

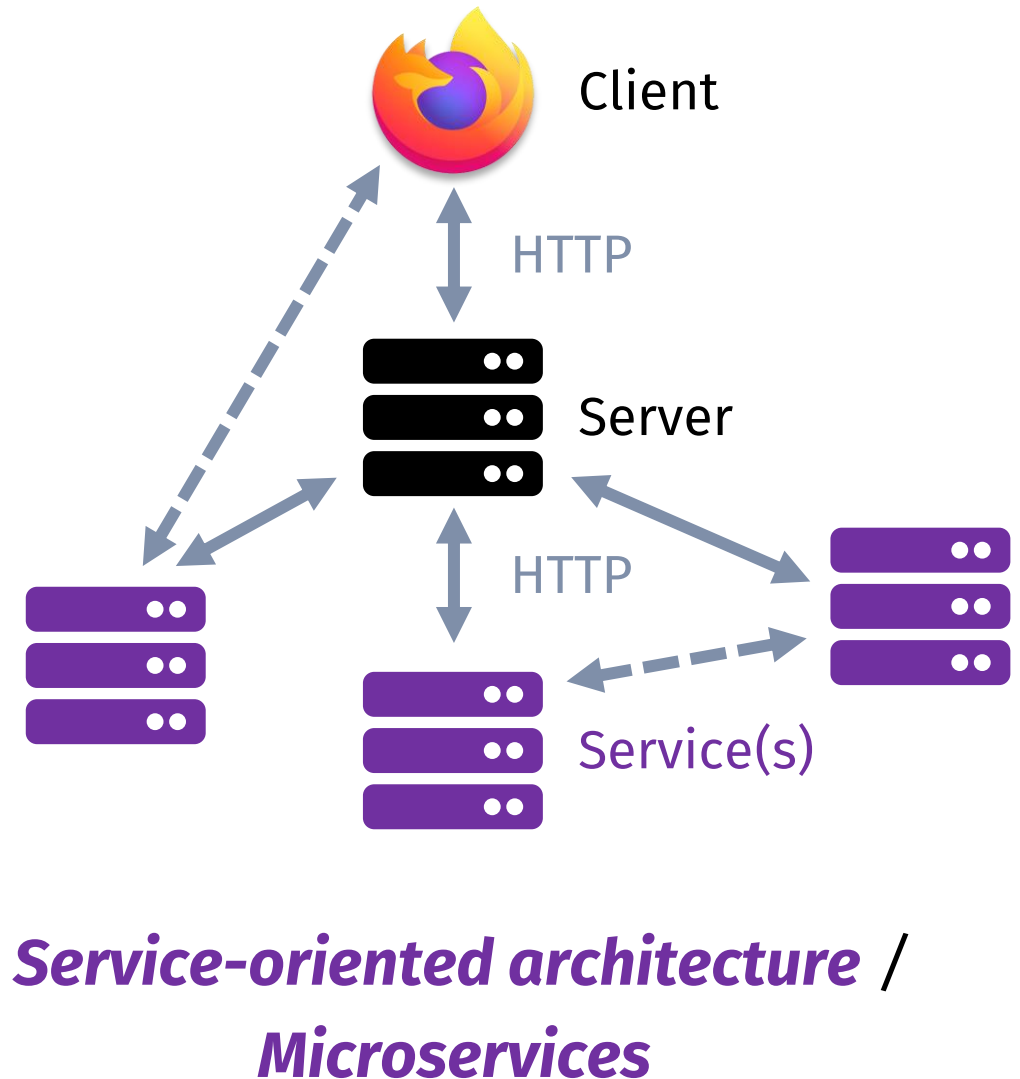
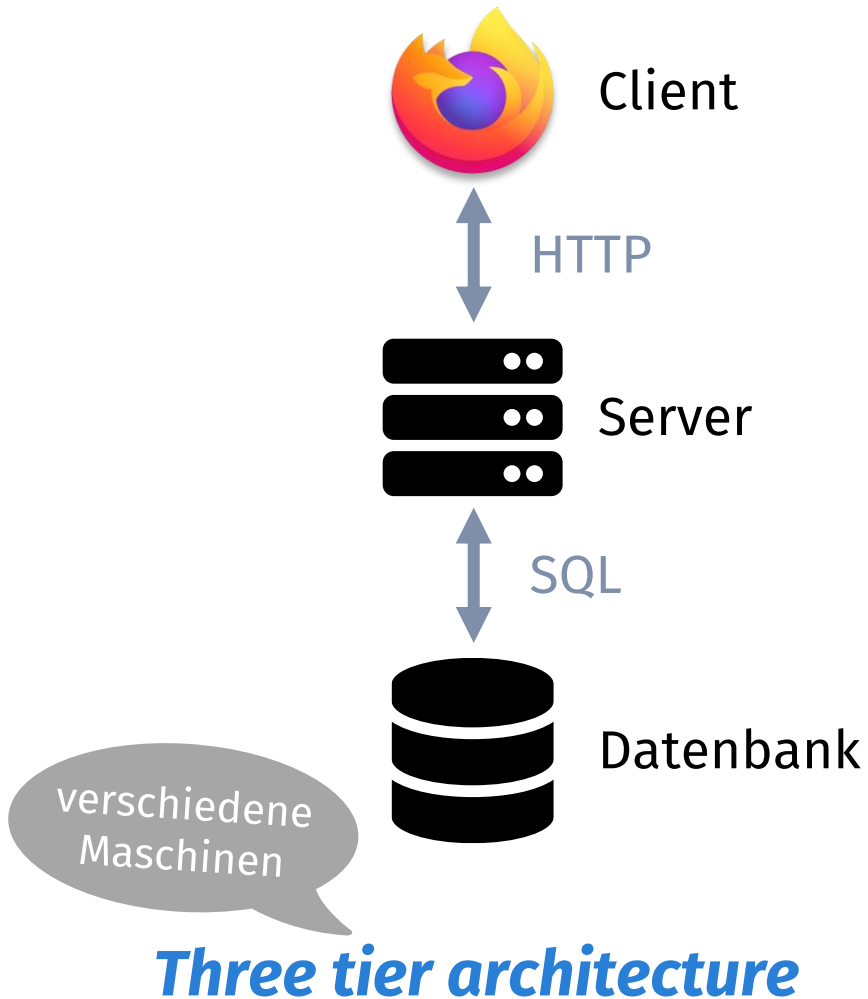
Web Engineering

