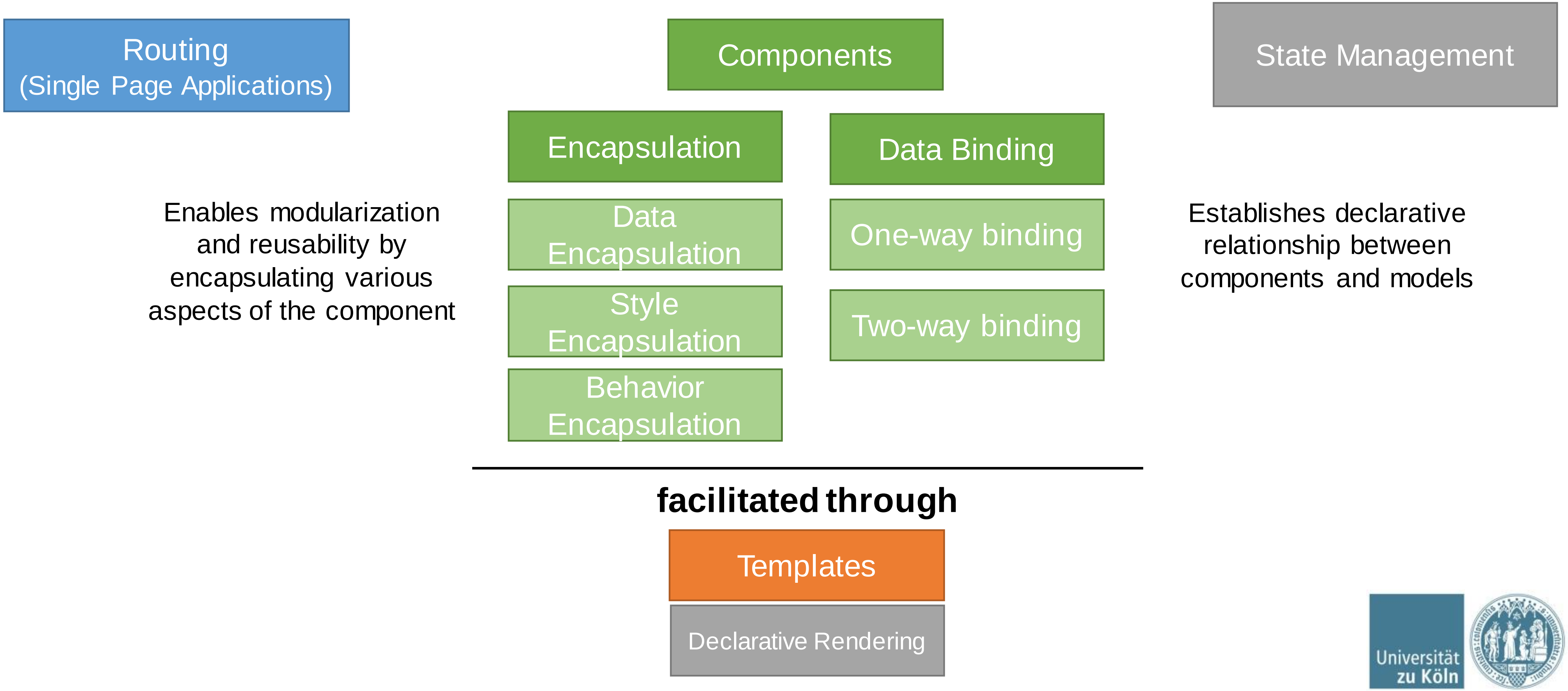
Transform	`promise.then((value) => value * 2);`	`Obs.pipe(map(value) => value * 2);`

Subscribe	`promise.then((value) => {`
	`  console.log(value)`
	`});`
		`const sub = obs.subscribe((value) => {`
		`  console.log(value)`
		`});`

Unsubscribe	N/A	`sub.unsubscribe();`

Abstractions in Web Frontends

Frontend Abstractions



Routing
(Single Page Applications)

