

Tværfagligt projekt

Event finder – Produktrapport

Alexander Jensby Thomsen
16-01-2025

Indhold

Case beskrivelse	2
Problemformulering	2
Kravspecifikation	3
Testbehov	5
Produktarkitektur	6
Produktkonfiguration og – udarbejdelse	7
Database interrogation	7
Server integration	9
Repositorylaget – kommunikation med databasen	9
Servicelaget – appen's forretningslogik og bruger sikkerhed	10
Routinglaget – Kald og opsætning af API'er	11

Case beskrivelse

Sociale arrangementer er en vigtig del af kulturen og spiller en central rolle i at samle mennesker. For mange kan det dog være svært at opdage nye og nærtliggende arrangementer, både store og små. Mange eksisterende platforme tilbyder brede løsninger, der fokuserer på at forbinde mennesker på tværs af verden, men mangler et lokalt perspektiv. Dette skaber et behov for en løsning, der gør det muligt at begrænse mængden af begivenheder inden for en bestemt afstand og baseret på individuelle interesser.

Ved at udnytte geografisk placering kan brugerne lettere finde relevante og nærtliggende arrangementer, som de ellers måske ikke ville have opdaget. Dermed adresseres både behovet for arrangørerne, der ønsker at nå ud til flere, og de individuelle brugere, der søger nye sociale oplevelser og ønsker at bringe mennesker tættere sammen.

Appen vil udnytte eksterne API'er til at hente data om aktuelle arrangementer og kombinere dette med brugernes geografiske placering for at levere målrettede anbefalinger. Ved hjælp af en relationsdatabase gemmes brugerpræferencer og historik. Backend vil være designet til at håndtere forespørgsler effektivt og levere realtidsopdateringer

Problemformulering

Hvordan kan en digital løsning designes og implementeres, så den effektivt forbinder brugere med relevante sociale arrangementer i deres nærområde baseret på geografisk placering og individuelle interesser?

For at besvare dette spørgsmål vil jeg undersøge, hvordan nuværende teknologier som eksterne API'er og geolokation kan udnyttes til at levere målrettede anbefalinger. Jeg vil vurdere, hvordan en backend kan designes til at håndtere forespørgsler effektivt og levere realtidsopdateringer om aktuelle arrangementer. Jeg vil også se på, hvordan løsningen kan tilgodese både brugernes behov for at opdage nye oplevelser og arrangørernes ønske om at nå ud til flere mennesker.

Perspektiveringen vil undersøge, hvordan en sådan applikation kan sammenlignes med eksisterende platforme og deres globale tilgang, og hvordan det lokale fokus kan skabe merværdi i sociale og kulturelle sammenhænge.

Kravspecifikation

Denne kravspecifikation beskriver de nødvendige funktioner og kvaliteter for appen og er opdelt i to hovedkategorier: funktionelle krav og ikke-funktionelle krav. Denne opdeling sikrer en klar differentiering mellem, hvad appen skal kunne (funktionelle krav), og hvordan den skal opføre sig (ikke-funktionelle krav).

For at skabe en overskuelig struktur og let identificere de enkelte krav anvendes en ID-struktur, hvor:

Funktionelle krav er markeret som nF (f.eks. 1F, 2F, osv.).

Ikke-funktionelle krav er markeret som nI (f.eks. 1I, 2I, osv.).

Database-funktionelle krav er markeret som nD (f.eks. 1D, 2D, osv.).

Denne tilgang hjælper med at sikre, at appen opfylder både tekniske behov, brugerinteraktioner, ydeevne og sikkerhedskrav. Tabellen med kravene inkluderer også prioriteringer og kommentarer, der skal vejlede udviklingsprocessen og sikre fokus på de vigtigste funktioner først.

