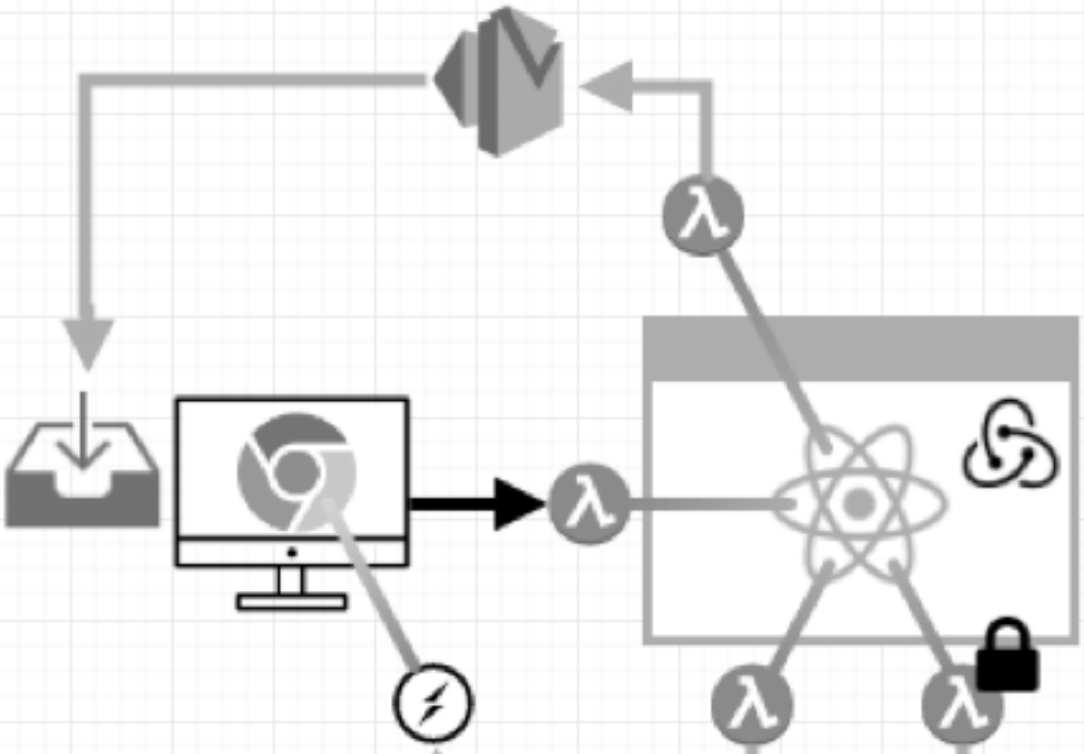
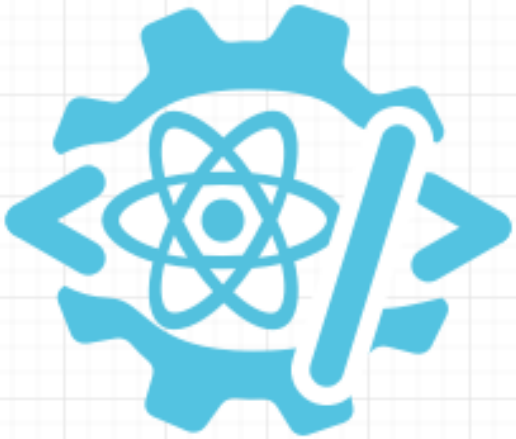
React-Architect

