RICH CLIENT: SERVER

WIEDERHOLUNG

VERANTWORTLICHKEITEN - JSF

- View-Management
- State-Management
- Rendering
- Events
- Routing
- Validation
- Data-Management
- Persistence

VERANTWORTLICHKEITEN - RICH CLIENT

- View-Management
- State-Management
- Rendering
- Events
- Routing
- Ensurance

VERANTWORTLICHKEITEN - WEBSERVICE

- Validation
- Data-Management
- Persistence

WEBSERVICE

WEBSERVICE

Drei grundlegende Eigenschaften:

- Stateless
- Scalable
- Untrusting

Speaker notes

- Keine Übertragung von Zustandsänderungen
- Keine Übertragung von unzusammenhängenden Informationen

WEBSERVICE - STATELESS

- Kein Zustand
- Keine Session
- Anfrage ausschließlich mit fachlichen Informationen

Speaker notes

- Keine Übertragung von Zustandsänderungen
- Keine Übertragung von unzusammenhängenden Informationen

WEBSERVICE - STATELESS

- Keine nicht-persistenten Informationen
- Transparentes Caching ausgenommen
- Persistierung in Datenbank oder Dateisystem
- Transparente Datenbank oder Dateisystem

Speaker notes

- Keine anfrageübergreifende Informationen
- Transparente Caches agieren auf persistenten oder berechenbaren Daten
- Transparente Caches agieren niemals auf flüchtigen Daten
- Transparente Caches für persistente Daten müssen kurzlebig oder Änderungen bewusst sein
- Transparente Datenbanken/Dateisysteme (ver)teilen über mehrere Instanzen

WEBSERVICE - SCALABLE

- Abhängig von ausschließlich externen Informationen
  - Eingaben des Clients
  - Daten der Persistence
- Instanzen sind identitätslos
- Dynamisches hoch-/runterfahren von Instanzen

WEBSERVICE - UNTRUSTING

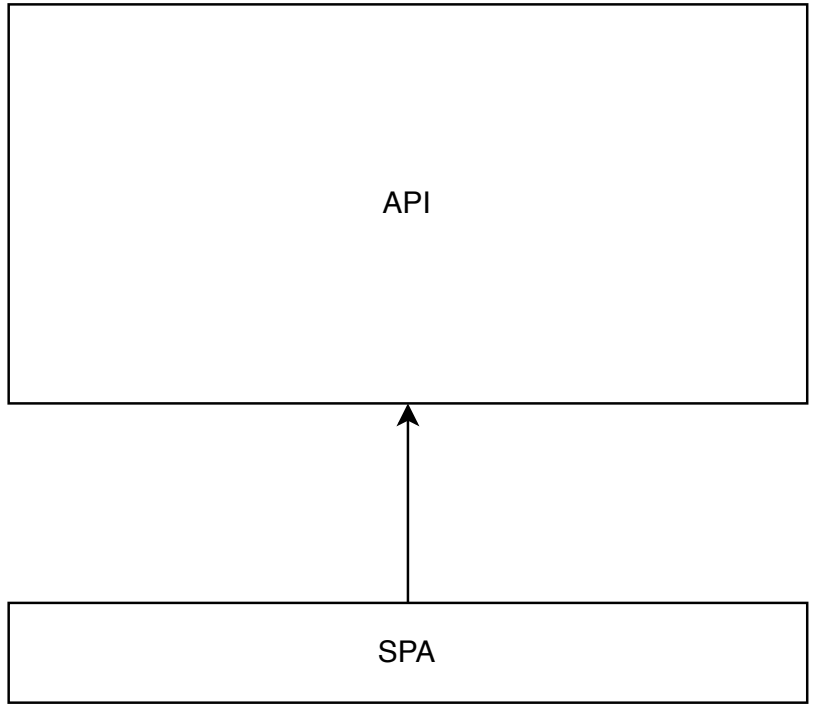
- Validierung aller Eingaben
- Isolierung aller Eingaben
- Durchgehende Prüfung der Authorisierung

Speaker notes

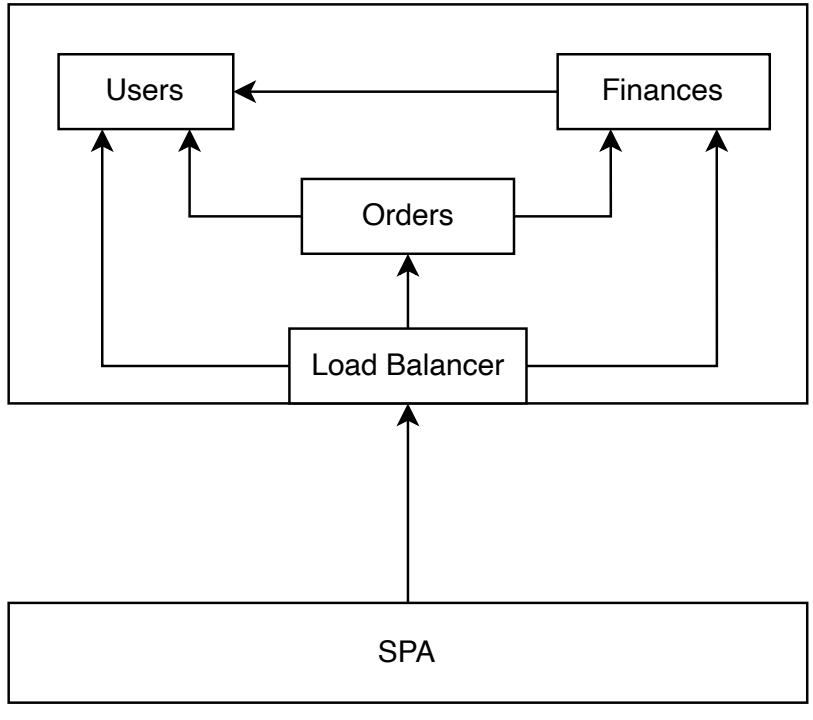
- Validierung stellt Korrektheit sicher
- Validierung stellt keine Sicherheit sicher
- Isolierung durch formlose Betrachtung der Eingaben
- Isolierung durch z.B. Prepared-Statements
- Mindestens Validierung des Tokens (zumeist über Signature/Secret)
- Eventuelle Überprüfung der Zugriffsrechte

