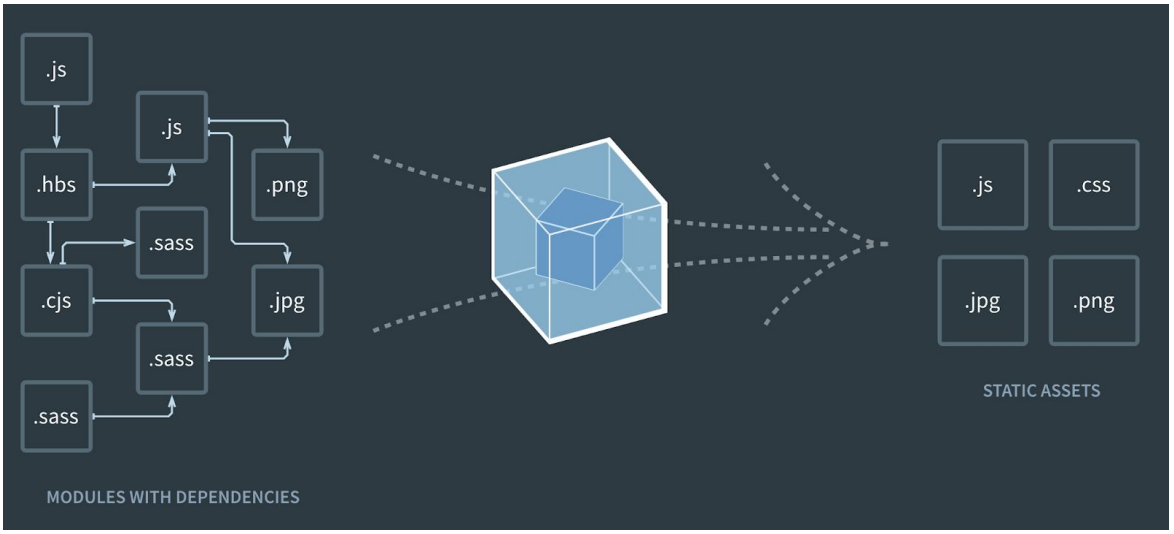
10. Deployment

Introduction

The goals of development and production builds differ greatly. During development, we want to have a good source mapping, localhost server with live reloading etc. On the other hand in production, our goal shifts to minified bundles, lightweight source maps and optimized assets in order to provide best performance.

Build

When using `create-react-app` (CRA) a lot of things are already provided, so that you can create a production build easily. You just need to run a `yarn build` command, which will create a build directory, with all the necessary files. Inside the *build/static* folder, there will be your JavaScript and CSS files. It may seem like an easy process, but a lot of things are happening under the hood. CRA uses webpack internally to build your project. Below is a simplified view of what is actually happening during the build process.



Styles and Assets

There are various options of styling the application. You can, for example, include pure CSS files alongside your React components, use SASS stylesheets, or CSS modules. No matter what option you choose, the styles need to be compiled into CSS and included in your build static files, or will be dynamically generated during runtime. During the build, there can be some PostCSS tools (like Autoprefixer) applied, that can optimize them, add vendor prefixes to help you ensure cross-browser compatibility.

```
/* Before auto-prefixing */
.example {
   transition: all .5s;
}
```

```
/* After auto-prefixing */
.example {
   -webkit-transition: all .5s;
   -o-transition: all .5s;
   transition: all .5s;
}
```

Static assets work similarly to styles. You often import these files directly into the JS modules and the underlying bundling engine will include them in the bundle. The imported file will be eventually translated into an actual path of the image.

```
import React from 'react';
// Tell webpack this file uses logo img
import logo from './logo.png';

console.log(logo); // /logo.84287d09.png

function Header() {
 // Import result is the URL of your image
 return <img src={logo} alt="Logo" />;
}

export default Header;
```

Typescript files

Since browsers understand only JavaScript, we again need to transform our code into a browser-understandable format. The transformation process includes compiling Typescript into JavaScript and omitting the type annotations, converting JSX, polyfilling/transpilling the code to ensure cross-browser compatibility and minifying the resulting code.

```
// ORIGINAL CODE
const App : React.FC = () => <div>Hello world</div>;

// TRANSPILED CODE
var App = function App() {
 return /*#__PURE__*/React.createElement("div", null, "Hello world");
};
```

