

WEBTECHNOLOGIEN

04 – NODE.JS UND DATENBANKEN, MODEL-VIEW- CONTROLLER

PROF. DR. MARKUS HECKNER

AGENDA

Datenbanken und Web-Apps

Model-View-Controller

URL-Parameter

WARUM DATENBANKEN

- Webanwendungen verarbeiten oft strukturierte Massendaten (Profile, Likes, Posts, Playlisten, Kommentare, etc.)
- Die Daten müssen aus der Datenbank in die Webanwendung und die Webanwendung muss die Daten einfügen, aktualisieren und löschen können (CRUD – Create Read Update Delete)
- Zur Verwaltung dieser Daten stehen zahlreiche Datenbankmanagementsysteme (DBMS) zur Verfügung, z.B.
 - MySQL
 - Oracle 12c
 - **PostgreSQL**
 - SQLite
 - MongoDB (nicht-relational)
 - ...

DATENBANKEN – HEUTE SQL MIT POSTGRES SQL



- Relationales DBMS
- Open Source
- Packages für den Zugriff über Node.js verfügbar
- Kostenlose Datenbank bei Heroku verfügbar

IN DIESER SESSION NUR EINE TABELLE

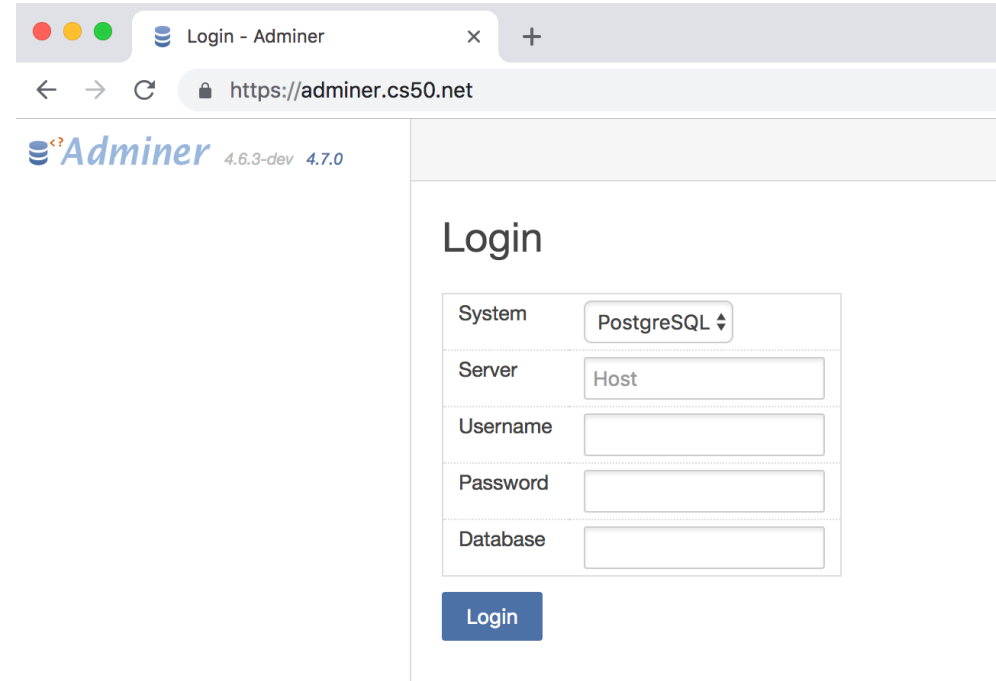
```
CREATE TABLE dbdemo_playlists (  
    ID SERIAL PRIMARY KEY,  
    TITLE VARCHAR  
);
```

```
INSERT INTO dbdemo_playlists (TITLE) VALUES ('Happy Mood');  
INSERT INTO dbdemo_playlists (TITLE) VALUES ('Iconic songs');
```

ZUGRIFF AUF DIE POSTGRESQL DATENBANK MIT ADMINER

- Weboberfläche zum Management von Datenbanken
- Direkte Eingabe von SQL-Queries möglich
- Zugangsdaten zur Datenbank von Heroku (vgl. Lab)

<https://adminer.cs50.net/>



The screenshot shows a web browser window with the title "Login - Adminer" and the URL "https://adminer.cs50.net". The Adminer logo and version "4.6.3-dev 4.7.0" are visible. The login form is titled "Login" and contains the following fields:

System	PostgreSQL ▾
Server	Host
Username	<input type="text"/>
Password	<input type="password"/>
Database	<input type="text"/>

Below the form is a blue "Login" button.

DB DEMO – ÜBERSICHT ALLER PLAYLISTEN

[Playlist 1](#) [Dashboard](#) [About](#)