ID	Kategori	Krav	Prioritet	Kommentar
1F	Funktionelle krav	Appen skal kunne vise en liste over events baseret på brugerens lokation	Høj	Kræver tilladelse af brugerens placering
2F	Funktionelle krav	Brugeren skal kunne filtrere events baseret på kategori, dato og afstand	Høj	
3F	Ikke-funktionelle krav	Appen skal vise events på et interaktivt kort	Mellem	Brug af Google Maps API eller Mapbox. Et mere nice to have krav.
1I	Ikke-funktionelle krav	Appen skal fungere offline ved hjælp af caching af seneste data.	Høj	Implementér Service Workers til offline-tilstand.
4F	Funktionelle krav	API'et skal overholde CRUD	Høj	CRUD for Create-Read-Update-Delete
1D	Database-krav	Databasen skal kunne gemme informationer om events, tidspunkt, lokation og arrangør	Høj	

5F	Funktionelle krav	API'et skal kunne have en UI hvor man kan vise data	Høj	Swagger er et godt valg
6F	Funktionelle krav	Ved brug af Service Worker, skal gemte data fra et event gemmes i cachen.	Høj	
2I	Ikke-funktionelle krav	Brugeren skal kunne se informationer om sidste sete events mens offline.	Høj	Brug Service Worker, og ligger op til 6F
3I	Ikke-funktionelle krav	Der vises en liste over events indenfor en givet radius	Høj	Listen er vigtigere end kortet.
2D	Database-krav	Der skal gemmes informationer om events, tidspunkt, lokation og arrangør	Høj	
3D	Database-krav	Der gemmes information om brugeren, senest lokation	Høj	
7F	Funktionelle krav	Opret route decorators for API'et	Høj	I .NET bruger man Controllers, her bruger man route decorators.

Testbehov

Formålet med acceptancetesten er at verificere, at applikationen opfylder kravene specificeret i kravspecifikationen. Hvert krav testes individuelt med fokus på funktionalitet, performance og brugervenlighed.

Acceptance test ID	Krav ID	Type	Betingelse	Hvordan det testes
T1	1F	Acceptance	En bruger kan få vist informationer fra begivenheder igennem appen	Brugeren får vist mulige begivenheder indenfor en radius, og kan derfra få vist informationerne
T2	2F	Acceptance	En bruger kan ændre i indstillingerne for søgebehovet, efter brugerens eget behov	Brugeren får mulighed for at kunne ændre radius samt nøgleord for event typer i appen.
T3	2I	Acceptance	Seneste events skal kunne fremvises uden brug af internet	De seneste gemte events skal gemmes i cachén på enheden, således de er tilgængelige for brugeren
T4	4F	API endpoints	Alle API'er skal testes ved brug af et HTTP kald igennem Postman.	Til route decorators skal alle tilgængelige endpoints testes i Postman. Testene skal opsættes således at input og output giver en response 201. Dette sikre også at databasen også virker.
T5	3I	GUI/Intergration	Listen med events bliver rettet til med rette events, i forhold til den rette radius	Brugeren tester appen, og justere radiussen, listen med events burde ændre sig i forhold til størrelsen.
T6	2D	Unit	Databasen virker, og gemmer rette informationer i rette tabeller	Ved at køre unittests på metoder som kalder databasen, vil man kunne teste om databasen virker som den skal.
T7	2F	GUI/Intergration	Listen med events bliver rettet til med rette events, i forhold til ændringerne i valgt kategori og lokation	Ved at brugeren bevæger sig i andre områder og ændre kategorierne, skal listen med events ændre sig.

Produktarkitektur

Projektet består af flere teknologier, der arbejder sammen for at levere en funktionel løsning. Her er en liste over produktets teknologier:

1. 1. Overordnede teknologier

- a. **Frontend:** Vue.js (PWA) – Brugergrænsefladen, som brugerne interagerer med.
- b. **Backend:** Python FastAPI – Håndterer API-kald og forretningslogik.
- c. **Database:** PostgreSQL – Gemmer og håndterer eventdata.
- d. **Hosting:** Docker + Azure – Applikationen kører i containere, som er hostet i skyen.
- e. **Offline-funktionalitet:** Service Workers – Cacher data, så appen kan bruges uden internetforbindelse.

2. 2. Teknologiernes roller

- a. Vue.js kommunikerer med FastAPI via REST API'er.
- b. FastAPI henter og gemmer data i MSSQL.
- c. Docker bruges til at pakke og køre både backend og database i containere.
- d. Service Workers sikrer, at appen fungerer offline ved at cache de seneste data.

3. 3. Samspil mellem teknologierne

- a. Brugeren åbner PWA'en i browseren.
- b. Vue.js sender en HTTP-request til FastAPI for at hente events.
- c. FastAPI læser fra PostgreSQL og returnerer data som JSON.
- d. Hvis brugeren mister internetforbindelsen, bruges Service Workers til at vise de sidst hentede data.