Webservices mit REST

Adrian Herzog

(basierend auf der Arbeit von Michael Faes, Michael Heinrichs & Prof. Dierk König)

Architektur von Web-Apps



Ziele von Webservices

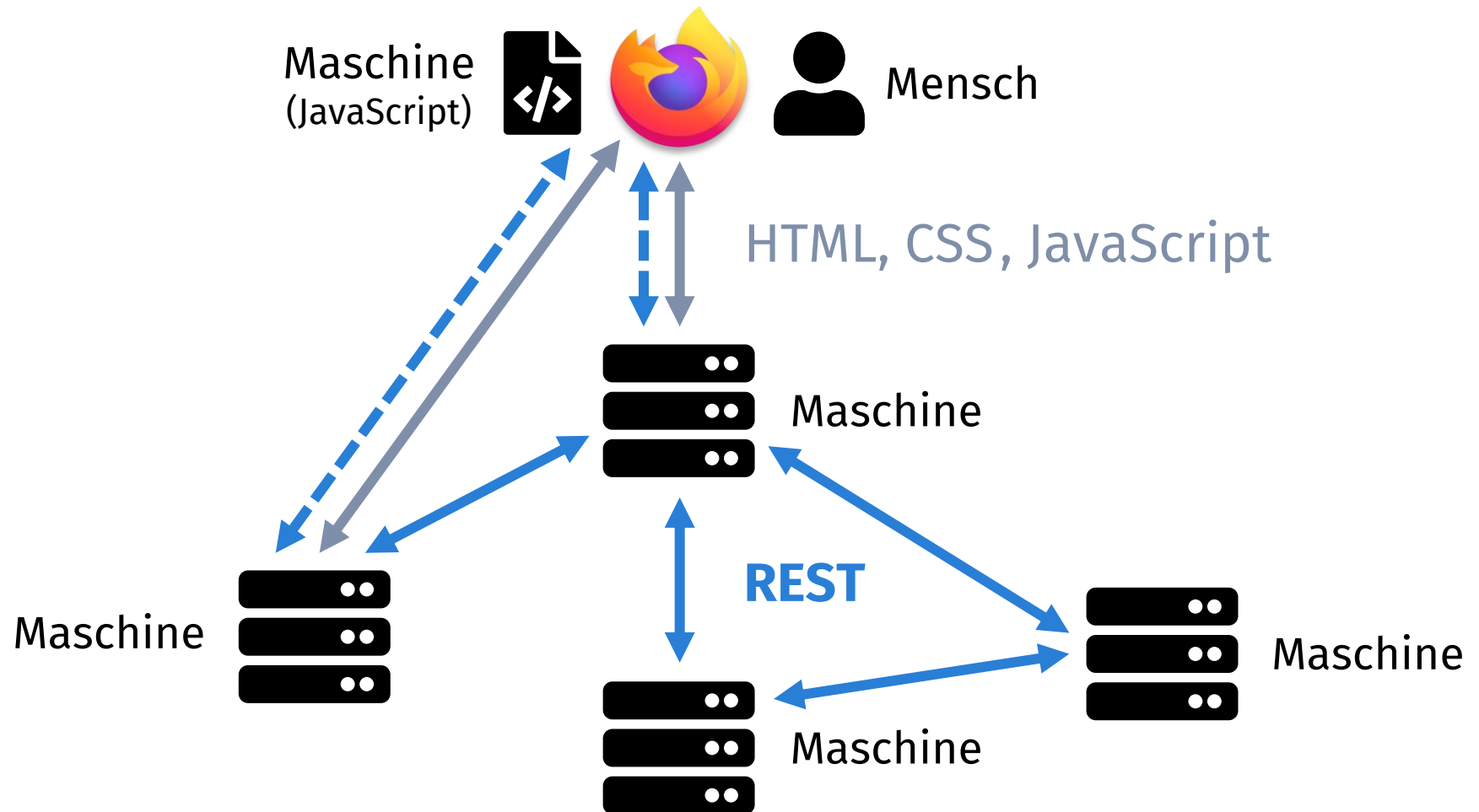
1. Starke Entkopplung von verschiedenen Teilen einer Applikation

