Tech-Stack in Unserem Unternehmen

1. Einleitung: Was ist ein Tech-Stack?

Der Tech-Stack ist die Zusammenstellung von Tools, Programmiersprachen, Frameworks und Libraries, die ein Unternehmen verwendet, um Anwendungen zu entwickeln und zu betreiben. Ein gut gewählter Tech-Stack verbessert die Produktivität, die Skalierbarkeit und die Wartbarkeit von Softwarelösungen und beeinflusst auch, welche Architektur-Entscheidungen sinnvoll sind.

2. TypeScript

2.1 Was ist TypeScript?

TypeScript ist eine von Microsoft entwickelte Programmiersprache, die eine statische Typisierung für JavaScript bereitstellt. Sie wurde 2012 eingeführt und ist eine Obermenge von JavaScript, was bedeutet, dass jede gültige JavaScript-Anweisung auch gültige TypeScript-Anweisungen sind.

2.2 Gründe für die Entwicklung von TypeScript

Hintergrund und Probleme:

- Fehlende Typen und Fehlerbehandlung: JavaScript bietet keine statische Typprüfung, was bedeutet, dass Fehler erst zur Laufzeit auftreten können. Dies erschwert das Debugging und die Wartung von Anwendungen erheblich.
- Mangel an Struktur: Ohne die strengen Typisierungen und Strukturierungsfunktionen ist es schwierig, die Absicht von Code zu verstehen, insbesondere bei komplexen Datenstrukturen und Objekten.

2.3 Ziele von TypeScript

- Fehlerfrüherkennung: Durch statische Typisierung können viele Fehler bereits während der Entwicklungsphase erkannt werden.
- Verbesserte Codequalität und Lesbarkeit: Die klare Definition von Typen und Schnittstellen verbessert die Dokumentation und Verständlichkeit des Codes.

- Unterstützung für moderne JavaScript-Features: TypeScript unterstützt die neuesten JavaScript-Funktionen (wie ES6/ES7) und transpiliert sie in eine Version, die in älteren Browsern ausgeführt werden kann.

2.4 Wichtigste Features von TypeScript

- **Statische Typisierung**: Ermöglicht die Definition von Variablen mit bestimmten Typen. Dies verhindert Typfehler und verbessert die Lesbarkeit.

```
let userName: string = "Max"; // String-Typ
let age: number = 30; // Number-Typ
```

- **Interfaces**: Interfaces definieren die Struktur eines Objekts und sorgen für eine klare Schnittstelle zwischen verschiedenen Teilen der Anwendung.

```
interface User {
    id: number;
    name: string;
    isActive: boolean;
}

const user: User = {
    id: 1,
    name: "Max",
    isActive: true,
};
```

- **Typen und Union Types**: Neben den grundlegenden Typen (string, number, boolean) können auch benutzerdefinierte Typen erstellt werden. Union Types erlauben die Kombination mehrerer Typen.

```
let value: string | number; // Wert kann entweder string oder number sein
value = "Hello";
value = 42;
```

- **Type Any**: Der Typ any wird verwendet, wenn der Typ einer Variablen nicht bekannt ist. Er bietet Flexibilität, sollte aber sparsam eingesetzt werden, da er die Vorteile der statischen Typprüfung reduziert.

```
let dynamicValue: any;
dynamicValue = "This is a string";
dynamicValue = 10; // Gültig
```

- **Generics**: Generics ermöglichen es, Funktionen und Klassen zu definieren, die mit verschiedenen Datentypen arbeiten können, ohne dabei die Typensicherheit zu verlieren.

```
function identity<T>(arg: T): T {
    return arg;
}

let result = identity<string>("Hello");
```

2.5 Vorteile von TypeScript über JavaScript

- Frühe Fehlererkennung: Fehler werden bereits während der Kompilierung entdeckt, was die Debugging-Zeit reduziert.
- Bessere Wartbarkeit: Dank der statischen Typisierung und der Strukturierung mit Interfaces und Klassen ist der Code besser lesbar und wartbar.
- Erweiterte Entwicklungsumgebung: Viele IDEs bieten bessere Unterstützung für TypeScript, z.B. Autovervollständigung, IntelliSense und Refactoring.
- Verbesserte Zusammenarbeit in Teams: Klare Typen und Interfaces helfen, die Kommunikation zwischen Entwicklern zu verbessern und Missverständnisse zu vermeiden.
- Support für moderne JavaScript-Features: TypeScript ermöglicht die Nutzung der neuesten JavaScript-Funktionen, während es sicherstellt, dass der Code auch in älteren Umgebungen funktioniert.