ARCHITEKTUREN

ARCHITEKTUREN



ARCHITEKTUREN



ARCHITEKTUREN

- Aufteilung von Verantwortlichkeiten
- Abgrenzung einzelner Komponenten
- Interaktion zwischen Komponenten

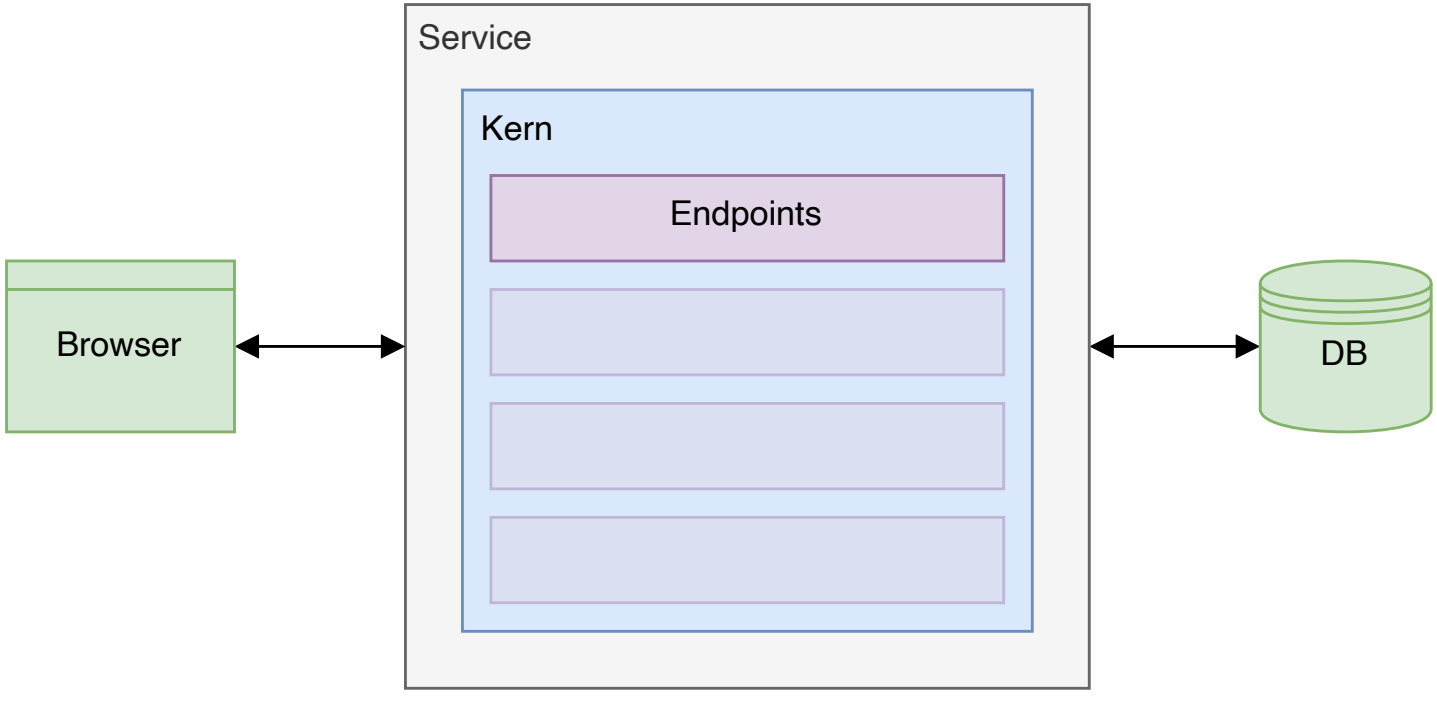
ARCHITEKTUREN

- Architektur bedingt API
- API bedingt nicht die Architektur

ARCHITEKTUREN

- Monolith
- Modulith
- Services
- Microservice

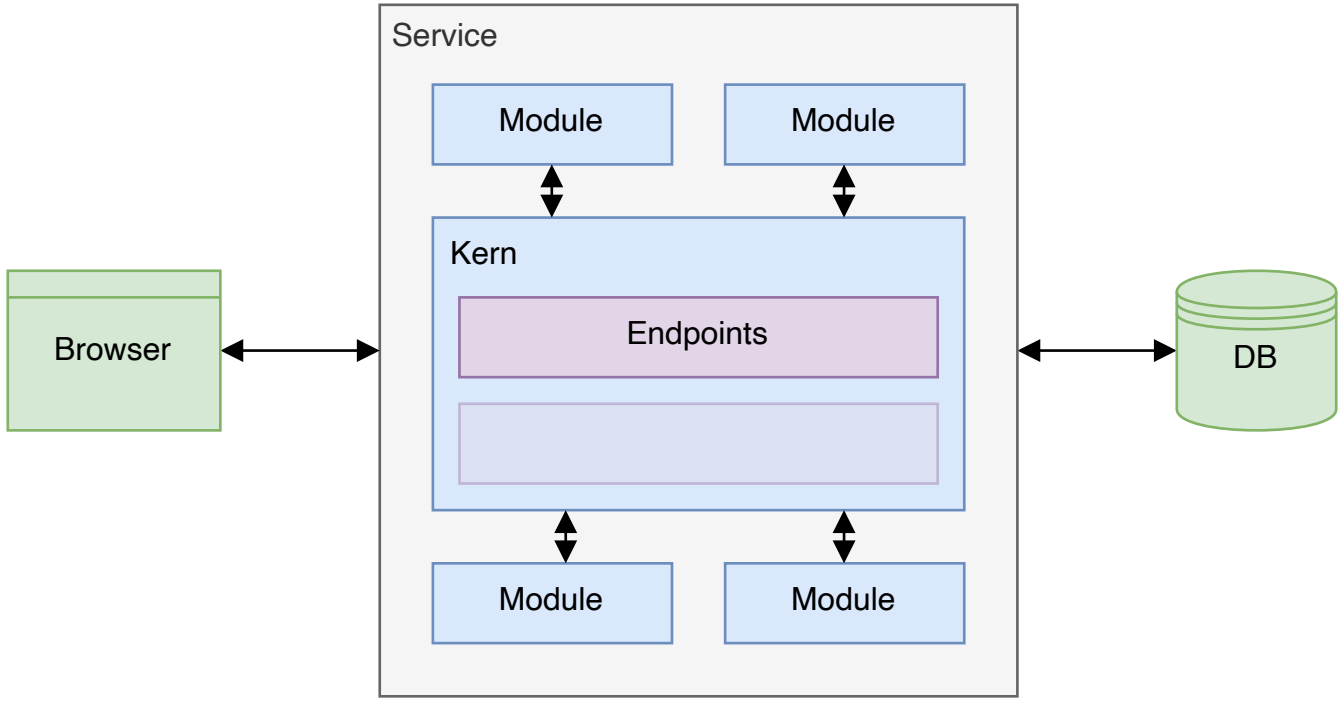
ARCHITEKTUREN - MONOLITH



ARCHITEKTUREN - MONOLITH

