<WA1/>

2020

Client-Server interactions in React

Re-Connecting to REST APIs

Enrico Masala
Fulvio Corno
Luigi De Russis

CC BY NC SA

Outline

- Handling API calls in React
- The “two servers” problem
  – Two servers + CORS
    - Variant: React Development Server
  – Build + Express (single server)
  – Also: Understanding Build (webpack, imports, etc)

https://www.robinwieruch.de/react-fetching-data

The Road to Learn React, Chapter “Getting Real with APIs”

Taming the State in React, Chapter “Local State Management”

React as a REST Client

HANDLING API CALLS IN REACT

Main questions

- How to integrate remote (REST) APIs
- Where/when to load data from remote APIs?
- Delays and “loading…”
- Updating remote data

Different kinds of state

Entity State (or Application State)

- Retrieved from the back-end
- Should update the back-end
  - On user-initiated add/modify/delete actions
- Should periodically check for updates
  - Caused by other users, by other open sessions, or by connected systems
- Globally managed, accessible by various components

View State (or Presentation State)

- Not stored in the backend
- Does not need to persist
- Lives and dies within the controlling Component
- Implemented ad Local State
  - this.state
  - this.setState

Remote Application state

- Application State is stored into a remote database, accessible by REST APIs
- **Dehydrating** state means extracting the state from the React application
  - May happen at several times during application execution
  - Should happen whenever something (in the Application State) is modified
- **Rehydrating** state means retrieving state from the database
  - Must happen when the application bootstraps
  - **Best place**: inside `componentDidMount()` lifecycle method

Rehydrating at mount time

```
componentDidMount() {
  fetch('/api/items')    // backend API call
  .then(response => response.json())
  .then(archivedItems => this.setState(rehydrateItems(archivedItems)));
}

function rehydrateItems(archivedItems) {
  return function(prevState) { // the setState callback function
    return {    // build new state
      archivedItems: [
        ...prevState.archivedItems,   // existing state (if any)
        ...archivedItems              // merged with new state (if any)
      ]
    };
  };
}
```

https://reactjs.org/docs/faq-ajax.html

Dehydrating during updates

```
<ItemList ... addItem={this.addItem} .../>

addItem = (newItem) => {
  fetch('/api/items', {
    method: 'post',
    body: JSON.stringify(newItem)
  }).then(...).catch(...);

  this.setState((state)=>{ [...state.items, newItem] }) ;
}
```

The two updates (remote API, local state) run **in parallel**.

Optimistic state update: assumes that remote state will be updated without errors => **Risky**

“Loading…” indicators

- Rehydrating may require some time, while the component renders “empty” (with the initial state)
- Use a local state variable (e.g., `loading`) to indicate whether rehydrating has completed
- The `render()` will display a message (or a spinner image) if `loading===true`

```
componentDidMount() {
  this.setState(
    {loading: true});

  apiClient.loadItems()
  .then(people => {
     this.setState({
       loading: false,
       people: people});
  });
}
```

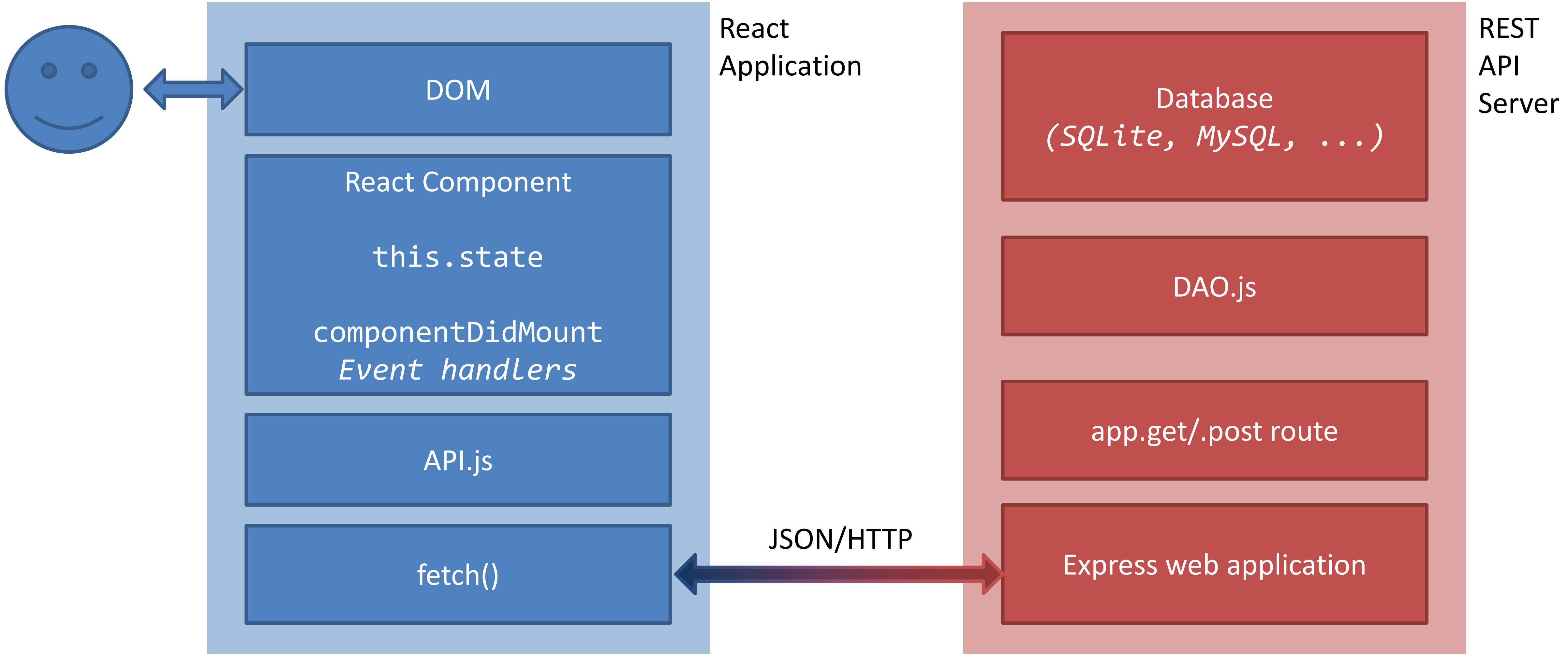
What component should “fetch” the data?