Produktkonfiguration og – udarbejdelse

Database interrogation

Systemet er bygget op omkring en Microsoft SQL Server (MSSQL) database, der administreres via SQL Server Management Studio (SSMS). Databasen indeholder tre primære tabeller: **Events**, **Locations** og **Users**.

- **Events-tabellen** lagrer oplysninger om arrangementer, herunder navn, dato og en reference til den tilknyttede lokation.
- **Locations-tabellen** indeholder information om eventlokationer, såsom adresse og kapacitet.
- **Users-tabellen** gemmer data om registrerede brugere, inklusiv brugernavn og adgangskode.

For at sikre dataintegritet er der oprettet **foreign key constraints** mellem **Events** og **Locations**, hvilket garanterer, at et event altid er knyttet til en gyldig lokation.

Alle databasekald udføres via **stored procedures**, hvilket optimerer ydeevnen ved større forespørgsler og forbedrer sikkerheden. Dette eliminerer risikoen for **SQL-injections**, da serveren aldrig sender rå SQL-forespørgsler direkte, men i stedet kun kalder definerede procedurer.

Databasen håndterer desuden filtrering af data direkte i SQL Server. For eksempel anvendes geografiske beregninger til at returnere events baseret på en brugers **geolokation** og en angivet radius.

Stored procedure over hvordan events bliver hentet ud fra lokation og radius.

```
USE [Event_Tracker]
GO
/***** Object: StoredProcedure [dbo].[GetEventsWithinRadius]  Script Date: 25-02-2025 07:58:50 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[GetEventsWithinRadius]
    @user_latitude FLOAT,
    @user_longitude FLOAT,
    @radius_km FLOAT
AS
BEGIN
    -- Select statement to fetch events within the specified radius in KM
    SELECT
        e.Event_ID,
        e.Event_Name,
        e.Event_Start_Time,
        l.Location_Name
    FROM
        Events e
    JOIN
        Location l ON e.Location_ID = l.Location_ID
    WHERE
        geography::Point(@user_latitude, @user_longitude, 4326).STDistance(
            geography::Point(
                CAST(SUBSTRING(l.Location_LatLong, 1, CHARINDEX(',', l.Location_LatLong) - 1) AS FLOAT),
                CAST(SUBSTRING(l.Location_LatLong, CHARINDEX(',', l.Location_LatLong) + 1,
                    LEN(l.Location_LatLong)) AS FLOAT),
                4326)
            ) <= (@radius_km * 1000);
END;
```

Den stored procedure ovenover viser hvordan at det er databasen ansvar at hente de relevante events og returnere dem til serveren.

Server integration

Repositorylaget – kommunikation med databasen

Som beskrevet i afsnittet om **teknologivalg**, er systemet bygget efter en **klient-server-arkitektur**, hvor database, server og klient er klart adskilte.

Al kommunikation med databasen håndteres via **repository-klasser**, hvilket sikrer en struktureret og kontrolleret adgang. Kun disse repository-klasser har adgang til at kalde databasen – og kun via stored procedures.

Et eksempel på denne tilgang ses i **event_repo.py**, hvor metoden `get_events_within_radius()` kalder stored proceduren **GetEventsWithinRadius** med parametre leveret fra klienten.

```
# event_repo.py

from Database.db import Database
import pyodbc
from dotenv import load_dotenv

load_dotenv()

def get_events_within_radius(radius, latitude, longitude):

    db = Database()
    db.create_connection()

    conn = db.conn
    cursor = conn.cursor()

    sql = """
EXEC GetEventsWithinRadius
@user_latitude = ?,
@user_longitude = ?,
@radius_km = ?;
"""

    cursor.execute(sql, (latitude, longitude, radius))

    results = cursor.fetchall()

    cursor.close()
    conn.close()

    return results
```

Det ses at her i `event_repo.py` en metode som kalder `GetEventsWithinRadius` med parameter givet fra klienten.

Servicelaget – appen's forretningslogik og bruger sikkerhed

Et vigtigt aspekt af systemet er sikker håndtering af følsomme data, især loginoplysninger. Al brugerhåndtering foregår i **login_service.py**, hvor adgangskoder krypteres og saltes før lagring i databasen.