- Alle Aspekte der Anwendung in einem Projekt
- Keine Trennung zwischen Fachlichkeiten
- Keine externen Abhängigkeiten zur Laufzeit

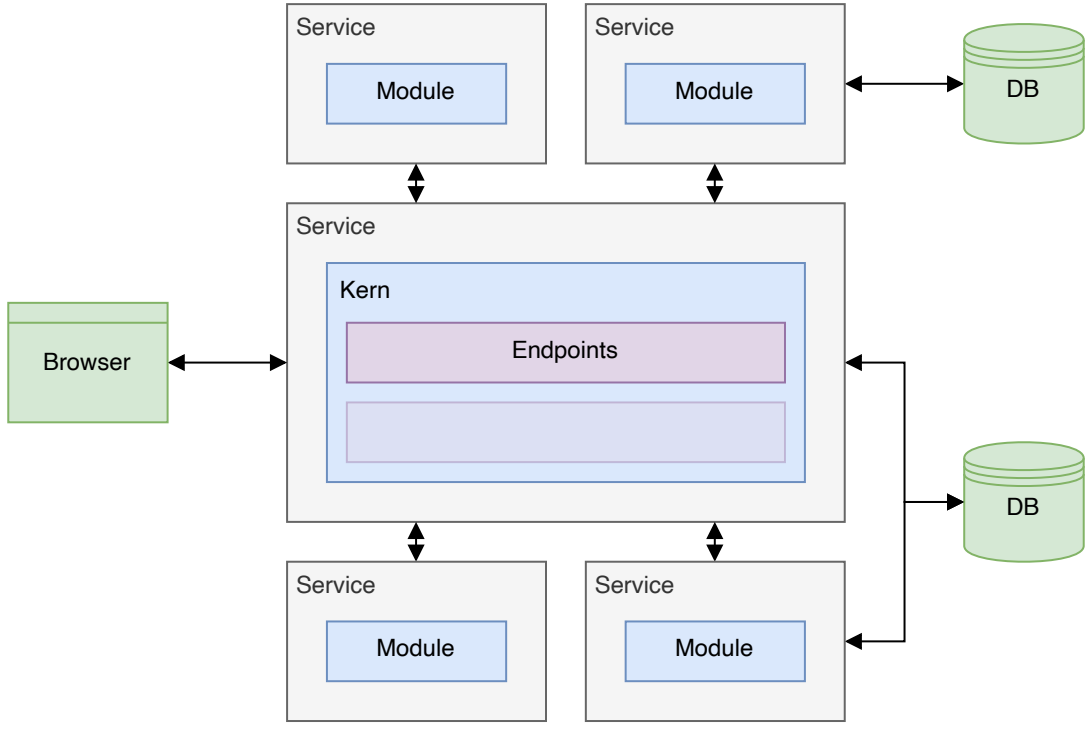
ARCHITEKTUREN - MODULITH



ARCHITEKTUREN - MODULITH

- Unterteilung der Anwendung in Fachlichkeiten
- Auslagerung der Fachlichkeiten in Module
- Module definieren öffentliche Schnittstellen
- Auslagerung in Form von Package, Modul, Projekt
- Keine Auslagerung zur Laufzeit
- Zusammengeführt durch Kern