Full-Stack React App Development and Serverless Deployment



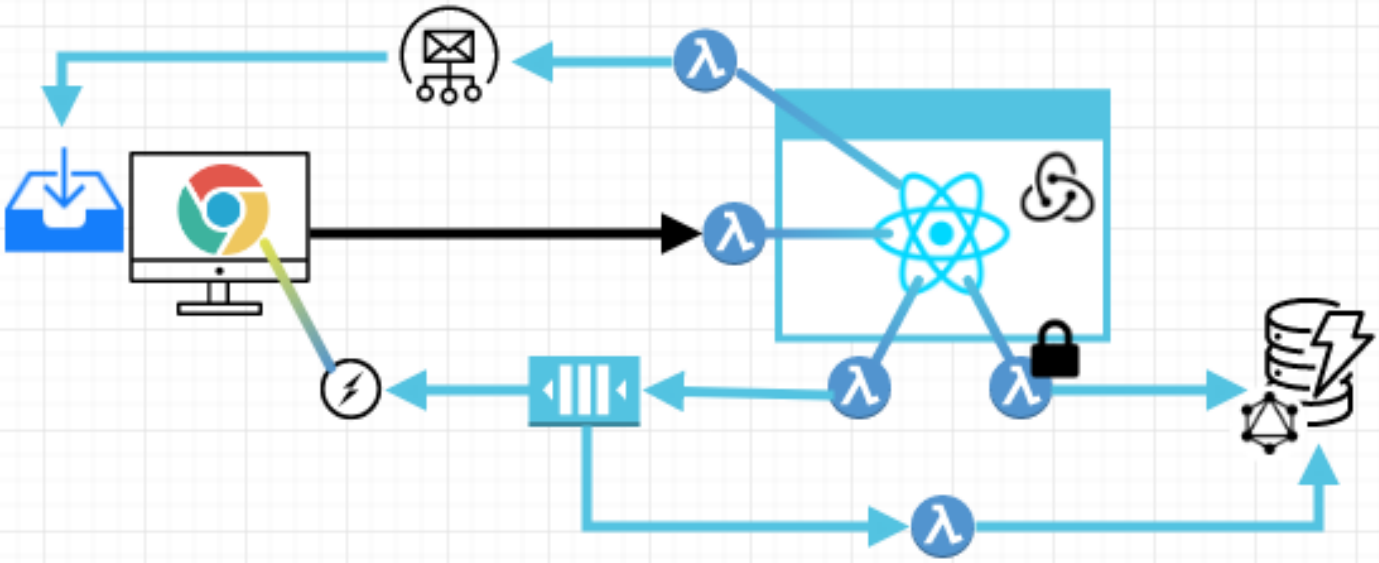


Dr. Frank Zickert

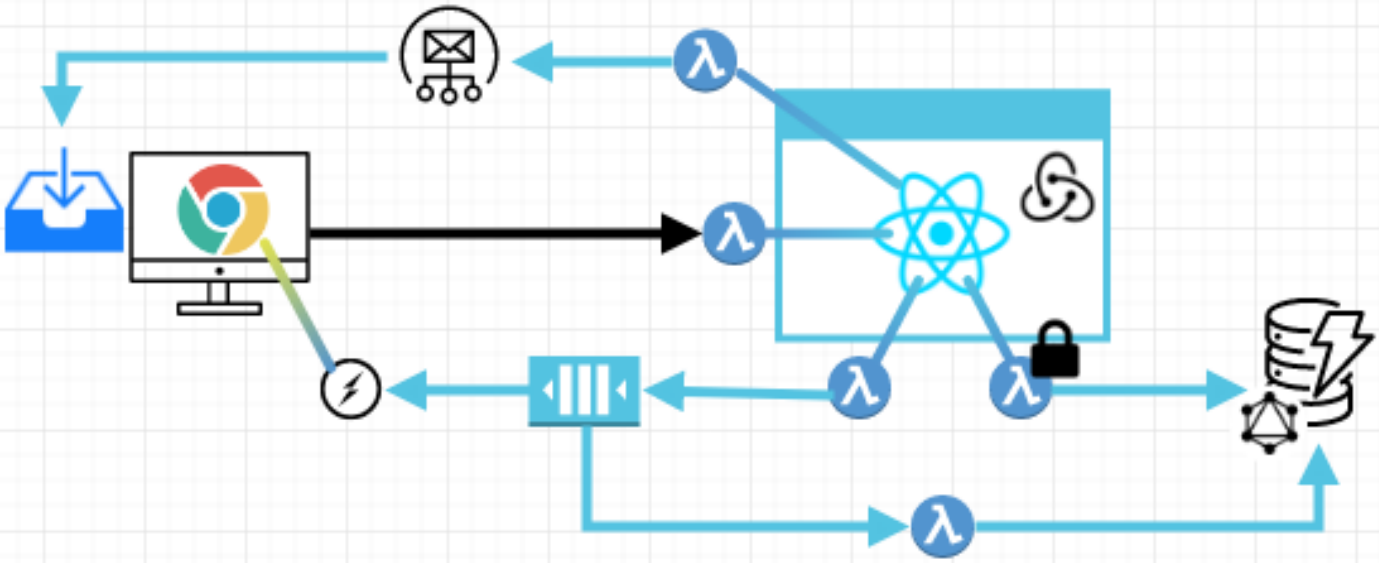
PUBLISHED BY REACT-ARCHITECT

www.react-architect.com

Release 1.0, May 2020



Contents



1. Introduction

Welcome to *React-Architect: Full-Stack React App Development and Serverless Deployment*. This book is your guide to creating full-stack applications with React.

React is a great library to develop web-applications with. It is easy to get started with, yet powerful. It is more than a library that supports building nice user-interfaces. With React, you can configure the infrastructure of your app, style your components, and implement business logic.

This book aims to be your comprehensive guide on solving problems from end-to-end. You'll not only learn how to write the code of a full-stack application. But you'll also learn what it takes to think and act like a full-stack developer.

This book ships with lots of practical solutions to real-world problems. We'll go through all the code examples in detail. We'll look at how the code works. You'll learn how to create full-stack applications with different architectures. In a practical and applied manner.

By the time you finish this book, you'll be well equipped to create applications that solve your problem at hand. You'll have gained hands-on knowledge that you'll need to be successful in the real world.

You will have a deep understanding of full-stack software architectures. You will no longer be limited to creating Single-Page-Apps. You won't hack together a bunch of code snippets anymore. But you'll be able to create production-ready, well-structured solutions to real-world problems.

Find a comfortable spot, open up your IDE, and start your journey to full-stack React mastery.

1.1 Why Should I Even Bother With Full-Stack Development?

Writing code has long been the biggest challenge of software development. How do you combine if-then-else-structures with loops to solve a business problem?

The user interface design was done on the side. The developer was the decisive factor. What she was able to build was what the software looked like. Some developers had talent concerning user interfaces. Some did not.

This changed with the rise of the internet. Graphical user interfaces moved into focus. Even more importantly, those user interfaces did not need to be programmed. The Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) enabled designers to build websites. These designers were technically skilled. Of course. But they did not need to be programmers.

At the time, JavaScript was not mandatory. Not very long ago, the company I am working for distinguished "front-end" developers. In those who do the graphical design with HTML and CSS (only). And in those who are capable of working with JavaScript.

JavaScript development was a hacky concept. Front-end developers manipulated the Document Object Model (DOM). That is the page's hierarchical structure of the HTML-elements. They added, removed, and changed the classes and the other attributes of an HTML-element. Seen from the eyes of a software engineer, this was all hacky! It did not have anything to do with the craft of software engineering.

Would you let a JavaScript developer implement business logic? Would you let her do something you consider critical for the success of your business?

For heaven's sake, no!

Implementing business logic was left to the back-end developers. These developers took the styled HTML and inserted it into the business logic that they wrote. PHP became popular for making it quite easy to render HTML on the server-side.

But something happened!

It may have been caused by the introduction of HTML5 and "Asynchronous JavaScript and XML" (AJAX). It may have been caused by the advent of Ecma Script. Websites started to act and behave more like applications. An increasing proportion of the business logic was closely coupled to the user interface. With the greater responsibility, frameworks evolved in JavaScript. Front-end development lost its hacky image and became software engineering, too.

Supposedly, front-end development became more of software engineering than back-end development is! Back-end got reduced to work with databases, provide the data through APIs, and take care of DevOps. They cope with all the things that require interconnectivity and integration.

But moving business logic to the front-end did not free the capacities at the back-end. For plausibility, security, and integrity, all the logic still had to run at the server-side. Because the front-end could be hacked by attackers. The code had to be duplicated!

Of course, writing code twice comes with problems. There is an avoidable effort of doing things twice. And there is the risk of inconsistencies between the logic at the front-end and the back-end side.

Full-stack development promises to avoid duplicate code and thus, prevent those inconsistencies.

There is a good chance that this promise will be kept.

JavaScript runs on any modern browser. With Node.js, it runs on the server, too.

React.js removed the boundaries between HTML and JavaScript. It uses the Javascript Syntax Extension (JSX) that complements basic Javascript source code with HTML.

Styled-Components added a convenient way of styling your React-components. There's no need for global style sheets anymore.

With Infrastructure-Components, you can integrate REST-services into your React app. As well as database schemes and queries. You can even configure your serverless infrastructure.

Now, you can use React from the very front-end of your application to the back-end and even DevOps. Within the same code. Without duplicates.

The technology stack has never been so homogenous. It has never been easier before to become a full-stack developer.

With the technology stack moving closer together, the demand for full-stack developers even rises. Consider isomorphic React apps. These are apps that use the same React-components to run on the server and the client-side. If you considered yourself as either a pure front-end or a back-end developer, you would not be up for that challenge.

This is less of a programming challenge. It is not about learning many different programming languages. This is rather a challenge to the skills and the mindset.

Creating intuitive user interfaces requires more than HTML, CSS, and JavaScript. You need to have the ability to craft compelling user experiences.

Writing business logic requires more than combining if-then-else-structures with loops. You

need to understand the implications of the logic you implement.

Writing full-stack applications requires you to understand software architectures. You need to understand how to compose your software. The code of your front-end and back-end might be identical. But it matters whether the code runs on the front-end or the back-end. It matters for performance, security, and sometimes even meaning.