Components

State Management

Encapsulation

Data Binding

Enables modularization and reusability by encapsulating various aspects of the component

Data Encapsulation

Style Encapsulation

Behavior Encapsulation

One-way binding

Two-way binding

Establishes declarative relationship between components and models

facilitated through

Templates

Declarative Rendering

Frontend Architecture

Model-View-ViewModel (MVVM)

A design pattern often used in frontends

Model: Data access layer for data that is shown to the user and can be manipulated

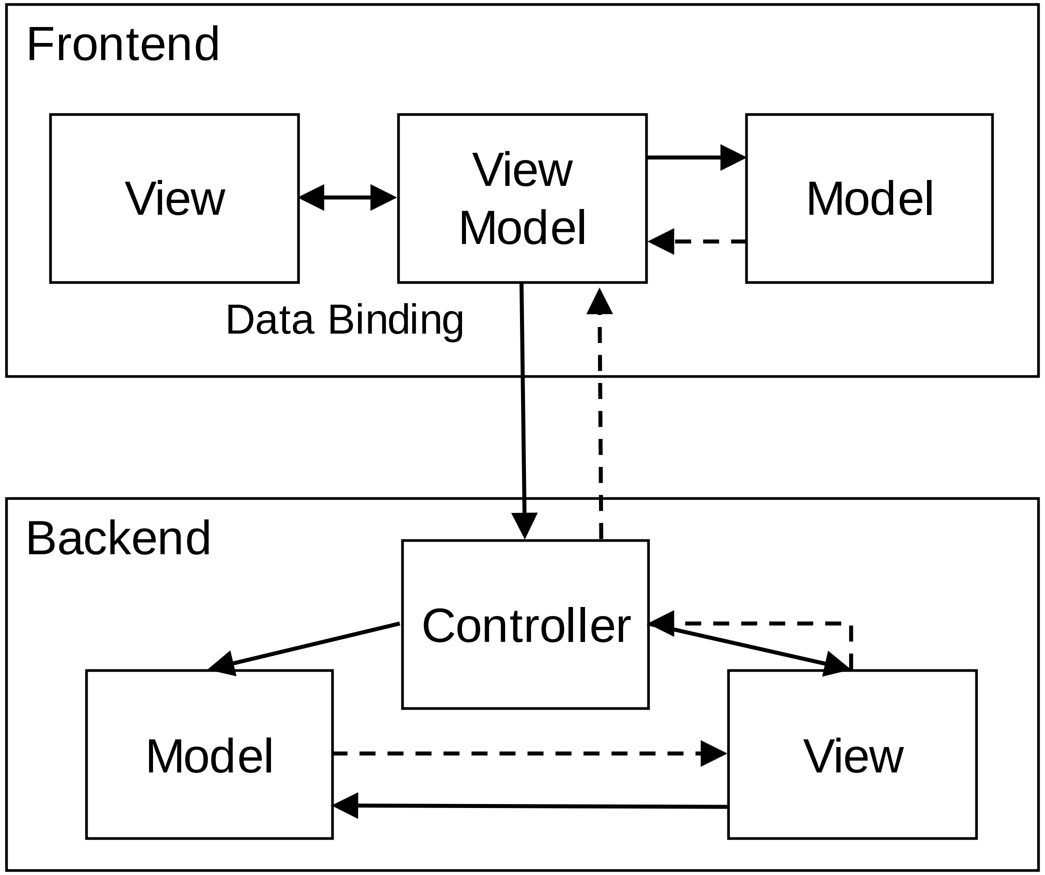
View: Structure, layout, and appearance of what a user sees on the screen

ViewModel: Contains the UI logic and connects the view with the model.

Data Binding: Declarative binding between view and view model.

