01 - WebDev: Client-Server-Interaktion und HTTP-Grundlagen

Einleitung

Dieses Kapitel behandelt die fundamentalen Konzepte der Webkommunikation, die als Grundlage für das Verständnis jeder Webanwendung dienen – unabhängig davon, ob es sich um Frontend-Entwicklung mit React oder Backend-Entwicklung mit Django handelt. Es beleuchtet die Kernmechanismen, die den Datenaustausch im Internet steuern, einschließlich der Interaktion zwischen Webbrowsern und Servern, der Struktur von Adressen wie URLs und der Bedeutung von Endpoints. Darüber hinaus wird eine grundlegende Einführung in die Prinzipien von REST gegeben, die für moderne Web-APIs von zentraler Bedeutung sind.

---

1. Das Client-Server-Modell

Die Funktionsweise des Internets basiert maßgeblich auf dem **Client-Server-Modell**. In diesem Modell übernehmen zwei Hauptakteure unterschiedliche Rollen:

1.1 Der Client

Der Client ist in der Regel ein Webbrowser (wie Chrome, Firefox oder Safari), kann aber auch eine mobile Anwendung oder ein anderes Programm sein, das Informationen anfordert. Der Client initiiert Anfragen.

1.2 Der Server

Ein Server ist ein spezialisierter Computer, der darauf ausgelegt ist, Anfragen von Clients zu empfangen, zu verarbeiten und entsprechende Antworten zurückzusenden. Er fungiert als Bereitsteller von Informationen oder Diensten.

Interaktion: Wenn eine Webseite aufgerufen wird, sendet der Browser des Nutzers (Client) eine Anfrage an einen Webserver. Der Server verarbeitet die Anfrage und liefert die angeforderten Daten, beispielsweise eine HTML-Seite, Bilder oder andere Medieninhalte, zurück an den Browser des Clients.

---

2. Das HTTP-Protokoll: Die Sprache des Webs

Das **HTTP-Protokoll (Hypertext Transfer Protocol)** bildet das grundlegende Regelwerk für die Kommunikation im Web. Es definiert, wie Clients und Server miteinander interagieren, basierend auf einem Anfrage-Antwort-Prinzip.

2.1 HTTP-Request (Die Anfrage)

Ein **HTTP-Request** ist die Nachricht, die ein Client an einen Server sendet, um eine bestimmte Aktion auszulösen. Ein Request besteht aus mehreren Komponenten:

2.1.1 Methode (Verb)

Diese definiert die Art der gewünschten Aktion auf dem Server. Die gängigsten Methoden umfassen:

- **GET:** Fordert Daten vom Server an (z.B. eine Webseite laden, ein Bild abrufen).
- **POST:** Sendet Daten an den Server, um eine neue Ressource zu erstellen (z.B. ein Formular absenden, einen neuen Beitrag erstellen).
- **PUT:** Sendet Daten an den Server, um eine bestehende Ressource vollständig zu aktualisieren.
- **DELETE:** Fordert den Server auf, eine bestimmte Ressource zu löschen.
- **PATCH:** Übermittlung von Daten zur teilweisen Aktualisierung einer bestehenden Ressource.
- **HEAD:** Anforderung der Header einer Antwort, ohne den eigentlichen Inhalt.

2.1.2 URL (Uniform Resource Locator)

Die Adresse der Ressource, auf die sich die Anfrage bezieht.

2.1.3 Header

Zusätzliche Metadaten über die Anfrage oder den Client (z.B. akzeptierte Datentypen, Authentifizierungsinformationen).

2.1.4 Body (optional)

Die eigentlichen Daten, die mit dem Request gesendet werden, relevant bei Methoden wie POST, PUT oder PATCH.