ARCHITEKTUREN - SERVICES



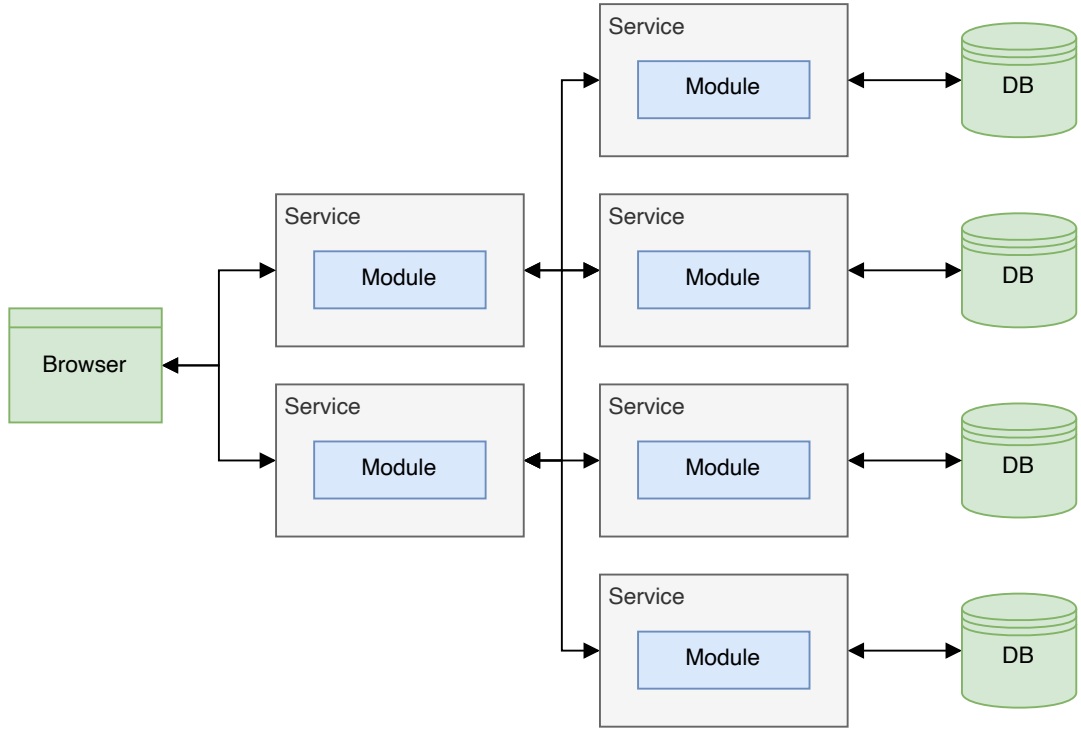
ARCHITEKTUREN - SERVICES

- Modulith als Kern
- Auslagerung einzelner Module in Services
- Services haben eigene Datenhaltung

Speaker notes

- Daten müssen nicht repliziert werden, da Kern das Mapping übernimmt

ARCHITEKTUREN - MICROSERVICES



ARCHITEKTUREN - MICROSERVICES

- Auslagerung jedes Modules in Services
- Expliziter Kern durch implizite Abhängigkeiten zwischen Services ersetzt
- Services replizieren Daten in eigener Datenhaltung

Speaker notes

- Daten müssen repliziert werden, da es keinen Kern fürs Mapping gibt

VERGLEICH

VERGLEICH - KRITERIEN

- Initialaufwand
- Wartungsaufwand
- Betriebsaufwand
- Personalaufwand

VERGLEICH - KRITERIEN

- Abhängigkeit
- Ausführbarkeit
- Testbarkeit
- Skalierbarkeit
- Zuverlässigkeit
- Ausfallsicherheit

VERGLEICH - INITIALAUFWAND

- Aufsetzten der Architektur

VERGLEICH - INITIALAUFWAND

Monolith	Modulith	Services	Microservices
Gering	Mittel	Mittel	Hoch

Speaker notes

- Monolith braucht kein Konzept
- Modulith braucht einfaches Konzept
- Services braucht erweitertes Konzept
- Microservices braucht allgemeines Konzept

VERGLEICH - WARTUNGSAUFWAND

- Einführung neuer Features
- Entfernung alter Features
- Behebung von Fehler
- Aktualisierung der Abhängigkeiten
- Refactoring

Speaker notes

- Abhängigkeiten sind Sprache, Frameworks, Libraries, Services, Datenbanken, Schnittstellen

VERGLEICH - WARTUNGSAUFWAND

Monolith	Modulith	Services	Microservices
Hoch	Mittel	Mittel	Gering

Speaker notes

- Monolith: stark erhöhter Aufwand
- Modulith: Module leicht erweiterbar; Kern erhöhter Aufwand
- Services: Aufwand abhängig von Kern oder Service
- Microservices: können vollständig neugeschrieben werden

VERGLEICH - BETRIEBSAUFWAND

- Betreiben der Services
- Instandhaltung der Umgebung
- Behebung von Störungen

VERGLEICH - BETRIEBSAUFWAND

Monolith	Modulith	Services	Microservices
Gering	Gering	Mittel	Hoch

Speaker notes

- Monolith: wenige, große Instanzen; wenige Server, physische Umgebung
- Modulith: s.h. Monolith
- Services: wenige, große Instanzen + kleine, evt. häufige Services; einige Server, evt. partiell virtualisierte Umgebung
- Microservices: viele, kleine Instanzen; viele Server, virtualisierte Umgebung

VERGLEICH - PERSONALAUFWAND

- Teamgröße sowie Teamanzahl
- Erhöhte Komplexität erfordert mehr Personal
- Mehr Personal erfordert erhöhte Flexibilität

