I. Introduction of Web

I.1 Benefits

- No need for backup
- Platform independent
- No software update
- Lower investment costs
- Software as a Service

I.2 Liabilities

- No data sovereignity
- Limited calibration possibilities
- Limited/restricted hardware access
- No operation system access
- More expensive deployment strategies

II. What is Routing?

- Links multiple application parts together
- Provides the concept of information architecture (IA)

II.1 Basics

- Routing is accomplished **completely** client-side
  - No page reload, no roundtrip, server isn't involved
  - Page transition is managed by JS completely
  - Working back-button and bookmarks
- Entry Point [View UI controller] is enforced by the given route
  - Controller provides features behind a View (UI) and bootstraps it
- Router provides client-side event hooks during navigation - Lifecycle management

II.2 Client-side routing concepts

- **The old way** - Earlier, we used anchors (#). Don't use these anymore!
- **The HTML5 way**
  - JavaScript API window.history is used
  - window.history.pushState causes the address bar to show the URL, but won't cause the browser to load it (or even check, if it's valid)
  - window.onpopstate can be used to listen for route changes
  - **warning:** configuration adjustments needed on server-side (all sub-routes must return root-files)

II.3 Example routeConfig

Listing 1: routeConfig.js

```
let router = new ui.Router({
    rootPath: "/demo3",
    initialRoute: "index",
    routes: {
        "index": () => { controller.indexAction(routerOutletView); }
    }
});
```

III. Data Bindings

```
<p>Your team is {{counter.team}}<p>
<p>Your current count is: {{counter.count}}</p>
<form>
    <button data-click="up">Count Up</button>
</form>
```

```
class CounterModel {
    constructor(team, count) {
        this.team = team || "unspec";
        this.count = count || 0;
    }
}
```

IV. Services

- ...contain the major application logic
- ...are generally the source of all application data **Data Services**
- Provide microtesting of smalles possible logic units
- Completely decoupled from UI
- **UI Services** are usually seen in the communication between UI controllers.

```
class CounterController {
    constructor(counterService) {
        this.counterService = counterService;
    }

    indexAction(viewRef) {
        // the service returns the model and the
        // view is rendered using the returned model.
        this.counterService.load((model) => {
            this.renderIndexView(viewRef, model);
        });
        $(viewRef).on('click', '[data-click=up]', (e) => {
            this.counterService.up(model) => {
                this.renderIndexView(viewRef, model);
            }
            e.preventDefault();
        });
    }
```

V. Bundling SPAs

- All JS code must be delivered to the client over potentially metered/slow networks
- Bundling and minifying the source leads to smaller SPA footprint
- Larger SPAs with many modules need a reliable dependency management
- Initial footprint can be reduced by loading dependent modules on-demand

V.1 WebPack as bundler

- **Entry** - The entry point (modules to be bundled) tells webpack where to start and follows the graph of dependencies to know what to bundle.
- **Output** - Tell webpack where to bundle your application
- **Loaders** - Loaders in webpack transform these files into modules as they are added to your dependency graph.
- **Plugins** - Loaders only execute transformations on a per-file basis, plugins are most commonly used performing actions and custom functionality.

Listing 2: webpack.js

```
context: rootDir,
entry: {
    di: srcDir + scriptsDir + "/di.js",
    ui: srcDir + scriptsDir + "/ui.js"
}, output: {
    path: distDir + scriptsDir, filename: "[name].js"
}, module: {
    loaders: [ { test: /\.js$/,
        exclude: /(node_modules|tmp)/,
        loader: 'babel-loader' }]
}, plugins: [
    new HtmlWebpackPlugin({
        title: 'Index',
        filename: 'index.html', // Rel. path from "output" dir
        template: srcDir + '/index.html' // Src file
    }),
    new webpack.optimize.UglifyJsPlugin({
        compress: { warnings: false }
})]
```

VI. Angular 2

VI.1 Angular CLI

```
npm install -g @angular/cli // Install the CLI globally
ng new my-app // Create a new angular app
ng serve --open // Serve the Angular app and open the browser
ng build // Just build the angular app
ng test // Build the angular app and execute the test runner
ng generate module core
```

VI.2 Architectural Overview

- **Modules** - A cohesive block of code dedicated to closely related set of capabilities.
- **Directives** - Provides instructions to transform the DOM.
- **Components** - A component is a directive-with-a-template; it controls a section of the view.
- **Templates** - A template is a form of HTML that tells Angular how to render the component.
- **Metadata** - Describes a class and tells Angular how to process it.
- **Services** - Provides logic of any value, function or feature that your application needs.