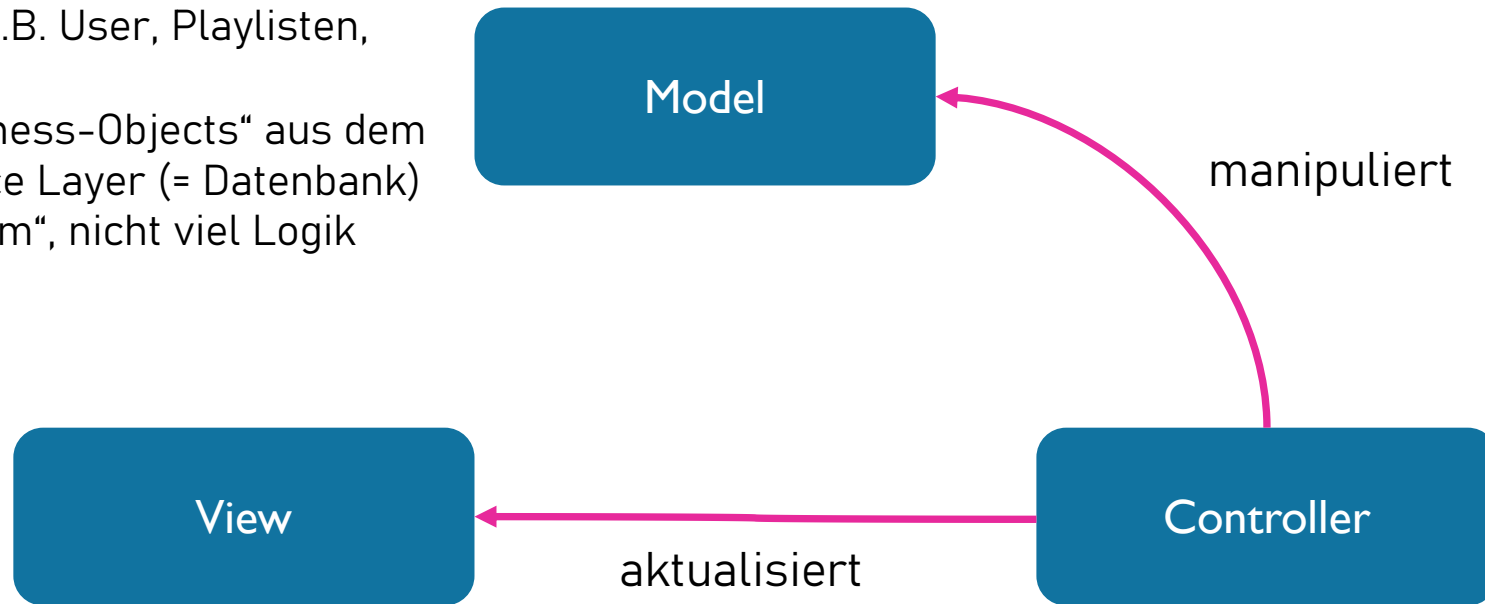
Dashboard

Happy Mood

Iconic songs

MODEL VIEW CONTROLLER IST EIN WEIT VERBREITETES SOFTWARE-ENTWURFSMUSTER

- Persistiert Daten aus „Business-Objects“ (z.B. User, Playlisten, Songs, ...)
- Holt „Business-Objects“ aus dem Persistence Layer (= Datenbank)
- Eher „dumm“, nicht viel Logik



- Holt Daten aus den Models
- Speichert Daten in den Models
- Fügt Daten in die Views ein und senden Daten diese an Client zurück

DAS MODEL STEHT DEN CONTROLLERN ZUR VERFÜGUNG UM DATEN ABZURUFEN UND ZU SPEICHERN

data-store.js

```
const datastore = require("../data-store.js");
const datastoreClient = datastore.getDataStore();
const logger = require("../utils/logger.js");

const playlistStore = {
  async getAllPlaylists() {
    const query = 'SELECT * FROM playlist2_playlists';
    try {
      let result = await datastoreClient.query(query);
      return result.rows;
    } catch (e) {
      logger.error("Error fetching all playlists", e);
    }
  },
};

module.exports = playlistStore;
```

} Model benötigt Zugriff auf die Datenquelle
(hier: PostgreSQL) – data-store.js stellt diesen Zugriff her
– Alle Models nutzen data-store.js (mehr später)

Objekt `playlistStore` ist das Model und bietet
über seine Methoden Zugriff auf die Daten

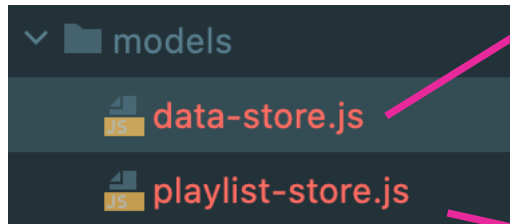
SQL-Query-String

Absenden des Queries mithilfe des `dataStoreClient`-
Objekts – Rückgabe ist Objekt `result` mit einem
Array `rows` (= Ergebnis der Abfrage)

Rückgabe des Ergebnisses an
den Controller

Error Handling – Logging des
DB Fehlers

ZUSAMMENSPIEL ZWISCHEN DATA-STORE.JS UND DEM MODEL PLAYLIST-STORE.JS



Model benötigt Zugriff auf die Datenquelle
(hier: PostgreSQL) – data-store.js stellt diesen Zugriff her
– Alle Models nutzen data-store.js (mehr später)

Model zur Abfrage und Ablegen von Daten zur Playlist

ASYNC UND AWAIT ERLEICHTERN ASYNCHRONE PROGRAMMIERUNG

```
const datastore = require("../data-store.js");
const datastoreClient = datastore.getDataStore();
const logger = require("../utils/logger.js");

const playlistStore = {
  async getAllPlaylists() {
    const query = 'SELECT * FROM playlist2_playlists';
    try {
      let result = await datastoreClient.query(query);
      return result.rows;
    } catch (e) {
      logger.error("Error fetching all playlists", e);
    }
  },
};

module.exports = playlistStore;
```

`getAllPlaylists` enthält ein `await`, d.h. das Keyword `async` muss vor der Funktionsdefinition hinzugefügt werden!

Methode `query` ist asynchron, d.h. gibt nicht sofort ein Ergebnis zurück – Aber mittels `await` verhält sich der Code synchron – d.h. bis das Ergebnis von `query` zurückgegeben wird kann der Node Server andere Dinge tun, aber die Ausführung in `getAllPlaylists` geht erst weiter, wenn `query` ein Ergebnis zurückliefert!

!

Asynchrone Programmierung mit JS ist eine größere Baustelle (Stichworte Callbacks, Promises) – Für diesen Kurs reicht die Verwendung von `async` / `await`!