Code splitting

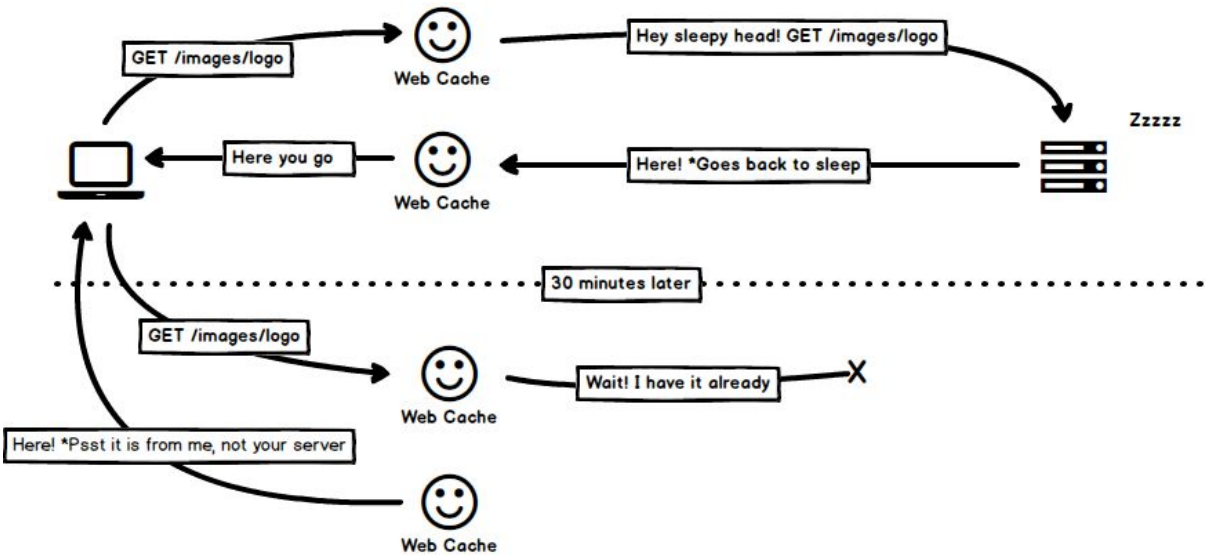
Build tools like Webpack in CRA compile your code into bundles, which allows for much greater optimizations and minification compared to keeping code in modules (each file is a ES6 module). Downside of this is that when a user wants to access your single page app, he/she needs to download ALL of it's bundled code even for parts he may never even go to. Code splitting is a technique that allows you to split your codebase into multiple bundles via dynamic imports that will be fetched only when necessary. This feature can have a major impact on load time when used correctly.

Caching and hashes

Since fetching resources over the network is both slow and expensive:

- Page needs to wait for them to be loaded,
- Limited mobile data plan charges money for the downloaded data.

We need to provide a way of limiting the unnecessary resource fetching. The solution to this issue is the browser's HTTP Cache. It is effective, supported in all browsers and does not require too much work to establish.



All HTTP requests that the browser makes are routed to the browser cache to check whether there is a valid cached response that can be used to fulfill the request. If it finds a match, the response is loaded from cache and there is no need to fetch them over the network.

Even though the HTTP Cache solved our problems of unnecessary resource fetching, it created a new problem. If we want to upload a new version of the resources, we need to make sure that the browser will use the new resource, rather than the cached one. To solve our new problem, we need to change the URL of the resource, every time we change the file, to download the version. Typically, this is done by embedding a fingerprint of the file, in its filename.

The process of providing fingerprint is part of the build process. If we want to use the “best of the both worlds” approach, we can split our code into an application code and a vendor code. Vendor code includes modules that were imported from node_modules and it often tends to change less frequently than our application code, and we will be able to enroll new features, without redownloading the whole codebase.
The production build, created by CRA will have following javascript files:

- `main.[hash].chunk.js`
  - Our core application code App.js, etc
- `[number].[hash].chunk.js`
  - Vendor code, or other code splitting chunks
- `runtime-main.[hash].js`
  - Webpack runtime logic files

Metadata and SEO

After your page is built and deployed, it may be accessed in many different ways and actual real people opening the page is just one use case. Features like preview links or indexability by search engines are expected to work and that’s something that frontend developers should properly set up. Getting deep into Search Engine Optimizations isn’t a part of this course (since React’s client-side rendering doesn’t play well with indexing bots) but as an example we can prepare at least some basics.