VERGLEICH - PERSONALAUFWAND

Monolith	Modulith	Services	Microservices
Gering	Mittel	Mittel	Hoch

Speaker notes

- Monolith: sehr kleines Entwicklungsteam benötigt; Operations durch Entwickler
- Modulith: sehr kleines bis kleines Entwicklungsteam benötigt; evt. extra Personal für Operations
- Services: Entwicklungsteam abhängig von Größe des Projektes; extra Personal für Operations
- Microservices: ein, großes bis mehrere, kleine Entwicklungsteams; klein bis mittleres Operationsteam

VERGLEICH - ABHÄNGIGKEIT

- Trennung der Fachlichkeiten
- Freiheit der Technologien

VERGLEICH - ABHÄNGIGKEIT

Monolith	Modulith	Services	Microservices
Hoch	Hoch	Mittel	Gering

Speaker notes

- Monolith: sehr starke Kopplung, keine Kohäsion
- Modulith: mäßige Kopplung, gewisse Kohäsion
- Services: losere Kopplung, partiell hohe Kohäsion; mehrere Sprachen/Frameworks möglich
- Microservices: lose Kopplung, hohe Kohäsion; mehrere Sprachen/Frameworks/Images möglich

VERGLEICH - AUSFÜHRBARKEIT

- Ausprobieren neuer Features
- Nachstellen von Fehler
- Aufsetzen der Umgebung

VERGLEICH - AUSFÜHRBARKEIT

Monolith	Modulith	Services	Microservices
Hoch	Hoch	Mittel	Gering

Speaker notes

- Monolith: nur eine Instanz benötigt
- Modulith: s.h. Monolith
- Services: mehrere Instanzen benötigt
- Microservices: mehrere Instanzen und Umgebung benötigt

VERGLEICH - TESTBARKEIT

- Validierung der Korrektheit
- Absichern von Entwicklungen

VERGLEICH - TESTBARKEIT

Monolith	Modulith	Services	Microservices
Gering	Mittel	Mittel	Hoch

Speaker notes

- Monolith: nur in Gesamtheit testbar
- Modulith: einzelne Module testbar, Kern nur in Gesamtheit
- Services: einzelne Module/Services testbar
- Microservices: einzelne Services modular testbar

VERGLEICH - SKALIERBARKEIT

- Reaktionsfähigkeit bei Fluktuationen
- Effiziente Nutzung der Resourcen

VERGLEICH - SKALIERBARKEIT

Monolith	Modulith	Services	Microservices
Keine	Gering	Mittel	Hoch

Speaker notes

- Monolith: nicht skalierbar
- Modulith: einzelne Module über Threading skalierbar
- Services: einzelne Services können skaliert werden
- Microservices: alle Services können skaliert werden

VERGLEICH - ZUVERLÄSSIGKEIT

- Störungsanfälligkeit
- Kommunikationsabbrüche
- Fehlerhafte Zustände
- Netzwerke, Hardware, Software

VERGLEICH - ZUVERLÄSSIGKEIT

Monolith	Modulith	Services	Microservices
Hoch	Hoch	Mittel	Gering

Speaker notes

- Monolith: kaum Netzwerkverbindung; keine Interaktion zwischen Servergruppen; wenige Teile
- Modulith: s.h Monolith
- Services: Kommunikation zwischen Kern und Service anfällig; mehrere Teile
- Microservices: Netzwerkvirtualisierung und Servergruppen anfällig; viele Teile

VERGLEICH - AUSFALLSICHERHEIT

- Ausfallsicherheit
- Redundanz

VERGLEICH - AUSFALLSICHERHEIT

Monolith	Modulith	Services	Microservices
Gering	Gering	Mittel	Hoch

Speaker notes

- Monolith: Single-Point-of-Failure
- Modulith: s.h Monolith
- Services: Kern Single-Point-of-Failure, Services redundant
- Microservices: alle Services redundant

VERGLEICH - ZUSAMMENFASSUNG

	Monolith	Modulith	Services	Microservices
Initialaufwand	Gering	Mittel	Mittel	Hoch
Wartungsaufwand	Hoch	Mittel	Mittel	Gering
Betriebsaufwand	Gering	Gering	Mittel	Hoch
Personalaufwand	Gering	Mittel	Mittel	Hoch
Abhängigkeit	Hoch	Hoch	Mittel	Gering
Ausführbarkeit	Hoch	Hoch	Mittel	Gering
Testbarkeit	Gering	Mittel	Mittel	Hoch
Skalierbarkeit	Keine	Gering	Mittel	Hoch
Zuverlässigkeit	Hoch	Hoch	Mittel	Gering
Ausfallsicherheit	Gering	Gering	Mittel	Hoch

VERGLEICH - ANFORDERUNGEN

Monolith	Modulith	Services	Microse
Unbekannt - Einfach	Einfach - Umfangreich	Umfangreich - Komplex	Komplex

Speaker notes

- Anforderungen und Teamgröße limitieren Architekturmöglichkeiten
- Architektur nach Konvergierung von Anforderungen und Personalaufwand wählen
- Teamgröße muss sich mit Anforderungen decken

VERGLEICH - TEAMGRÖSSE

Monolith	Modulith	Services	Microservices
Klein	Klein - Groß	Mittel - Groß	Groß - Mehrere

Speaker notes

- Anforderungen und Teamgröße limitieren Architekturmöglichkeiten
- Architektur nach Konvergierung von Anforderungen und Personalaufwand wählen
- Teamgröße muss sich mit Anforderungen decken