VI.3 About Modules

- Every app has at least one Angular module (the **root** Module)
- Modules export features (directives, services, ...) required by other modules
- NICHT zu verwechseln mit ES6 Modules (ES6=pro file; Angular=logischer Block von mehreren ES6 Modulen)
- Library Modules:
  - May accomodate multiple Angular modules
  - Contain and export also other facilities (classes, functions, ...)
  - Angular ships as multiple library modules (all with the @angular-prefix)
  - As an ES6 module, the module library provides single export with all containing features (also known as **barrel** export)

VI.4 Modules

- **Root Module** - By convention named AppModule (app.module.ts). Provides the main view, called the root component, that hosts all other app views. Is bootstrapped by the main.ts
- **Core Module** - Provides globally required services and components directly needed by the root module. The core module should help keep the Root Module clean. Only the root Module should import the Core Module.
- **Shared Module** - Provides globally used components/directives/pipes. It's a global UI component module. Do not specify app-wide singleton providers (services) in a shared module (use Root Module instead).
- **Feature Module** - Splits the application into cohesive feature sets. Allows to assign development responsibilities to different teams. Feature modules are designed to extend the app. A feature module can expose or hide it's implementation from other modules.
- **Lazy Module** - Provides similar features such as Feature Modules. Reduces initial footprint of your SPA. Lazy loaded when invoked by a lazy route. Has it's own DI Container (a child of the root injector).

Listing 3: app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {HttpModule} from '@angular/http';
import {CoreModule} from './core/core.module';
import {AppComponent} from './app.component';
import {AppRoutingModule} from './app-routing.module';
import {AuthModule} from './auth/auth.module';
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
import {DashboardModule} from './dashboard/dashboard.module';
import {DashboardRoutingModule} from './dashboard/dashboard-routing.module';

@NgModule({
  declarations: [ AppComponent ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule,
    DashboardRoutingModule,

    NgbModule.forRoot(),
    CoreModule.forRoot(),
    AuthModule.forRoot(),
    DashboardModule.forRoot(),

    AppRoutingModule
  ],
  providers: [ ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 4: dashboard.module.ts

```
import {NgModule, ModuleWithProviders} from '@angular/core';

import {AuthService} from '../auth/services/auth.service';
import {DashboardComponent} from './components/dashboard/dashboard.component';
import {DashboardRoutingModule} from './dashboard-routing.module';
import {RouterModule} from '@angular/router';

@NgModule({
  // declarations (Components / Directives) used
  // from/within the Module
  declarations: [ DashboardComponent ],
  // Other Modules to import (imports the exported
  // Components/Directives from the other module)
  imports: [ DashboardRoutingModule, RouterModule ],
  // components/Directives (or even Modules)
  // to export (available for other modules; and forRoot() )
  exports: [ ],
  // DI Providers (Services, Tokens, Factories...),
  // may be instantiated multiple times
  providers: [ AuthService ]
})
export class DashboardModule {
  static forRoot(config?:{}):ModuleWithProviders {
    return {
      ngModule: DashboardModule,
      providers: [ ]
    };
}}
```

VI.5 @NgModule() Metadata

- `declarations[Type1, Type2, ...]` - The *view classes* that belong to this module. Angular has 3 view classes: components, directives and pipes.
- `exports[Type1, Type2, Module1, Module2, ...]` - The subset of declarations that should be visible and usable in the component templates of other modules. Can re-export other modules, which are automatically included when importing this module.
- `imports[Module1, Module2, ...]` - Specifies the modules which exports/providers should be imported into *this* module.
- `providers[Provider1, Provider2, ...]` - Creators of services that this module contributes to the global collection of services (Dependency injection container); they become accessible in all parts of the app.
- `bootstrap[Component]` - The main application view, called the *root component*. Only the *root module* should set this property (enables usage of the root HTML tag: <app-root>).