DATA-STORE.JS STELLT VERBINDUNG ZUR DB HER

- Stellt mittels eines DB-Connection-Strings die Verbindung zur DB her
- Ist eine Schnittstelle zur Datenbank
- Hier PostgreSQL, beliebige andere relationale und nicht-relationale DBMS integrierbar

```
let pg = require("pg");
const logger = require("../utils/logger.js");
const conString = process.env.DB_CON_STRING;

const dbConfig = {
  connectionString: conString,
  ssl: { rejectUnauthorized: false }
}

if (conString == undefined) {
  logger.error("ERROR: environment variable DB_CON_STRING not set.");
  process.exit( code: 1);
}

let dbClient = null;

const datastore = {
  getDataStore() {
    if (dbClient !== null) {
      return dbClient;
    } else {
      dbClient = new pg.Client(dbConfig);
      dbClient.connect();
      return dbClient;
    }
  },
  async endConnection() {
    await dbClient.end();
  }
}
```

DATA-STORE.JS (1/2)

Liest String für Zugriff auf die Datenbank aus der Datei .env (bzw. in replit aus den Secrets)

```
let pg = require("pg");
const logger = require("../utils/logger.js");
const conString = process.env.DB_CON_STRING;

const dbConfig = {
  connectionString: conString,
  ssl: { rejectUnauthorized: false }
}

if (conString == undefined) {
  logger.error("ERROR: environment variable DB_CON_STRING not set.");
  process.exit( code: 1);
}
```

Config

Error-Handling

DATA-STORE.JS (2/2)

Gibt dbClient für das Absenden von Anfragen an die Models zurück

```
let dbClient = null;

const datastore = {
  getDataStore() {
    if (dbClient !== null) {
      return dbClient;
    } else {
      dbClient = new pg.Client(dbConfig);
      dbClient.connect();
      return dbClient;
    }
  },
  async endConnection() {
    await dbClient.end();
  }
}
```

Mehr ist erstmal nicht wichtig!