The demand for full-stack developers no longer results from the businessmen's urge to hire fewer developers who can do more. The demand for full-stack developers stems from the increasing power of the tools. React is no longer a front-end library. Today, you can use React to write back-end software and to configure your infrastructure.

Full-stack development no longer requires multiple programming languages. But it still requires versatile skills. It requires you to think and act like a full-stack developer.

React-Architect: Full-Stack React App Development and Serverless Deployment is your guide on this journey. You'll learn how to think and act like a full-stack developer.

1.2 Who This Book Is For

Are you interested in Full-Stack React Development?

Then, I wrote this book for you! It does not matter whether you are a developer, a student, or even a project manager. It does not matter whether you are just getting started with JavaScript or whether you're already a senior front-end engineer.

Just Getting Started with JavaScript and React?

Don't worry. We'll go through the basics of web-app development with JavaScript and React. You'll be fine as long as you know the basic JavaScript syntax and feel confident with functions, if-then-else, and loops. We will cover all the other techniques like lambda-functions and React-hooks. It'll be great if you are familiar with basic HTML and CSS structures. But you don't need to be an expert in it.

We will introduce the advanced programming concepts step-by-step. Thus, as a novice developer, I'd suggest you read the book in its order. Regardless of your skill level, trust me, you will not get left behind. By the time you finish this book, you'll be proficient in full-stack web-app development.

Already a Senior Front-End Engineer?

This book isn't just for beginners – there's advanced content in here too. As a senior front-end engineer, you most likely know what it takes to manipulate the DOM. You already do 'async' API calls. And you have your Redux-store well-managed.

This book will complement your skills beyond programming. We'll cover how to think like

a full-stack developer. How do you write and secure your endpoints? How do you structure your data in the database? How do you authenticate your users?

What if I don't write code at all?

React-Architect: Full-Stack React App Development and Serverless Deployment contains more than code. We'll cover all many more aspects of full-stack app development. Does the app satisfy the non-functional requirements? Does it scale? What are the implications of different software architectures?

A compelling journey

This book is for you, even if you just want to read about full-stack React development.

Maybe you want to start with the background and come back to the code later? Here's my confession: This is what I do. I read textbooks during my holidays. While I'm sitting in the mountains or lying on the beach (unfortunately, this isn't my most frequent way of reading). I read textbooks on the train while I commute to work. And I come back to the code later.

1.3 Book Organization

React-Architect: Full-Stack React App Development and Serverless Deployment is your comprehensive end-to-end guide to full-stack React development with lots of code.

Throughout the book, we'll create full-stack React applications. We'll select the architecture that best caters to the needs of the problem at hand. And we'll explain and discuss the details of our software architectures.

Step-by-step, we'll introduce new concepts, programming techniques, libraries, and components.

This book combines practical, hands-on parts with the theoretical background. I recommend that you try to work with the code yourself. I'd even suggest that you change the examples. Make them serve your individual interests or business needs. Likely, you'll run into some problems. But solving problems is what software development is all about. It will add to your experience.

Don't worry. I don't leave you alone with the problems. I am convinced that the chapters will cover most of the problems you are dealing with. And if not: please let me know. I'll do my best to add another chapter that takes up this problem and proposes a solution.

1.4 Let's Start Thinking Like a Full-Stack Developer

Your team and you just deployed your application to production. Everybody seems to feel sort of relieved. But before the big party starts tonight, someone asks: "How many visits do we have today?"

The answer to this question is a single number. But getting this number requires a full-stack app! Before we start developing this app, let's do some thought work.

We start with the definition of "today". I am not kidding!

"Easy", you say? "It is the period from 12 am (or 0:00) to 11:59 pm (or 23:59) at the current day, month, and year." This is correct!

How do you measure "today"?

"This is easy, too. I use the JavaScript Date class. The following snippet prints out the current date. We can use this directly in our web-app!"

Listing 1.1: Printing the current date

```
const datestring = (d) => (
  d.getFullYear()
  + "-" + (d.getMonth()+1)
  + "-" + d.getDate()
);
console.log(datestring(new Date());
```

Unfortunately, the internet is some kind of global thing. The visitors may live in different time zones. This snippet works with each visitor's local time.

What's the consequence of working with the visitors' local times?

Let's consider the difference between the New Zealand Kiwi and the Hawaiian monk seal.

Figure 1.1: A Kiwi bird and a Hawaiian monk seal

The Kiwi is an early bird who gets up at 12 am and visits your app. This is 12 am at New Zealand Standard Time (NZST).

The seal looks like a late riser who visits your page at 11 pm. This is 11 pm Hawaii Standard Time (HST) on the same day.

NZST is twelve hours before UTC that is ten hours before HST. Thus, 12 am HST is 22 hours later than 12 am NZST. Additionally, the seal visited your app at 11 pm. This is 23 hours later than 12 am. In total, the seal visits your page 45 hours later than the Kiwi. But it counts the same day!

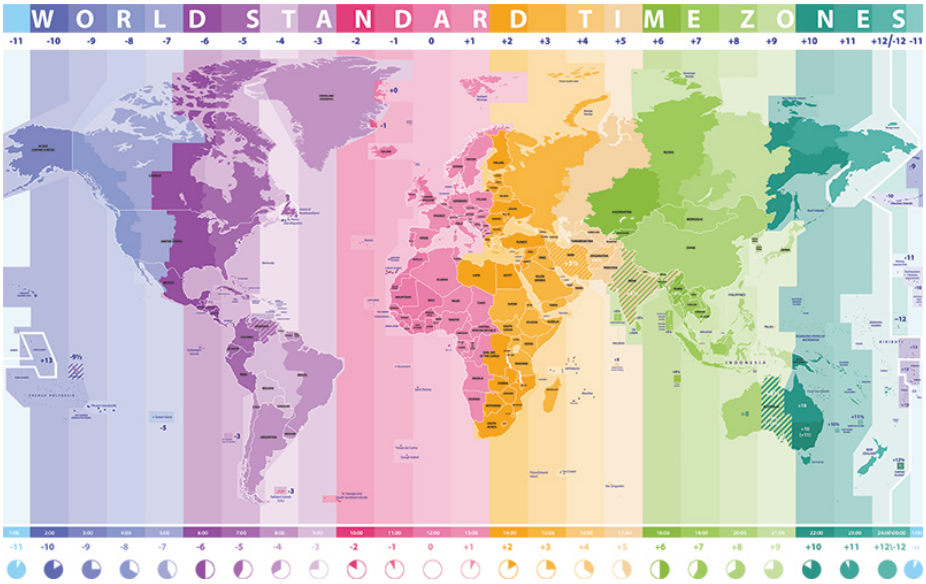


Figure 1.2: The time zones of the World

What if the Kiwi and the seal visited your app concurrently. The seal just had lunch on Sunday at 11 am (HST). The Kiwi just had breakfast at 9 am (NZST), the next Monday! These concurrent visits would add to the counts of two different days!

