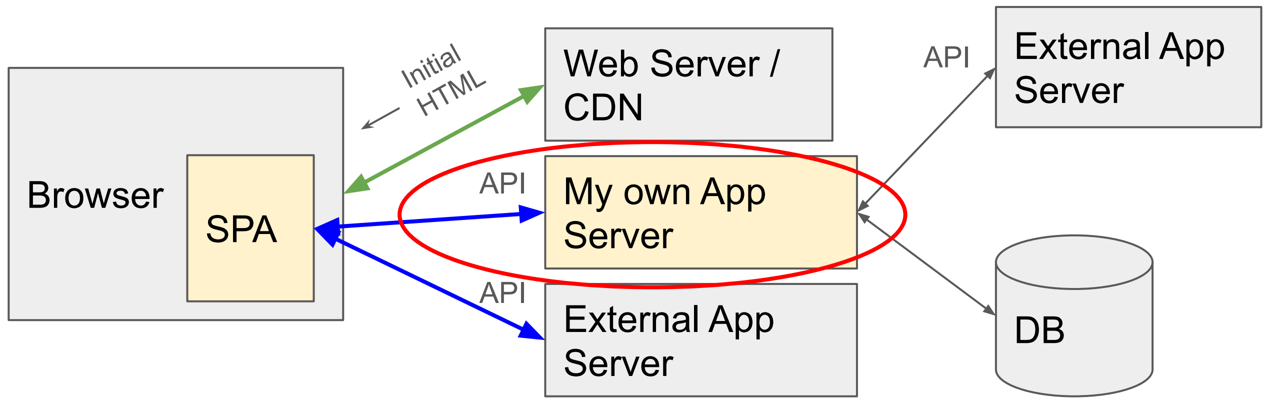
TIA - backend

Jana Kostičová

Server side / Backend



🟨 = my business logic

Set up node.js / express.js backend

(You must have node.js installed locally)

1. Create basic express app
   https://expressjs.com/en/starter/generator.html

```
npx express-generator my-app-be --no-view

cd my-app-be
npm install
```

2. Install nodemon

```
npm install --save-dev nodemon
```

3. Modify package.json (add “dev” script)

```
"scripts": {
    "start": "node ./bin/www",
    "dev": "nodemon ./bin/www"
  },
```

4. Run the server with automatic restarts

```
npm run dev
```

Enabling ESM syntax (optional)

- Package.json - add `"type": "module",`
- Update files: `bin/www`, `app.js`, `routes/*`

1. Update imports

```
const express = require('express');
```
$\longrightarrow$
```
import express from 'express';
```

2. Update exports

```
module.exports = router;
```
$\longrightarrow$
```
export default router;
```

3. Use import.meta.url to mimic __dirname (if necessary)

Create backend API (getting messages from server)

Backend

1. *data/messages.js*: Create mock data (see sources)
2. *routes/api_v1/messages.js*: Create new router for messages
3. *app.js***:** Add messagesRouter
4. Test in browser

2. **Backend:** New file *routes /api_v1/messages.js*

```
var express = require('express'); // ESM: import
const messages = [....]; // sample data

var router = express.Router();

router.get('/', function(req, res, next) {
  res.json(messages);
});

module.exports = router; // ESM: export
```

3. **Backend:** Update *app.js*

```
var messagesRouter = require('./routes/api_v1/messages'); // ESM: import
...
app.use('/api/v1', messagesRouter);
```

4. Test in browser: http://localhost:3000/api/v1/messages (adjust if needed)

BE ↔ FE communication

Frontend

- *vite.config.js:* Add to *defineConfig*:

```
server: {
   proxy: {
     '/api': 'http://localhost:3000', // Adjust if needed
   },
 },
```

- (Restart dev server)
- Modify sources to fetch data from server / upload data to server

Modify frontend (getting messages from server)

1. **Frontend:** Update *src/messageService.jsx - getMessages* function
2. **Frontend:** Update *App.jsx* - *useEffect* hook