- The fetching component should be:
  - A common parent for all components **interested in this data**
  - A common parent for all components that must **show loading indicators**
  - A common parent for all components that need to **display error messages**
- May be the component holding the state do be Rehydrated
- May be a component **below** the one holding the state, if a rehydrating function is passed down as a prop

API Client Classes

- Recommendation: keep your fetch methods in a separate JS module (e.g., `API.js`)
- Keeps details of REST methods inside the API module
  - API should not depend on React or application state/props
  - Application code should not call fetch or have any REST information
- Allows easy swapping with “stub” methods for testing

Conceptual architecture



Local Storage

- Modern browsers also have client-side [semi-]permanent storage
- Key-value stores
  - Local Storage: permanently stored in the browser, can be retrieved also on future visits
  - Session Storage: expires when the browser session is closed
- Follow the same logic as remote storage (but they are synchronous and fast)
- May be combined with remote storage

https://www.robinwieruch.de/local-storage-react

Data caching

- The result of read APIs (GET) may be *cached* in the Local Storage (or application state, or application Context)
- The API Client module may implement a caching layer
- Future GETs for the same REST Resource may return the result from the cache, rather than the remote server

- But remember: cache invalidation is one of the hardest problems in Computer Science!

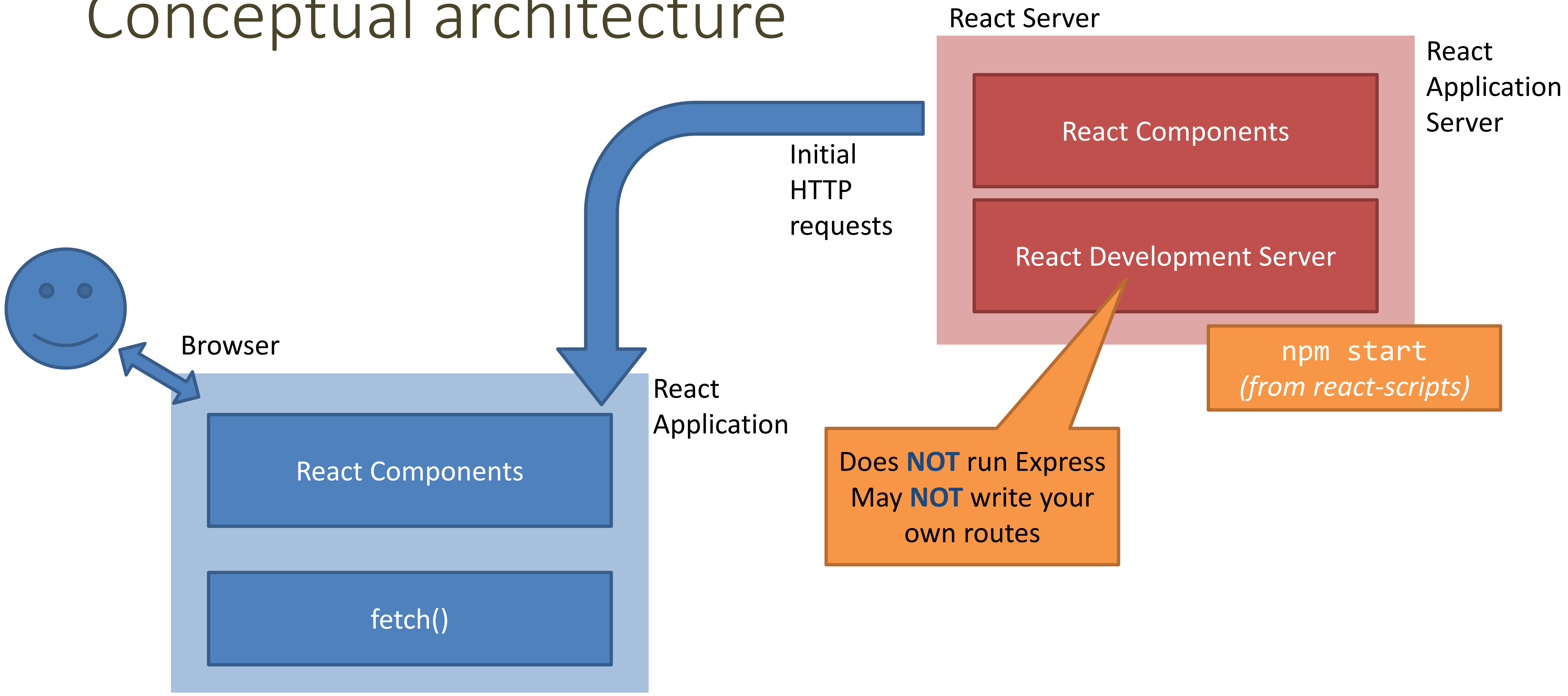
https://www.robinwieruch.de/react-fetching-data

Full Stack React, Chapter “Using Webpack with Create React App”