3. Angular

3.1 Was ist Angular?

Angular ist ein von Google entwickeltes Webframework, das die Entwicklung von Single Page Applications (SPAs) erleichtert. Es wurde 2010 als AngularJS (oder Angular 1) veröffentlicht und 2016 als Angular 2 (kurz einfach „Angular“) neu gestaltet. Angular 2 und die nachfolgenden Versionen (z.B. Angular 11, Angular 12) basieren auf TypeScript, was eine bessere Struktur und Typensicherheit bietet.

3.2 Angular CLI

Was ist Angular CLI?

Angular CLI (Command Line Interface) ist ein leistungsstarkes Werkzeug, das Entwicklern hilft, Angular-Anwendungen schnell zu erstellen, zu verwalten und zu testen.

Beispiele für CLI-Kommandos:

- **Projekt erstellen**:

```
ng new my-angular-app
```

Dieser Befehl erstellt eine neue Angular-Anwendung mit dem Namen my-angular-app, einschließlich aller notwendigen Konfigurationen und Dateien.

- **Komponente erstellen**:

```
ng generate component my-component
```

Dies erstellt eine neue Komponente namens my-component mit den Standarddateien und der zugehörigen Registrierung im Modul.

3.3 Komponenten in Angular

Was sind Komponenten?

Komponenten sind die grundlegenden Bausteine von Angular-Anwendungen. Jede Komponente enthält die Logik und das Template für einen bestimmten Teil der Benutzeroberfläche. Sie fördern die Wiederverwendbarkeit und erleichtern das Testen und die Wartung des Codes.

Beispiel für eine App-Komponente:

```
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'Willkommen in meiner App';
}
```

Hier ist die AppComponent die Hauptkomponente der Anwendung. Sie definiert die Struktur und das Verhalten der gesamten App. Das Template der App-Komponente könnte andere Komponenten wie den Header, Footer und Content enthalten.

Template der AppComponent:

```
<h2>{{ title }}</h2>
<app-header></app-header>
<router-outlet></router-outlet>
```

In diesem Template wird die HeaderComponent durch den Selector `<app-header>` eingebunden. Der `<router-outlet>`-Tag ist der Platzhalter für die routenbasierten Komponenten, die abhängig von der aktuellen URL angezeigt werden.

Routing zwischen Komponenten:

Angular verwendet das Router-Modul, um zwischen verschiedenen Komponenten zu navigieren. Wenn eine Route aktiviert wird, wird die zugehörige Komponente angezeigt, und die vorherige Komponente wird zerstört. Dies ermöglicht das Laden dynamischer Ansichten ohne ein vollständiges Neuladen der Seite.

Routing-Konfiguration:

```
const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule {}
```

Hier zeigt die Anwendung HomeComponent, wenn der Benutzer die Startseite besucht, und wechselt zu AboutComponent, wenn die Route `/about` aufgerufen wird. Der Vorteil ist die Möglichkeit, dynamische Ansichten zu laden, ohne die gesamte Seite neu laden zu müssen.

Beispiel für eine Header-Komponente:

```
@Component({
    selector: 'app-header',
    templateUrl: './header.component.html',
    styleUrls: ['./header.component.css']
})
export class HeaderComponent {}
```

HTML:

```
<nav>
    <ul>
        <li><a routerLink="/">Home</a></li>
        <li><a routerLink="/about">About</a></li>
    </ul>
</nav>
```

3.4 Angular-Lifecycle-Hooks

Was sind Lifecycle-Hooks?

Angular bietet verschiedene Lifecycle-Hooks, die es Entwicklern ermöglichen, auf bestimmte Phasen im Lebenszyklus einer Komponente oder eines Dienstes zu reagieren.