VI.6 Module metadata and provider accumulation mechanisms

- Default import - Imports all components, Pipes, Directives from the given ForeignModule. *Declarations will be re-instantiated on the current module level.* Providers are registered into the current DI container, if registration not yet made.
- `forChild(config?)` import - Represents a static method on a module class (by convention). It is nearly the same as a default import, but allows you to configure services for the current Module level. It returns an object with a `providers` property and an `ngModule` property.
- `forRoot()` import - Represents a static method on a module (by convention, see `forChild()` import). This type of import is useful when you want to enforce that the same provider **won't** be loaded twice by lazy modules.
  - **Only root modules** should import foreign Modules by calling `forRoot()`
  - Declare your providers in `@NgModule` declaration OR in `forRoot()`, but **never** in both.
  - The `providers` are added to the DI container on **root** level
  - Also, the other ForeignModule are imported by the `NgModule` property.
  - Providers from `ForeignModule.forRoot()` take precedence over the `providers` from the module definition.

VI.7 Components

Components control and support the view (Controller in MVC / ViewModel in MVVM). Declared as a TS class with an `@Component` function decorator. The lifecycle is managed by Angular (Hydration, Update, Dehydration)

Listing 5: payment.component.ts

```
import {Component, OnInit} from '@angular/core';
import {NgForm} from '@angular/forms';
import {AuthService} from '../../../auth/services/auth.service';
import {AccountsService} from '../../../services/accounts.service';

@Component({
  selector: 'app-payment',
  templateUrl: './payment.component.html',
  styleUrls: ['./payment.component.css']
})
export class PaymentComponent implements OnInit {
  @Output() click = new EventEmitter<any>();
  @Input() title: string;
  private sender: AccountViewModel;
  private recipient: AccountViewModel = new AccountViewModel();
  private amount: number = 0;

  constructor(private authSvc: AuthService,
              private accSvc: AccountsService) { }

  ngOnInit() {
    this.sender = new AccountViewModel(this.authSvc.authenticatedU
  }

  public recipientChanged(event) {
      this.accSvc.fetchAccountOwner(this.recipient.accountNr)
        .subscribe((nr) => { this.recipient.nr = nr; });
  }
}
```

VI.8 Templates

- Almost all HTML syntax is valid template syntax (except <script> for security reasons). Some legal HTML doesn't make much sense in a template (<head>, <body>)
- Angular extends the HTML vocabulary of your templates with:
  - Interpolation
  - Template Expression/Statements
  - Binding Syntax
  - Directives
  - Template Reference Variables
  - Template Expressoin Operators (Advanced)
- Binding Syntax
  - Two Way Binding [()]: `<input type="text"[(ngModel)]="counter.team">`
  - One Way (View to Model / Event Binding) (...): `<button (click)="counter.eventHandler">`
  - One Way (Model to View / Property Binding) [...] or {{ ...}}: `<input type="text"[(ngModel)]="counter.team">`
- Binding to targets must be declared as Inputs or Outputs (like in the example above)

VI.9 Directives

Similar to a component, but without a template. Declares as a Typescript class with an `@Directive()` function decorator. Two different kind of directives exist: Strucutral directives (Modifies the structure of your DOM) and Attribute directives (Alter the appearance or behavior of an existing element)

- Attribute Directive
  - **NgStyle Directive** `<div [style.font-size]=ïsSpecial ? 'x-large' : 'smaller'>`
  - **NgClass Directive** `<div [class.special]=ïsSpecial>`
- Structural Directives
  - Asterisk is ßyntactic sugar"for something a bit more complicated
  - Angular desugars in two stages: First it translated the `*directive="..."` into a template *attribute*, `template="directive ..."`. Then it translates the attribute into a `<template>` Element.
  - Example: `<div *ngIf="hasTitle">` results in `<template [ngIf]="hasTitle"><div>`

Template reference variables

- References a DOM element within a template
- Can also be a reference to an Angular component or directive
- Reference variables can be used anywhere in the template
- A hash symbol (#) declares a reference variable
- Example:

```
<input placeholder="phone-number" #phone>
<!-- phone refers to the input element -->
<button (click)="callPhone(phone.value)">Call</button>
```