AGENDA

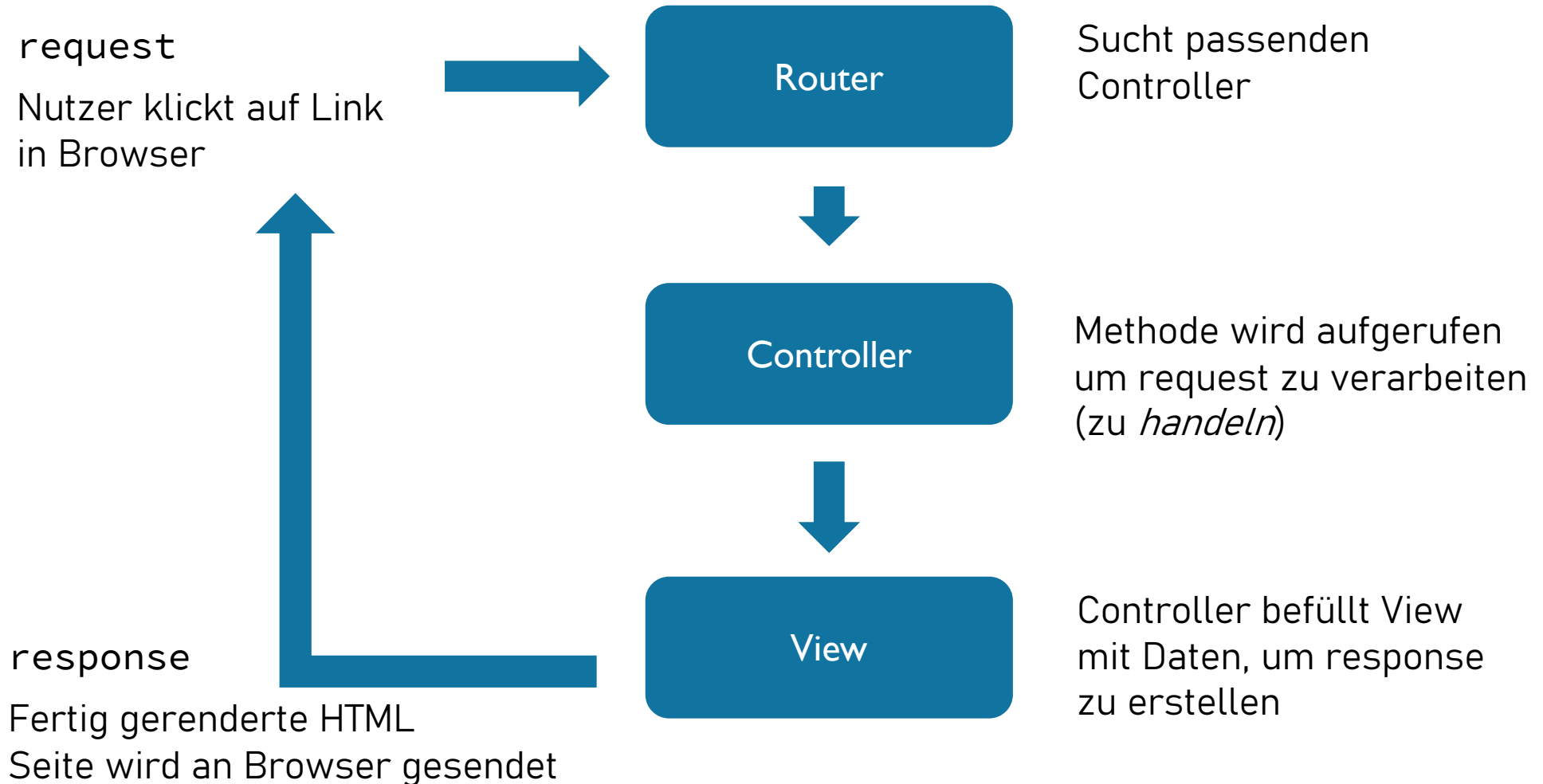
Datenbanken und Web-Apps

Model-View-Controller

URL-Parameter

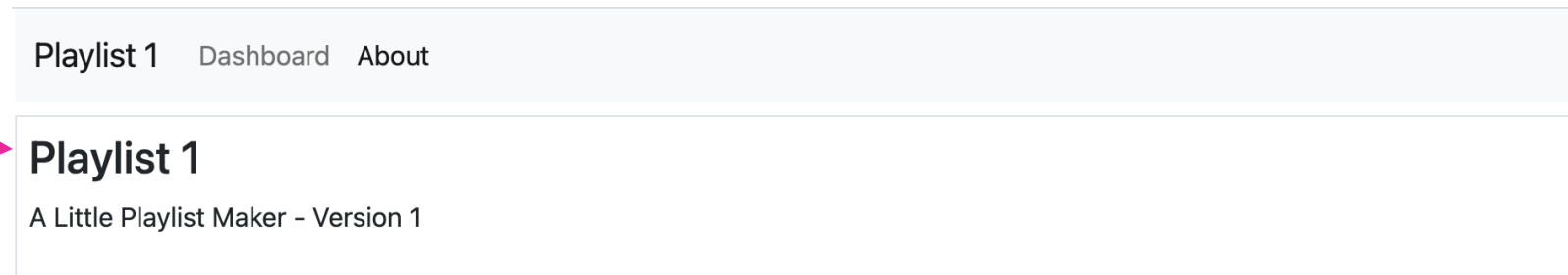
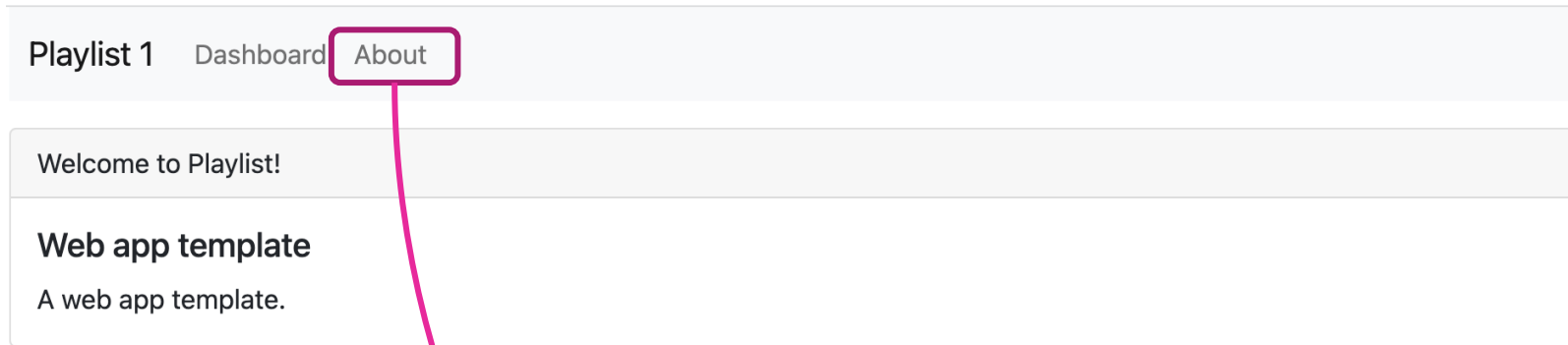
ROUTER CONTROLLER VIEW VS. ROUTER CONTROLLER MODEL VIEW