Wichtige Lifecycle-Hooks:

- **ngOnInit()**: Wird aufgerufen, nachdem die Komponente initialisiert wurde. Ideal für die initiale Logik.

```
ngOnInit() {
    // Initialisierungscode
}
```

- **ngOnDestroy()**: Wird aufgerufen, kurz bevor die Komponente zerstört wird. Hier können Abonnements oder andere Ressourcen aufgeräumt werden.

```
ngOnDestroy() {
    // Aufräumcode
}
```

3.5 Angular Services

Was sind Services?

Services sind Klassen, die häufig benötigte Funktionen bereitstellen, z.B. zur Datenverwaltung oder zur Durchführung von HTTP-Anfragen. Sie fördern die Wiederverwendbarkeit und die Trennung von Anliegen (Separation of Concerns).

Beispiel für einen Service:

```
@Injectable({ providedIn: 'root' })
export class DataService {
    constructor(private http: HttpClient) {}

    getData() {
        return this.http.get('/api/data');
    }
}
```

3.6 Angular Material

Was ist Angular Material?

Angular Material ist eine UI-Bibliothek, die eine Sammlung von vorgefertigten UI-Komponenten bereitstellt, die nach den Prinzipien des Material Designs gestaltet sind.

Beispiele für Angular Material-Komponenten:

```
<mat-toolbar color="primary">
    <span>Mein App-Titel</span>
</mat-toolbar>

<mat-card>
    <mat-card-title>Beispielkarte</mat-card-title>
    <mat-card-content>Hier ist der Inhalt der Karte.</mat-card-content>
</mat-card>
```

Diese Komponenten verbessern das visuelle Erscheinungsbild und die Benutzererfahrung der Anwendung erheblich.

3.7 RxJS und Angular

Was ist RxJS?

RxJS (Reactive Extensions for JavaScript) ist eine Bibliothek für reaktive Programmierung, die das Arbeiten mit asynchronen Datenströmen erleichtert.

Wie wird RxJS in Angular verwendet?

Angular nutzt RxJS intensiv für die Datenverwaltung, insbesondere bei HTTP-Anfragen und der Verarbeitung von Benutzerereignissen.

Beispiel für die Verwendung von RxJS in Angular:

```
this.dataService.getData().subscribe(data => {
    this.items = data;
});
```

Hier wird `getData()` aufgerufen, um Daten zu laden. RxJS-Observables ermöglichen es, auf Änderungen in den Daten zu reagieren.

4. Was ist Maven?

Maven ist ein Build-Management-Tool für Java-Projekte. Es ermöglicht die Automatisierung des Build-Prozesses, die Verwaltung von Abhängigkeiten und die Ausführung von Tests. Mit Maven können Entwickler komplexe Projekte mit verschiedenen Abhängigkeiten effizient verwalten.

POM-Datei:

In Maven wird eine Projektbeschreibung in einer Datei namens `pom.xml` (Project Object Model) gespeichert. Diese Datei definiert das Projekt, seine Abhängigkeiten und Konfigurationen.

Beispiel für eine einfache `pom.xml`-Datei:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-spring-boot-app</artifactId>
    <version>1.0.0</version>

    <properties>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

5. Spring Boot

5.1 Was ist Spring Boot?

Spring Boot ist ein Framework, das die Entwicklung von Java-Anwendungen vereinfacht, indem es die Einrichtung von Spring-Anwendungen automatisiert. Es ermöglicht

Entwicklern, produktionsbereite Anwendungen mit minimalem Aufwand zu erstellen. Spring Boot ist Teil des größeren Spring-Ökosystems und ist darauf ausgelegt, die Konfiguration und Bereitstellung zu vereinfachen.

5.2 Schichten in Spring Boot

In Spring Boot folgt man oft dem Schichtenarchitektur-Muster, das es ermöglicht, die Logik in separate, leicht wartbare Teile zu gliedern. Die drei Hauptschichten sind:

5.2.1 Controller

Controller sind für die Verarbeitung von HTTP-Anfragen verantwortlich. Sie empfangen die Anfragen von den Clients, leiten sie an die Service-Schicht weiter und senden die Antworten zurück.

Beispiel für einen Controller:

```java
package com.example.myapp.controller;

import com.example.myapp.service.UserService;
import com.example.myapp.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }
}
```

5.2.2 Service

Services enthalten die Geschäftslogik der Anwendung. Sie verarbeiten die Anfragen, führen Validierungen durch und kommunizieren mit der Repository-Schicht, um Daten zu lesen oder zu speichern.

