Web Programming
Lecture Notes

EDAF90, Lp3, spring 2020
Computer Science, LTH
Lund University

build date: January 19, 2020
http://cs.lth.se/EDAF90

Editor: Per Andersson

Contributors in alphabetical order:
Björn Regnell

Home: https://cs.lth.se/edaf90

Repo: https://github.com/lunduniversity/webprog

This compendium is on-going work.
Contributions are welcome!
Contact: per.andersson@cs.lth.se

Contents

7 Network **1**

7.1 URI and URL . . . 1

7.1.1 URL encoding . . . 3

7.2 HTTP . . . 3

7.3 JSON . . . 5

7.4 REST . . . 6

7.4.1 Client–server architecture . . . 6

7.4.2 Statelessness . . . 6

7.4.3 Cacheability . . . 7

7.4.4 Layered system . . . 7

7.4.5 Uniform interface . . . 7

7.4.6 REST in practice, CRUD . . . 7

9 Angular **9**

9.1 TypeScript . . . 9

9.2 Template Language . . . 10

9.3 Data binding . . . 12

9.4 @Input and @Output . . . 12

9.5 Service . . . 13

9.6 App.module.ts . . . 14

10 Security **16**

10.1 Same Origin Policy . . . 16

10.2 Cross Site Scripting . . . 16

10.3 Code Injection . . . 16

11 Authentication and JWT **18**

12 Server Send Data **20**

12.1 polling . . . 20

12.2 long polling . . . 20

12.3 Server-Sent Events . . . 20

12.4 WebSockets . . . 21

13 Service Workers **22**

14 Server Side Rendering **23**

Lecture 7

Network

This lecture covers topics related to communication between web applications and servers on the internet. The internet are commonly described with a four layered architecture of protocols, where each layer adds more functionality. The lowest layer, the *link layer*, deals with communication on a local network and includes details on how the digital bits are converted to and from the analog signal sent over the transmission channel. Example of protocols at this layer are ethernet and the WiFi Standards 802.11a/b/g/n. The layers above add more functionality. The next layer is the *Internet layer*. This layer makes it possible for packages to find it way around the globe. This is the responsibility of the IP-protocol. It also defines the IP-address and describes how the addresses are assigned to nodes on the internet. Each node on the internet have it own unique IP-address. Above this we have the transport layer and the TCP and UDP protocols. TCP is short for Transmission Control Protocol and provides flow-control, connection establishment, and reliable transmission of data. This means that TCP ensures a reliable stream of data. When data are lost, they will be retransmitted, so when using TCP, you do not need to bother about dividing your data into packages, you just write to a stream. TCP makes sure the receiver eventually get the same stream of data. For some applications, there is no value of data that arrives late. Then UDP is an alternative. UDP delivers packages, but some may be lost on the way and the packages may arrive in a different order than they were sent. On the top of the internet protocol stack you find the *application layer*. This layer is the focus of this course. This is the home of protocols describing how applications communicate.

7.1 URI and URL

A Uniform Resource Identifier (URI) identifies a resource, for example the ISBN number of a book. A Uniform Resource Locator (URL) is the location of a resource, i.e a web address. Both have the the same syntax:

```
URI = scheme:[//authority]path[?query][#fragment]
authority = [userinfo@]host[:port]
```

Web applications are mainly interested in fetching and posting data, so this text will focus on URLs. Here is one example:

```
https://cs.lth.se/edaf90/?firstname=per#url
```

The by far most commonly used protocol is `https`, but `http` and `ftp` are also common. A url can contain any protocol, for example Secure Shell (SSH) or Network Time Protocol (NTP). However you can of course only use protocols your network stack knows how to

handle. For web applications in a web browser, this means https/http which is supported by, `window.fetch`.

The authority part of the example url is the DNS name of the server. An IP-number can also be used. If a port is not given, the default for the protocol is used, 443 for https and 80 for http. In the labs you use port 3000, which is used by the node.js development http server.

The path part of a url commonly identifies a file. If the path is a directory on the servers file system, the server usually responds with a default file from that directory, i.e. `index.html` or `index.php`. This behaviour is part of the server configuration, so do not rely on it if you do not have control over the server.