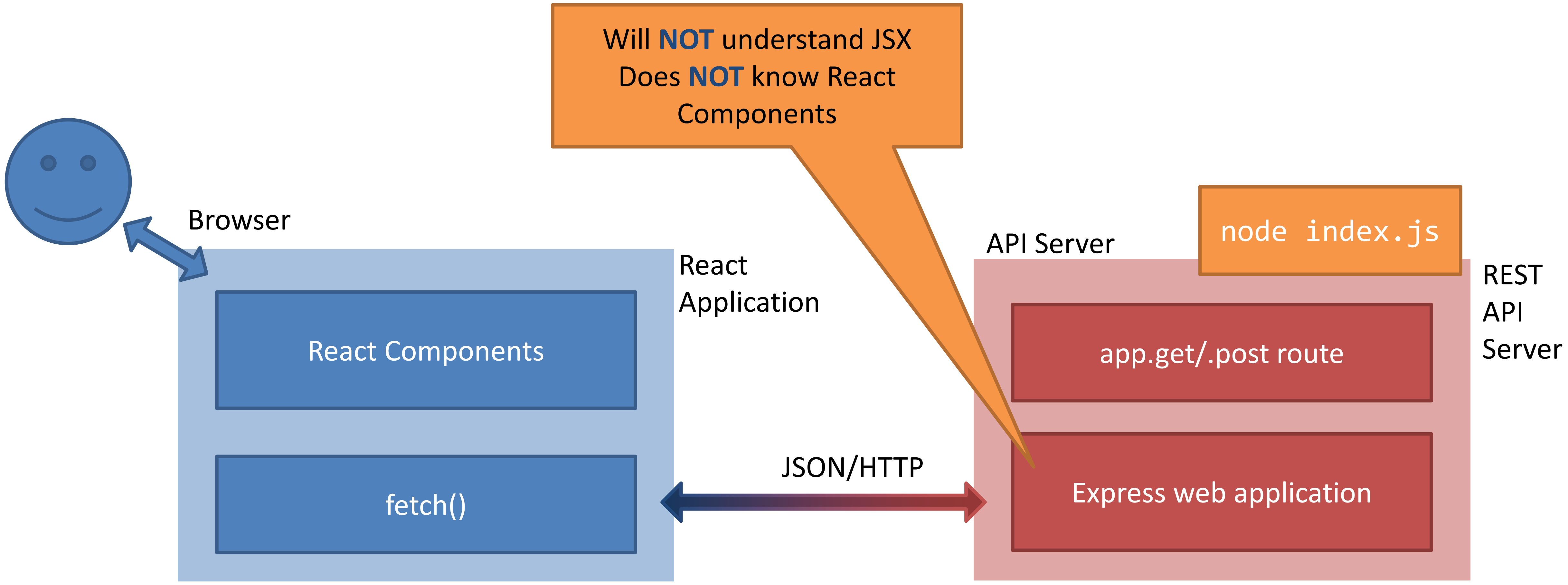
A Client and a Server walk into a bar…

THE “TWO SERVERS” PROBLEM

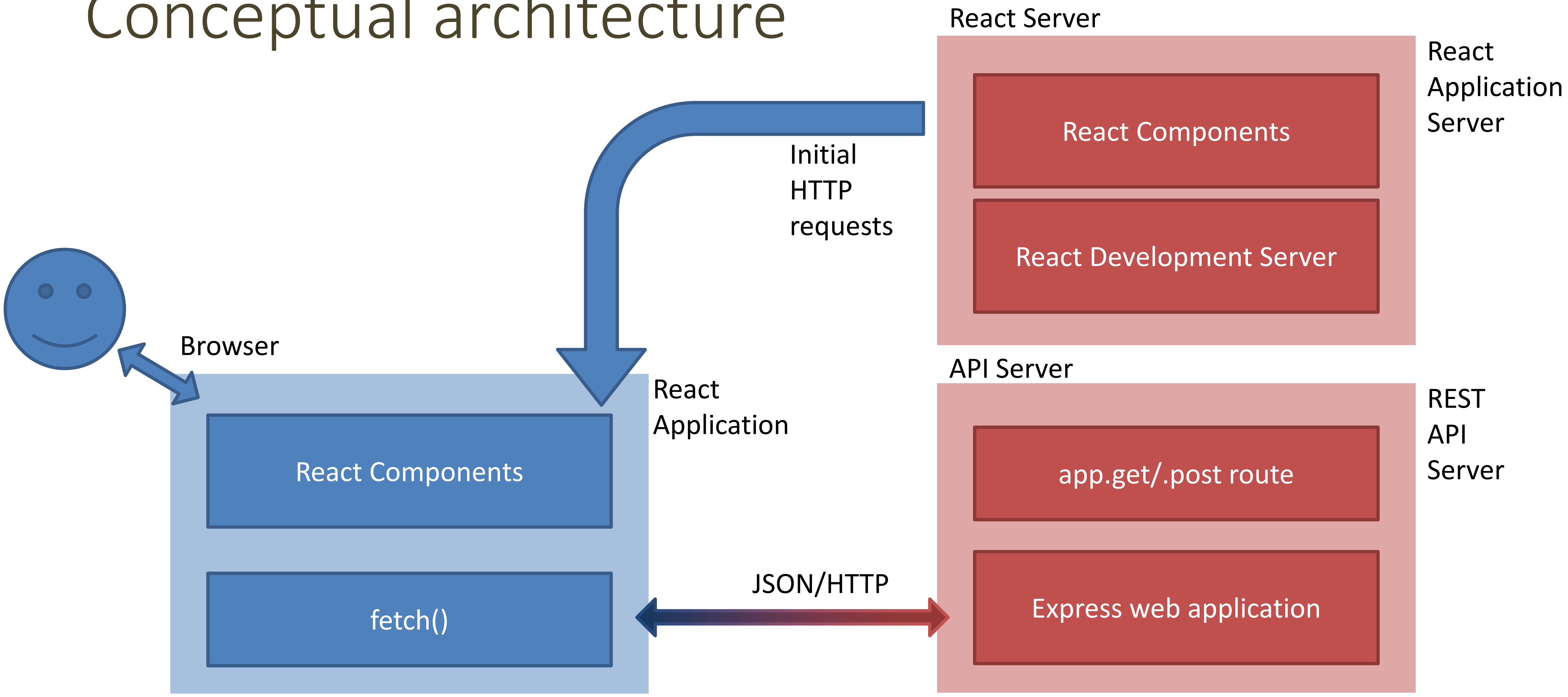
Conceptual architecture



Conceptual architecture



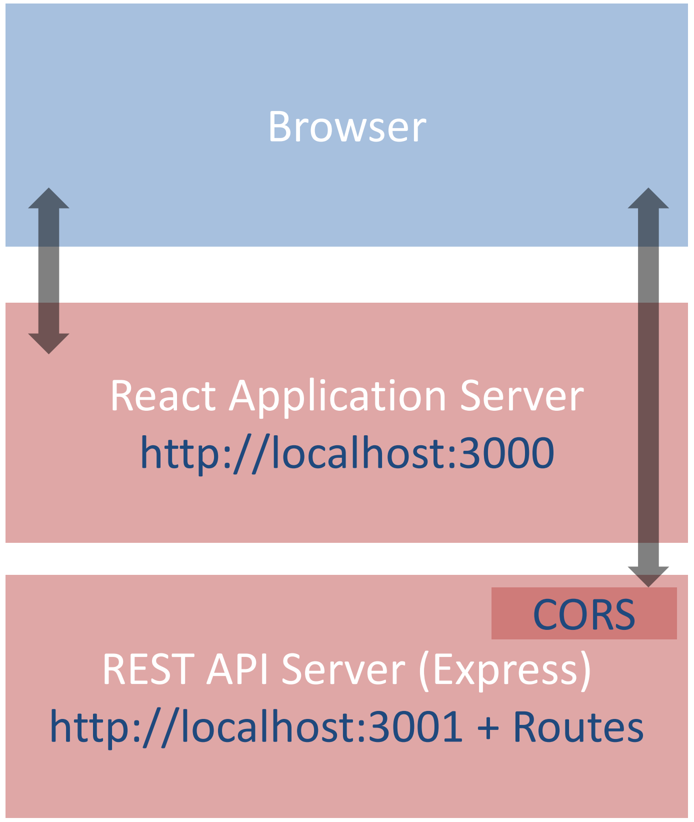
Conceptual architecture

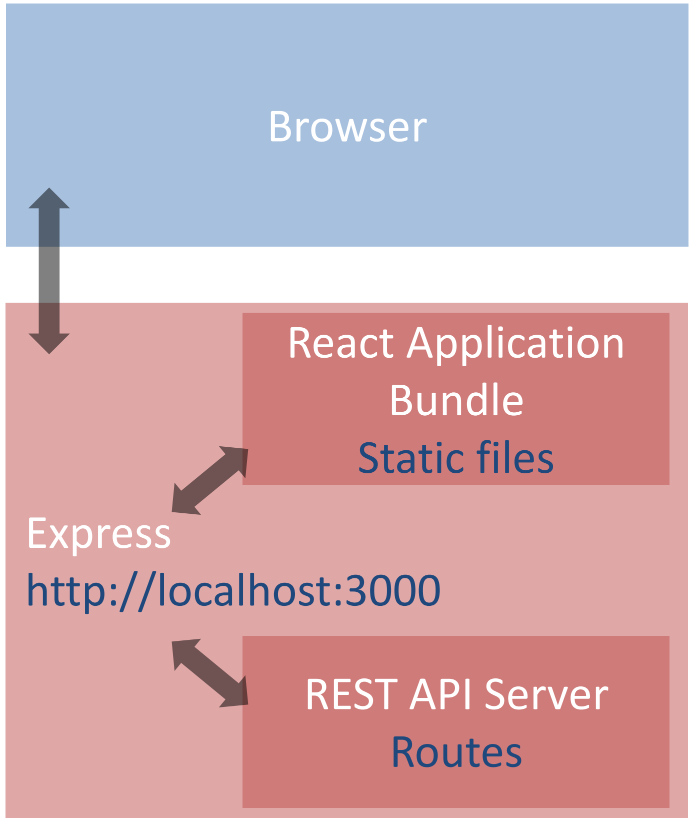