Når en bruger oprettes, krypteres adgangskoden med **bcrypt** og lagres derefter i databasen via repository-laget. Denne fremgangsmåde sikrer, at adgangskoder ikke gemmes i et læsbart format, hvilket forhindrer kompromittering i tilfælde af dataleak.

```
import bcrypt
import jwt
import datetime
import os

SECRET_KEY = os.getenv("SECRET_KEY", "my_secret_key")

def user_create(username, password):
    if not username or not password:
        return {"error": "Username and password are required"}, 400

    # Hash the password
    hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode('utf-8')

    try:
        user_repository.create_user(username, hashed_password)
        return {"message": "User created successfully"}, 201
    except Exception as e:
        return {"error": f"Error: {str(e)}"}, 500
```

Routinglaget – Kald og opsætning af API'er

- Systemets API er udviklet i **Flask**, der blandt andet tilbyder indbygget Swagger-support til test og dokumentation.
- Hovedfilen **app.py** fungerer som API'ets entrypoint og håndterer:
- **Opsætning af Flask-applikationen**
- **Aktivering af CORS (Cross-Origin Resource Sharing)**
- **Registrering af API namespaces (routes)**
- **Aktivering af Swagger-dokumentation på /doc**

API'et definerer namespaces til de forskellige entiteter:

- `/api/events` – håndterer event-relaterede kald
- `/api/users` – håndterer brugerrelaterede kald
- CORS er aktiveret med `supports_credentials=True`, hvilket tillader deling af cookies mellem klient og server.
- Et eksempel på en API-route ses i **events_routes.py**, hvor endpointet `/get_users_within_radius` håndterer POST-anmodninger.

```
from flask import Flask
from flask_cors import CORS
from flask_restx import Api
from Routes.user_routes import ns_events as users_ns
from Routes.event_routes import ns_events as events_ns

# Initialize Flask app
app = Flask(__name__)

# Enable CORS for all routes globally with credentials support
CORS(app, supports_credentials=True)

# Attach RESTX API to the main app (ensures Swagger works)
api = Api(app, doc='/doc')

# Register the namespaces
api.add_namespace(events_ns, path='/api/events')
api.add_namespace(users_ns, path='/api/users')

# Run Flask app
if __name__ == '__main__':
    app.run(debug=True)
```

I routingfilen, ses hvordan et API er blevet oprettet:

```
@ns_events.route('/get_users_within_radius', methods=['POST'])
class Handle_events(Resource):
    def options(self):
        return {"message": "CORS preflight successful"}, 200

    @ns_events.expect(location_model)
    def post(self):
        radius = request.json.get('radius')
        latitude = request.json.get('latitude')
        longitude = request.json.get('longitude')

        if not all([radius, latitude, longitude]):
            return {"error": "Missing required fields (radius, latitude, or longitude)"}, 400

        try:
            events = get_events_from_radius(radius, latitude, longitude)

            if not events:
                return {"message": "No events found within the specified radius"}, 404

            return jsonify({"events": events})

        except Exception as e:
            logging.error(f"Error while getting events within radius: {str(e)}")
            return {"error": f"An error occurred: {str(e)}"}, 500
```

Her er der også en del vigtige elementer som:

- **Route og metode:** Endepunktet /get_users_within_radius bruger POST-metoden.
- **CORS support:** Implementeret med en OPTIONS-metode for preflight requests.
- **Input-validering:** Sikrer, at radius, latitude og longitude er til stede i request-body.
- **Fejlhåndtering:**
 - Returnerer 400 ved manglende felter.
 - Returnerer 404, hvis der ikke findes events inden for radius.
 - Returnerer 500 ved interne fejl, som logges.
- **Funktionalitet:**
 - Kalder get_events_from_radius() for at hente events baseret på radius og koordinater.
 - Returnerer fundne events i JSON-format.
- **Logging:** Fejl logges for debugging.