Nice starting point can be https://metatags.io/ where you can prepare basic meta tags for various sites to use when showing preview links.

```
<!-- Primary Meta Tags -->
<title>Meta Tags — Preview, Edit and Generate</title>
<meta name="title" content="...">
<meta name="description" content="...">

<!-- Open Graph / Facebook -->
<meta property="og:type" content="...">
<meta property="og:url" content="...">
<meta property="og:title" content="...">
<meta property="og:description" content="...">
<meta property="og:image" content="...">

<!-- Twitter -->
<meta property="twitter:card" content="summary_large_image">
<meta property="twitter:url" content="...">
<meta property="twitter:title" content="...">
<meta property="twitter:description" content="...">
<meta property="twitter:image" content="...">
```

Environment variables

There are multiple environments, where the application can be running. The environments can generally be divided into categories like *Local*, *Testing*, *Staging* or *Production*. The actual number and types of environment can vary depending on the character of the application and internal company processes. But there will always be at least a Local and a Production environment present. If we want to use the same codebase for each of them, we need to provide a way of setting environment variables that will control the behavior of our application. Example of things, we often control can be following:

- Redirects for API endpoints
- Turning on/off analytic data processing and error reporting
- Authentication keys

There are two ways of providing Environment variables to the application, the runtime, and build time embedding.

Runtime variables

Runtime variables can be beneficial, when you need to change a configuration during the run time. It allows you to build the application only once and deploy the same build to multiple environments, change the configuration after deployment, load configuration from external source. Additionally, it will save time, since the build pipelines often take a long period of time to finish. Since CRA produces only static bundle, it is impossible to read them at runtime. One of the approaches, to use runtime variables, is adding a placeholder to the index.html file and making the server replace them before sending the response. In the example below, the server that serves our application, will replace the `__SERVER_DATA__` with a runtime value. This approach is not used as often as build time variables, since it is more difficult to implement and maintain.

```
<!doctype html>
<html lang="en">
 <head>
   <script>
     window.RUNTIME_ENV = __SERVER_DATA__;
   </script>
...
```

Build time variables

Most of the time, using the custom build time variables is enough for the most common use-cases, since it is easier to implement. The most commonly used library is *dotenv*, which allows you to load variables from `.env` file into `process.env` object. In this section, we will describe ways how you can define environment variables inside the CRA environment,

which is using *dotenv* library internally.

By default, you have available the `NODE_ENV` variable, and any other variables starting with `REACT_APP_`. As mentioned above, these environment variables will be defined for you inside the `process.env` object. For example if you create a variable named `REACT_APP_FOO`, you will find it in your JS as `process.env.REACT_APP_FOO`.

Value of the `NODE_ENV` variable is assigned based on the way you run the application. In the table below, you can find the corresponding values based on the way you start (build) the application.

command	**process.NODE_ENV value**
`yarn start`	`'development'`
`yarn test`	`'test'`
`yarn build`	`'production'`

Just with the default `NODE_ENV` variable, you can easily write code that will be conditionally run based on the value.

```
if (!process.env.NODE_ENV || process.env.NODE_ENV === "development"){
 // dev code
} else {
 // production code
}
```

In CRA, you have two options for providing your own environment variables:

Temporary Environment variables in shell

This approach is temporary and will only live for the life of the shell session. The way of setting the variable differs, depending on the OS where you run your application. Example:

```
$ REACT_APP_TEMP_VAR=tempVar yarn start
```

Permanent Environment variables in .env file

In order to declare permanent variables, you need to create a file called `.env` in the root of your project. The structure of the file is simple:

```
REACT_APP_VARIABLE_NAME=variable_value
```

In CRA, you can define multiple .env files that will be evaluated based on the environment, where your application runs. The options you will use most commonly are following:

- `.env` - default
- `.env.development`, `.env.test`, `.env.production`. - environment-specific

Progressive Web App

In short, Progressive Web Apps are websites that can behave as native apps, work offline and make use of caching techniques to load faster and be more responsive. Many websites (like Twitter) are PWAs without you even realizing, just try to disable your internet connection and you will see that they still load without content.

PWA in Create React App

Create React App implements everything a PWA needs to function. All you need to do is initialize the provided service worker (since CRA 4 no longer in default template). You can try out the template by creating a new react app:

```
yarn create react-app my-app --template cra-template-pwa-typescript
```

You will find a new `src/service-worker.ts` and `src\serviceWorkerRegistration.ts` files containing default configuration of caching and everything required to register a service worker. In order to use it, it needs to be registered in the `index.tsx` file first.

```
serviceWorker.register();
```

Behind the scenes, CRA is using the Workbox library that you can fully utilize in customizing your service worker. If you want to know more about the concept of Web Workers API that the service worker is based on, here is further reading.

manifest.json

The web app manifest is a JSON file that tells the browser about your Progressive Web App and how it should behave when installed on the user's desktop or mobile device. A typical manifest file includes the app name, the icons the app should use, and the URL that should be opened when the app is launched.
You can read about it’s fields and description here.