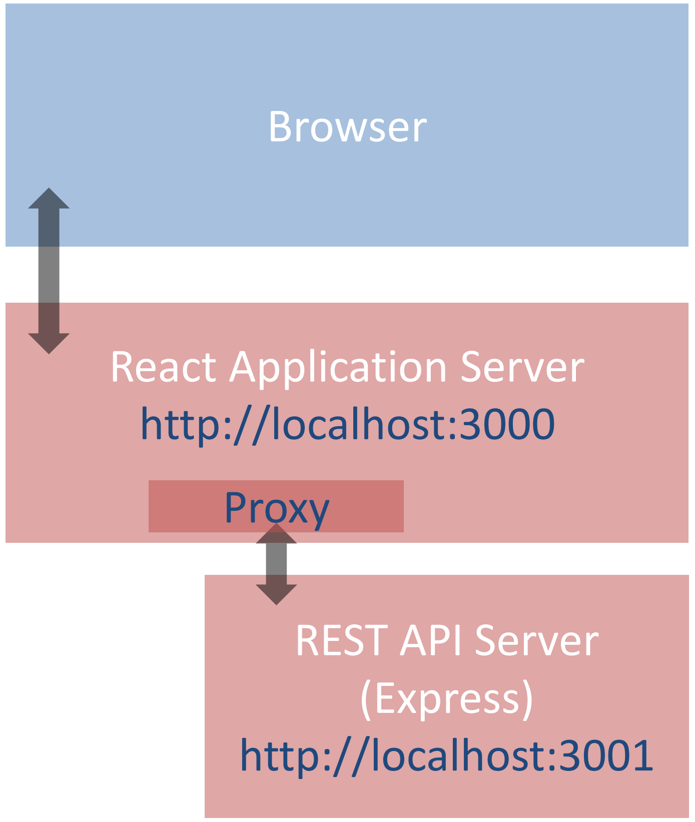
Issues

- Deployment
  - One-server-does-all or two-separate-servers?
  - Development vs Production trade-off
    - convenience/debug/turnaround time vs performance/security
  - Cross-Origin security limitations
- Opportunities
  - Separate the load
  - Use any API Server (even 3rd party ones)