VERGLEICH - FAZIT

- Anforderungen und Teamgröße limitieren jeweils Architekturmöglichkeiten
- Architektur aus Deckung der Architekturmöglichkeiten wählen
- Teamgröße muss sich mit Anforderungen decken

VERGLEICH - FAZIT

- Monolith für unbekannte Projekte
- Modulith für mehr Wartbarkeit
- Services für Skalierbarkeit
- Microservices für Zuverlässigkeit

UMSETZUNG

TECHNOLOGIEN - WEBENGINEERING I

- Servlets
  - Rohes HTTP
  - Erfordert eigene Implementierung
- JSP
  - Implementierung von Servlets
  - Ausschließlich HTML

TECHNOLOGIEN - WEBENGINEERING II

- JSF
  - Implementierung von Servlets
  - Quasi ausschließlich HTML
- ?
  - Vollständiges HTTP
  - Vielseitige Media-Types
  - Einfache Bedienung

TECHNOLOGIEN - WEBENGINEERING II

- JSF
  - Implementierung von Servlets
  - Quasi ausschließlich HTML
- Spring, Quarkus, Micronaut...
  - Vollständiges HTTP
  - Vielseitige Media-Types
  - Einfache Bedienung

SPRING-BOOT

SPRING

- Application Framework
- Dependency-Injection-Container

SPRING-BOOT

- Basiert auf Spring
- Erweitert um Java EE
- Convention-over-Configuration
- Annotation-Base Configuration
- Spring ursprünglich eigentlich XML

Speaker notes

- Spring besteht aus Modulen die miteinander kombiniert werden können
- Spring Boot erweitert Spring um Java EE (Servlets)
- Standardkonfiguration wird bevorzugt, am besten keine komplizierte Konfiguration
- Abweichend davon kostet es Konfigurationsaufwand

SPRING-BOOT - BOOTSTRAP

```
@SpringBootApplication
public class MySpringApplication {
    public static void main(String[] args) {
        SpringApplication.run(MySpringApplication.class, args)
    }
}
```

Speaker notes

- Startet eine leere Anwendung
- Trotzdem bietet die Anwendung oftmals bereits Endpunkte wie z.B. /health

SCHICHTEN

- Frontend
- Middleware
- Backend

SPRING SCHICHTEN - FRONTEND

- Schnittstelle zur Außenwelt
- Einordnung
  - Wird nicht referenziert
  - Referenziert Middleware

SPRING SCHICHTEN - MIDDLEWARE

- Implementiert Businesslogik
- Einordnung
  - Wird von Frontend und Middleware referenziert
  - Referenziert Backend

SPRING SCHICHTEN - BACKEND

- Persistence Ebene / andere Services
- Einordnung
  - Wird von Middleware referenziert
  - Referenziert nichts

FRONTEND

FRONTEND

- Implementierung erfolgt durch Controller
- API in ReST, GraphQL usw.

FRONTEND - CONTROLLER

```
@RestController
public class PersonController {
    ...
}
```

Speaker notes

- Ein Controller ist der Einstiegspunkt für alle Anfragen

FRONTEND - METHODS

GET /persons

```
@RestController
public class PersonController {
    @RequestMapping(
        method = RequestMethod.GET,
        path = "/persons"
    )
    public List<Person> getPersons() {
        ...
    }
}
```

FRONTEND - METHODS

POST /persons {"firstName": "John", "lastName": "Doe"}

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @RequestMapping(
        method = RequestMethod.POST
    )
    public void createPerson(@RequestBody Person person) {
        ...
    }
}
```

FRONTEND - METHODS

PUT /persons {"firstName": "John", "lastName": "Doe"}

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @RequestMapping(
        method = RequestMethod.PUT
    )
    public void updatePerson(@RequestBody Person person) {
        ...
    }
}
```

FRONTEND - METHODS

DELETE /persons/John%32Doe

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @RequestMapping(
        method = RequestMethod.DELETE,
        path = "/{name}"
    )
    public void deletePerson(@PathVariable String name) {
        ...
    }
}
```

FRONTEND - METHODS

Methoden haben Shorthands-Annotations

- `@RequestMapping(method = RequestMethod.GET)-> @GetMapping()`
- `@RequestMapping(method = RequestMethod.POST)-> @PostMapping()`
- `@RequestMapping(method = RequestMethod.PUT)-> @PutMapping()`
- `@RequestMapping(method = RequestMethod.DELETE)-> @DeleteMapping()`

FRONTEND - PATHS

GET /persons

```
@RestController
public class PersonController {
    @GetMapping(path = "/persons")
    public List<Person> getPersons() {
        ...
    }
}
```

FRONTEND - PATHS

GET /persons

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @GetMapping
    public List<Person> getPersons() {
        ...
    }
}
```

Speaker notes

- Ganze Pfade können in einem Controller zusammengefasst werden
- Alle Endpunkte des Controllers befinden sich implizit unter der Oberpfad

FRONTEND - PATHS

GET /persons/subpath

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @GetMapping(path = "/subpath")
    public String getSomething() {
        ...
    }
}
```

FRONTEND - PATH VARIABLES

GET /persons/John%32Doe

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @GetMapping(path = "/{name}")
    public Person getPerson(@PathVariable String name) {
        ...
    }
}
```

FRONTEND - PARAMETERS

GET /persons?firstName=John&lastName=Doe

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @GetMapping
    public Person getPerson(
        @RequestParam String firstName,
        @RequestParam String lastName
    ) {
        ...
    }
}
```

FRONTEND - BODIES

PUT /persons {"firstName": "John", "lastName": "Doe"}

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @PutMapping
    public void updatePerson(@RequestBody Person person) {
        ...
    }
}
```