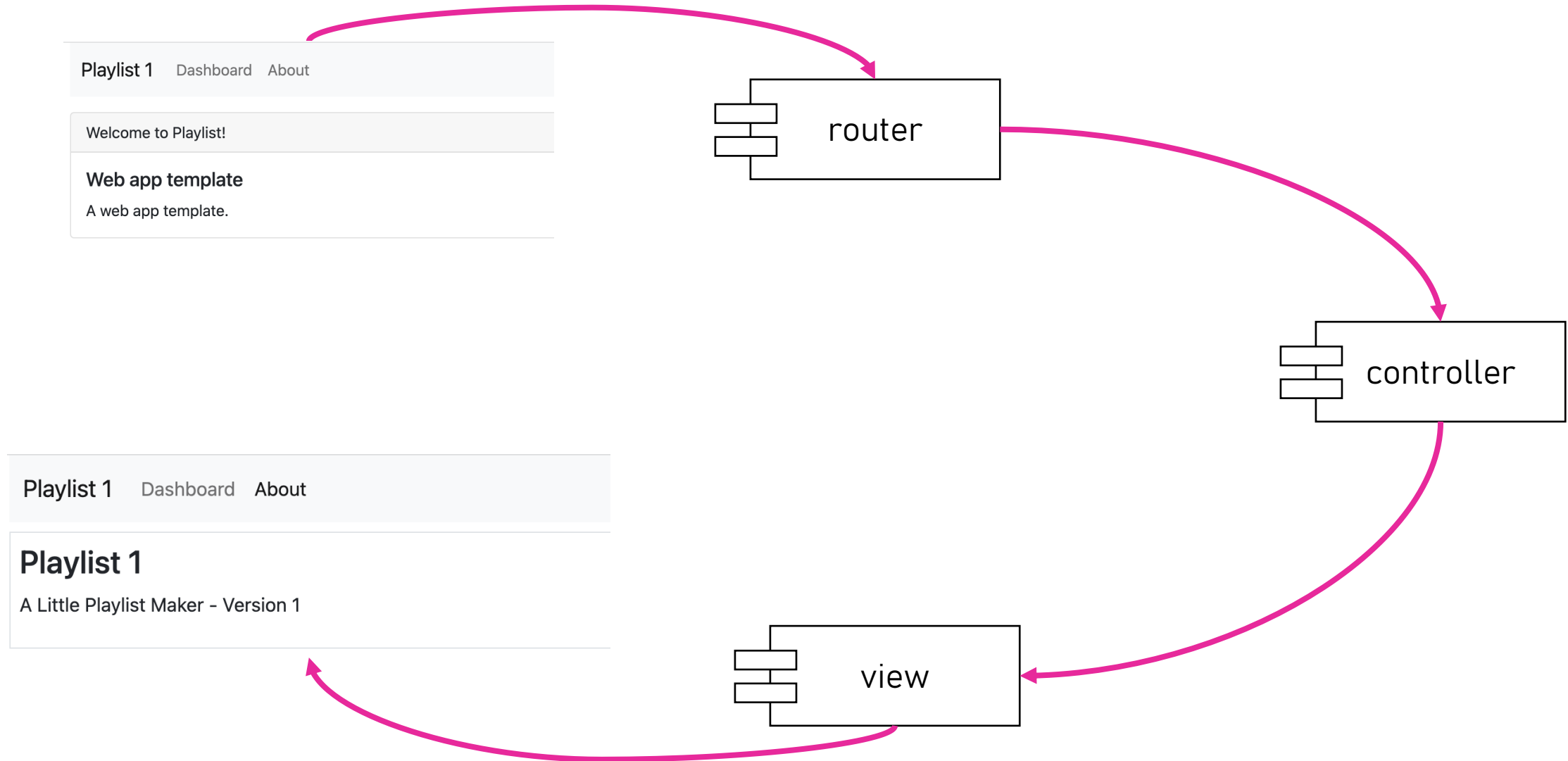
WIEDERHOLUNG: DER WEG VOM ROUTER ÜBER DEN CONTROLLER ZUM VIEW

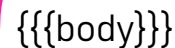
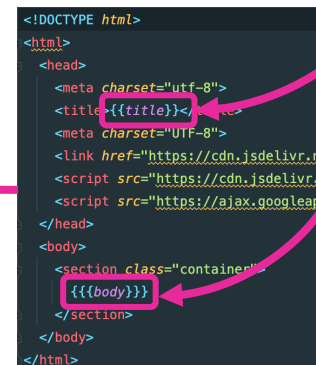
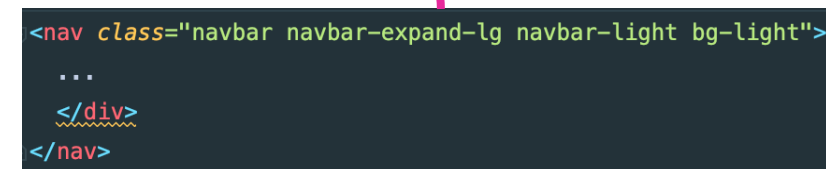
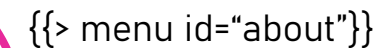
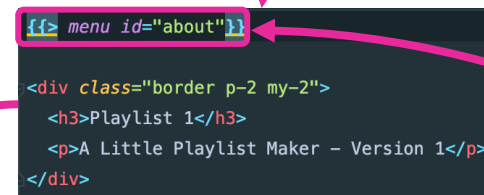
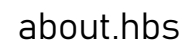
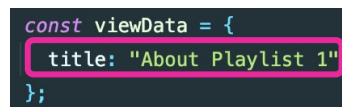
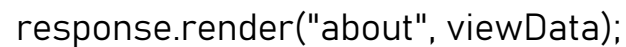
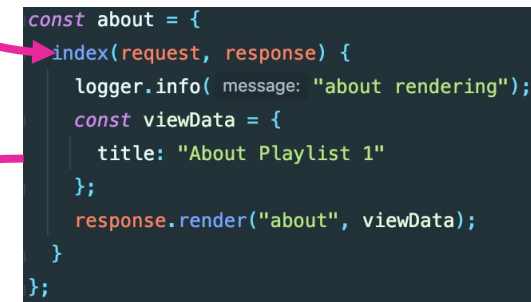
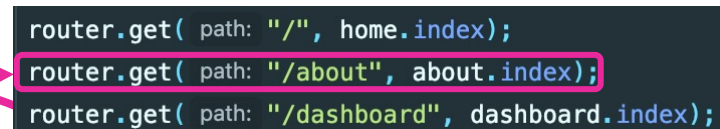
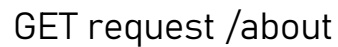