Are you warmed up? Let's talk about the temporal paradox!

The marketing team launches a new campaign. Of course, they want to target the emerging markets in the far-east. Ads run from Monday morning at 9 am Japan Standard Time (JST, +9h to UTC).

The campaign is a huge success and goes viral. It spreads around the world. You get a lot of visits from your US-based customers, too.

But wait, it is 8 pm the previous Sunday in New York (EDT, -4h to UTC). It is 5 pm the previous Sunday in Los Angeles (PDT, -7h to UTC). Your app gets a lot of visits on the day before the campaign started!

You see the effect prior to the cause... weird. Isn't it? Don't get lost in temporal paradoxes!

How do we prevent ourselves from the temporal paradox?

Figure 1.3: A starship appears from an anomaly, Copyright Paramount Pictures

"Just pick a single time-zone for all of the visits, stupid", you say?

This is a good idea! But we're still getting the time from the visitors' browsers, right? What if the visitors got their clocks wrong? What if they changed the date on their computers? These visitors would skew our visit count of today!

"You're doing something wrong here", you say? You're right, again! But the problem is not related to the code! It is related to the mindset that I applied.

I applied a pure front-end perspective.

Do you hear the back-end-developers? "Never trust the front-end!" They're not offensive. This phrase means that you need to treat data you get from the front-end carefully.

Usually, this phrase implies that there might be security issues with the front-end data. An attacker could replace your front-end and send malicious code instead of the data you expect.

But in our example, we don't even assume any bad intent. The visitor simply has a wrong time and date setting in her browser.

Let's apply a back-end perspective, for a change?

The first thing we need is a trust-worthy instance. It can be a server or some other computation resource in the cloud. The important point is that it is under our control.

We can use the following code directly at our back-end:

Listing 1.2: Printing the current date

```
const datestring = (d) => (
  d.getFullYear()
  + "-" + (d.getMonth()+1)
  + "-" + d.getDate()
);
console.log(datestring(new Date());
```

Yes, it is the identical code from above. But there is a huge difference in meaning and implication. Now, we take the local time of the server. This is the same for all visitors.

Two concurrent visits add to the count of the same day. Other events, such as a marketing campaign, fit seamlessly into the time series. We prevent temporal paradoxes.

Don't brand me as a back-end sympathizer. Because I am not. I'll prove it right away.

Let's come back to our question: "How many visits do we have today?"

What if this number did not only matter to us developers? What if it mattered to our visitors, too? Let's assume we provide booking services. Like Airbnb or a hotel reservation platform. We want to show the number of visits that a certain place or hotel got today.

Let's take the count from our back-end. Our back-end time is UTC (+0). When our New Zealand Kiwi visits our app at 11 am (NZST, +12 to UTC), she sees yesterday's stats. Because the back-end time is 11 pm UTC. As a result, the number of today's visits is high. Then, at noon in New Zealand, our back-end counter switches days. When the Kiwi returns to the page later that day, she sees a number that is lower than before. For the same day!

This is no good user experience, is it? Your counts may be correct, technically. But are they correct in the given context? Do these counts satisfy the purpose? All Kiwi visitors in the afternoon would get the feeling that there's not much traffic on the app. Or that the hotels are not in high demand. If they visited the app after midnight (NZST), they would see much higher counts. Would they believe that a hotel is in such high demand right after midnight?

"You're doing something wrong here", you say? And once again, you're right! The problem is still not related to the code! It is related to the mindset that I applied.

I applied a pure back-end perspective.

If you want to become a Full-Stack React Developer, then you need to stop thinking like a front-end-developer or a back-end-developer. You need to start thinking like a full-stack developer.

A full-stack developer uses all the tools in her toolbox. This is front-end and back-end!

When we collect the data, we can use the trustworthiness of our back-end. We can verify the visitor's local time and her time-zone with our back-end time. Does it all add up or is it inconsistent? We can store all this data.

When we display our data, we can use the context of our front-end. What is the location and time of our visitor? With the raw data at hand, we can calculate the count of today's visits from an NZST-perspective for the Kiwi. And we can calculate the count of today's visits from an HST-perspective for the seal.

Neither the Kiwi nor the seal would see a counter reset during the day. We could even take into consideration the time-zone that the hotel on the visited page is on. If we liked it.

Full-stack enables us to achieve more than we could with a pure front-end or a pure back-end solution! But it requires us to think outside of our comfort zone. We need to think beyond the routines we got used to. And we need to think critically about what we want to achieve. Because with the more power we get, the more responsibility we have.

1.5 Programming Paradigm, Software Stack and Libraries

This book follows the functional programming paradigm. In functional programming, we organize our code as a hierarchical composition of functions. Like its mathematical counterpart, the value of a function depends only on its arguments. With the same value for its arguments, a function always produces the same result.

This is in contrast to imperative programming. There, not only a function's arguments matter. But the current state can affect a function's resulting value.

Functional programming aims to avoid such side-effects. Because side-effects result in unexpected behavior. They complicate understanding the code.

JavaScript and React have vibrant communities. The number and quality of open-source libraries increase every single day. Libraries prevent us from reinventing the wheel. In this book, we'll make intensive use of the following libraries:

- React
- React-Router
- React-Apollo (GraphQL)
- Styled-Components
- Express
- Infrastructure-Components

What's the value of an application, when you can't use it? Of course, you'll need to deploy an application to a runtime environment. This is an environment outside of your local development machine. In this book, we will deploy every single example. We'll make deployment a natural habit. It prevents us from the "works on my machine"-problem.

We'll use the serverless-framework. We host our applications on Amazon Web Services (AWS). We use:

- Lambda Functions
- API Gateway
- Simple Storage Service (S3)
- DynamoDB
- Simple E-Mail Service (SES)