FRONTEND - MEDIA-TYPES

POST /persons/convert {"firstName": "John", "lastName": "Doe"}

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @PostMapping(
        path = "/convert",
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_XML_VALUE
    )
    public Person convertPerson(@RequestBody Person person) {
        ...
    }
}
```

FRONTEND - MEDIA-TYPES

POST /persons/291/document/43006

```
@RestController
@RequestMapping(path = "/persons")
public class PersonController {
    @GetMapping(
        path = "/{id}/documents/{documentId}",
        produces = MediaType.APPLICATION_OCTET_STREAM_VALUE
    )
    public byte[] get(
        @PathVariable Long id,
        @PathVariable Long documentId
    ) {
        ...
    }
}
```

FRONTEND - MEDIA-TYPES

- Konvertierung zwischen Java und JSON, XML usw.
- Häufig genutzte Media-Types automatisch unterstützt
- Darunter z.B. auch Text, HTML oder Binär
- Serialisierung kann konfiguriert werden

MIDDLEWARE

MIDDLEWARE

- Implementierung erfolgt durch Services
- Teilt die Businesslogik in Fachlichkeiten auf

MIDDLEWARE - UMSETZUNG

- Empfohlen als Interfaces mit Implementierungen
  - Austauschbar
  - Konfigurierbar
- **aber** keine Pflicht

MIDDLEWARE - SERVICE INTERFACE

```
public interface PersonService {
    Person getPerson(String name);
}
```

MIDDLEWARE - SERVICE IMPLEMENTATION

```
@Service
public class PersonServiceImpl implements PersonService {
    @Override
    public Person getPerson(String name) {
        ...
    }
}
```

MIDDLEWARE - DEFAULT IMPLEMENTATION

```
@Service
@ConditionalOnMissingBean(PersonService.class)
public class DefaultPersonService implements PersonService {
    @Override
    public Person getPerson(String name) {
    ...
    }
}
```

MIDDLEWARE - CONDITIONAL IMPLEMENTATION

```
@Service
@ConditionalOnProperty("service.dumbMode")
public class DumbModePersonService implements PersonService {
    @Override
    public Person getPerson(String name) {
        ...
    }
}
```

MIDDLEWARE - SERVICE

```
@Service
public class DocumentService {
public byte[] getDocument(Person person) {
...
}
}
```

BACKEND

BACKEND

- Implementierung erfolgt durch Repositories / Services
- Datenbanken, Dateisystem und anderes Services

BACKEND - JPA ÜBERSICHT

- Objekt-Relationales-Mapping
- Entities
- Abbildungen von Objekten auf Tabellen
- Transaktionsmanagement
- Aggregation von zusammengehörigen Änderungen
- Gewährleistung von Datenintegrität

BACKEND - JPA ENTITIES

- `@Entity` zur Deklaration eine Entity
- `@Id` zur Markierung des ID-Feldes
- `@GeneratedValue` zur automatischen Generierung
- usw.

Speaker notes

- JPA Annotations sind zahlreich
- Objekt-Relationales-Mapping mithilfe von JPA ist nicht Teil des Rahmens der Vorlesung
- Für ausführliche Informationen zu JPA: https://www.baeldung.com/learn-jpa-hibernate

BACKEND - JPA REPOSITORY

```
@Repository
public interface PersonRepository

  extends JpaRepository<Person, Long> {
  ...
}
```

BACKEND - JPA REPOSITORY

- Implementiert Datenbankschnittstelle für eine Entity
- Verschiedene Arten von Datenbankschnittstelle möglich (JPA, ElasticSearch etc.)
- Deklaration als Interface, Implementierung erfolgt automatisch
- Queries werden anhand des Names automatisch generiert
- Standard-Methoden bereits vorgegeben

BACKEND - JPA REPOSITORY

```
List<Person> findAll();
```

```
SELECT * FROM person
```

BACKEND - JPA REPOSITORY

```
Person findById(Long id);
```

```
SELECT * FROM person WHERE id=:personId
```

BACKEND - JPA REPOSITORY

```
Person findByFirstNameAndLastName(
    String firstName,
    String lastName
);
```

```
SELECT * FROM person WHERE firstName=:firstName
    AND lastName=:lastName
```

BACKEND - JPA REPOSITORY

```
@Query("SELECT n FROM Person p "
        + "WHERE p.tag IN (:tags) "
        + "AND p.creationDate >= :timestamp")
List<Person> findWithTagsAfter(
    String[] tags,
    OffsetDateTime timestamp
);
```

JPA-REPOSITORY - BETTER PRACTICE

- Vielzahl an vordefinierten Operationen
- Wrapper-Klasse für explizite Schnittstellen
- Mehr Aufwand - Mehr Konsistenz
- Projekt-spezifisches Wording
- Verändern der Methodensignatur
- Keine ungewollten Operationen

JPA-REPOSITORY - BEISPIEL

```
@Repository
public class PersonRepository {
    private final SpringPersonRepository delegate;

    public PersonRepository(
        @Autowired SpringPersonRepository delegate
    ) {
        this.delegate = delegate;
    }

    public @Nullable Note find(@NotNull Long id) {
        return delegate.findById(id).orElse(null)
    }
}
```

REFERENZIERUNG

REFERENZIERUNG

- Spring ist ein Dependency-Injection-Container
- Bootstrap baut Objektgraphen auf
- Objektgraph ist normalerweise statisch
- Objektgraph erlaubt aber dynamische Erweiterung
- Zwei primäre Quellen für Objekte
  - Components
  - Configurations

Speaker notes

- Objektgraphen ist der Graph den der Dependency Injection Container aufbaut
- Quellen -> Einstiegspunkte für den Graphen
- zu jeder Zeit können Objekte zur Laufzeit hinzugefügt werden

REFERENZIERUNG - VERWENDUNG