2.2 HTTP-Response (Die Antwort)

Nach der Verarbeitung eines Requests sendet der Server eine **HTTP-Response** zurück an den Client. Eine Antwort enthält:

2.2.1 Statuscode

Eine dreistellige Zahl, die das Ergebnis der Anfrage kennzeichnet. Beispiele sind:

- `200 OK`: Die Anfrage war erfolgreich.
- `201 Created`: Eine Ressource wurde erfolgreich erstellt.
- `204 No Content`: Die Anfrage war erfolgreich, es wird jedoch kein Inhalt zurückgegeben.
- `301 Moved Permanently`: Die Ressource wurde dauerhaft verschoben.
- `302 Found` (oder `302 Redirect`): Die Ressource ist temporär unter einer anderen URL verfügbar.
- `400 Bad Request`: Die Anfrage des Clients war fehlerhaft.
- `401 Unauthorized`: Die Anfrage erfordert eine Authentifizierung.
- `403 Forbidden`: Der Server verweigert die Bearbeitung der Anfrage.
- `404 Not Found`: Die angeforderte Ressource konnte nicht gefunden werden.
- `500 Internal Server Error`: Ein allgemeiner Fehler auf dem Server.

2.2.2 Header

Zusätzliche Informationen über die Antwort oder den Server (z.B. den gesendeten Datentyp).

2.2.3 Body (optional)

Der eigentliche Inhalt der Antwort, wie HTML-Code oder JSON-Daten.

---

3. URLs, Ports und Endpoints

Für den Zugriff auf Ressourcen im Web sind präzise Adressen erforderlich.

3.1 URLs (Uniform Resource Locators)

Eine **URL** ist die Adresse einer Ressource im Internet. Die grundlegende Struktur einer URL ist wie folgt:

`scheme://host:port/path?query#fragment`

- **Scheme (Protokoll):** Gibt das für die Kommunikation genutzte Protokoll an (z.B. `http` oder `https` für sichere Verbindungen).
- **Host (Hostname/Domain):** Der Domainname oder die IP-Adresse des Servers (z.B. `www.example.com`, `localhost`).
- **Port:** Eine optionale Nummer, die einen spezifischen Dienst auf dem Server identifiziert. Standard-Ports (z.B. 80 für HTTP, 443 für HTTPS) werden oft weggelassen. Im Rahmen dieses Kurses werden häufig Ports wie `8000` (für den Django-Entwicklungsserver) oder `3000` (für andere lokale Dienste) verwendet.
- **Path (Pfad):** Der spezifische Pfad zur Ressource auf dem Server (z.B. `/produkte/kategorie/`).
- **Query (Query-Parameter):** Optionale Schlüssel-Wert-Paare, die zusätzliche Daten an den Server übermitteln. Sie beginnen mit einem `?` und werden durch `&` getrennt (z.B. `?suche=laptop&sortierung=preis`). Diese sind oft für Filter- oder Suchfunktionen.
- **Fragment:** Ein optionaler Teil, der auf einen spezifischen Abschnitt innerhalb der Ressource verweist (oft bei HTML-Seiten, beginnend mit `#`).

Beispiel: Die URL `http://localhost:3000/json?value=200` zerlegt sich wie folgt:

- `http`: Scheme
- `localhost`: Host
- `3000`: Port
- `/json`: Path
- `?value=200`: Query-Teil mit dem Parameter `value` und dem Wert `200`.

3.2 Endpoints

Ein **Endpoint** ist ein spezifischer URL-Pfad auf einem Server, der für die Interaktion mit einer bestimmten Ressource oder Funktion eines Dienstes (API) definiert ist. Er stellt den Zielpunkt einer API-Anfrage dar.

Beispiele:

- `/api/recipes/` (um alle Rezepte abzurufen oder ein neues Rezept zu erstellen)
- `/api/recipes/123/` (um ein spezifisches Rezept mit der ID 123 abzurufen, zu aktualisieren oder zu löschen)
- `/api/users/login/` (für die Anmeldung eines Benutzers)

---

4. REST-Konzepte (Grundlagen)

REST (Representational State Transfer) ist ein Architekturstil für verteilte Systeme, insbesondere für Webdienste. Es handelt sich hierbei nicht um eine Technologie, sondern um eine Sammlung von Prinzipien, die die Kommunikation zwischen verschiedenen Systemen standardisieren sollen.