VI.10 Services

- Provides any value, function, or feature that your application needs.
- Almost anything can be a service - it should do one thing and do it well.
- Typical services are logging service, data service, message bus, tax calculator, application configuration
- Strongly coupled to Dependency Injection (Angular uses DI to provide the services to the components who need them. Therefore services must be registered in teh DI Container)

Use the `@Injectable` decorator for services.

```
@Injectable()
export class CounterService { }
```

Then you need to register the service within the DI contianer

```
@NgModule({
 imports: [...],
 declarations: [...],
 providers: [ CounterService ],
 exports: [...]
})
export class CounterModule { }
```

To use the Service in a component, you can declare it in the constructor and it will be injected by the DI Container.

```
@Component(...)
export class CounterComponent {
 private counter : CounterModel;
 constructor(private counterService: CounterService) {
  this.counter = counterService.load();
 }
}
```

VII. Task Parallel Library (TPL)

```
CompletableFuture<Long> future = CompletableFuture.supplyAsync(() -> longQuery());
//other work
process(future.get());
future.thenAccept(result -> System.out.println(result));
CompletableFuture.allOf(future1, future2).thenAccept(continuation);
CompleteableFuture.any(future1, future2).thenAccept(continuation);
```

VIII. .net

- **Exception in Threads:** Exception in Threads fürht zu Abbruch des gesamten Programs.
- **volatile** auch von java kopiert
- **Lokale Variabeln:** Lokale Variabeln müssen nicht Read-only (final) sein.
- **Delegate:** Referenz auf Methode

IX. Actor

- **Vorteile** Aktive Objekte, kein Shared Memory, Kommunikation zwisschen Objekten, kein Race Condition

```
public class NumberPrinter extends UntypedActor {
 public void onReceive(final Object message) {
  if (message instanceof Integer) {
   System.out.print(message);
  }
 }
}
ActorSystem system = ActorSystem.create("System")
ActorRef printer = system.actorOf(Props.create(NumberPrinter.class
for (int i = 0; i < 100; i++) {
    printer.tell(i, ActorRef.noSender());
    //tell(message, sender)
    //getSelf() sefl ref, getSender() sender ref
}
Future<Object> result = Patterns.ask(actorRef, msg, timeout);
system.shutdown();
```

X. GPU Parallelisierung

- SM: Streaming Multiprocessor. Hat mehrere SP
- SP: Streaming Processor.
- SIMD: Single Instruction Multiple Data, Vektorparalleliserung
- NUMA: Non-Uniform Memory Access -¿ Host-Memory zu Device-Memory
- Grid: Hat mehrere Blöcke
- CUDA Block: Threads sind in Blöcke gruppiert
- Thread = virtueller Skalarprozessor
- Block = virtueller Multiprozessor
- Block müssen unabhängig sein, run-to-completion
- Blockgrösse vielfaches von 32
- Shared Memory: Per SM, schnell (4), nur zwischen Threads innerhalb Block sichtbar, paar KB
- Global Memory: Main memory, langsam (400-600), allen threads sichtbar, mehrer GB
- Warp: Block wird intern in 32-Threds Warp zerlegt
- Block läuft auf SM, Warp läuft auf SP eines einzigen SM
- Divergenz: Unterschiedliche Verzweigung im selben Warp, SM führt Verzweigung, die anderen warten
- Memory Coalescing: Zugriffssmuster der Threads, falls aufeinanderfolgende Daten -¿ in ein Memory Burst

XI. Cluster

- Head Node: Zugriffspunkt, rest Compute Nodes
- Job Manager für Monitoring
- HPC Job = vom Client lanciert, hat mehrer Tasks
- HPC Task = Zugriff auf Fle Shares, Ausführung eines Executables, Abhängigkeit zwischen Tasks möglich
- MPI: basiert auf Actor/CSP, Standard
- Communicator: Gruppen von MPI Prozessen
- Communicator-World: Alle Prozesse einer Ausführun

XII. Reactive Programming

- PLINQ Resultate ungeordnet, Java 8 Stream sind geordnet
- Pull: Pipeline-Schritt rückwärst, Input-Quelle is passiv, iteration
- Reactive = Push-Mechanismus: Input-Quelle/Arbeitsschritt is aktiv
- Rx: Observer und Observerbable, beides = Subject