1. **Frontend:** *src/messageService* - update *getMessage* function (.then / .catch syntax):

```
    function getMessages() {
        return fetch("/api/v1/messages").then(  // promise is resolved
            (response) => {
                if (!response.ok) { // HTTP status code NOT between 200-299
                    throw new Error("Error getting messages");
                }
                return response.json();
            }).catch((error) => {               // promise is rejected
                // Better way would be to throw error here and let the
                 // client handle (e.g. show error message)
                // Returning empty array for simplicity only!
                console.log("Error getting messages");
                return [];
            });
    }
```

Alternatives

- Async / await syntax - more friendly syntax in some cases
- Axios library
- …

Response object - properties

- response.ok:
  - True if HTTP response status code is in the successful range (200-299)
- response.status
  - HTTP status code (e.g., 200, 404, ..)
- response.statusText:
  - A description of the status code ("OK", "Not Found", ..)
- response.headers:
  - An object representing the headers of the response
- response.json():
  - A method to parse the response body as JSON (returns promise)
- response.text():
  - A method to parse the response body as text (returns promise)

2. **Frontend:** *App.jsx* - update *useEffect* hook

```
useEffect(() => {
  getMessages().then(
    (messages) => setMessages(messages)
  );

  const fetchMessagesInterval = setInterval(() => {
      getMessages().then(
        (messages) => setMessages(messages)
      );
    }, 10000);
  return () => clearInterval(fetchMessagesInterval);
}, []);
```

Test

- Run frontend development server - mock data from backend should appear
- Data flow Backend -> Frontend **established**

Opposite data flow (uploading a new message to server)

1. **Backend:** Update *routes/messages.js*
2. **Frontend:** Update *services/messageService.jsx - addMessage* function
3. **Frontend:** Don’t use callback in *pages/NewMessagePage.jsx*

1. **Backend:** Modify *routes /api_v1/messages.js*

```
var express = require('express'); // ESM: import
var router = express.Router();

router.get('/', function(req, res, next) {
  res.json(messages);
});

router.post('/', function(req, res, next) {
  messages.push(req.body);
  res.status(200);
});

module.exports = router; // ESM: export
```

2. **Frontend:** *services/messageService.jsx* - update *addMessage* function:

```
function addMessage(message) {
    return fetch("/api/v1/messages", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify(message)
    });
}
```

3. *pages/NewMessagePage.jsx: PublishMessage* `//setMessages(getMessages());`

- Data flow Frontend -> Backend **established**

BE ↔ DB (Postgres)

Backend

- Install pg package: npm install pg
- Add schema - *migrations/schema/tables.sql* (good practice)
- Add DB config - *config.secrets*
- Add model - *models/messages.js*
- Modify - *routes/api_v1/messages.js*

1. **Backend:** *migrations/schema/tables.sql*

```sql
CREATE TABLE "public"."users" (
    "id" varchar(100) NOT NULL,
    "avatar" varchar(100) NOT NULL,
    PRIMARY KEY ("id")
);

CREATE TABLE "public"."messages" (
    "id" varchar(100) NOT NULL,
    "user_id" varchar(100) NOT NULL,
    "text" text NOT NULL,
    PRIMARY KEY ("id")
    CONSTRAINT "messages_user_fk" FOREIGN KEY ("user_id")
         REFERENCES "public"."users" ("id") ON DELETE CASCADE
);


INSERT INTO "public"."users"(id, avatar) VALUES
     ('sampleUser123', 'images/person-circle.svg');
COMMIT;
```

Config.secrets template

```
// use your own configuration
exports.config = {
    db: {
        user: 'postgres',
        host: 'localhost',
        database: 'not_twitter',
        password: 'postgres',
        port: '5432'
    }
}
```

Config.secrets **must never be committed to the Git repository!** Add it to the .gitignore file.

2. Backend: *models/messages.js*:

- "mapping" relational data to objects

```
const {Pool} = require('pg');
const {config} = require('../config.secrets')

const pool = new Pool({
    user: config.db.user,
    host: config.db.host,
    database: config.db.database,
    password: config.db.password,
    port: config.db.port,
  });

exports.getMessages = function() {
    return pool.query(`
        select m.*, u.avatar
        from messages m
        left join users u on m.user_id=u.id
    `);
};

exports.addMessage = function(message) {
    return pool.query("insert into messages(id, user_id, text) values($1, $2, $3)",
     [message.id, message.user_id, message.text]);
};
```

DB credentials must not be present directly in the versioned code!

pool.query returns a promise
When calling getMessages / addMessage in routes, they must be handled with .then/.catch or async/await.