Die wesentlichen Prinzipien von REST, die hier grundlegend behandelt werden, umfassen:

- **Ressourcenorientierung:** Jede Komponente oder Datenmenge wird als Ressource betrachtet (z.B. ein Rezept, ein Benutzer, ein Kommentar), die über eine eindeutige URL identifizierbar ist.
- **Nutzung von Standard-HTTP-Methoden:** RESTful APIs verwenden die etablierten HTTP-Methoden (GET, POST, PUT, DELETE, PATCH) für Operationen auf diesen Ressourcen.
  - **GET** `/recipes/`: Ruft alle Rezepte ab.
  - **GET** `/recipes/{id}/`: Ruft ein spezifisches Rezept ab.
  - **POST** `/recipes/`: Erstellt ein neues Rezept.
  - **PUT** `/recipes/{id}/`: Aktualisiert ein spezifisches Rezept vollständig.
  - **DELETE** `/recipes/{id}/`: Löscht ein spezifisches Rezept.
- **Statelessness (Zustandslosigkeit):** Jede Anfrage vom Client an den Server muss alle Informationen enthalten, die für die Verarbeitung dieser Anfrage notwendig sind. Der Server speichert keine clientseitigen Sitzungsinformationen zwischen den Anfragen; jede Anfrage wird unabhängig von vorherigen behandelt.
- **Client-Server-Trennung:** Client und Server agieren unabhängig voneinander. Der Client ist nicht über die interne Logik des Servers informiert, und der Server hat keine Kenntnis über die Benutzeroberfläche des Clients.
- **Uniform Interface:** Ein standardisierter Interaktionsweg zwischen Clients und Servern, unabhängig von deren spezifischer Implementierung.

RESTful APIs bilden die Basis für die Kommunikation zwischen einem Django-Backend (das Daten bereitstellt) und einem React-Frontend (das diese Daten zur Darstellung der Benutzeroberfläche nutzt).

Cheat Sheet: Skript 1.1 - Web Development Grundlagen

HTTP-Methoden (Verbs)

- `GET`: Daten vom Server anfordern (Lesen).
- `POST`: Daten an den Server senden (Erstellen).
- `PUT`: Vorhandene Ressource vollständig aktualisieren.
- `DELETE`: Ressource löschen.
- `PATCH`: Vorhandene Ressource teilweise aktualisieren.

Wichtige HTTP-Statuscodes

- `200 OK`: Erfolgreich.
- `201 Created`: Ressource erfolgreich erstellt.
- `400 Bad Request`: Client-Fehler in der Anfrage.
- `401 Unauthorized`: Authentifizierung erforderlich.
- `403 Forbidden`: Kein Zugriff erlaubt.
- `404 Not Found`: Ressource nicht gefunden.
- `500 Internal Server Error`: Server-Fehler.

URL-Struktur

`scheme://host:port/path?query#fragment`

- `scheme`: Protokoll (z.B. `http`, `https`)
- `host`: Serveradresse (z.B. `www.example.com`, `localhost`)
- `port`: Dienstnummer (optional, z.B. `8000` für Django Dev-Server)
- `path`: Pfad zur Ressource (z.B. `/api/data/`)
- `query`: Schlüssel-Wert-Paare für zusätzliche Parameter (z.B. `?id=1&sort=asc`)

Konzepte

- **Client-Server-Modell:** Die Interaktion zwischen einem Anfrager (Client) und einem Lieferanten (Server).
- **HTTP-Protokoll:** Das Regelwerk für die Kommunikation im Web.
- **HTTP-Request:** Die vom Client an den Server gesendete Anfrage.
- **HTTP-Response:** Die Antwort des Servers auf eine Client-Anfrage.
- **Endpoint:** Ein spezifischer URL-Pfad für die Interaktion mit einer API.
- **REST (Representational State Transfer):** Ein Architekturstil für Webdienste, der Standard-HTTP-Methoden auf Ressourcen anwendet und auf Zustandslosigkeit basiert.