Beispiel für einen Service:

```java
package com.example.myapp.service;

import com.example.myapp.model.User;
import com.example.myapp.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User createUser(User user) {
        return userRepository.save(user);
    }
}
```

5.2.3 Repository

Repositorys sind für den Datenzugriff zuständig. Sie bieten Methoden zum Speichern, Lesen, Aktualisieren und Löschen von Daten in einer Datenbank. In Spring Boot wird häufig Spring Data JPA oder Spring Data JDBC verwendet, um die Interaktion mit der Datenbank zu vereinfachen.

Beispiel für ein Repository:

```java
package com.example.myapp.repository;

import com.example.myapp.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Hier können benutzerdefinierte Abfrage-Methoden definiert werden
}
```

5.3 Spring Data JDBC

Spring Data JDBC ist ein Teil des Spring Data-Projekts, das die Arbeit mit relationalen Datenbanken vereinfacht. Es nutzt die JDBC-API für die Datenbankinteraktion und mappt die Daten automatisch auf die Java-Objekte.

Automatisches Mapping

Entität und Tabelle: Spring Data JDBC verwendet einfache Entitäten, die direkt mit Datenbanktabellen verknüpft sind. Jede Entität wird als Klasse dargestellt, und die Felder dieser Klasse werden den Spalten in der Tabelle zugeordnet.

Beispiel für eine Entität:

```java
package com.example.myapp.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

@Table("users")
public class User {

    @Id
    private Long id;
    private String name;
    private String email;

    // Getter und Setter
}
```

5.4 Spring Boot Security

Spring Boot Security bietet eine leistungsstarke und anpassbare Sicherheitsarchitektur. Es schützt Anwendungen durch Authentifizierung und Autorisierung.

Grundlegende Funktionen

- **Authentifizierung:** Überprüfung der Identität eines Benutzers (z. B. durch Benutzername und Passwort).
- **Autorisierung:** Festlegung, welche Ressourcen für authentifizierte Benutzer zugänglich sind.

5.5 Spring Boot DevTools

Spring Boot DevTools ist ein nützliches Tool, das Entwicklern hilft, die Produktivität während der Entwicklung zu steigern. Es bietet Funktionen wie automatisches Neuladen der Anwendung und verbesserte Konfiguration während der Entwicklungsphase.

5.6 Anwendungseigenschaften

Die Konfiguration der Anwendung erfolgt in der Datei `application.properties`, die im Verzeichnis `src/main/resources` liegt. Hier können Sie verschiedene Einstellungen vornehmen, wie z. B. Datenbankverbindungen oder Serverkonfigurationen.

Beispiel für eine einfache `application.properties`-Datei:

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

5.7 Vorteile von Spring Boot

- **Schnelle Entwicklung:** Entwickler können schnell Prototypen erstellen und ihre Anwendungen in Produktionsumgebungen bereitstellen.
- **Einfaches Setup:** Mit der Autokonfiguration von Spring Boot ist die Initialisierung von Anwendungen schneller und einfacher.
- **Wachstumsfreundlichkeit:** Da Spring Boot Teil des Spring-Ökosystems ist, können Entwickler problemlos Funktionen wie Sicherheit (Spring Security) und Datenzugriff (Spring Data) integrieren.
- **Aktive Community:** Spring Boot hat eine große und aktive Community, die Unterstützung und Ressourcen bereitstellt.

6. Jenkins

6.1 Was ist Jenkins?

Jenkins ist ein Open-Source-Automatisierungsserver, der hauptsächlich für Continuous Integration (CI) und Continuous Delivery (CD) eingesetzt wird. Entwickelt wurde Jenkins von Kohsuke Kawaguchi und das Tool ist heute eine der meistgenutzten Lösungen zur Automatisierung von Softwareentwicklungsprozessen.

Jenkins unterstützt Teams bei der Automatisierung von Builds und Tests, um Fehler frühzeitig zu erkennen. Es arbeitet mit Plugins und kann flexibel an verschiedene Anforderungen angepasst werden.