Naming gets confusing

- Backend model vs. frontend model
- Backend controller vs. frontend controller
- Backend service vs. frontend service



MVVM in Angular

Files formats

Model: A class defined in TypeScript (usually only attributes and no methods)

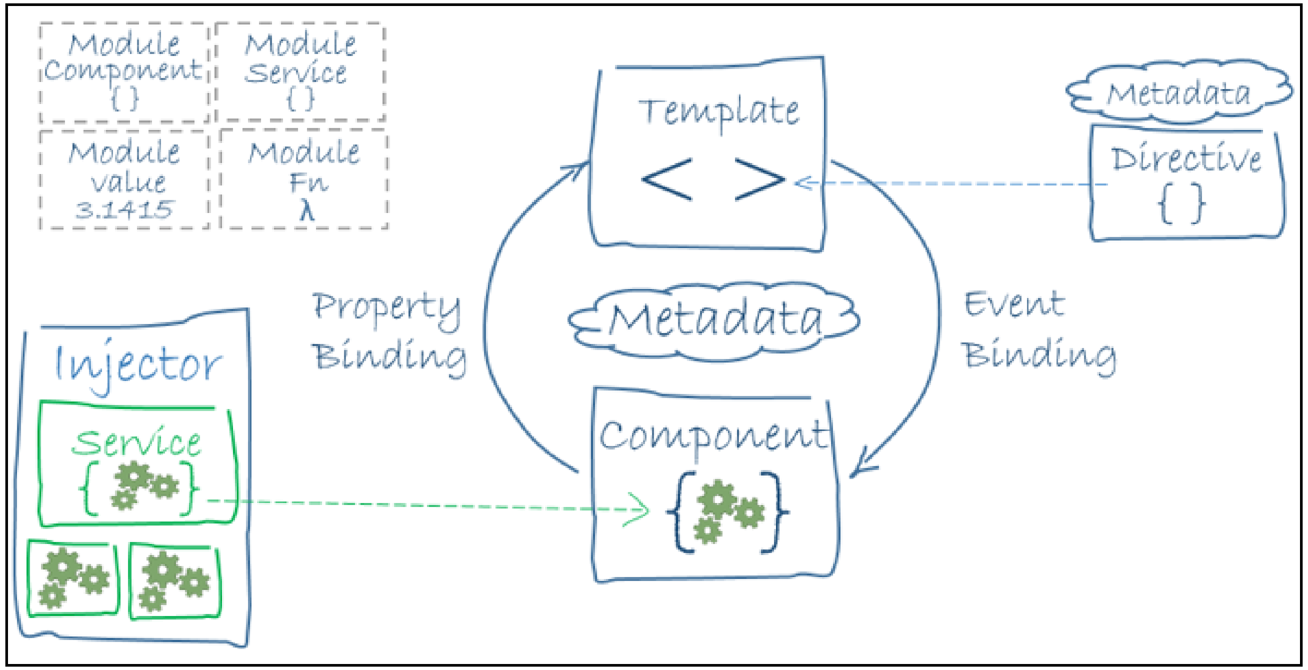
View: An HTML template

ViewModel: A TypeScript class containing event-handling behavior

Data Binding:

- Property Binding: Binding (changing) data to UI elements)
- Event Binding: Binding events in the view to actions in the ViewModel

Services: A TypeScript class containing supporting and reusable (across ViewModels) functionality