- Jeder Service kann passende Technologie verwenden (Bibliotheken, Frameworks, DB, Sprache, Hardware, ...)
- **Einheitliche Schnittstellen**, Implementationsdetails von aussen versteckt
- Services können einzeln ersetzt werden
- Services werden von verschiedenen Teams entwickelt und betrieben
- Oder sogar von verschiedenen Organisationen!
<https://github.com/public-apis/public-apis>

2. Performance-Vorteile

- Jeder Service kann mittels **Load Balancing** separat skaliert werden
- **Effektives Caching**

Maschine-Mensch / Maschine-Maschine



REST = HTTP + JSON?

Nicht REST, Beispiel 1:

```
POST /api/contacts HTTP/1.1
Content-Type: application/json
```

```
{
  "method": "searchContacts",
  "args": {
    "searchTerm": "guppy",
    "maxResults": 1
  }
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "results": [
    {
      "firstName": "Mabel",
      "lastName": "Guppy",
      "email": [],
      "phone": [
        "405-580-6403"
      ]
    }
  ]
}
```

REST = HTTP + JSON?

Nicht REST, Beispiel 2:

```
POST /api/contacts HTTP/1.1
Content-Type: application/json
```

```
{
  "firstName": " Mabel",
  "lastName": "Guppy"
}
```

Zustand!



```
POST /api/contacts/last HTTP/1.1
Content-Type: application/json
```

```
{
  "company": "Livepath"
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "firstName": "Mabel",
  "lastName": "Guppy"
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
  "firstName": "Mabel",
  "lastName": "Guppy",
  "company": "Livepath"
}
```

REpresentational State Transfer

REST ist ein Architekturstil!

Eigenschaften/Einschränkungen:

Zustandslose Kommunikation

- Zustand wird jeweils «übertragen»