The query part of a url is commonly generated as part of a http form submit. Here is an example:

```
<form action="/view/" method="get">
  Enter item id:<br>
  <input type="text" name="item">
</form>
```

When the user submits the form, the browser send a request for (this can be interrupted using JavaScript, as you do in the labs):

```
https://cs.lth.se/view?item=42
```

If you set the metod to POST, the form data is submitted in the body of a http POST request. POST is the only alternative if you want to upload files directly from a html form.

The last part of a url, the framgent, makes it possible to refer to a specific element in a html file. The fragment is the id of the referred element, for example:

```
<h2 id="url">URI and URL</h2>
...
<a href="https://cs.lth.se/edaf90/#url">
```

A few years ago, the path in a url referred always referred to a file on the server, and any additional information the server needed to respond to the http request was passed in the query part of the URL. Let's look at an example. A web shop have a page for viewing items. Using the traditional approach, the URL to item 42 is:

```
https://my-shop.com/view?item=42
```

This looks like a form submission, and some customers might hesitate to click on such link. Also, thinking of the URL as a pointer to a recourse, you want the path to refer to the item, not the code that visualises the data. Today another way to structure the URLs has become popular:

```
https://my-shop.com/view/2
```

This looks like a normal link. It is still a URL. The only change is that the path no longer identifies the server side script. The server needs to do some pattern matching to figure out that the script is `view` and 42 is a parameter to the script. Not all web servers support this, for example apache can not do this out of the box (you can get the functionality using a few rewrite rules). An alternative is to use yet another structure for the same information:

```
https://my-shop.com/#view/2
```

The part befor the fragment (#) identifies the script. The script needs to interpret the "view/2" part. The router in react support the two last alternatives: `BrowserRouter` handles the `/view/2` structure, and `HashRouter` supports the `#view/2` alternative.

7.1.1 URL encoding

URLs are text strings, specifically a sequence of 7 bit ASCII characters. 7 bit ASCII can not represent many of the characters in latin, asian, or arabic alphabets, such as åääö. Also, some characters, for example `/#?` have special meaning in a url. Clearly, there is a need to represent these characters in URLs, especially when a URL containing the submit of a html form. This is done by URL-encoding, also known as percent-encoding. The principle is to represent any non safe character with a three character sequence, `%nn`, where `nn` is the hexadecimal value of the 8 bits of the character, for example "ett två" is encoded as "ett%20tv%C3%A5" (space is ASCII character 20, "å" is a two byte sequence in utf-8: 0xC3 and 0xA5). The safe characters, those that do not need percent-encoding, are the letters `a` to `z`, `A` to `Z`, numbers `0` to `9`, and `- _ . ~` (minus, underscore, dot and tilde). According to the latest standard, utf-8 encoding should be used for url-encoding. However, in the context of html form submission, web browsers will use the character encoding of the web page. In JavaScript, url-encodeing is handled by the function `encodeURIComponent()`.

7.2 HTTP

HTTP is the protocol used for almost all communication between web browsers and servers. It is text based, simple and designed to be human readable. The communication is always initialised from the client, the server only responds to requests. The client sends a command to the server, for example a requests for a *resource*. The resource is commonly a file, or a web page. The client may also provide additional information in the request header, for example authentication details. The server execute the command and responds with the result, and possible extra information in the response header. Lets look at an example of how to fetch a page from `cs.lth.se`:

```
GET /edaf90/ HTTP/1.1
HOST: cs.lth.se
```

The first line always has three parts: 1, the command `GET` 2, the resource `/edaf90/` 3, protocol version `HTTP/1.1`.

The http protocol defines the following commands: `GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH`. We discuss them in more details in the section about REST. More information can also be found at mozilla's documentation: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

The resource is the "`path[?query][#fragment]`" part of a url. Do not put the authority part of the url here (server name). This information goes to the header section.

The optional header section starts at line 2. Each line contain one header field following the syntax: `header-name: value`. The value is the text until the next new line character. Leading white spaces are trimmed from the value. A complete list of the http standard headers can be found here https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

You are free to use any name, so the protocol can be extended to the needs of your application. In earlier versions of the protocol custom headers should start with X-, but that was later dropped as some custom headers later became part of the standard.