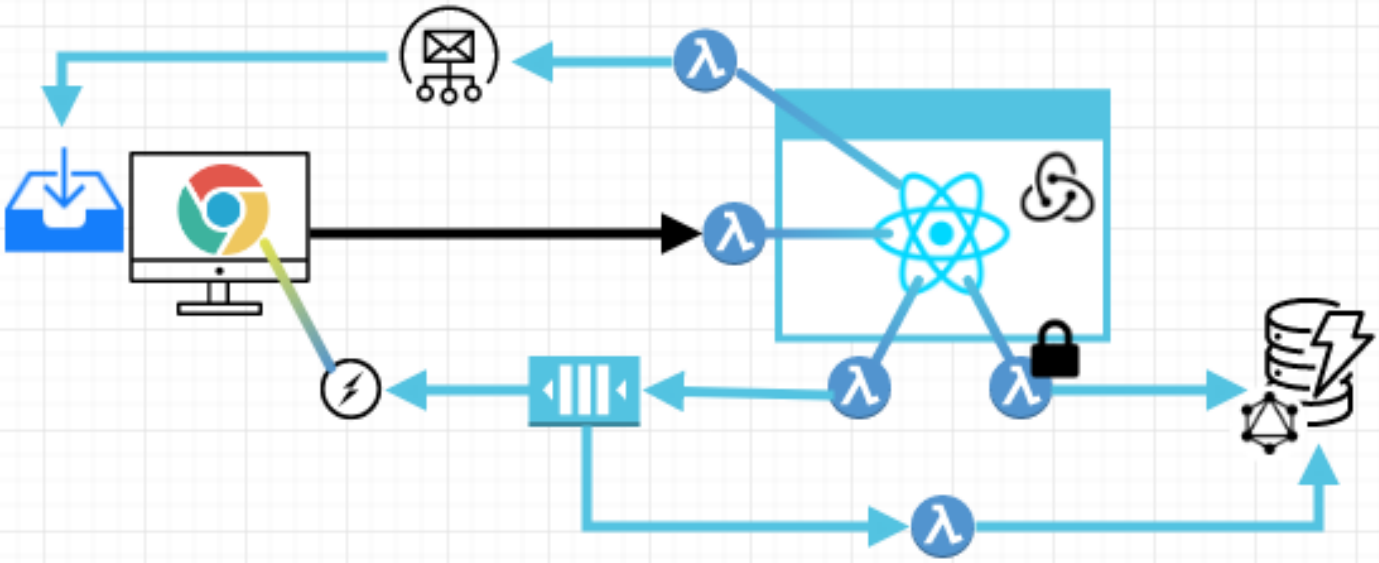
You may be afraid of having to fight a lot with configuration files. Don't. We won't spend hours and days on complex configurations. We'll specify the architecture of our apps with React. We'll make use of the "Architecture-as-a-Function"-paradigm.

Haven't you heard of `{insert arbitrary technology here!}` yet?

Yes. This software stack is opinionated. But it serves the purpose of this book: **Full-Stack Development and Deployment**.

This stack lets us use a single programming language. This is JavaScript.

If you're not convinced, yet. May I ask you for some leeway? I don't want to argue that this stack is better than another one. But I'd like to demonstrate that this stack lets us create and deploy a full-stack easily.



2. Get Started

2.1 Create the app

Enough with the theory! Let's build our example from above! We create and deploy an app that counts the visits at the back-end, stores the data in a database, and shows the number of today's visits, depending on your user's timezone.

There are manifold ways of how you can create a React app. `create-react-app` is the most famous one. It supports you when you develop a single-page-app. But it does not create a full-stack app for you. And it leaves you unsupported when it comes to deploying your app.

We use the library Infrastructure-Components. These React-Components let us define our infrastructure architecture as part of our React-app. We don't need any other configuration like Webpack, Babel, or Serverless anymore.

> In this book, we use serverless AWS resources. Currently, Amazon provides generous free tiers for the resources used in this book. However, Amazon may charge you for the resources you use. Please make sure to control the AWS resources you use. If you don't have an AWS account, you can create it at https://aws.amazon.com.

Infrastructure-components aim to automate as many things as possible. However, there are a few things we need to do manually. Because these things require explicit confirmation. For our first example, we need a technical user (with programmatic access / API-key).

In your AWS-console, open the IAM menu and create a new user with the following policy:

Listing 2.1: The policy of your technical user

```
{
 "Statement": [
  {
   "Action": [
    "s3:*",
    "apigateway:*",
    "lambda:*",
    "logs:*",
    "cloudformation:*",
    "cloudfront:*",
    "acm:ListCertificates",
    "route53:ListHostedZones",
    "route53:ListResourceRecordSets",
    "route53:ChangeResourceRecordSets",
    "route53:GetChange",
    "iam:CreateRole",
    "iam:DeleteRole",
    "iam:DeleteRolePolicy",
    "iam:GetRole",
    "iam:PassRole",
    "iam:PutRolePolicy",
    "dynamodb:CreateTable",
    "dynamodb:DeleteTable",
    "dynamodb:DescribeTable",
    "dynamodb:DeleteItem",
    "dynamodb:GetItem",
    "dynamodb:PutItem",
    "dynamodb:Scan",
    "dynamodb:Query",
    "execute-api:ManageConnections",
    "cloudfront:UpdateDistribution"
   ],
   "Effect": "Allow",
   "Resource": "*"
  }
 ],
 "Version": "2012-10-17"
}
```

You'll get an `AWS_ACCESS_KEY_ID` and an `AWS_SECRET_ACCESS_KEY`. Put these into the `.env`-file in your project root.

Listing 2.2: The content of your `.env`-file

```
AWS_ACCESS_KEY_ID=***...***
AWS_SECRET_ACCESS_KEY=***...***
```

You can create the files we use in this project manually or you can download a customized boilerplate project at www.infrastructure-components.com. Just enter the email address you specified when downloading this book. You'll get a '.zip'-file for download.

You'll find the complete source code in this GitHub-Repository

Unpack the `.zip`-file to a folder of your choice. You'll get the following file structure.

Listing 2.3: The file structure of your boilerplate project

```
project/
|- src/
|   |- index.tsx
|- .env
|- .gitignore
|- package.json
```

The `.env` file contains your credentials. Please keep it safe. Because this file is all you need to get access to your project. Do not put it into a repository, like git.

The `.gitignore` file contains some basic rules. It keeps your git-repository free from files you don't want to version control. Like the `.env`. The following snippet provides a basic version.

Listing 2.4: A basic version of the `.gitignore`-file

```
# Infrastructure-Components
.infrastructure_temp/

# Environment variables
*.env

# Serverless files
.serverless
.dynamodb
.s3

# output-folders
container/
build/
dist/
node_modules/
v8-compile-cache-0/
```

The `package.json` describes your project from a technical perspective. In the `scripts` section, you should have a single definition: the `build`-script.

Further, the `package.json` specifies all the dependencies of your project.

Listing 2.5: The `package.json`

```
{
 "name": "visit-count",
 "version": "0.0.1",
 "scripts": {
  "build": "scripts .env build src/index.tsx"
 }
 "dependencies": {
  "@apollo/react-hooks": "^3.1.3",
  "@babel/polyfill": "^7.8.0",
  "express": "^4.17.1",
  "graphql": "^14.5.8",
  "graphql-tag": "^2.10.1",
  "infrastructure-components": "^0.3.10",
  "isomorphic-fetch": "^2.2.1",
  "react": "^16.12.0",
  "react-apollo": "^2.5.8",
  "react-dom": "^16.12.0",
  "react-router": "^5.1.2",
  "react-router-dom": "^5.1.2",
  "react-helmet": "^5.2.0",
  "request": "^2.88.0",
  "serverless-http": "^2.1.0",
  "styled-components": "^4.4.1"
  "universal-cookie": "^4.0.1"
 },
 "devDependencies": {
  "infrastructure-scripts": "^0.3.11",
  "serverless-dynamodb-local": "^0.2.38",
  "serverless-offline": "^4.9.4",
  "serverless-pseudo-parameters": "^2.5.0",
  "serverless-s3-local": "^0.5.3",
  "serverless-single-page-app-plugin": "^1.0.2"
 }
}
```

You'll need `Node.js` and `npm` installed on your development machine to get started.