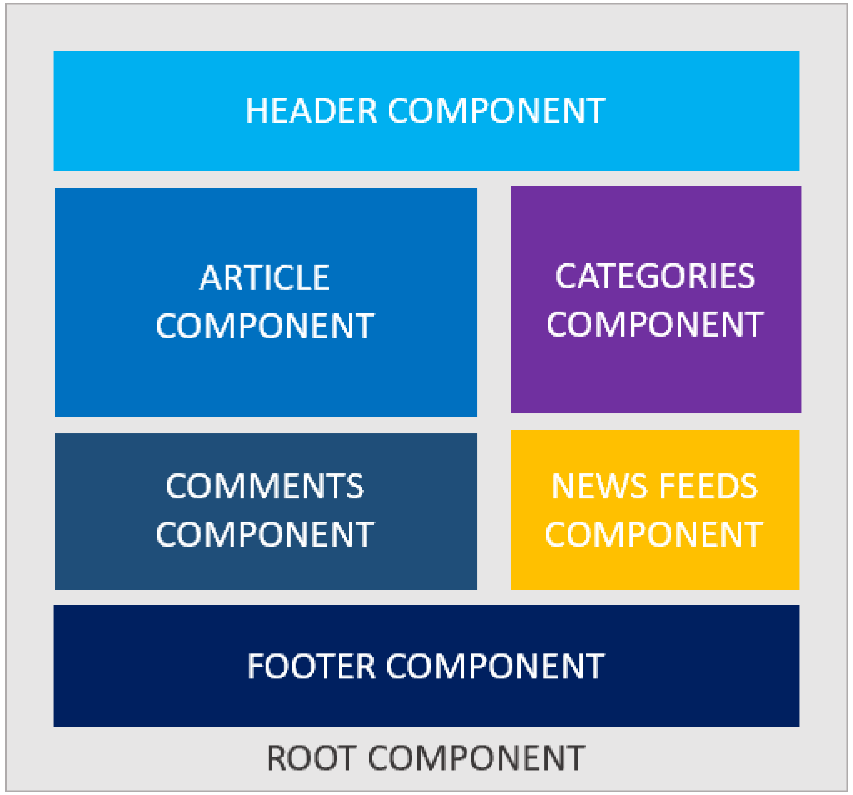


Frontend Components

Frontend Components

Components are reusable building blocks

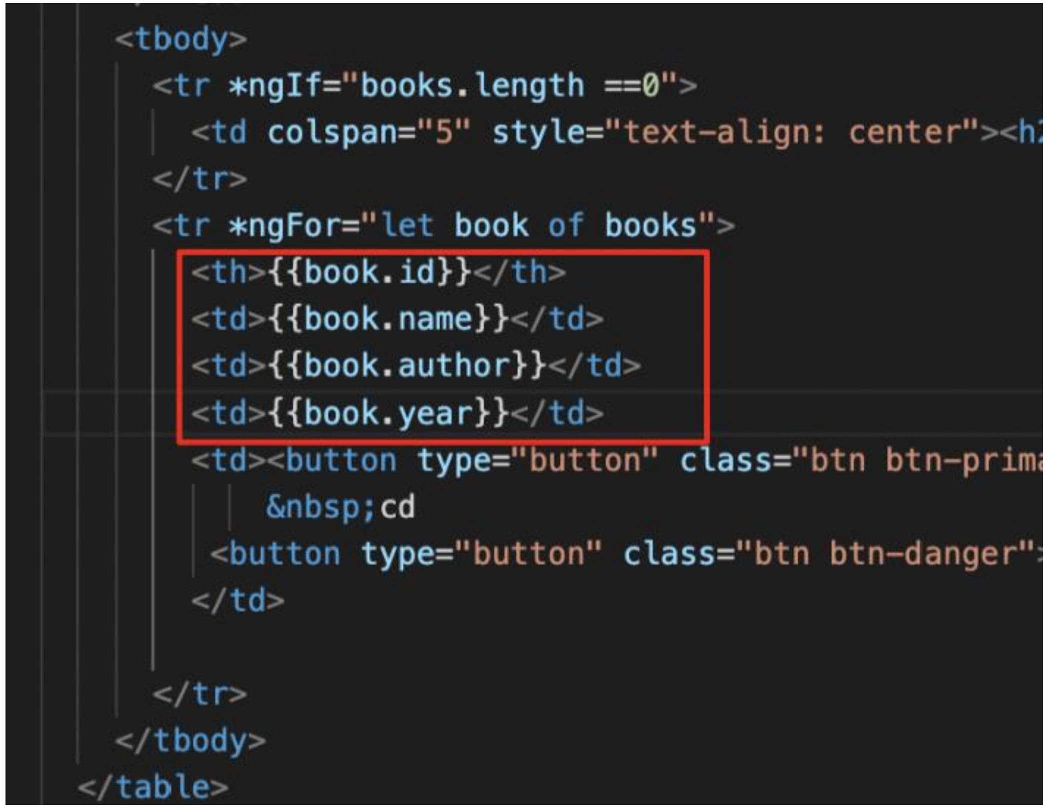
- Template HTML code in *View*
  - Declares binding to internal model and properties through *interpolations*; Syntax: `{{data}}`
  - Supports bounded loops and conditional rendering
- Behavior in ViewModel
  - Input parameters that become part of the internal model (properties)
  - Functions to deal with event handling (methods)
- Encapsulate (scoped) styles that are bound to the component