3 Possible Solutions







https://www.newline.co/fullstack-react/articles/using-create-react-app-with-a-server/

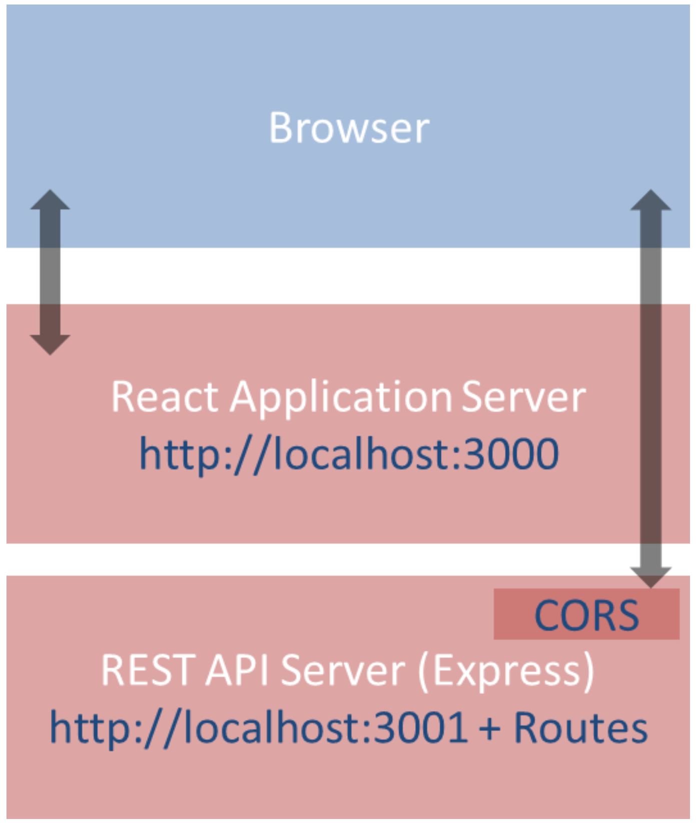
Full Stack React, Chapter "Using Webpack with Create React App / Using Create React App with an API server"

Side-by-side deployment

RUNNING TWO SEPARATE SERVERS

Double-Server Setup

- React Web Server and REST API server are hosted separately
  - Different hosts, and/or
  - Different ports
- The browser:
  - Receives the React application
  - Directs the API requests to the REST server



Double-Server Setup

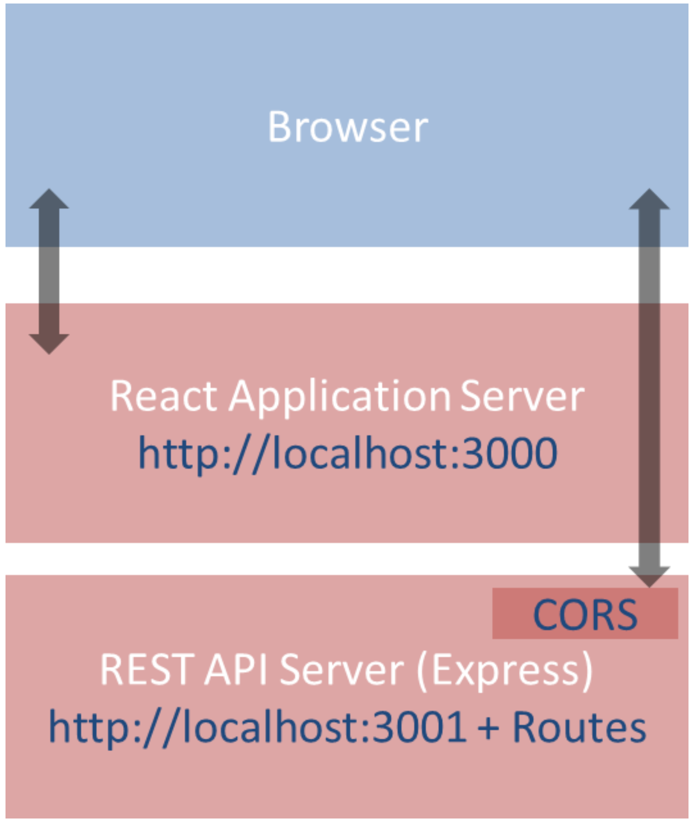
- Must run two web servers
  - React project: `npm start`
  - Express project: `node index.js`
  - Two projects, in two different directories (or different servers)
- Problem: handle CORS

<WA1/>
2020

CORS

Cross-Origin Resource Sharing

Enrico Masala
Fulvio Corno
Luigi De Russis



Advantages and disadvantages

- Servers are easy to deploy
- Scalable solution: requests are sent to the appropriate server
- Only possible configuration if the REST API is provided by a third party
  - Public APIs

- Need to configure cross-origin resource sharing (CORS) on API server
- Requires using absolute URLs to access APIs
- Wrongly configured CORS might be a security risk (undesired access to APIs from e.g. mock websites)

How to configure

- Configure CORS on API server for development

```
// index.js (node express server)

//Enable All CORS Requests (for this server)
app.use(cors());
//Use ONLY for development, otherwise restrict domain
```

- In production mode, use different domains for React and API servers, NEVER allow CORS requests from any origin, always specify origin

Example

API.js in the React Application

```
const APIURL=new URL('http://localhost:3001');

async function getCourses() {
  return fetch(new URL('/courses', APIURL))
    .then((response)=>{
      if(response.ok) {
        return response.json() ;
      } else {
        throw response.statusText;
      }
    })
    .catch((error)=>{
      throw error;
    });
}
```

Called in componentDidMount()

index.js for the API Server

```
const express = require('express');
const port = 3001;
const cors = require('cors');
const app = express();
app.use(cors());

app.get('/courses', (req, res) => {
  dao.listCourses()
    .then((courses) => res.json(courses))
    .catch((err)=>
      res.status(503)
         .json(dbErrorObj));
});

app.listen(port, () => console.log(`Example app 
listening at http://localhost:${port}`));
```