3a. Backend: modify *routes/api_v1/messages.js*

```
router.get('/', function(req, res, next) {
  getMessages().then(
    (messages) => {
      res.json(messages.rows);
    }
  ).catch(
    (err) => {
      console.log(err);
      res.status(500);
    }
  );
});
```

3b. Backend: modify *routes/api_v1/messages.js*

```
router.post('/', function(req, res, next) {
  addMessage(req.body).then(
      (r) => res.status(200)
    ).catch(
      (e) => {
        console.log(e);
        res.status(500);
      }
    );
});
```

Not-twitter tags

- Fontend+Backend+mock data: https://github.com/jkosticova/not-twitter/releases/tag/Frontend_Backend_mock_data
- Frontend+Backend+DB: https://github.com/jkosticova/not-twitter/releases/tag/Frontend_Backend_DB

HTTP Request Methods (verbs)

Indicate the desired action to be performed for a given resource

Most common:

- GET - get selected representation of the resource - does not alter state of server (safe)
- POST - request resource to process the entity based on resource’s rules (NOT idempotent)
- PUT - replace the resource with entity
- DELETE - delete the specified resource
- PATCH - apply partial modifications to a resource

Idempotency

An HTTP method is idempotent if the effect on the resource of a single request is the same as the effect of making several identical requests. E.g. increasing update counter in the resource is not idempotent

See also:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Rest API

A very brief introduction:

1. Endpoints are organized around resources (entities), not actions
   - Examples: */users, /products. /messages*
2. Actions are expressed by HTTP verbs
   - Instead of embedding actions in the URL (/getUsers, /deleteUser), Rest APIs use HTTP methods:
   - GET /users → Fetch users
   - POST /users → Create a user
   - PUT /users/1 → Update user with ID 1
   - DELETE /users/1 → Remove user with ID 1

- It might be difficult to achieve pure Rest API, deviations are acceptable (e.g., see authentication example later)

Query parameters

/api/v1/products?category=electronics&price_min=100&price_max=500

```
router.get('/products', (req, res) => {
  // extract query parameters
  const category = req.query.category;
  const price_min = req.query.price_min;
  const price_max = req.query.price_max;

  // process query parameters (e.g., filter products from a database)
  // this is just a placeholder response
  res.json({
        message: `Fetching products in the ${category} category with a
                price range between ${price_min} and ${price_max}`
  });
});
```

Topics not covered

- Transferring avatar
- HTTP error codes
- Rest API in more detail

Simple authentication using sessions

1. User logs in via login form

```
fetch("/api/v1/login", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ username, password }),
  credentials: "include"
});
```

2. The frontend (React) sends a POST request to the server with the username and password

3. Server validates credentials. If valid, it creates a session and sends back a session ID in an **HttpOnly** cookie

(it is first necessary to set up express-session middleware)

```
router.post("/", (req, res) => {
    const { username, password } = req.body;
    if (isValidUser(username, password)) {
        req.session.user = { username };  // Creates session
        res.status(200).json({ message: "Login successful" });
    } else {
        res.status(401).json({ error: "Invalid credentials" });
    }
});
```

Logout: The frontend calls a logout endpoint, and the server clears the session.

Alternatives

- Stateless models
  - Token authentication (JWT)
- Stateful models
  - OAuth
  - OpenID
- …

Server session

Way of maintaining state information associated with user’s interaction. Essentially a dictionary associating “session id” with data. Can be local for server or shared (e.g. in-memory vs shared database)

References

- https://expressjs.com/
- Axios library: https://axios-http.com/docs/intro
- https://www.postgresql.org/docs/current/
- Rest API: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- HTTP Request methods: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- TIA presentation 2023/24 (M. Kostič): https://micro.dcs.fmph.uniba.sk/dokuwiki/_media/sk:dcs:tia:tia_3_-_web_application_development_walkthrough_3_1_.pdf