Frontend Templates (View)

Templating

Template engines replace variables in static template files and control structures (conditionals and loops) with values passed from the program.



```
    <tbody>
      <tr *ngIf="books.length ==0">
        <td colspan="5" style="text-align: center"><h
      </tr>
      <tr *ngFor="let book of books">
        <th>{{book.id}}</th>
        <td>{{book.name}}</td>
        <td>{{book.author}}</td>
        <td>{{book.year}}</td>
        <td><button type="button" class="btn btn-prima
            &nbsp;cd
         <button type="button" class="btn btn-danger"
        </td>

      </tr>
    </tbody>
  </table>
```

Backend vs. frontend templating

Backend Templates

- The backend receives a request, retrieves/computes data, and generates HTML files
- Templates are static markup files that are expanded based on data/values
  - Template variables are replaced with values
  - Loops: Iterate over lists of values and generate HTML for each instance
  - Conditionals: Generate different HTML depending on values

Frontend Templates

- Conceptually very similar to backend templates (template variables, loops, conditionals)
- Reactive: Values might change based on model changes
  - Model changes can be triggered by user input
  - Model (changes) can be retrieved from backend
- DOM is updated

Data Binding

One-Way Binding

Declares binding to internal model and properties

- Bindings as part of DOM content nodes are declared through interpolation syntax:
  `{{ data }}`
  - (Interpolations are inline expressions, i.e., can be any JavaScript code)
- Bindings as part of attributes are defined using directives (property binding)
  `<img [src]=“standardImage”>`
- **One-way refers to the direction of data-flow**
  Values from the model and properties are bound to the template variables to create the output when expanded

```
<div>
  // interpolation
  <a href="{{ link }}">{{ pizza.name }}</a>
  <p>Ingredients: {{ pizza.ingredients.join(', ') }}</p>
</div>

<div>
  // property binding
  <img [src]="link">
  <p>Pepperoni Pizzeria!</p>
</div>
```

```
@Component({
  selector: 'app-pizza',
  templateUrl: './pizza.component.html',
  styleUrls: ['./pizza.component.css']
})
export class PizzaComponent {
  link = 'https://pepperoni-pizzeria.com/awesome-pizzas'
  pizza =  {
    name: 'Pepperoni Pizza',
    ingredients: ['anchovies', 'tomatoes']
  }
}
```

Data Binding

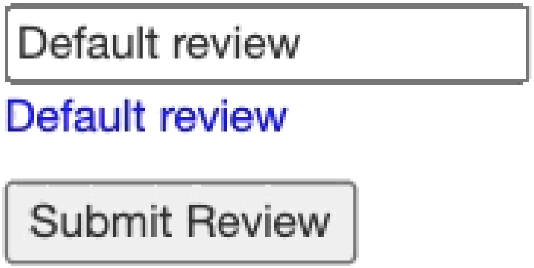
Two-Way Binding

Declares binding to and from the internal model (form inputs)

- Model changes are reflected in the view (as in one-way binding)
- Changes in the view are reflected in the model (and consequently in all bindings that have been established on the model)
- Binding through ngModel directive
  `<input [(ngModel)]="name">`

```
<div class="container">
    <input type="text" [(ngModel)]="review">
    <p>{{ review }}</p>
    <button>Submit Review</button>
</div>
```

```
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  review="Default review";
}
```