Geschichte von Jenkins

- **Entwicklung:** Jenkins wurde ursprünglich 2004 als „Hudson“ entwickelt und später in „Jenkins“ umbenannt.
- **Popularität:** Heute ist Jenkins eines der beliebtesten CI/CD-Tools und wird von vielen großen und kleinen Unternehmen weltweit eingesetzt.

6.2 Git in Jenkins

Git ist ein weit verbreitetes Versionskontrollsystem, das es Entwicklern ermöglicht, Codeänderungen zu speichern, nachzuverfolgen und zu versionieren. Jenkins nutzt Git, um Code automatisch abzurufen und basierend auf dem aktuellen Stand einen Build durchzuführen.

- **Code-Quelle:** Jenkins kann auf ein Git-Repository zugreifen und den aktuellen Code-Stand abrufen.
- **Automatischer Build:** Jenkins kann so eingerichtet werden, dass es täglich zu einer bestimmten Uhrzeit (z. B. jeden Tag um 01:00 Uhr) den Code aus dem Repository zieht und einen neuen Build startet.

6.3 Docker in Jenkins

Docker ist eine Plattform, die es ermöglicht, Anwendungen in leichtgewichtigen Containern zu isolieren und bereitzustellen. Jenkins nutzt Docker, um Builds in einer stabilen, konsistenten Umgebung auszuführen und die Bereitstellung zu vereinfachen.

- **Containerized Builds:** Jenkins kann einen Docker-Container starten, der alle Abhängigkeiten für den Build enthält. So wird eine einheitliche Umgebung gewährleistet.
- **Image-Erstellung und Bereitstellung:** Jenkins kann Docker-Images bauen und speichern, die dann auf Servern bereitgestellt werden.

6.4 Jenkins-Pipeline: Aufbau und Ablauf

Ein Beispiel für eine Jenkins-Pipeline, die eine tägliche Routine für den Build und die Bereitstellung der Anwendung darstellt:

Ablauf:

1. **Täglicher Build-Trigger:** Jenkins wird so konfiguriert, dass es jeden Tag zu einer festen Uhrzeit (z. B. 01:00 Uhr) einen neuen Build startet.
2. **Code-Checkout von Git:** Jenkins holt den aktuellen Code-Stand aus dem Git-Repository.
3. **Build und Test in Docker:** Der Build erfolgt in einem Docker-Container, und automatisierte Tests werden ausgeführt.
4. **Fehlerbenachrichtigung:** Schlägt der Build oder Test fehl, wird das Team benachrichtigt (z. B. per E-Mail oder Chat-Integration).
5. **Image-Erstellung und Bereitstellung:** Wenn alles erfolgreich ist, wird ein neues Docker-Image erstellt und auf den Server bereitgestellt.

Pipeline-Beispiel:

```
pipeline {
    agent any
    triggers {
        cron('H 1 * * *') // Täglicher Trigger um 01:00 Uhr
    }
    stages {
        stage('Checkout Code') {
            steps {
                git url: 'https://github.com/user/repository.git', branch:
'main'
            }
        }
        stage('Build in Docker') {
            agent {
                docker { image 'maven:3.6.3-jdk-11' } // Container für Build
            }
            steps {
                sh 'mvn clean package' // Maven-Build ausführen
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test' // Tests ausführen
            }
        }
        stage('Build Docker Image') {
            steps {
                script {
                    docker.build("myapp:${env.BUILD_ID}") // Docker-Image
erstellen
                }
            }
        }
        stage('Deploy') {
            steps {
                // Bereitstellungslogik hier einfügen
            }
        }
    }
}
```

6.5 Vorteile der Jenkins-Pipeline mit Git und Docker

- **Automatisierung:** Jenkins-Pipelines ermöglichen die vollständige Automatisierung des Build- und Bereitstellungsprozesses.
- **Konsistenz:** Die Verwendung von Docker-Containern stellt sicher, dass die Anwendung in einer einheitlichen Umgebung läuft, was Probleme durch Umgebungsunterschiede verringert.
- **Frühes Erkennen von Fehlern:** Durch automatisierte Tests und Builds können Fehler frühzeitig im Entwicklungsprozess erkannt werden, was die Qualität der Software erhöht.
- **Flexibilität:** Jenkins bietet viele Plugins und Integrationen, die es Teams ermöglichen, ihre CI/CD-Prozesse anzupassen und zu erweitern.