XIII. Software Transactional Memory

- Atomarice Sequenzen von Operationen
- keine inkonsistente Zwischenzustände
- ACI TX: Atomicity (vollständig oder gar nicht sauber) , Consistency (programm vor und nach TX gültig), Isolation (as-if-seriell)
- Deskriptiv: was ist atomar, automatisch isolation, nur Speicherzugriff isoliert
- Problem: Starvation gefahr, Seiteneffekt bei SW-TX bleibt sichtbar
- Nested TX: Commit bei Top-Level TX
- HW Support Intel TSX
- Scala: Wrapping von Variable
- Scala: Write Skew nicht möglich, Starvation problem

```
final Ref.View<Integer> balance = STM.newRef(0);
void deposit(int amount) {
  M.atomic(() -> {
    balance.set(balance.get() + amount);
  });
}
void withdraw(int amount) {
  STM.atomic(() -> {
    if (balance.get() < amount) {
      STM.retry();
    }
    balance.set(balance.get() - amount);
  });
}
// bei Exception wird rollback
// write sekew:
atomic { if (b.onDuty) { a.onDuty = false; } }
atomic { if (a.onDuty) { b.onDuty = false; } }
```

XIV. Misc

```
Collections.synchronizedList(list);
/..Collection(...) /..Map...()
// Lockfreie Datenstrukturen
ConcurrentLinkedQueue<V>, ConcurrentLinkedDeque<V>
ConcurrentSkipListSet<V>, ConcurrentHashMap<K, V>
ConcurrentSkipListMap<K, V>
```

- OutOfMemory Gründe: Kosten zwischen 128kB bis 1MB pro Thread
- notify() vs notifyAll(): Notify() reicht aus, wenn alle Threads auf eine Bedingnug warten.

```
public class UpgradeableReadWriteLock {
  private ReadWriteLock readWriteLock =
  new ReentrantReadWriteLock(true);
  private Lock mutex = new ReentrantLock(true);

  public void readLock() throws InterruptedException {
    readWriteLock.readLock().lock();
  }
  public void readUnlock() {
    readWriteLock.readLock().unlock();
  }
  public void upgradeableReadLock()
  throws InterruptedException {
    mutex.lock();
  }
  public void upgradeableReadUnlock() { mutex.unlock(); }
  public void writeLock() throws InterruptedException {
    mutex.lock();
    readWriteLock.writeLock().lock();
  }
  public void writeUnlock() {
    mutex.unlock();
    readWriteLock.writeLock().unlock();
  }
}
//--
CompletableFuture<String> as = CompletableFuture.supplyAsync(() ->
});
as.thenAccept(result -> {});
// ForkJoinPool
invokeAll(a, b) == a.fork(); b.fork(); b.join(); a.join();
// --
// lock free stack - herausnehmen, falls platzmangel
public class LockFreeStack<T> implements Stack<T> {
    private AtomicReference<StackNode<T>> topNode;
    private StackNode<T> bottomElement = new StackNode<T>(null)
    public LockFreeStack(){
        topNode = new AtomicReference<>(bottomElement);
    }
    public void push(T value) {
        StackNode<T> currentTop;
        StackNode<T> nextTop;
        do{
            currentTop = topNode.get();
            nextTop = new StackNode<>(currentTop, value);
        }while (!topNode.compareAndSet(currentTop, nextTop));
    }

    public T pop() {
        StackNode<T> currentTop;
        do{
            currentTop = topNode.get();
        }while(currentTop != bottomElement
        && !topNode.compareAndSet(currentTop, currentTop.getNextEl
        return currentTop.getValue();
    }
}
// AtomicInteger a = new AtomicInteger(10);
// a.updateAndGet(i -> i + 2);
```

XV. Checklist

- ThreadPool shutdown nicht vergessen
- GPU: Boundry Check wegen zusätzlichen Threads
- wenn wait() oder Condition.await() -¿ InterruptedException nicht vergessen
- try-finall nicht vergessen, wenn lock
- Beim eigenen Code: parameter checks (null check, negative check)
- Bei CyclicBarrier.await() ist BarrierBrokenException möglich
- Spurious wakeup auch möglich als Fehler.