Default review

Default review

Data Binding

Event Binding

Declares reactions to events

- Model changes are reflected in the view (as in one-way binding)
- Binding through directive `<button (click)="onSubmit()" type="submit">`

```
@Component({
  selector: 'app-root',
  template: '
    <div>
      <app-pizzeria
       (reviewSubmitted)="onReviewSubmitted($event)">
      </app-pizzeria>
    </div>
  ',
  styleUrls: ['./pizzeria.component.css']
})
export class PizzeriaComponent {
  reviews = [];

  onReviewSubmitted(review: string) {
    this.reviews.push(review);
  }
}
```

```
<div>
  <textarea rows="4" columns="50" [(ngModel)]="review">
    Enter review here...
  </textarea>
  // event binding!
  <button (click)="onSubmit()" type="submit">
    Submit Review</button>
</div>
```

```
@Component({
  selector: 'app-pizzeria',
  templateUrl: './pizzeria.component.html',
  styleUrls: ['./pizzeria.component.css']
})
export class PizzeriaComponent {
  @Output() reviewSubmitted = new EventEmitter<string>();
  review = '';

  // this method will execute on click
  onSubmit() {
    this.reviewSubmitted.emit(this.review);
  }
}
```

Conditional Rendering

Conditional Rendering

Render elements only if expression evaluates to true

- Controlled by directives `ng-if` and `ng-template`

```
<div *ngIf="!isLoggedIn">
  Please login, friend.
</div>
```

```
export class AppComponent {
  isLoggedIn = true;
}
```

```
<ng-container
  *ngIf="isLoggedIn; then loggedIn; else loggedOut">
</ng-container>

<ng-template #loggedIn>
  <div>
    Welcome back, friend.
  </div>
</ng-template>
<ng-template #loggedOut>
  <div>
    Please friend, login.
  </div>
</ng-template>
```

Bounded Loops (List Rendering)

List Rendering

Map elements in an array to HTML elements

- Controlled by directive `ng-for`

```
<table>
  <thead>
    <th>Name</th>
</thead>
  <tbody>
    <tr *ngFor="let hero of heroes">
      <td>{{hero.name}}</td>
    </tr>
  </tbody>
</table>
```

Routing

Routing

Browser-like navigation for Single-Page Applications

- Simulate standard navigation by manipulating the browser history
- URL fragments allow linking to different logical “pages” while staying on the same browser page `https://www.example.com/#/config/437568`
- Router library
  - Same concept as server-side routing
  - Can pass URL parts as props to components

```
import { Routes } from '@angular/router';

import { HomeComponent} from './home.component'
import { ProductComponent} from './product.component'
import { ErrorComponent} from './error.component'

export const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'product', component: ProductComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full` },
  { path: '**', component: ErrorComponent }
];
```

Map actions to routes

```
<li>
  <a [routerLink]="['product']">Product</a>
</li>

<router-outlet></router-outlet>
```