Når serveren kører, og man ser på swagger, vil man kunne se for dette API, både hvad den forventer, metode type og hvad forventer at få tilbage:

POST `/api/events/get_users_within_radius`

Parameters

Name	Description
payload * required object (body)	<div>Example Value Model</div> <pre>{ "radius": "string", "latitude": "string", "longitude": "string" }</pre> <div>Parameter content type application/json ▼</div>

Responses

Code	Description
200	Success

Frontend – Applikationen

Routing

Inden jeg taler om de views og komponenter, er det vigtigt at forstå hvordan strukturen er lavet, der er kun én side, app.vue. Så ændringerne i siderne er komponenter og views som bliver routet, i forhold til om man har adgang (Er logget ind), og hvilke knapper man klikker på. Dette gøres ved brug af router.beforeeach(), som har til formål at se om man er logget ind, og dertil vil vise det rette indhold. Routing, index.js:

```
router.beforeEach(async (to, from, next) => {  
  
  // Tillad adgang til login-siden uden at tjekke autentifikation  
  
  if (to.name === 'UserLogin') {  
  
    next()  
  
    return  
  
  }  
  
  try {  
  
    const response = await fetch('http://127.0.0.1:5000/api/users/check', {  
  
      method: 'GET',  
  
      credentials: 'include'  
  
    })  
  
    // Håndter 401 UNAUTHORIZED  
  
    if (response.status === 401) {  
  
      globalAuthState.loggedIn = false // Opdater global state  
  
      next({ name: 'UserLogin' }) // Omdiriger til login-siden  
  
      return  
  
    }  
  
    // Hvis statuskoden er 200, fortsæt til den ønskede rute  
  
    if (response.ok) {  
  
      globalAuthState.loggedIn = true // Opdater global state  
  
      next()  
  
      return  
  
    }  
  
  }  
})
```

Ved at kalde <http://127.0.0.1:5000/api/users/check>, kalder appen serveren og sender ens cookie med, hvis man har en. Serveren vil validere cookien, og dertil sende en 200 hvis cookien er OK. Hvis brugeren ikke har en cookie, eller er udløbet, vil appen ændre komponentet i app.vue til at indeholde UserLogin.vue, som er et komponent der bruges til at indeholde login siden.

Når brugeren har fået en cookie, og er blevet valideret, vil først menuen blive opdateret til den som de validere brugere for.

App.vue

App.vue:

```
<template>

<div id="app">

  <MainMenubar v-if="!loggedIn" />

  <LoggedInMenubar v-else />

  <router-view />

</div>

</template>
```

Læg mærke til at her bruges der en if statement, som sørger for at menu baren ændre sig, om brugeren har en valideret cookie. Når brugeren starter appen, vil brugeren også blive spurgt efter om appen må bruge enhedens lokation, dette bruges til at finde brugerens latitude og longitude.

```
<router-view />
```

Router-view bruges til at opdatere det HTML som app.vue skal indeholde, eftersom der sker ændringer i routing.

```
Setup () {

  const loggedIn = ref(globalAuthState.loggedIn)

  // Opdater loggedIn når globalAuthState ændres
  watchEffect(() => {

    loggedIn.value = globalAuthState.loggedIn

  })

  // Tilføj latitude og longitude som reactive variabler
  const latitude = ref(null)
  const longitude = ref(null)

  // Metode til at hente brugerens geolokation
```



```
const getLocation = () => {  
  if (navigator.geolocation) {  
    navigator.geolocation.getCurrentPosition(  
      (position) => {  
        latitude.value = position.coords.latitude  
        longitude.value = position.coords.longitude  
      },  
      (error) => {  
        console.error('Error getting location:', error)  
      }  
    )  
  } else {  
    console.error('Geolocation is not supported by this browser.')  
  }  
}  
  
// Kald getLocation, når komponenten er oprettet  
getLocation()
```

Når brugerens lokation er hentet, har brugeren nu mulighed for at hente de begivenheder som brugeren ønsker indenfor den ønskede radius. Dette sker i EventListTemplate.vue, som også er en komponent.

EventListTemplate.vue

```
// Hent enhedens lokation

if (navigator.geolocation) {

  navigator.geolocation.getCurrentPosition(

    async (position) => {

      this.latitude = position.coords.latitude

      this.longitude = position.coords.longitude

      // Tjek om der er internetforbindelse

      if (navigator.onLine) {

        try {

          // Forsøg at hente data fra API'et

          const response = await axios.post('http://127.0.0.1:5000/api/events/get_users_within_radius', {

            radius: this.radius,

            latitude: this.latitude,

            longitude: this.longitude

          })

          this.data = response.data // Gem API-svaret

          // Cache API-svaret for fremtidig brug

          if (navigator.serviceWorker) {

            const cache = await caches.open('events-api-cache')

            cache.put('http://127.0.0.1:5000/api/events/get_users_within_radius', new

Response(JSON.stringify(this.data)))

          }

        }

      }

    }

  )

}
```