Caching von (gewissen) Antworten

Einheitliche Schnittstellen

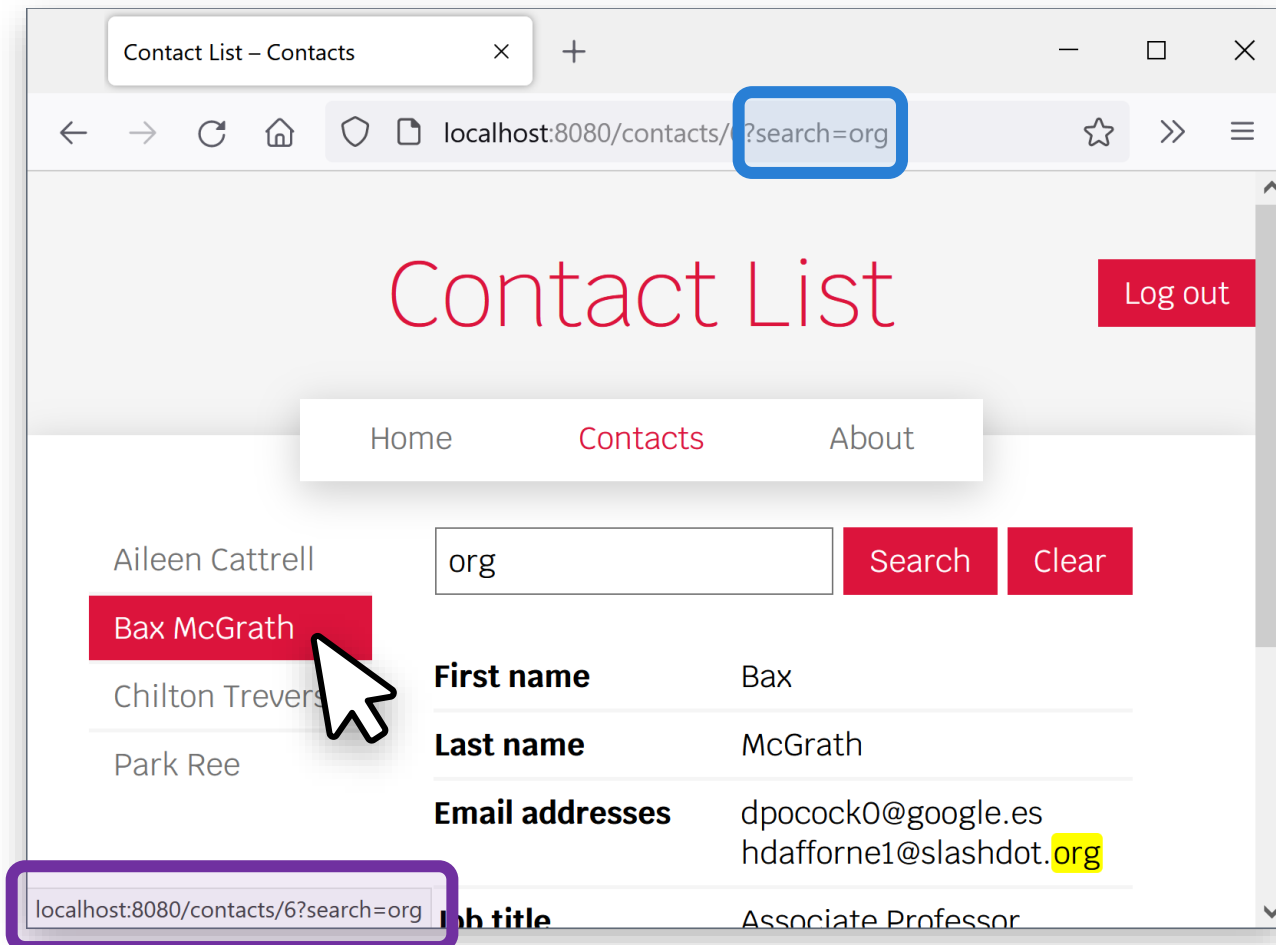
- Adressierbare *Ressourcen*
- Verschiedene *Repräsentationen* von Ressourcen möglich
- Selbstbeschreibende Nachrichten

Heute typisch: «HTTP + JSON»

- Ressourcen durch URLs adressiert
- Repräsentation durch *Medientyp* beschrieben (oft application/json)
- Standardisierte Operationen mit definierter Bedeutung:
HTTP-Verben (GET, POST, ...)

Zustandlose Kommunikation

Beispiel für zustandlose Kommunikation und Übertragung von Zustand: «persistente» Suche



1. Client startet Suche
→ schickt an Server
2. Server antwortet mit Resultaten und *Links, welche Suchzustand erhalten!*
3. Client folgt Link
→ schickt Zustand wieder an Server

REST mit HTTP + JSON: Beispiel

Abfragen eines Kontakts:

selbstbeschreibende
Standardmethode

Adresse der Ressource

```
GET /api/contacts/1 HTTP/1.1
Accept: application/json
```

Repräsentation
der Ressource

HTTP/1.1 200 OK

Content-Length: 183

Content-Type: application/json

```
{
  "firstName": "Mabel",
  "lastName": "Guppy",
  "email": [],
  "phone": [
    "405-580-6403"
  ],
  "jobTitle": "Librarian",
  "company": "Photolist"
}
```

(https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/JSON)

Übung 1: Zugriff auf REST-API

In der Vorlage dieser Woche findest du das Projekt «rest-clients» und darin das Programm `WeatherClient`. Dieses verwendet eine öffentliche REST-API zum Abrufen des aktuellen Wetterberichts.