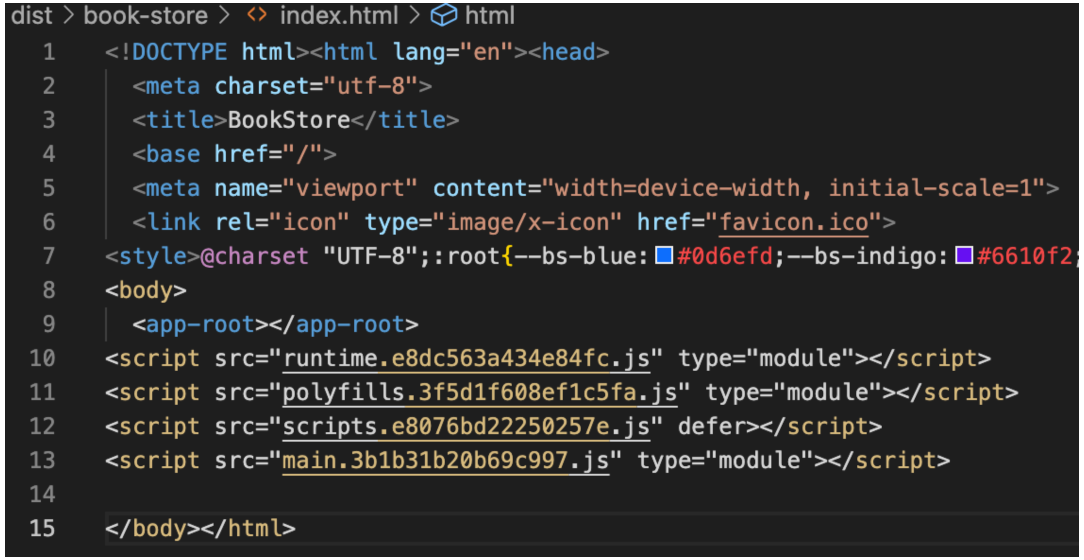
Deployment

Deploying Frontend Applications

- Frontent applications themselves need to be *served* by a web server
- Often, these are (virtual) single-page applications
    - There is only a single html file on the server (*index.html*)
    - Every “page-like” navigation is dynamically handled by the application on client-side

Deployment in Angular

- **`ng serve`** runs a local web server that provides the application
- **`ng build`** creates a set of static files that can hosted on most web servers
- **`ng deploy`** directly deploys the app to a hosting service (e.g., Amazon Cloud S3, Firebase, GitHub Pages)



```html
<!DOCTYPE html><html lang="en"><head>
  <meta charset="utf-8">
  <title>BookStore</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
<style>@charset "UTF-8";:root{--bs-blue:#0d6efd;--bs-indigo:#6610f2
<body>
  <app-root></app-root>
<script src="runtime.e8dc563a434e84fc.js" type="module"></script>
<script src="polyfills.3f5d1f608ef1c5fa.js" type="module"></script>
<script src="scripts.e8076bd22250257e.js" defer></script>
<script src="main.3b1b31b20b69c997.js" type="module"></script>

</body></html>
```

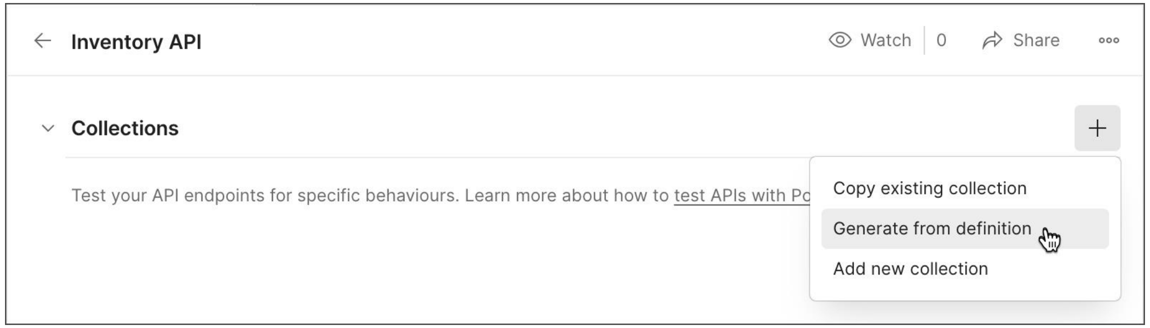
Testing Web Applications

Backend Testing

- Unit and integration tests: Standard testing frameworks for the language of the backend
  - JUnit tests for Java/Spring backends
- System Tests: External tools for testing RESTfulAPIs (e.g., via Postman)