Calls DAO.js

https://create-react-app.dev/docs/proxying-api-requests-in-development/

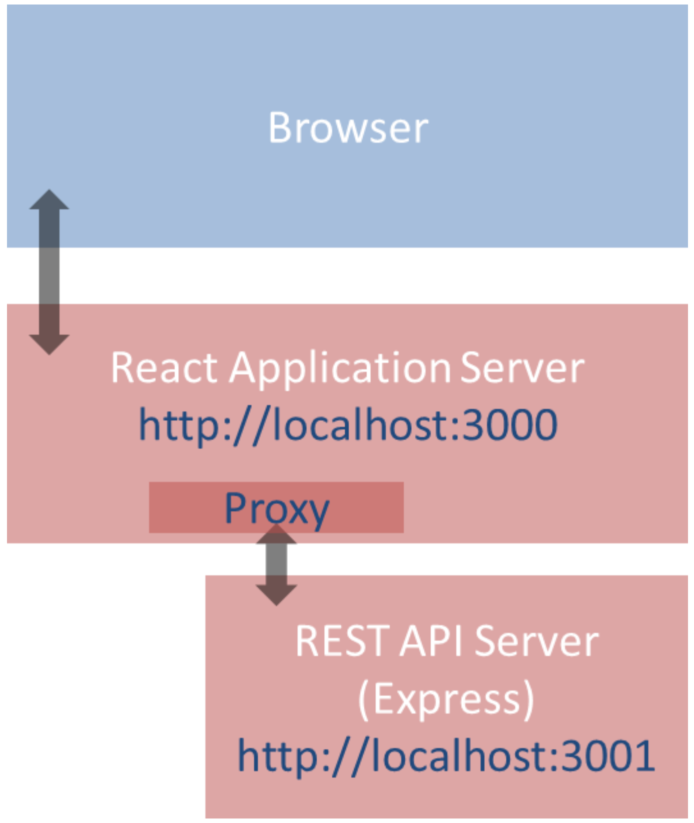
Full Stack React, Chapter “Using Webpack with Create React App / Using Create React App with an API server”

Double-Server made Easier

USING THE REACT DEVELOPMENT PROXY

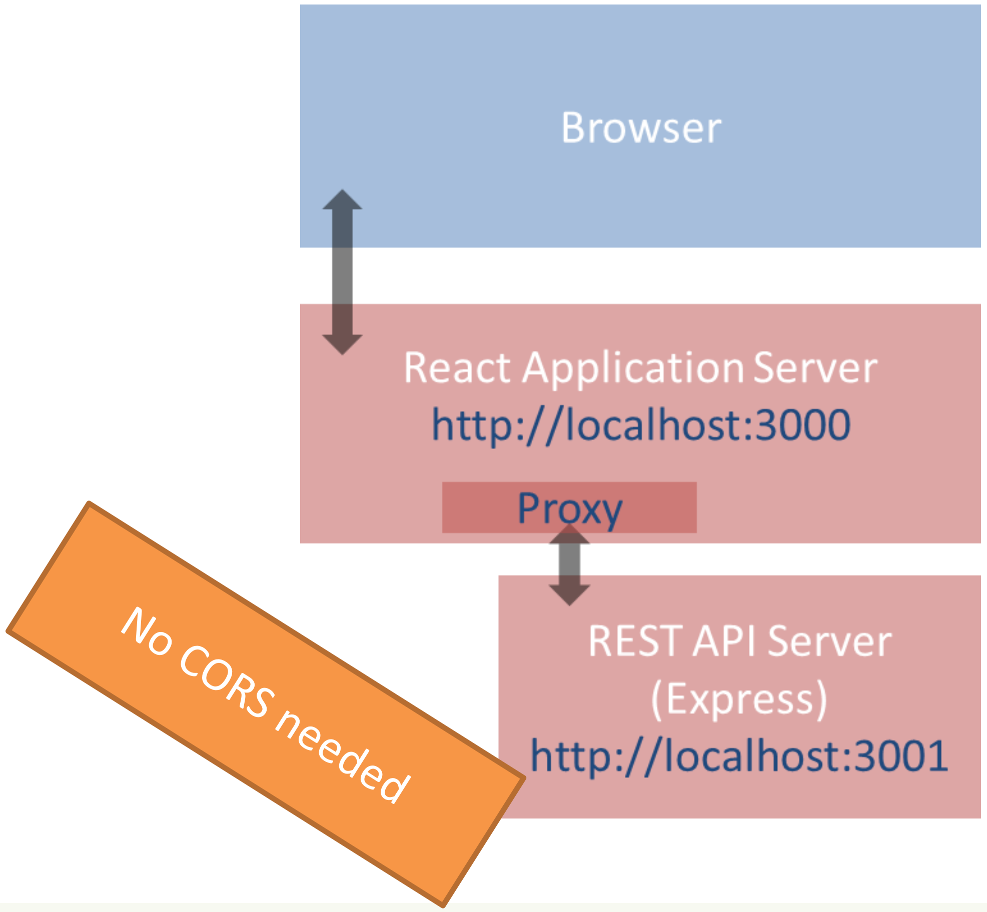
API Server behind Application Server

- A feature provided by the React Development Server
  - uses react-scripts development modules
- Avoids the need to set-up CORS
- The Browser thinks there is only one server



API Server behind Application Server

- Browsers access only one server: the React application server
- The React web server is configured to act as a proxy for certain requests
- Those requests are sent to another web server via a proxy mechanism
- The proxy returns the response unaltered as its own response



How to configure

- Just add one line in `package.json` originally written by create-react-app

```
// package.json
{...
 ...,
"proxy": "http://localhost:3001",
}
```

- NB: Works only in development mode while using the infrastructure of the create-react-app package