ROUTER CONTROLLER VIEW

MVC

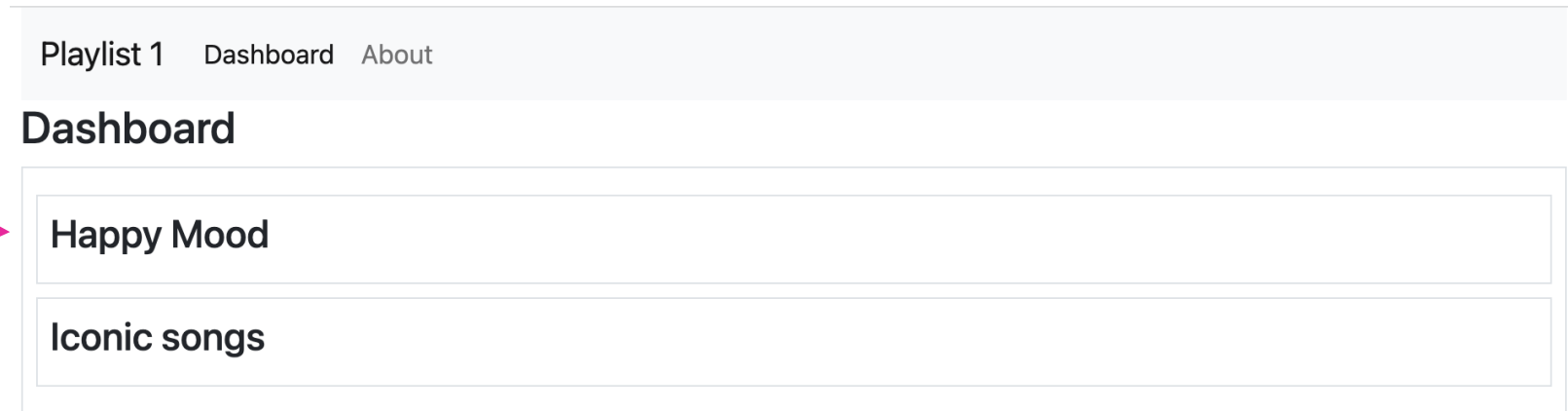
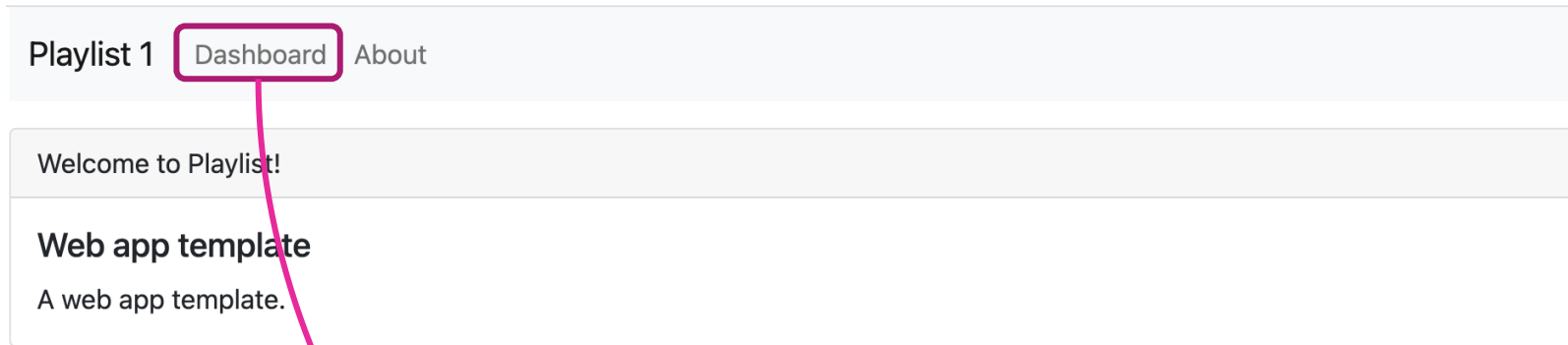