Hosting options

There are many free hosting services where you can host your single page React application. We will take a look at Github's Github Pages service and Google's Firebase Hosting.

Github Pages

This is by far the easiest option (assuming you are using Github for hosting your git repository). Each repository can be published directly from your versioned source code.

Prerequisites are:

- Repository must be public (or with Pro account you can publish private repositories)
- Select where the source files are (branch and/or specific folder)

And that's it.
You can set it up in the settings tab in the web interface of your Github repository. By default, the page will be hosted on *https://GITHUB_USERNAME.github.io/REPOSITORY_NAME* url.

There is also one special case where if you create a repository named *GITHUB_USERNAME.github.io* it will be hosted on *https://GITHUB_USERNAME.github.io* directly.

To redeploy the page all you need to do is push a new commit to the GHPages source branch.

gh-pages package

To make it even easier for deploying CRA apps specifically, there is a `gh-pages` package which you can use to deploy your app with a single command.

```
yarn add -D gh-pages
```

After adding this package to your dev dependencies you need to make some changes to your `package.json`.

```
{
  // ...
  // homepage is optional with GITHUB_USERNAME.github.io special case
  "homepage": "https://GITHUB_USERNAME.github.io/REPOSITORY_NAME",
  // ...
  "scripts": {
```

```
    "predeploy": "yarn build",
    "deploy": "gh-pages -d build",
    // ..
```

Now by running `yarn deploy` your app will get built by react-scripts and pushed to gh-pages branch in your repository.

Conclusion

There are few negatives to keep in mind though. Deployed site is completely static which means that in order to achieve some server-side functionality you would have to look for other solutions.

Another problem is that if you want to use client side routing (React Router), you will need to find some workaround (HashRouter in React Router's case) that works around the fact that there is no way of setting up custom routes for GH Pages hosted sites.

Firebase Hosting

Second hosting option we will discuss is one provided by Firebase. This option is free* in a sense that it is limited to 10GB of storage and 360Mb of data transferred per day. But for smaller hobby projects it should be more than enough.

To deploy to Firebase Hosting you will need to install the `firebase-tools` package. Note we are using yarn global add instead of yarn add since we are installing a CLI tool and not a project dependency.

```
yarn global add firebase-tools
```

Note:
You also need to add the yarn bin directory to your path so this global CLI too can be easily run from your command line. You can see your yarn bin path by running `yarn global bin`.

Then you need to login to your Firebase account using the firebase tools CLI.

```
firebase login
```

Now we can initialize the hosting project. This will take us through a setup process, where you can configure everything required.

```
firebase init hosting
```

First choose the existing project option and select a firebase project you want your hosted site to be linked to.

```
? Please select an option: Use an existing project
? Select a default Firebase project for this directory: project-id (Project)
```

Then set the public directory to the build folder where `index.html` file will be since that is the output directory of `react-scripts build`. Also we want to set up single page application routing.

```
? What do you want to use as your public directory? build
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
```

One of quite recent features is integration with Github repositories. It will automatically set up Github Actions that build and deploy your page when you commit to the repository.

```
? For which GitHub repository would you like to set up a GitHub workflow?
(format: user/repository) user/repository
? Set up the workflow to run a build script before every deploy? Yes
? What script should be run before every deploy? yarn && yarn build
? Set up automatic deployment to your site's live channel when a PR is merged?
No
? What is the name of the GitHub branch associated with your site's live
channel? master
```

Note:
This is the most basic setup and there are many ways to improve it. For example instead of deploying on every commit to `master` you can change it so it's deployed from a new `production` branch on PR merge.

Now that initialization is done you should see few new files added `firebase.json`, `.firebaserc`, few yml files setting up Github actions in `.github\workflows\` folder. After committing these changes your page should be automatically deployed by GIthub Actions and accessible on either *PROJECT_ID.web.app* or *PROJECT_ID.firebaseapp.com* urls.

Conclusion

Unlike GH Pages, Firebase Hosting works with SPA client-side routing which is nice. Also since you already need to have a Firebase project set up, you can more easily use other services such as Cloud Functions etc.
One thing to keep in mind though is that Firebase Hosting has limits for free accounts, which you most likely won't hit with some hobby project, but it's another reason to keep optimisations such as caching in mind.