Further, running the DynamoDB locally (offline on your development machine) requires Java 8 JDK. Here's how you can install the JDK.

You can install the specified libraries with the command `npm i`. (Run this command from the root of your project folder.)

If you use `yarn` instead of `npm i`, please use:
`yarn install --ignore-engines`

2.2 Define the architecture

Let's create a Serverless isomorphic React app. This is a React app that combines an interactive front-end with server-side rendering. The following image depicts the basic architecture of a Serverless isomorphic app.

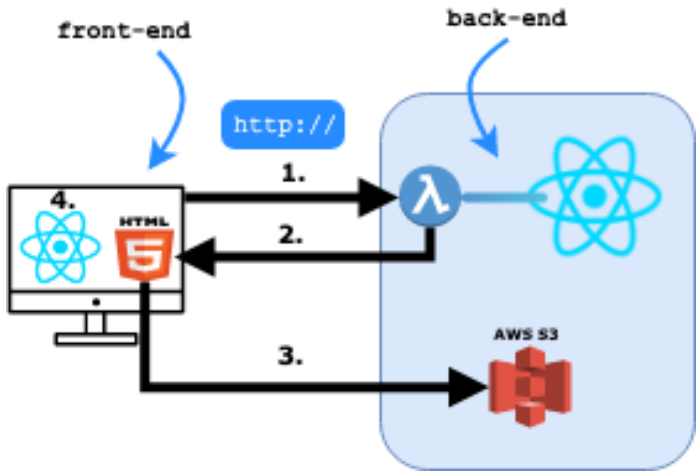


Figure 2.1: The architecture of a serverless isomorphic React app

This app works as follows:

1. The user's browser requests (over HTTP(S)) a page from your application. The back-end of your app runs on a Serverless Lambda function. This is an on-demand computation resource. AWS starts the Lambda function automatically when a browser requests a page of your app. The Lambda function stops again once it finishes processing the request.
2. The Lambda function renders (dehydrates) your React app into HTML. This HTML does not contain the JavaScript part of your app. The Lambda function returns the HTML as the response to the HTTP-request. Once the browser receives the response, it directly displays the HTML.
3. The HTML code contains a small script. This script downloads the JavaScript code of your React app.
4. The browser runs your React app locally. React combines (rehydrates) the existing HTML with the JavaScript code. This enables the user to interact with the page. It completes your React app in the browser.

React applications consist of components that you compose in JSX (`.jsx`-file ending). It complements basic JavaScript source code with an XML-like syntax. The file ending `.tsx` declares a JSX-file that supports TypeScript, too.

The XML-like components form a hierarchy. A hierarchy provides a clear and readable structure. Since your app consists of a hierarchy of components, it has a single top-level

component. You need to ‘export default‘ this component in the file `src/index.tsx`. This is the file the `build`-script of your `package.json` refers to.

Let’s create the most basic form of an isomorphic app. The following code is the full code of a simple isomorphic app that displays "Hello React-Architect!".

Listing 2.6: An isomorphic React app

```
import React from 'react';
import {
  Environment,
  IsomorphicApp,
  Route,
  WebApp
} from 'infrastructure-components';

export default (
  <IsomorphicApp
    stackName = 'visit-count'
    buildPath = 'build'
    assetsPath = 'assets'
    region='eu-west-1'>

    <Environment name='dev' />

    <WebApp
      id='main'
      path='*'
      method='GET'>

      <Route
        path='/'
        name='React-Architect'
        render={(props)=> (
          <div>Hello React-Architect!</div>
        )}/>

    </WebApp>
  </IsomorphicApp>
);
```

In Infrastructure-Components-based projects, the top-level-component defines the overall architecture of your app. This is an `<IsomorphicApp/>`.

The `<IsomorphicApp/>`-component takes only a few parameters. The `stackname` is the (arbitrary) name of your app. Please use only lower case characters and hyphens for the name serves as an identifier within AWS (when you deploy your app). The `buildPath` is the relative path to the folder where you want to put the build-resources. You may want to add this name to your `.gitignore` file to keep your repository free from compiled files.

The `assetsPath` is the relative path to resources at runtime. The `region` is the AWS-region you want your infrastructure to reside after deployment

The sub-components (children) of the `<IsomorphicApp />` refine the app's behavior and add functions. Our app requires a `<Webapp />`-component as a child. This is the React web-app that combines server-side-rendering with client-side-rendering.

The `<WebApp/>`-component requires the following parameters. The `id` is the (arbitrary) name of the web app. It lets you start the web-app locally in hot-dev mode: `npm run {id}`. Please use only lower case characters. The `path` specifies the relative paths that this app will serve, use "*" as a placeholder for "all". The `method` is the HTTP-method that this app will work with, e.g. "GET", "POST", "PUT", "DELETE".

Your `<WebApp/>`-component consists of `<Route/>`- components. These are the pages of your app. They work like the `<Route/>`s of `react-router`. They contain the user interface of your app. Everything you see. Everything you can interact with. The `path` of a route specifies the page's path in the URL. This page should be within the scope of the `<WebApp/>`'s path. The `name` specifies the title of the page. The `render`-function lets you render your page's React-component.

In our JSX-components declaration, we define a JavaScript code-block through a pair of curly-brackets ("{" and "}"). In this code-block, we use an arrow function. An arrow function is an anonymous function. This is simply a function without a name, which we implement directly at the place of its use.

An arrow function has the following form:

Listing 2.7: The structure of an arrow function

```
1 (params) => (returned value)
```

The `render` function provides `props` as a parameter. But right now, we don't use the `props`. We simply render a basic HTML-`<div/>` component.

2.3 Run your app locally and deploy it to AWS

Your `<IsomorphicApp />`-component has an `<Environment />` as a child. The `<Environment/>`-component defines a runtime of your app. You can start and deploy each `<Envrionment/>` separately.

The build-command (`npm run build`) prepares your project environments. You only need to run it when you add an `<Environment />`. Once the `build`-script finishes, it adds three more scripts to your project-configuration (to the `package.json`) for each `<Envrionment/>`.

These are:

- `npm run {webapp-id}` starts your `<WebApp/>` in hot-development-mode (replace {webapp-id} with the `id` of your web-app). It runs at `localhost:3000`. This mode does not start the back-end. You won't have server-side-rendering. But in this mode, changes in your source code apply instantly. You just need to refresh your browser.
- `npm run start-{environment-name}` starts the whole software stack offline. It runs at `localhost:3000`, too. Open the developer tools of your browser and have a look at the Network-tab. Have a look at the preview of the `localhost` resource (this is the page your browser received from the server). You can see that it contains the HTML, already. You don't need to run the build-command if you change your source-code. But you need to stop the running stack (ctrl + c) and restart it in this mode to see your changes.
- You can also deploy your app with a single command: `npm run deploy-{environment-name}`. Replace `{environment-name}` by the name you specified in the `<Environment/>` component. Infrastructure-Components support multiple environments per project, e.g. a dev-environment and a prod-environment. Once you started the deploy command, the scripts create the whole infrastructure stack. The deployment may take a few minutes. You need an AWS account to deploy your app. Make sure your `.env`-file contains the AWS credentials.