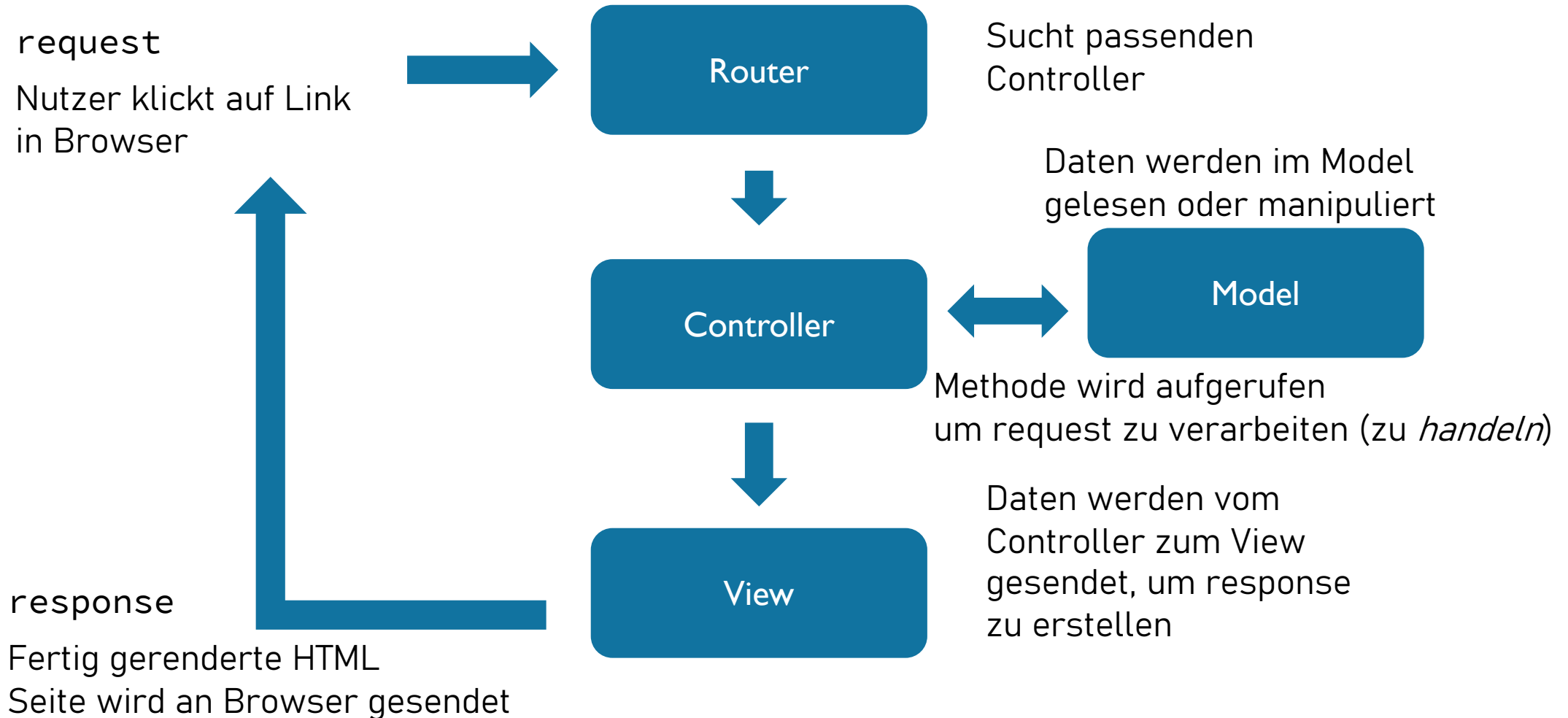


ROUTER CONTROLLER MODEL VIEW

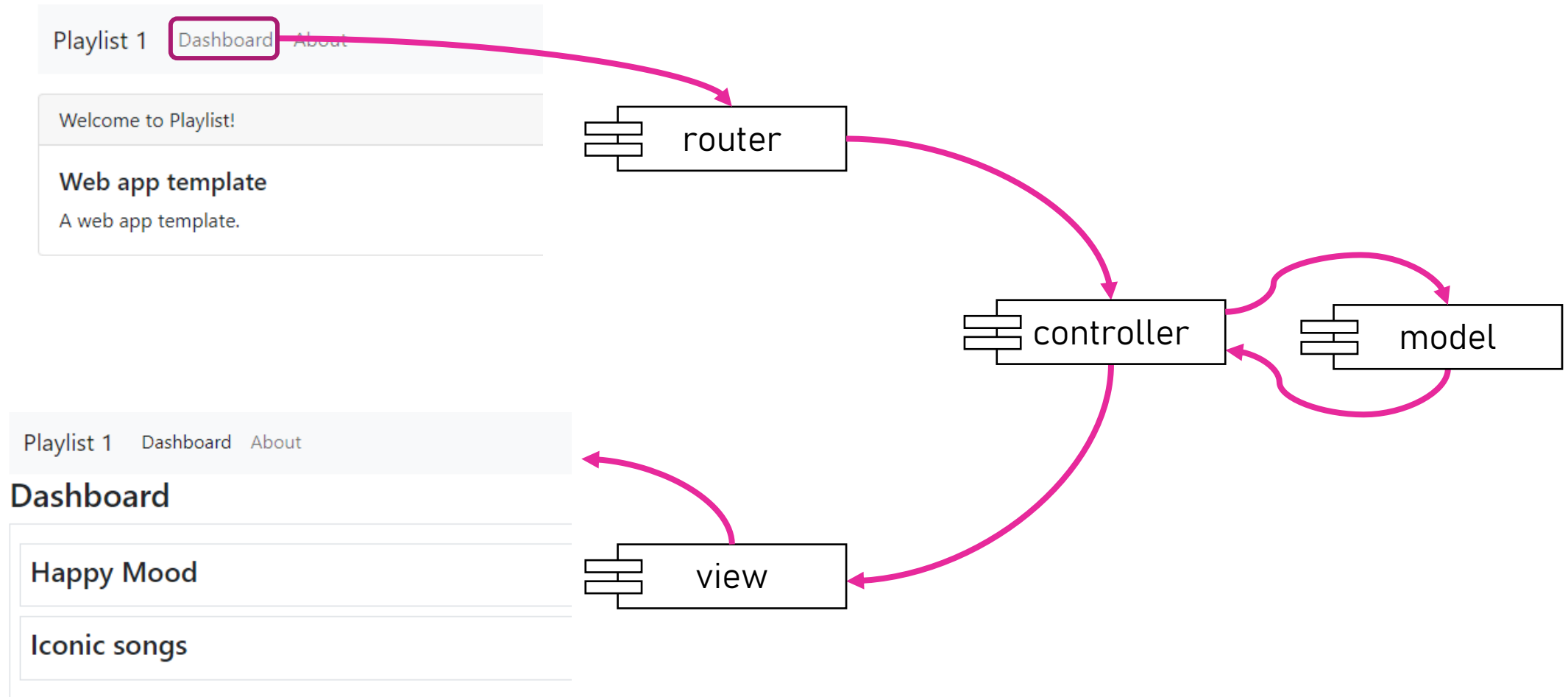
MVC

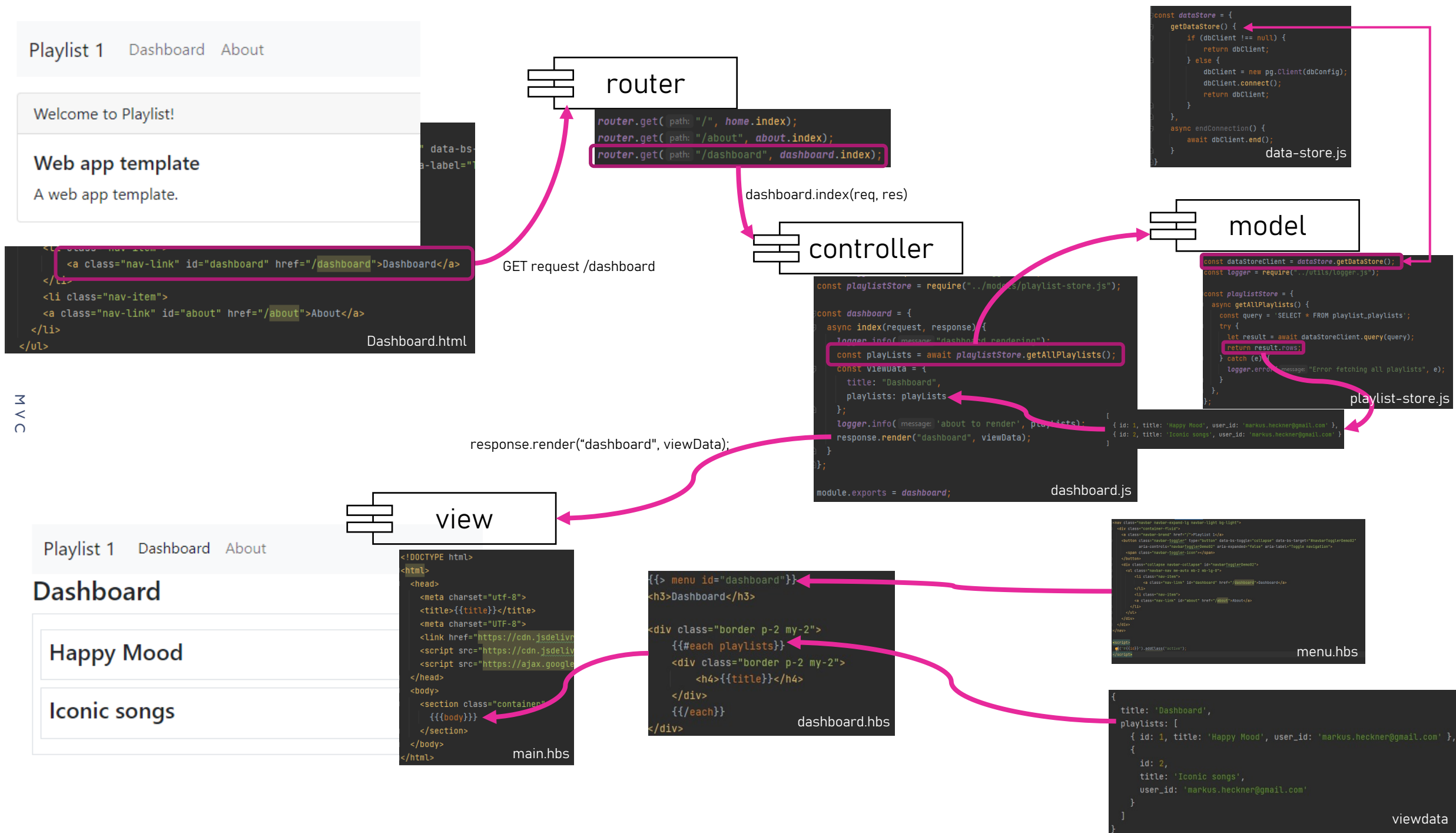


DER WEG VOM ROUTER ÜBER DEN CONTROLLER UND DAS MODEL ZUM VIEW



ROUTER CONTROLLER MODEL VIEW





CONTROLLERS UND MODELS IM DETAIL

Controller

- Funktionen werden vom Router aufgerufen (Nutzer greift auf eine Seite zu) oder von anderen Controllern (z.B. Controller zeigt Dashboard nach Login (später))
- Holen Daten aus den Models
- Speichern Daten in den Models
- Verwenden utils (falls nötig – später)
- Fügen Daten in die Views ein und senden Daten diese an Client zurück

Models

- Persistieren Daten aus „Business-Objects“ (z.B. User, Playlisten, Songs, ...)
- Holen „Business-Objects“ aus dem Persistence Layer (= Datenbank)
- Eher „dumm“, nicht viel Logik

Utils

- Verarbeiten Daten aus den Models (z.B. für Analytics, Konvertierungen)

AGENDA

Datenbanken und Web-Apps

Model-View-Controller

URL-Parameter

TYPISCHE URL FÜR DAS ANZEIGEN ALLER OBJEKTE EINES BESTIMMTEN TYP

- URL enthält den Namen des Objekts (eigtl. playlists)
- Keine weiteren Bestandteile in der URL

/dashboard

oder alternativ:

/playlist

Playlist 1 Dashboard About

Dashboard

Happy Mood

[View](#)





Iconic songs

[View](#)

TYPISCHE URL FÜR DIE ABFRAGE EINES BESTIMMTEN OBJEKTS

- URL enthält den Namen des Objekts (eigtl. playlists)
- ID (Primärschlüssel) des Objekts als Teil der URL