Frontend Testing

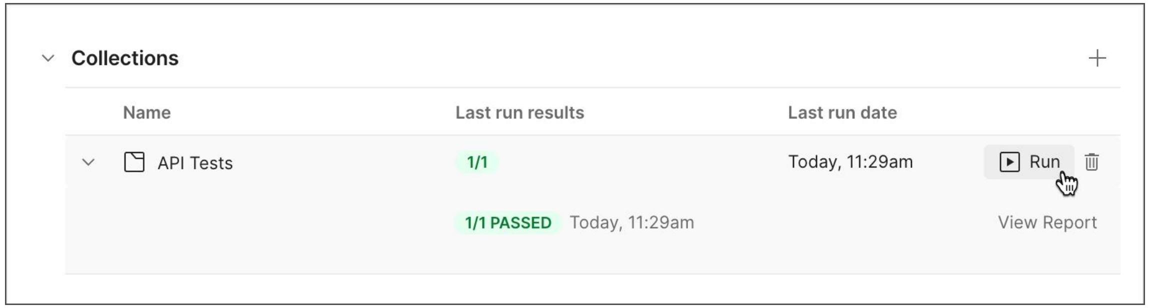
- Unit and integration tests: Testing frameworks for frontend frameworks
  - Jasmine for Angular
- System Tests: External tools for testing web GUIs
  - Work on the HTML elements
  - Often record-and-playback



Inventory API

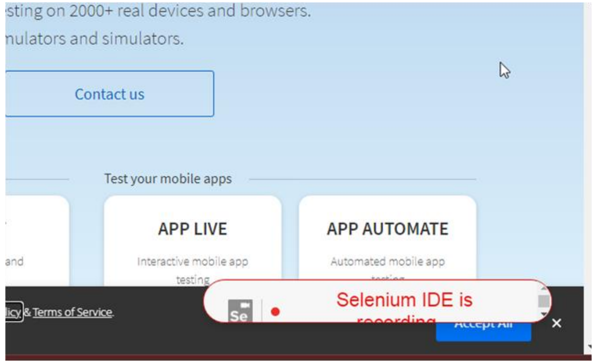
Collections

Test your API endpoints for specific behaviours. Learn more about how to test APIs with Po



Collections

Name	Last run results	Last run date
API Tests	1/1	Today, 11:29am
	1/1 PASSED Today, 11:29am	



sting on 2000+ real devices and browsers.
nulators and simulators.

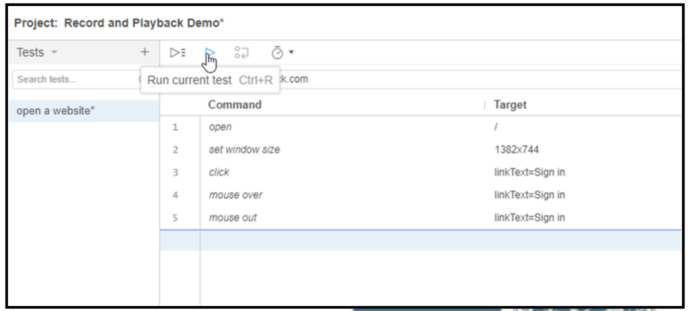
Test your mobile apps

APP LIVE
Interactive mobile app testing

APP AUTOMATE
Automated mobile app

licy & Terms of Service.

Selenium IDE is recording



Project: Record and Playback Demo*

	Command	Target
1	open	/
2	set window size	1382x744
3	click	linkText=Sign in
4	mouse over	linkText=Sign in
5	mouse out	linkText=Sign in

Frontend Frameworks

Endless Variety of Frontend Frameworks

- Different philosophies
- Different corporate backing
- Same concepts and abstractions

Summary

SE for Web Applications II: Frontend

- Websites are written in HMTL
- With JavaScript, you can manipulate websites to react to events (user input)
- JavaScript is used to define dynamic behavior on websites (incl. calls to a backend)
- Web frontends consider the client side of web applications
- Modern frontend follow a Model-View-ViewModel architecture (or similar)
- Frontend components help creating modular applications