The header section and the request body are separated by one empty line. The body contains additional information sent to the server, for example the content of a http-form for `POST` requests. The body is normally empty for `GET` requests.

Here is the server response for the example above:

```
HTTP/1.1 200 OK
Date: Sat, 09 Feb 2019 15:51:26 GMT
Server: Apache/2.4.6 (Red Hat Enterprise Linux) OpenSSL/1.0.2k-fips PHP/7.0.27
Content-Language: sv
ETag: "7dc5c74268a1b0ea90ce76b44ebe3181"
Content-Type: text/html; charset=utf-8
x-url: /edaf90/
Age: 44
X-Cache: cached
X-Cache-Hits: 1
Content-Length: 38430
Connection: keep-alive
Accept-Ranges: bytes

<!DOCTYPE html>
... more text, a total of 38430 bytes
```

The response have the same form as the request. The first line has three fields: `protocol-version status-code human-readable-status`. The protocol version is the latest the server supports, but never newer than what the client asked for in the request.

Status codes form groups:

- 1xx (Informational): The request was received, continuing process.
- 2xx (Successful): The request was successfully received, understood, and accepted.
- 3xx (Redirection): The client needs to do additional actions in order to complete the request.
- 4xx (Client Error): The request contains bad syntax or cannot be fulfilled. The client must rephrase the request before any later attempts.
- 5xx (Server Error): The server failed to fulfill an apparently valid request. For example, because a database is down for maintenance. The client may try the same request again later.

The normal response is 200 — OK. You probably also have seen 404 — page not found. A list of status codes can be found at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`, or simply google for “http status code”.

Http is text based, so any binary information, for example images, are always base64 encoded. Make sure you encode/decode any binary data before you send/reads it. Check the documentation of your http-library for the details. More information about base64 encoding can, for example be, found at wikipedia, `https://en.wikipedia.org/wiki/Base64`.

The latest version of the http standard is 2.0. This version keeps the same basic structure: command-header-body. Version 2.0 mainly focus on improving efficiency. The headers can be compressed, so they are no longer human readable in the network stream. The server is also allowed to send more data than was requested, for example when a client asks for a web page, it probably will ask for the images in that page later. The server can bundle all content of the page in the first response.

As a web application developer you do not need to bother about the inner details of the http protocol. All network accesses are made using libraries, both on the client and server side. They handle the formatting of the request, compression et.c. Base64 and url encoding may be handled different in different libraries, so check the documentation of your framework. To use the libraries you do need to know the basics: command, response status code, header and body as these are part of any http-api.

If you want to examen the inners of http-requests, you can use the `curl` command in a terminal.

```
curl -X GET http://cs.lth.se/edaf90
```

7.3 JSON

JSON is a popular text based file format commonly used in RESTful APIs. JSON is a subset of JavaScript. JSON always use unicode character encoding. There are five values:

- a literal, one of **`true`**, **`false`** or **`null`**
- a number, for example `3.1415` or `42`
- a string, a sequence of unicode code points wrapped with quotation marks ", (U+0022). Note, you may not use singel quotation marks `'`, which JavaScript allows.
- an object, a unordered collection of comma separated name-value pairs wrapped with `{ }`. The name and the value are separated by :. The are no restrictions on the name. It may contain any character, but it must be a string (wrapped with quotation marks). In javaScript a name only needs quotation marks if it contains special characters, like space.
- an array, an ordered list of comma separated values wrapped with `[ ]`. You can mix different types of values in the same array.

Objects and arrays can be nested. Example:

```
{
  "Salad + Pasta": {price: 10, foundation: true, gluten: true},
  "'Norsk fjordlax": {price: 30, protein: true},
  "Cashewnötter": {price: 5, extra: true, vegan: true},
  "Sojabönor": {price: 5, extra: true, vegan: true},
  "Tomat": {price: 5, extra: true, vegan: true},
  "Ägg": {price: 5, extra: true},
  "Caesardressing": {price: 5, dressing: true, lactose: true},
  "info": ["open 8-17", {"delivery":true}, [null, true, false]]
}
```