2.4 Add a database

The `<DataLayer/>` creates a DynamoDB instance. This is a key-value database (NoSQL). It delivers high performance at any scale. But unlike relational databases, it does not support complex queries.

The following code snippet depicts the hierarchy of components and how the `<DataLayer/>` blends in.

Listing 2.8: The structure of components with a DataLayer

```
<IsomorphicApp />
|- <DataLayer />
|   |- <Entry />
|   |- <WebApp />
|       |- <Route />
|- <Environment />
```

The `<DataLayer/>` takes the `<WebApp/>` as a child. This has a simple reason. We want our `<WebApp/>` to have access to the database. Further, the `<DataLayer/>` takes `<Entry/>`-components as children.

An `<Entry/>` describes the type of items in your database. It has three fields: `primaryKey`, `rangeKey`, and `data`. This is important. Because you can only find items of an `<Entry/>`

in the database through its keys. You need the `primaryKey`, the `rangeKey`, or both. With this knowledge, let's have a look at our `<Entry/>` that stores the visits.

Listing 2.9: The structure of components with a DataLayer

```
<Entry
  id="visitentry"
  primaryKey="visittimestamp"
  data={{
    visitcount: GraphQLString
  }}
/>
```

We use the timestamp of a visit as the `primaryKey`. We only use this one key. But the timestamp of a visit is not unique. Concurrent visits at the same time have the same timestamp. Thus, the database would not be able to distinguish these two concurrent requests.

We could easily use our `rangeKey` and make them distinguishable. But we don't store any more data related to a visit. Our interest is the number of visits. Nothing else. Thus, we can count the visits at each unique timestamp just as well.

Why does the data-property have two pairs of curly-brackets {{ }} ?
The `data`-property takes a JavaScript-object as its value. We need the first pair of brackets because we want to integrate JavaScript code into the JSX-syntax. The second pair of brackets specifies a JavaScript-object.

There are two things we need to do with our data:

1. We want to count the requests made to the server. This means, each request counts as a visit. We use the clock of the server. This does not mean that we need to use the server's timezone. But when we calculate a local time, like UTC, we can trust our server's clock to be set correctly. We prevent from receiving wrong data from the front-end.
2. We want to show the number of today's visits. The user's local time determines the term "today" here.

2.5 Store the data

First, we need to count the visits and store this count in the database. A `<Middleware/>`-component allows us to run some code explicitly on the server upon a request. We add the `<Middleware/>` as a child to the `<Route/>`.

Listing 2.10: The structure of components with a middleware

```
<IsomorphicApp />
|- <DataLayer />
|  |- <Entry />
|  |- <WebApp />
|     |- <Route />
|        |- <Middleware />
|- <Environment />
```

Let's look at the details of how the `<DataLayer/>` and the `<Middleware/>` integrate into our app. In this example, all the other components are reduced to their names. Don't worry, at the end of the chapter, there is a listing of the full source code.

The `<DataLayer />` has an `id` (use an arbitrary lower case name). And it has two children: an `<Entry />` and the `<WebApp/>`.

The `<Entry/>` has an `id` and a `primaryKey`. The `primaryKey` specifies the key-name of an attribute. `data` is a JavaScript object. The keys of this object are arbitrary strings. These determine the fields of your data. In our example, `visitcount` is the only data we need. The value assigned to the key specifies the type of data. Currently, `GraphQlString` is the only supported type. But this is not a problem because we can easily transform it into numbers, too.

The `<WebApp/>` is the same as above. It has a single `<Route/>` as a child. The `<Route />` takes the `<Middleware/>` as a child. This `<Middleware/>` contains the code that runs at the server-side in the form of a callback-function. This function runs before React renders the `<Route/>`.

The callback function takes three parameters: `request`, `response`, and `next`. The `request` contains all the data we get from the browser. The `response` is a prepared object we can use to respond to the browser request. If we don't want to respond to the request directly, we can call the `next` function. This calls the next `<Middleware/>` or the `render` function of the `<Route/>` if there are no more `<Middleware/>`s (as in our example).

In the example, we wrapped the implementation of the callback into another function: `serviceWithDataLayer` that we import from `infrastructure-components`. Furthermore, the callback we provide has another parameter (at the first position): `dataLayer`. Adding this parameter is the whole purpose of the `serviceWithDataLayer`-function. It lets you access the `dataLayer` within the callback-function.

We declare the callback function as an `async function`. This makes the function asynchronous. It does not run on the main thread anymore. Within asynchronous functions, we can use the `await` declaration. This declaration lets the outer function wait for other asynchronous functions to resolve.

Listing 2.11: Integration of the DataLayer and the Middleware

```
import React from 'react';
import { GraphQLString } from 'graphql';
import {
  DataLayer,
  Entry,
  Environment,
  IsomorphicApp,
  Middleware,
  Route,
  serviceWithDataLayer,
  update,
  WebApp
} from 'infrastructure-components';

export default (
  <IsomorphicApp>
    <DataLayer id='datalayer'>
      <Entry
        id='visitentry'
        primaryKey='visittimestamp'
        data={{ visitcount: GraphQLString }}
      />

      <WebApp>
        <Route path='/'
          name='React-Architect'
          render={(props)=><div>Hello React-Architect!</div>}>

          <Middleware callback={serviceWithDataLayer(

            async function (dataLayer, request, response, next) {
              await update(
                dataLayer.client,
                dataLayer.updateEntryQuery('visitentry', data => ({
                  visittimestamp: utcstring(new Date()),
                  visitcount: data.visitcount ?
                    parseInt(data.visitcount) + 1 : 1
                }))
              );
              next();
            }
          )}/>
        </Route>
      </WebApp>
    </DataLayer>
  </IsomorphicApp>
);
```

With the `dataLayer` at hand and the ability to wait for other asynchronous functions, we can now store data in the database. We use the `update` function that we import from `infrastructure-components`. This function takes two parameters. The first parameter is the GraphQL-`client`-object. The second parameter is a GraphQL-query that specifies what we want to update.

We get the GraphQL-`client`-object directly from the `dataLayer`-object. The `dataLayer`-object also helps us to build the GraphQL-query. The function `updateEntryQuery` creates a GraphQL update query for us that we can feed directly into the `update`-function.