```
@RestController
public class PersonController {
    public PersonController(
      @Autowired PersonService personService
    ) {
        ...
    }
}
```

REFERENZIERUNG - VERWENDUNG

```
@Service
public class PersonServiceImpl {
    public PersonServiceImpl(
        @Autowired PersonRepository personRepository
    ) {
        ...
    }
}
```

REFERENZIERUNG

- Benötigt Aufruf durch Dependency-Injection-Container
- Auflösung der Referenzen über Typ
- Mehrfach vorhandene Objekt über Namen ggf. Classifier
- Boostrap scheitert wenn Referenz nicht auslösbar
  - kein entsprechendes Objekt
  - mehrere entsprechende Objekte

REFERENZIERUNG

- Zwei Einstiegspunkte in den Objektgraphen
  - Components
  - Configurations

REFERENZIERUNG - COMPONENTS

- Klassen direkt oder indirekt annotiert mit `@Component`
- Durch `@RestController`, `@Service` und `@Repository` indirekt annotiert

REFERENZIERUNG - COMPONENTS

```
@Component
public class IndependentComponent {
    ...
}
```

REFERENZIERUNG - COMPONENTS

```
@Component
public class DependentComponent {
    public DependentComponent(
        @Autowired RequiredComponent component
    ) {
        ...
    }
}
```

REFERENZIERUNG - CONFIGURATIONS

- Klassen annotiert mit `@Configuration` über Methoden annotiert mit `@Bean`

REFERENZIERUNG - CONFIGURATIONS

```
@Configuration
public class SomeConfiguration {
    @Bean
    public IndependentComponent createComponent() {
        ...
    }
}
```

REFERENZIERUNG - CONFIGURATIONS

```
@Configuration
public class SomeConfiguration {
    @Bean
    public DependentComponent createComponent(
        @Autowired RequiredComponent component
    ) {
        ...
    }
}
```

CONFIGURATIONS

CONFIGURATIONS

- Indirekte Deklaration von Objekten
- Ändern und Erweitern bestehender Objekte
- Aufruf durch den Dependency-Injection-Container

CONFIGURATIONS - BEISPIEL

```
@Configuration
public class MyConfiguration {
    @Bean
    public MyComponent createComponent() {
        ...
    }
}
```

CONFIGURATIONS - BEISPIEL

```
@Configuration
@EnableWebMvc
public class WebConfiguration implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**");
    }
}
```

VALIDATION

VALIDATION

- Überprüfung der Eingaben
- Client ist nicht vertrauenswürdig
- Spring unterstützt `javax.validation` Annotations

VALIDATION - VERWENDUNG

- `@Valid` zur Markierung von zu validierenden Parametern
- `@NotNull`
- `@NotBlank` mindestens EIN nicht Whitespace-Charakter
- `@Size` Einschränkung der Länge von Strings & Collections
- `@Min`, `@Max` Einschränkung des numerischen Wertebereichs
- `@Email` erwartet eine valide Email-Adresse

Speaker notes

- Für ausführliche Informationen über javax.validation: https://www.baeldung.com/javax-validation

VALIDATION - BEISPIEL

```
public class Note {
    ...
    @NotBlank(message = "description must not be blank")
    private String description;
    ...
}
```

VALIDATION - BEISPIEL

```
@PostMapping
public Note createNote(
    @RequestBody @Valid NoteProposal proposal
) {
    ...
}
```

PRAXIS

PRAXIS - AUFGABEN

- TODOs abfragen
- TODO anlegen
- TODO abfragen
- TODO ändern
- TODO löschen

PRAXIS - VORBEREITUNG

- Repository: `SpringTodoRepository` anlegen, in `TodoRepository` einbinden
- Service: `TodoServiceImpl` anlegen
- Controller: `TodoService` in `TodoController` einbinden

PRAXIS - TODOS ABFRAGEN

`GET /todo`

- Repository: Methode zum Abrufen alle TODOs anlegen
- Service: Repository aufrufen
- Controller: Endpunkt anlegen, Service aufrufe

PRAXIS - TODO ANLEGEN

```
POST /todo {"title": "Cleanup"}
```

- Repository: Methode zum Speichern anlegen
- Service: Aus Proposal ein TODO machen, in Repository speichern
- Controller: Endpunkt anlegen, Service aufrufen

PRAXIS - TODO ABFRAGEN

`GET /todo/{id}`

- Repository: Methode zum Abrufen eines TODO anlegen
- Service: Repository aufrufen und Existenz überprüfen
- Controller: Endpunkt anlegen, Service aufrufen

PRAXIS - TODO ÄNDERN

`PUT /todo/{id} {"done": true}`

- Repository: -
- Service: TODO laden, anpassen, in Repository speichern
- Controller: Endpunkt anlegen, Service aufrufen

PRAXIS - TODO LÖSCHEN

`DELETE /todo/{id}`

- Repository: Methode zum Löschen anlegen
- Service: TODO laden, in Repository löschen
- Controller: Endpunkt anlegen, Service aufrufen

PRAXIS - VALIDIERUNG

- `TodoProposal.title`
  - nicht Leer
  - maximal 50 Zeichen