More info about JSON can be found at https://www.json.org

In JavaScript there is a global object called JSON. It contains handy functions, like `parse()` and `stringify()`.

```
const str = '{ "Clavin": {"alter ego": "Spaceman Spiff"}}';
const myObj = JSON.parse(str);
console.log(JSON.stringify(myObj));
```

7.4 REST

Modern web applications are complex and have more in common with desktop applications than html forms from the early days of the web. The web browser comes with basic support for fetching pages and submitting forms. Using these mechanisms, the entire page needs to be fetched and rendered whenever the application need to interact with the server. This is not enough for modern web applications. The application needs to have direct communication with the server to exchange information without freezing the user interface. Web browsers have support for this, see **??**. These communication APIs makes it possible for a web application to send http requests to any server. Http i just text, so every application needs to add its own protocol on top of http. There are many pitfalls and issues you need to be aware of when designing an api for a distributed system, which all web applications inherently are. REST is a set of principles and guidelines to help you develop an api on top of http. Terminology: when following these principles, you say that you have a RESTful api, providing a REST service running on a REST server.

The core principles of Representational State Transfer (REST) are:

- Client–server architecture
- Statelessness
- Cacheability
- Layered system
- Uniform interface

7.4.1 Client–server architecture

The principle behind the client–server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains

7.4.2 Statelessness

Each request from any client contains all the information necessary to service the request, and session state is held in the client.

This eliminates many potential buggs. Let's illustrate this with bugg caused by a stateful protocol. The api of a web shop have two operations `/view-item/` and `/order-last-viewed-item/`. The customer do this:

```
/view-item/13
/view-item/42
  Go to the previous page using the browser history.
  The user sees item 13 on the screen
/order-last-viewed-item
```

Pages in the browser history are often rendered from the browser cache. The server can not know that the customer was viewing item 13 when placing the order, so an order of item 42 was made. The same problems occurs when a user opens the same application in different tabs. You will get a copy of the DOM in the new tab, and once the user starts to interact with the application, the DOMs in the tabs diverges. There is no way for a server to keep track of which tab is the origin of a request.

7.4.3 Cacheability

As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data. Use http headers to control the cacheability of each response.

7.4.4 Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. They can also enforce security policies. Layered system comes for free when building ontop of http.

7.4.5 Uniform interface

This is all about making things simple, more on this in 7.4.6. The four constraints for this uniform interface are:

- Resource identification in requests
- Resource manipulation through representations – When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
- Self-descriptive messages – Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.
- Hypermedia as the engine of application state

Part of the text above comes from wikipedia,
https://en.wikipedia.org/wiki/Representational_state_transfer

7.4.6 REST in practice, CRUD

The text above is written from a theoretical point of view, and in general terms, talking about resources and interfaces. In practice things get relay simple. Resources are usually rows in databases and the operations we want to do are Create, Read, Update, and Delete (CRUD). REST leverage on http, so we use the http commands POST, GET, PUT,

and DELETE. The status code of the respons indicate if the operation was successful, for example 200-ok or 404-not found.

Where do we put the data? Well, this can be debated. There is no right and wrong to this question. Be consistent and adopt to the context of your system. A common approach is to follow the html way of doing things:

- *Resource identification* is part of the url:

```
/students/
/students/42/
```

  The first request selects all students, and the second a specific student.

- *Filtering* can be done using the query part:

```
/students/?name=Per
```

- *Names:* plural nouns are commonly used in URLs. Use lower case letters. If you have multi word names, replace space with minus sign, i.e. `piano-char`. Then things looks nice after url-encoding (you do not want `/piano%20char/`). Avoid CamelCase since the path of the URLs are case insensitive. Avoid verbs in the path, the http command contains this information
- *Http headers:* use it for cache control and authentication.
- *Http body:* data returned from the server goes into the response body. Create and Update sends its data in the body, but Read and Delete only identifies a resource using the URL (empty body).
- JSON: use json format in the http body.