Koden viser to vigtige ting:

1. **Online funktion:** Henter data fra serveren
2. **Offline funktion:** Henter data fra cachen

Hvis brugeren starter appen, uden at være på internettet, vil der ingen data være gemt i cachen. Inden jeg fortæller om hvordan data bliver gemt i cachen, vil jeg først vise hvordan den relevante service worker virker i komponentet:

```
// Hvis der ikke er internetforbindelse, forsøg at hente cachelagrede data

const cache = await caches.open('events-api-cache')

const cachedResponse = await cache.match('http://127.0.0.1:5000/api/events/get_users_within_radius')

if (cachedResponse) {

  const cachedData = await cachedResponse.json()

  this.data = cachedData // Vis cachelagrede data

  this.error = 'Ingen internetforbindelse. Viser cachelagrede data.'

} else {

  this.error = 'Ingen internetforbindelse, og der er ingen cachelagrede data.'

}
```

Service workeren vil kalde caches.open, og hente de gemte data, og sætte this.data til at vise det relevante data.

Service Workeren

Service workeren er en funktion som bruges til at udnytte funktionaliteter som bl.a. offlinefunktion, en service worker bliver først lavet ved at registrere den i registerServiceWorker.js, hvor efter man kan lave sin service worker, service-worker.js. Ved at specificere hvilket endpoint det er man vil cache, kan man ændre parameter. Her bliver der gemt 10 begivenheder, samt at de bliver gemt i 1 time.

```
registerRoute(

  ({ url }) => url.pathname === '/api/events/get_users_within_radius',

  new NetworkFirst({

    cacheName: 'events-api-cache',

    plugins: [

      new CacheableResponsePlugin({

        statuses: [0, 200] // 0 for offline, 200 for succesfulde kald

      }),

      new ExpirationPlugin({

        maxEntries: 10, // Behold de 10 seneste responses

        maxAgeSeconds: 60 * 60 // Gem cache i 1 time

      })

    ]

  })

)
```

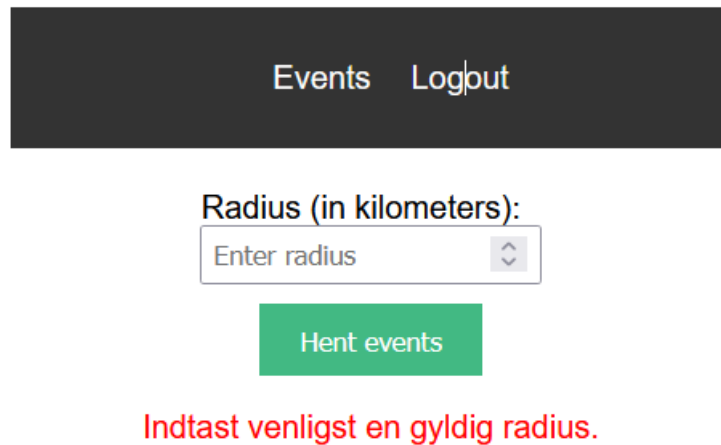
Brugervejledning

1. Når brugeren har downloadet appen, vil man blive mødt med at skulle logge ind

[Home](#) [Login](#)

Login

2. Når brugeren er logget ind har brugeren nu mulighed for at indtaste ens ønsket radius til afgrænsning af listen af begivenheder



The screenshot shows a dark grey header bar with the text "Events" and "Logout" in white. Below the header, the text "Radius (in kilometers):" is displayed. Underneath is a text input field with the placeholder text "Enter radius" and a small dropdown arrow icon on the right. Below the input field is a green button with the text "Hent events" in white. At the bottom of the form, there is a red error message: "Indtast venligst en gyldig radius."

3. Her kan brugeren nu se detaljer om de enkelte events

The Kilkennys
Start Time: 2.11.2025, 19.00.00
Location: Europahallen

Huxi Bach - Ufatteligt
Start Time: 8.11.2025, 19.00.00
Location: Europahallen

DRIVHUSET x The Comedy Store
Start Time: 9.11.2025, 18.00.00
Location: Europahallen

DRIVHUSET - Jacob Aksglæde og TBA
Start Time: 12.11.2025, 18.00.00
Location: Europahallen

API-et er ikke tilgængeligt. Viser
cachelagrede data.

4. Hvis brugeren er uden for netværk, vil der blive vist cached data.