- a) Studiere das Programm und dessen Ausgabe. Ändere auch mal den Ort und beobachte die Änderungen in der Ausgabe.
- b) Erweitere das Programm so, dass es weitere Wetterdaten abfragt und auf der Konsole ausgibt, z. B. Luftfeuchtigkeit und Windgeschwindigkeit. Ziehe die Dokumentation der API zur Rate: <https://open-meteo.com/en/docs>

HTTP-Statuscodes (Wiederholung)

| Codes | Kategorie | Beschreibung |
|------------|---------------|--|
| 1xx | Information | Anfrage dauert noch an... |
| 2xx | Erfolgreich | Anfrage wurde bearbeitet, Resultat kommt zurück (optional). Kann auch Location -Header enthalten. |
| 3xx | Umleitung | Weitere Schritte nötig, z. B. weil Ressource verschoben wurde. Enthält Location -Header. |
| 4xx | Client-Fehler | z. B. nicht-existierende Ressource (404) |
| 5xx | Server-Fehler | Verarbeitungsfehler auf dem Server |

HTTP-Verben (upgedated)

| Verb | Beschreibung | Safe | Idempotent |
|---------|--|------|------------|
| GET | Ruft eine «Ressource» vom Server ab | ✓ | ✓ |
| HEAD | Wie GET, aber Server schickt nur Header | ✓ | ✓ |
| POST | Erstellt neue Ressourcen oder überschreibt vorhandene | ✗ | ✗ |
| PUT | Erstellt/überschreibt die Ressource unter einer URL | ✗ | ✓ |
| PATCH | Ändert die Ressource unter einer URL | ✗ | ✗ |
| DELETE | Löscht die Ressource unter einer URL | ✗ | ✓ |
| OPTIONS | Fragt die möglichen Verben für einen Server oder eine URL ab | ✓ | ✓ |

Safe: ändert Zustand des Servers nicht

Idempotent: wiederholte Anfragen ändern Serverzustand **nicht weiter**

Beispiel: PUT

Erstellen/Überschreiben eines Kontakts (Client definiert Ort):

```
PUT /api/contacts/2 HTTP/1.1  
Content-Type: application/json
```

```
{  
  "firstName": "Lauree",  
  "lastName": "Clouter",  
  "email": [  
    "alyman0@economist.com"  
  ],  
  "phone": [],  
  "jobTitle": "Senior Editor",  
  "company": "Livepath"  
}
```

```
HTTP/1.1 200 OK
```

kein Body

Beispiel: POST

Erstellen eines neuen Kontakts (Server wählt Ort):

```
POST /api/contacts HTTP/1.1
Content-Type: application/json

{
  "firstName": "Lauree",
  "lastName": "Clouter",
  "email": [
    "alyman0@economist.com"
  ],
  "phone": [],
  "jobTitle": "Senior Editor",
  "company": "Livepath"
}
```

```
HTTP/1.1 201 Created
Location: /api/contacts/2
Content-Type: application/json
```

Ort der
erstellten
Ressource

Body *kann*
leer sein

REST mit Java & Spring

Data-binding für JSON

`JSONObject`-Bibliothek ist ok, aber für intensiven Gebrauch unhandlich.

Besser: Automatische Umwandlung von/zu Java-Objekten.

```
{
  "id": 2,
  "firstName": "Lauree",
  "lastName": "Clouter",
  "email": [
    "alyman0@economist.com"
  ],
  "phone": [],
  "jobTitle": "Senior Editor",
  "company": "Livepath"
}
```

Data-binding

```
public class Contact {

  private int id;
  private String firstName;
  private String lastName;
  private List<String> email;
  private List<String> phone;
  private String jobTitle;
  private String company;

  ...
}
```

«JPA für JSON»

Jackson-Databind

Bibliothek *Jackson-Databind* bietet Data-binding für primitive Typen, viele eingebaute Typen (`String`, `LocalDate`, etc.) und eigene Klassen:

```
var mapper = new ObjectMapper();  
Contact contact = ...  
String json = mapper.writeValueAsString(contact);  
System.out.println(json);
```

```
{"id":1,"firstName":"Mabel","lastName":"Guppy" ... }
```

```
String json = ...  
Contact contact = mapper.readValue(json, Contact.class);  
System.out.println(contact.getFirstName());
```



Mabel

Demo: WeatherClient2



Data-Binding konfigurieren

Jackson bietet eigene Annotationen zum Anpassen des Data-Binding.