Lecture 9

Angular

Angular is a framework for singel page web applications. An angular application is a set of angular components, and each component have:

- a tag name, so it can be instantiated in html templates
- TypeScript code containing state and behaviour
- a html template, describing how to render the component
- a styling template describing how to style the component (commonly a css file)

Here is an example:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.html',
  styleUrls: ['./my-component.scss']
})
export class MyComponent implements OnInit {
  form = {todo: '', done: false};

  constructor(public todoService: TodoService) { }

  ngOnInit() {
  }

  handleSubmit($event) {
    $event.preventDefault();
    console.log('submitting ' + JSON.stringify(this.form));
  }
}
```

Angular uses class decorations to add metadata to a component, for example the tag name and the names of the template files.

9.1 TypeScript

Angular is written in TyepScript. TypeScript is a superset of JavaScript, so you can always stick to JavaScript, however there are a few convenient things in TypeScript:

- Class attributes, in the example above, `form` is an attribute in the `MyComponent` class. You can add data to your prototypes. The JavaScript way to do the same thing is:

  ```
  MyComponent.prototype.form = someValue;
  ```

  The html template is not part of the class, so you probably want to avoid `private` attributes.

- Look at the code:

  ```
  constructor(public todoService) { }
  ```

  It does two things: 1, declares a `public todoService` attribute 2, initialises the attribute to the value passed to the constructor, that is, it is shorthand for:

  ```
  export class MyComponent implements OnInit {
    public todoService: TodoService;
    constructor(todoService: TodoService) {
      this.todoService = todoService;
    }
  }
  ```

- Types, TypeScript adds static typing to JavaScript. The type syntax looks like json, for example:

  ```
  public form: {todo: string, done: boolean};
  ```

In the course, you should know enough TypeScript to use it in the labs. I will not ask you to write TypeScript code on the exam.

9.2 Template Language

Angular uses html as template language. There are a few additions to the html syntax. Read more about the angular template syntax here: https://angular.io/guide/template-syntax. Here is an example:

```
<form class="example-form" (ngSubmit)="handleSubmit($event)">
  <mat-form-field class="example-full-width">
    <input matInput name="todo" placeholder="What to do"
               [(ngModel)]="form.todo">
  </mat-form-field>
  <mat-checkbox name="done" [(ngModel)]="form.done" class="example-full-width">
    Check when item is done
  </mat-checkbox>
  <br><button type="submit" mat-raised-button>Submit</button>
</form>
{{form|json}}
```

- The `<mat-*>` tags are angular components from the angular material module, see https://material.angular.io/components/form-field/overview

- JavaScript expressions: you can embed JavaScript code in the html code. Note, the embedded code can not have side effects, so avoid assignments `+=`, `++` et.c.

```
<h2>{{myJavaScriptFunction()}}</h2>
```

  A convenient operation that can be used in angular templates is `a?.b`, which is equivalent to `a ? a.b : a`. It might not look like a big difference, but consider rewriting `a?.b?.c?.d?.e` using the `? :` notation.

  The expression context is typically the component instance (name scope). You do not need to write `this` to access attributes from the component class.

- Angular pipes, you can modify the data from an embedded expressions by passing it trough pipes, similar to the pipe in a unix terminal. Pipes can be chained, see https://angular.io/guide/pipes

```
<h2>{{ birthday | date | uppercase}}</h2>
```

  There exists selection of predefined pipes which make your life easier. I commonly use `json`, which is equivalent to `JSON.stringify()`, and `async`, which transform an `Observable` to the last resolved value, for example:

```
<span>The configuration fetched from the server:
{{ http.get(configUrl) | async | json}}</span>
```

  Here is a complete list of the built in pipes: https://angular.io/api?type=pipe.

- Directives looks like html attributes. Behind each directive is a piece of code, which interacts with the underlaying DOM-elements, so basically directives can do anything. You can define your own directives using the `@Directive` class decoration, but that is beyond the scope of this course. Let's start by looking at some structural directives. By convention, all structural directives have names starting with *:

```
<div *ngIf="orders.length>0">Your Shopping Basket</div>

<ul>
  <li *ngFor="let salad of orders">{{salad.toString()}}</li>
</ul>
```

  There are more details to the NgFor, you can get more info from the structure you iteration over:

```
*ngFor="let salad of orders; let i=index; let odd=odd; trackBy: trackById"
```

  For a complete explanation, please visit
  https://angular.io/guide/structural-directives.

  Other directives add functionality or style to a element, for example:

```
<input matInput name="food" placeholder="What to do">
```

  The `matInput` directive will tell angular material to style the `<input>` element. `name` and `placeholder` are standard html attributes associated with any `<input>` element.

9.3 Data binding

The syntax to use for data binding depends on the direction the data flows.When data passes into a component, use the `[target]="expression"` syntax, for example:

```
<button [disabled]="isUnchanged">Save</button>
```

In some cases, you can also use the traditional html syntax:

```
<div class="special">Mental Model</div>
```

Note, `"special"` is a string, not an expression.

When data flows from a component, for example a DOM event, use the `(event)="expression"` syntax:

```
<button (click)="onSave($event)">Save</button>
```

The source event can be found under the name `$event` in the expression string.

Angular support two way data binding, commonly used in forms:

```
<input name="todo" [(ngModel)]="form.todo">
```

This works exactly as bound components in react, but you do not need to write the `onChange()` function. If you change the JavaScript object `form.todo`, angular will update the `<input>` value, and if the user writes in the form field, angular will update the `form.todo` object.

More about bindings can be found here:
https://angular.io/guide/template-syntax#binding-syntax-an-overview.

9.4 @Input and @Output

Components can have input, use the `@Input` decoration to bind a class attribute to a input:

```
@Component({
  selector: 'app-hello-world',
  template: `
    <h3>Hello World</h3>
    <p> {{text}}
  `
})
export class HeroChildComponent {
  @Input() text: string;
```

```
<app-hello-world [text]="'Per was here!'"></app-hello-world>
```

Output from a component is wrapped in an event emitted from the component, and follows the data binding pattern outlined above:

```
@Component({
    selector: 'app-child',
    template: `<button (click)="valueChanged()">Click me</button> `
})
export class AppChildComponent {
    @Output() valueChange = new EventEmitter();
    Counter = 0;
    valueChanged() { // You can give any function name
        this.counter = this.counter + 1;
        this.valueChange.emit(this.counter);
    }
}
```

```
<app-child (valueChange)='displayCounter($event)'></app-child
```

How data is passed in and out of components is described here: https://angular.io/guide/component-interaction.

9.5 Service

In angular components can interact trough services. Services enables bi-directional communication between any components in an angular app. A service is a JavaScript object which provides functionality to the components (functions they can call). Let's look an example, `SaladOrderService` which can mange orders of different salads:

```
import { Injectable } from '@angular/core';
import { Subject }    from 'rxjs';

@Injectable()
export class SaladOrderService {
  private orders = [];
  private orderSource = new Subject<Salad[]>();
  orders$ = this.orderSource.asObservable();

  orderSalad(salad: Salad) {
    this.orders.push(salad);
    this.saladSource.next(this.orders);
  }
}
```

There is one function, `orderSalad()`, adding a salad to the order, and one RxJS Observable `orders$`, which can be used to listen to changes in the orders.

Any angular component can use a service simply by requesting a parameter of the service type in its constructor:

```
@Component({
  selector: 'app-compose-salad',
```

```
  template: `
    <h2>Compose Salad</h2>
    <form (ngSubmit)="submit()">...</form>
  `
})
export class ComposeSaladComponent {
  salad = {};  // will be filled by the html form
  constructor(private orderService: SaladOrderService) {  }
  submit() {
    this.orderService.orderSalad(this.salad);
  }
}
```

```
@Component({
  selector: 'app-view-order',
  template: `
    <h2>Orders</h2>
    <ul>
      <li *ngFor="let salad of orders">{{salad.toString()}}</li>
    </ul>
  `
})
export class ViewOrderComponent {
  orders = [];
  constructor(private orderService: SaladOrderService) {
    orderService.orders$.subscribe(order => this.orders = orders);
  }
}
```

9.6 App.module.ts

Angular needs to know the tag names of all angular components, and the types of all services. This is normally managed in the module of the application, the `app.moudle.ts` file:

```
@NgModule({
  declarations: [
    AppComponent,
    ViewOrderComponent,
    ComposeSaladComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,