- ID kann im Controller ausgelesen werden und an das Model weitergegeben werden für Abfragen, Löschen, Manipulieren

/playlist/1

Playlist 1 Dashboard About		
Happy Mood		
Song	Artist	
Valerie	Amy Winehouse	
22	Taylor Swift	
Happy	Pharrell Williams	
a	a	

Router:

```
router.get('/playlist/:id', playlist.index);
```

Controller:

```
const playlistId = request.params.id;
```

URLS KÖNNEN AUCH MEHRERE PARAMETER ENTHALTEN

- URL enthält den Namen des Objekts (eigtl. playlists)
- Hier: Löschen des Songs mit der ID 2 aus der Playlist mit der ID 1

`/playlist/1/deletesong/2`

Router:

```
router.get('/playlist/:id/deletesong/:songid', playlist.deleteSong);
```

Controller:

```
const playlistId = request.params.id;  
const songId = request.params.songid;
```

FAZIT

- Datenbanken ermöglichen in Verbindung mit Webanwendungen die Verwaltung von größeren Datenmengen (z.B. Nutzer, Songs, Playlisten, Posts, Wetterstationen, Wettermesswerte, etc.)
- Wir verwenden PostgreSQL, im Prinzip lässt sich nahezu jede relationale und nichtrelationale Datenbank mit Node.js und Express einsetzen
- Models speichern Daten und rufen Daten ab, Controller verwenden die Models um Daten zu speichern und abzurufen, Views sind das Frontend der Web-App und werden vom Controller mit Daten befüllt und an den Client zurückgeschickt
- In der URL lassen sich Parameter integrieren, die von der Web-App ausgelesen werden können (vgl. z.B. `playlist/1` zum Abruf der Playlist mit der ID 1)