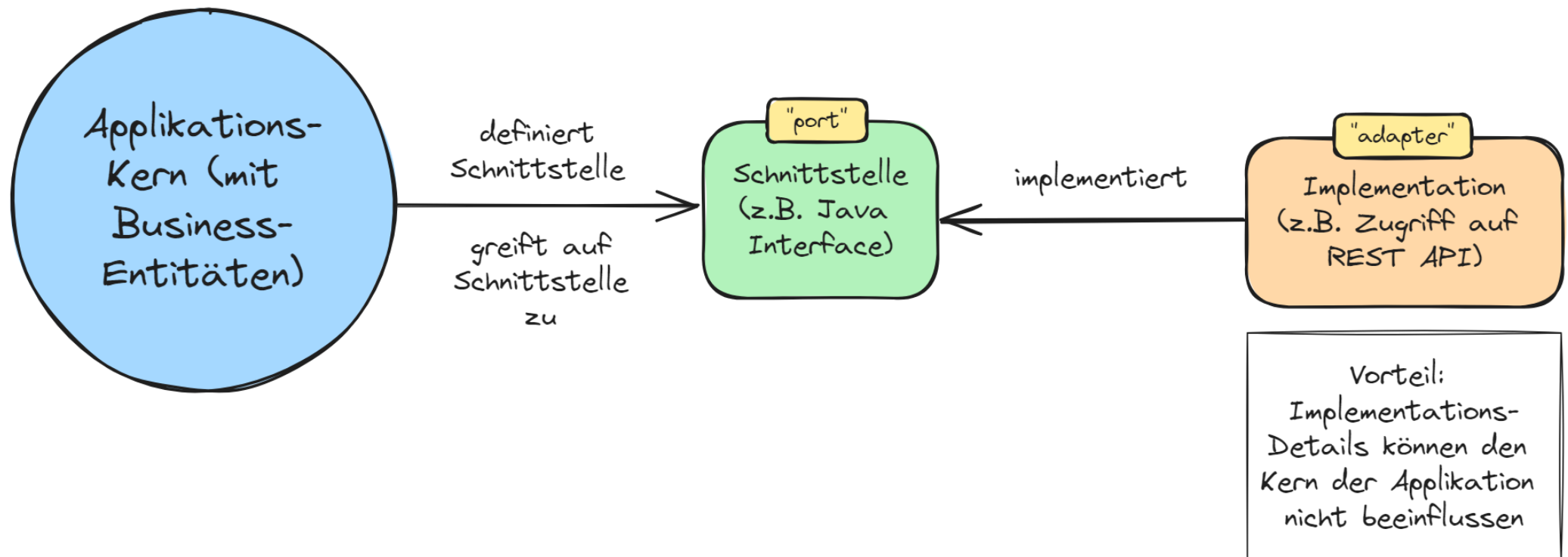
Beispiele: Entfernen der ID, Anpassen der Property-Namen:

```
public class Contact {  
  
    @JsonIgnore  
    private int id;  
  
    @JsonProperty("first-name")  
    private String firstName;  
  
    @JsonProperty("last-name")  
    private String lastName;  
  
    ...  
}
```

```
{  
    "first-name": "Lauree",  
    "last-name": "Clouter",  
    ...  
}
```

<https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations>

Die saubere Art, ein REST-API anzubinden



Umsetzung im Code: weatherclient3 im Ordner rest-clients

REST-APIs mit Spring Boot erstellen



Im Prinzip durch Controller abgedeckt! (Strings zurück geben...)

Spezielle Unterstützung für typisches Setup mit JSON (*Jackson*):

```
@RestController
@RequestMapping("/api/contacts")
public class ContactsRestController {

    @GetMapping
    public List<Contact> getAll() {
        return ...
    }

    @GetMapping("/{id}")
    public Contact get(@PathVariable long id) {
        return ...
    }
}
```



GET /api/contacts ...

GET /api/contacts/*2* ...

Übung 2: REST-Endpunkt mit Spring

- a) Erstelle einen ersten einfachen API-Endpunkt, welcher es erlaubt, unter [/api/contacts](#) die gesamte Liste von Kontakten abzurufen. Erstelle dazu einen Rest-Controller `ContactsRestController` und füge eine entsprechende Methode hinzu. Verwende den `ContactService` und erweitere ihn wie benötigt.
- b) Teste den Endpunkt einmal im Browser, einmal mittels `curl` oder IntelliJ-HTTP-Client und schliesslich noch mit dem Programm `ContactsClient` im Projekt «rest-clients».

Fragen?