    MatButtonModule,
```

```
    MatCheckboxModule,
    MatFormFieldModule,
    MatInputModule,

    AppRoutingModule
  ],
  providers: [SaladOrderService],
  bootstrap: [AppComponent]
})
```

`declarations` is a list of all components declared in this module. `providers` is a list of services declared in this module. `imports` is a list of other angular modules this module is depending in. After importing a module, you can use the components, pipes, directives, and services declared in that module.

Lecture 10

Security

This chapter gives a brief introduction to web security. It is not enough if you developing web applications. Please take a course on web security to complement the web programming course.

10.1 Same Origin Policy

The web is based on trust. In the beginning of the internet, the trust came from that everyone knew everyone. As the internet grew, this have changed. Instead a number of mechanisms have been added to improve security. One is the *same origin policy*. A web application is typically divided to several parts: html, css, and javascript files. On the internet, resources is considered to come from the same origin if they have the same:

1. URI scheme
2. host name
3. port number

Resources from the same origin are considered as part of the same application and are trusted to share data, for example access to cookies, `window.localStorage` and can make http requests to the same origin.

10.2 Cross Site Scripting

Cross-site scripting attacks use known vulnerabilities in web-based applications, their servers, or the plug-in systems on which they rely. Exploiting one of these, attackers fold malicious content into the content being delivered from the compromised site. When the resulting combined content arrives at the client-side web browser, it has all been delivered from the trusted source, and thus operates under the permissions granted to that system. By finding ways of injecting malicious scripts into web pages, an attacker can gain elevated access-privileges to sensitive page content, to session cookies, and to a variety of other information maintained by the browser on behalf of the user. Cross-site scripting attacks are a case of code injection. (text from wikipedia).

10.3 Code Injection

One common technique for cross site scripting attacks is code injection. If the application takes user input and places it in a context where it can be executed as code, it is open for attacks. Let me give you one example: When you search on google, you will get a page

that starts with “search results for: *the text you wrote*”. If this part of the page is created by placing the user input directly into the html code, it will be parsed by the browser and any `<script>` tags will be executed. Other scenarios where this can happen is systems that allow html formated text, for example in messages, or profile presentations. This code will have the same privileges as the rest of the applications, potentially full access to a REST api as an authenticated user. To avoid this, any data from a user must always be escaped befor placed in a html file, the DOM, or sql query.

In practice, how much of this work do I need to do and how much will the frameworks help me? Several common cases are covered by the framework, but check the documentation for the details. In angular:

```
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
```

Is safe. htmlSnippet is escaped, and any tags are displayed on screen. In other context html code is interpreted:

```
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

Angular will sanitise htmlSnippet, removing any `<script>` tag, so this is also safe. The output is however different, now the html code is interpreted.

Lecture 11

Authentication and JWT

In this section I will outline how you can use Json Web Tokens (JWT) for authentication. There are many ways to deal with authentication and I do not claim this is the best, but it is a popular technique and used by for example Auth0.

Username and passwords are sensitive data and should never be stored, either in the client nor on the server. The server saves a hash(password) so it can validate users at login. The hash() function should be a one way function, it is easy to compute the hash value given a password, but given the hash value it is impossible to gess/compute the password. To login the user needs the password, the hash value is not enough. This makes the system less vulnerable to leaked password entries.

How do we deal with the (user, password) data on the client side? We do not want to send the (user, password) data with every request. The old way to deal with this is to let the server remember who's loggen in as part of its session data. This involves setting a cooking with a session id. That cookie will be automatically injected into the header of any http-request by the browser, for example http form submissions. This makes you vulnerable to cross site scripting. Also, the session data does not work well with stateless REST apis.

The modern approach to authentication is to use authentication tokens. When the user loggs in the (user, password) is sent to the server for validation. If the data is correct an authentication token is returned to the client. This authentication token is later passed to the server as proof of a correct login, commonly in the http header:

```
auth: the-token-as-a-string
```

Desired characteristics of the authentication token:

- only the server should be able to generate valid tokens
- the client should be able to verify that the token is genuine (and not produced by a man in the middle)
- the token should have a limited lifetime, limiting the damage of a leaked token.
- the token should stor metadata, such as user name and privileges.