Proxy rules

- The React development server will serve requests **directly** if:
  - It is a recognized static asset (e.g., image, css, …)
  - The HTTP Accept header is set to `text/html`
- Otherwise, it will *attempt* to send the request to the **proxy**
  - The proxy response is returned
- If the resource is not found, it will serve the default html page

- Browsers use `text/html` only when expecting HTML content (e.g., first page)
- Best practice: avoid conflicting paths in URLs, if the path is found in React folders, it is served, otherwise it is passed to the proxy
  - Use unique path prefix for REST requests, e.g., `/api`

Use in production mode

- The approach may be useful in production mode if the REST API server should not / cannot be accessed directly from the Internet
  - For instance, application server with private IPs or other network/security configuration reasons
- The main web server (Apache, nginx, etc.) should be able to determine which requests must be redirected to the other web server
  - For instance, depending on URLs (e.g., /api/… requests)

```
# nginx web server
location /api/ {
    proxy_pass http://backend-server;
}
```

```
# Apache web server

ProxyPass /api/ http://backend-server
```

Common errors

- You are still running two web servers, on different ports
  - Remember to start the REST API server before launching the React application
  - May automate it by tweaking the startup scripts in `package.json`
- CORS is not needed
  - Remove it from the Express server
  - The API server is more secure, it will be accessed by React, only
- Production will be different
  - Need to configure the “real” proxy in production in order to be compatible with the same application path and API prefix

Common errors

- You are still running two web servers, on different ports
  - Remember to start the REST API server before launching the React application
  - May automate it by tweaking the startup scripts in `package.json`
- CORS is not needed
  - Remove it from the Express server
  - The API server is more secure, it will be acce
- Production will be different
  - Need to configure the "real" proxy in production in order to be compatible with the same application path and API prefix

Examples:
- https://www.freecodecamp.org/news/how-to-make-create-react-app-work-with-a-node-backend-api-7c5c48acb1b0/
- https://www.newline.co/fullstack-react/articles/using-create-react-app-with-a-server/

https://create-react-app.dev/docs/deployment/#static-server

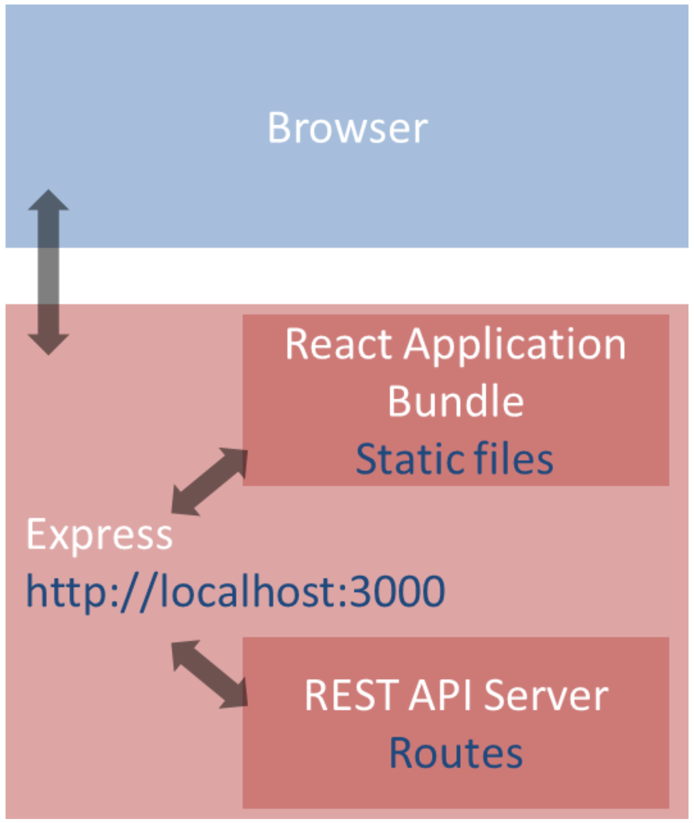
Full Stack React, Chapter “Using Webpack with Create React App / Creating a production build”

Packing and moving the React application into any web server

DEPLOYING A BUILD INSIDE A SERVER

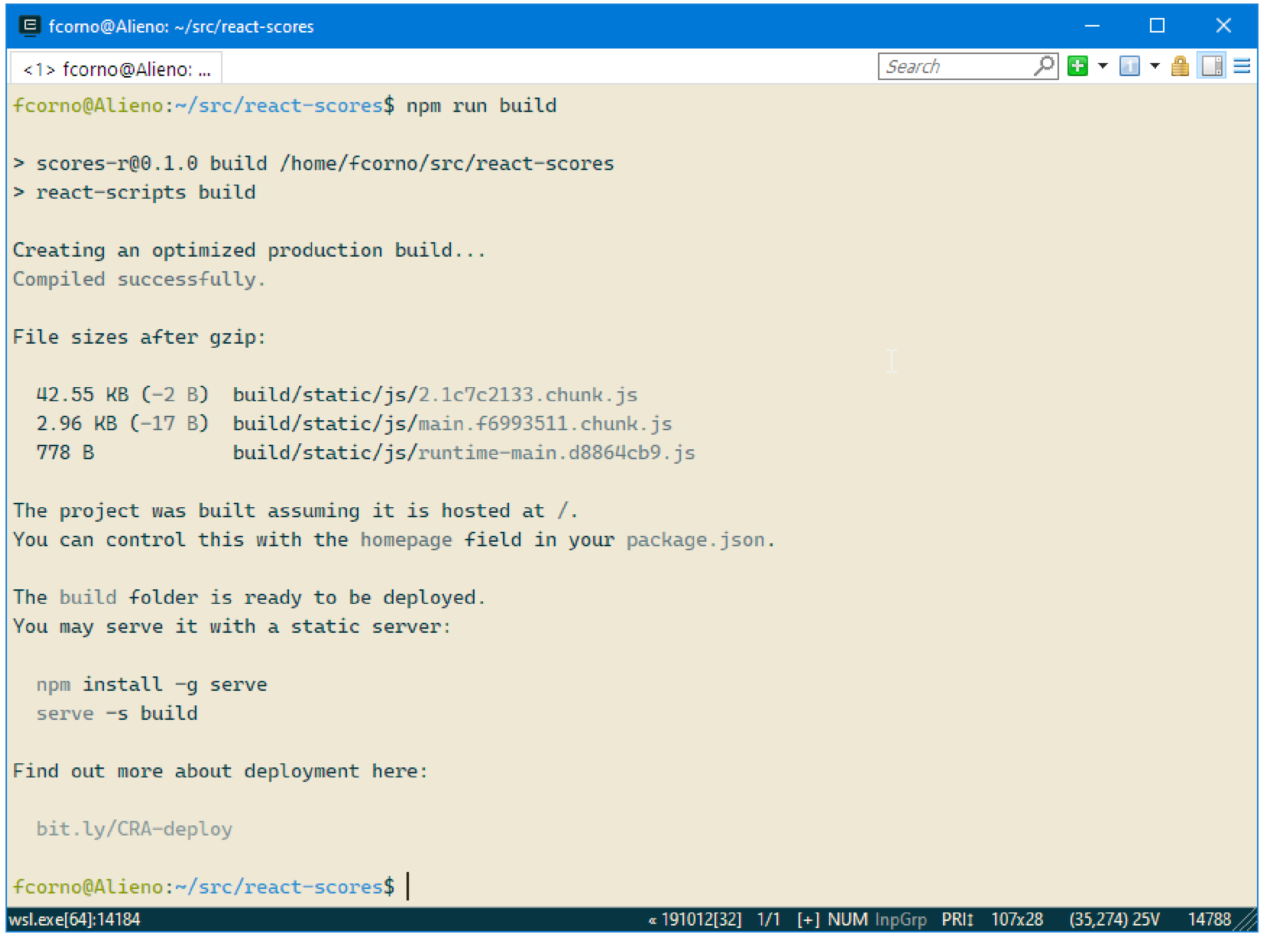
Deploying the React Bundle

- React does not need to run in the Development Server
- `npm run build` will create a “production bundle” with all the contents needed to run the application
- This bundle is composed of static files (html, js, assets) and may be served by *any webserver* (including Apache, nginx, php, …)



Build command

`npm run build`


fcorno@Alieno: ~/src/react-scores
<1> fcorno@Alieno: ...
Search
wsl.exe[64]:14184
« 191012[32] 1/1 [+] NUM InpGrp PRI↕ 107x28 (35,274) 25V 14788

```
fcorno@Alieno:~/src/react-scores$ npm run build

> scores-r@0.1.0 build /home/fcorno/src/react-scores
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  42.55 KB (-2 B)  build/static/js/2.1c7c2133.chunk.js
  2.96 KB (-17 B)  build/static/js/main.f6993511.chunk.js
  778 B            build/static/js/runtime-main.d8864cb9.js

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  bit.ly/CRA-deploy

fcorno@Alieno:~/src/react-scores$ |
```

Creates everything under `./build`

Publish from / or from 'homepage' property

https://create-react-app.dev/docs/deployment/

What does “build” do?

- Most of the work in “building” the static application is done by Babel and Webpack
  - Babel translates all JSX (and new JS syntax) into basic JS (according to the ‘`production`’ property in `package.json`)
  - Webpack packs and minimizes all JS code into a single file
  - Prepares an index.html that loads all the JS code
- The content of the “build” folder is self-contained and may be moved to the deployment server
- All debugging capabilities are removed

Hosting the build in Express

- `cd express-api-server`
- `cp –r ..../react-app/build .`
- Define a static route in `server.js`

```
app.use(express.static('./build'));

app.get('/', (req,res)=> {res.redirect('/index.html')} );
```

- In the application, you may call APIs locally
  - `fetch('/api/courses')...`

Pros and Cons

- Simple to deploy the final application (anywhere)
- May include the application inside the API server (in production, too)
- The JS code runs on every browser (thanks to polyfills and transpiling)

- The build cannot be directly modified
- Need a save/build/copy/reload cycle for every modification

Other “magic” by Webpack

- Packing of all imported modules
- Bundling of Assets
  - Images
  - CSS files
- CSS Modules

In development mode

- `npm start` runs the “Webpack development server” (WDS)
- All our code is transpiled and packed into a `bundle.js` that is automatically inserted into `index.html`
  - Contains all our code, plus React, plus imported modules
  - Also handles imports of non-JS files
- `bundle.js` does not exist – it’s kept in-memory by the WDS
- Sets up hot-reloading and synchronized error messages (via websockets)

https://create-react-app.dev/docs/adding-a-stylesheet
https://create-react-app.dev/docs/adding-a-css-modules-stylesheet
https://create-react-app.dev/docs/adding-images-fonts-and-files

Imports in Webpack

- `import logo from './logo.svg';`
- `import logo from './logo.png';`
  - Will include the image reference inside the bundle (placed under static/media)
  - Small files are rendered inline
- `import './Button.css’;`
  - This component will use these CSS declarations
  - All CSS will be concatenated into a single file, but here we are stating the dependency
- `import styles from './Button.module.css’;`
  - Files ending with .module.css are CSS modules
  - Styles may be applied with `className={styles.primary}`
  - Class names are *renamed to be unique*: no conflict with other Components’ styles

https://create-react-app.dev/docs/using-the-public-folder

Why use imports

- Scripts and stylesheets get minified and bundled together to avoid extra network requests.
- Missing files cause compilation errors instead of 404 errors for your users.
- Result filenames include content hashes so you don’t need to worry about browsers caching their old versions.

- They are an optional mechanism. “Traditional” loading (with link, img) still works, if you save your files in the public directory

References

- Taming the State in React, Robin Wieruch (2017)
  http://leanpub.com/taming-the-state-in-react
- The Road to learn React, Robin Wieruch (2019)
  http://leanpub.com/the-road-to-learn-react

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/