Json Web Tokens have these properties and are commonly used as authentication tokens. JWT is an open standard, RFC 7519. A JWT have three parts:

1. header — a json object contains information about the JWT, for example the algorithm used for the signature

   ```
   {
     "alg": "HS256",
   ```

```
  "typ": "JWT"
}
```

2. payload — a json object contains any data. There are a few properties with predefined meaning, for example iss (issuer), exp (expiration time), sub (subject).

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

3. signature — A digital signature of the JWT. Several different algorithms are supported, but HS256 is commonly used. It uses a public/private key encryption algorithm, ensures that only the server can generate tokens. Any client with the public key can validate that this is an authentic token generated by the server.

All three parts of the JWT is BASE64 encoded and placed in a dot separated string, for example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmF
tZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwp
MeJf36POk6yJV_adQssw5c
```

Lecture 12

Server Send Data

In many scenarios there is a need for the server to initiate the communication, for example when a new message is received in a chat system. Using traditional web technology, any communication must be initiated by the client. Here are some techniques to deal with this.

12.1 polling

This is the simplest approach. The client periodically, for example every 5 minutes, send a request to the server. If no new data is available, the server simply responds with an empty reply. The drawback of this approach is of course that there is a lot of network traffic with no content. Also, the user might wait up to one period for new data to arrive.

12.2 long polling

This is a variant of polling. If the server do not have new data, it holds the connection open but do not send any data. When new data arrives, the server can directly send it to the client since it already have an open connection. There is a time limit on how long a connection can be open. When the http request is closed due to timeout, the client needs to deal with the error and open the connection again.

12.3 Server-Sent Events

This is an efficient technique for simple communication. The client sends a request to receive notifications, so the communication is still initiated from the client. The difference is that the server can respons to the request at any time. This is how you set up the communication:

```
if (!!window.EventSource) {
  var source = new EventSource('server-resource-url');
  source.addEventListener('message', function(e) {
    console.log(e.data);
  }, false);

  source.addEventListener('open', function(e) {
    // Connection was opened.
  }, false);

  source.addEventListener('error', function(e) {
    if (e.readyState == EventSource.CLOSED) {
```

```
      // Connection was closed.
    }
  }, false);

} else {
  // Result to polling :(
}
```

The server can send notifications in a plaintext response, served with a text/event-stream Content-Type:

```
data: {\n
data: "msg": "hello world",\n
data: "id": 12345\n
data: }\n\n
data: {\n
data: "msg": "second message",\n
data: "id": 12346\n
data: }\n\n
```

Notifications are separated with a blank line. One advantage with Server Send Events is that the browser will manage the connection. If it is closed, the browser will automatically reconnect.

12.4 WebSockets

Sockets is the traditional mechanism for communication between computers, commonly used by desktop applications for point to point communication. At the OS-level, all network communication is done using sockets. Sockets allow bi-directional, full-duplex communication and is probably the best choice if you need real time bi-directional communication, for example in games. Sockets only deals with streams of data, commonly using the TCP/IP protocol, so you need to add your own application protocol on to of it.

Lecture 13

Service Workers

Service workers makes it possible to run code in the background in the web browser. In principle, a service workers is intercepting the http request from an application. The service worker can forward the requests to the server, or decide to handle the request on it own. Service workers are primarily used for caching and server send notifications. In addition to the http requests (fetch api), the service worker have access to the indexedDB (offline data storage), however it can not access the DOM so it must communicate with the web application trough events.

Lecture 14

Server Side Rendering

React and angular can give the user a speedy interface since the whole web page do not need to be fetched and rendered when the user navigates inside the app. Loading the first page can however be slow since there is a lot of JavaScript and html-templates to load. To deal with this, modern web frameworks support server side rendered content. The principle is that at build time, a singe static web page is rendered. When the user loads the first page of the app, this static web page is sent to the browser. This is much faster than sending JavaScript code that generate the DOM. This static web page do not only contain html code. It also contains JavaScript, which fetches the single web page application in the background. When the whole app have been fetched, the static web page hands over control to the singe page webb application. This works seamless, without the user seeing anything. To make a server side rendered page, you basically only need to provide a configuration with the application state (router url et.c.) to the build tools. The tools will produce a bundle, containing the initial static web page.