Of course, we need to tell the `updateEntryQuery`-function what kind of query it should create for us. The function `updateEntryQuery` takes two parameters. The fist parameter is the `id` of the `<Entry/>` that we want to update. This is `"visitentry"`. Please note, the entry-id does not specify a unique item in the database. But it specifies the type of the item. In our example, this means we update an item in the database with two fields: `visittimestamp` and `visitcount`. The value of the field `visittimestamp` serves as the `primaryKey` that identifies a unique item. The second parameter provides the data of the individual item we want to update. We do this in the form of another arrow function.

Listing 2.12: Example of a function providing the data

```
1 data => ({
2   visittimestamp: utcstring(new Date()),
3   visitcount: data.visitcount ? parseInt(data.visitcount) + 1 : 1
4 })
```

This function takes one parameter (`data`) and returns a JavaScript object. The `data` parameter contains the old data of the item (if one exists, otherwise it is empty). The returned object specifies the item's new data.

We need to specify the data of the `primaryKey` and the fields of the `<Entry/>`'s `data`-property. Thus, we need to specify values for our two fields: `visittimestamp` and `visitcount`.

The `visittimestamp` is the `primaryKey`. This means each value of the `visittimestamp` specifies a unique item in the database. If we specify the same value of an item that already exists in the database, we'll overwrite it.

This is exactly what we aim for. We don't want to create a new item for each request. But we want to count them. We create a string value using the function `utcstring`. We define this function in the following arrow function format:

Listing 2.13: The utcstring function

```
1 const pad = (n) => n<10 ? '0'+n : n;
2
3 const utcstring = (d) => (
4   d.getUTCFullYear()
5   + "-" + pad(d.getUTCMonth()+1)
6   + "-" + pad(d.getUTCDate())
7   + "-" + pad(d.getUTCHours())
8 );
```

This function returns a string with the current year, month, date, and hour, in the Coordinated Universal Time (UTC).

The `getUTCMonth` of the `Date` objects starts counting at `0`. We add `1` to get the usual numbers (January is `1`, not `0`)

Further, we include leading `0`s when we have a month, day, or hour smaller than `10`. This is what the `pad`-function does for us.

Since we run this code at the back-end, we assure the time to be correct. Regardless of whether the user has a different time set. That means all requests made within the same hour get the same value. This also means that there is only one item in the database per hour.

When we look at the timezones of the world, we can see that they differ by the hour of the day. (There are some exceptions, though.)

When we want to calculate the total number of today's visit, it is sufficient to know the hour of the day a visit belongs to. It does not matter whether a visit occurred at 10:01 or 10:59.

We only have a single item per hour in the database. Thus, we need to count the visits during this hour. But to count the visits of an hour, we need to know how many visits we had thus far. Here, the `data`-parameter of the `updateEntryQuery`-callback comes into play.

This `data`-parameter contains the fields of the item with the specified `primaryKey` before the update. If it exists, we can take the old `visitcount` and add `1`. If it does not exist, we do not have an item with this `primaryKey` in the database, yet. Thus, it is the first one of the current hour. We store `1` as the correct value.

The values in the database get treated as strings. You need to parse a value with `parseInt` if you want to use it as a number.

Once we updated the item in the database, we proceed with the `next()`-function. We let React render the component and display the data.

2.6 Display the data

There's only one step left. We want to show the number of visits today. But today should depend on the local time of the user.

We do this directly in the `render`-function of our `<Route/>`. The listing skips all other components that do not interest us now.

Listing 2.14: The render-function of the <Route/>

```
<Route
  path='/'
  name='React-Architect'
  render={withDataLayer((props) => {
    const setDate = (d, hours) => (
      new Date(d.getFullYear(), d.getMonth(), d.getDate(), hours)
    );

    return <div>
      <Query {...props.getEntryScanQuery('visitentry', {
          visittimestamp: [
            utcstring(setDate(new Date(), 0)),
            utcstring(setDate(new Date(), 23))
          ]
      })} >{
        ({loading, data, error}) => {
          return (
            loading && <div>Calculating...</div>
          ) || (
            data && <div>Total visitors today: {
              data['scan_visitentry_visittimestamp'].reduce(
                (total, entry) => (
                  total + parseInt(entry.visitcount), 0
                )
              )
            }</div>
          ) || (
            <div>Error loading data</div>
          )
        }
      }</Query>
    </div>
  })}>
  <Middleware/>
</Route>
```

You may have noticed that we wrap our `render`-function into `withDataLayer`. Like the `serviceWithDataLayer`-function we used before, this function provides access to the `dataLayer`. The function `withDataLayer` adds the `dataLayer` and some helper-functions to the `props`.

Before we discuss the calculation of today's visitors, let's first look at how we use the `dataLayer`. We use the `<Query/>`-component that we import from `react-apollo`.

The `<Query/>`-component takes the GraphQL-`query` and an optional `context` as its properties. The `withDataLayer`-function adds the helper-function `getEntryScanQuery` to the `props` of our component.

`getEntryScanQuery` takes the `id` of the `<Entry/>` whose items we want to get. The second parameter is a JavaScript-object. In this object, we provide as a key the name of our `primaryKey`. That is `visittimestamp`. The value we assign to this key is an array with two values. These specify the lower and the upper limit of the `visittimestamp` of the items we are looking for.

We spread `(...)` the returned values of the `getEntryScanQuery`-function into the `<Query/>`-component. This component manages the database request for us.

The `<Query/>`-component provides three values to its children: `loading`, `data`, and `error`. These values represent the current state of the database request.

- While the request runs, `loading` is defined.
- When the request returns with an error, the `error` contains the message.
- When the request is successful, `data` is defined. This is a Javascript object. The key `"scan_visitentry_visittimestamp"` contains our list of visitentries. This key results from the kind of query we use (`scan`), the `id` of the `<Entry/>`, and the filter (`visittimestamp`) we applied.

We get back the data as a list of items. We can calculate the sum of the `visitcounts` using the array's `reduce` function. This function iterates through each item in the list. It takes an arrow-function with two parameters. The first is the result (that we name `total`) and the current item (that we name `entry`). Whatever this function returns is the next `total`. We simply add the counts. Again, we need to `parseInt` the `visitcount` to treat it as a number. The last parameter we provide to the reduce function is the initial value.

Listing 2.15: Sum the visitcounts with reduce

```
1 data['scan_visitentry_visittimestamp'].reduce(
2   (total, entry) => total + parseInt(entry.visitcount),
3   0
4 )
```

In our query, we need to collect all the items that refer to today - from the user's perspective. We need to calculate the first and the last hour of the given day.

We first create `Date`-objects of the first and the last hour of today. We use the following `setDate` function.

Listing 2.16: The `setDate`-function sets the hours of a date

```
const setDate = (d, hours) => (
  new Date(d.getFullYear(), d.getMonth(), d.getDate(), hours)
);
```

It takes a `Date`-object and a number of hours as inputs. We use the local time in this function. We call it with `0` hours for the start of the day. We call it with `23` hours for the end of the day.

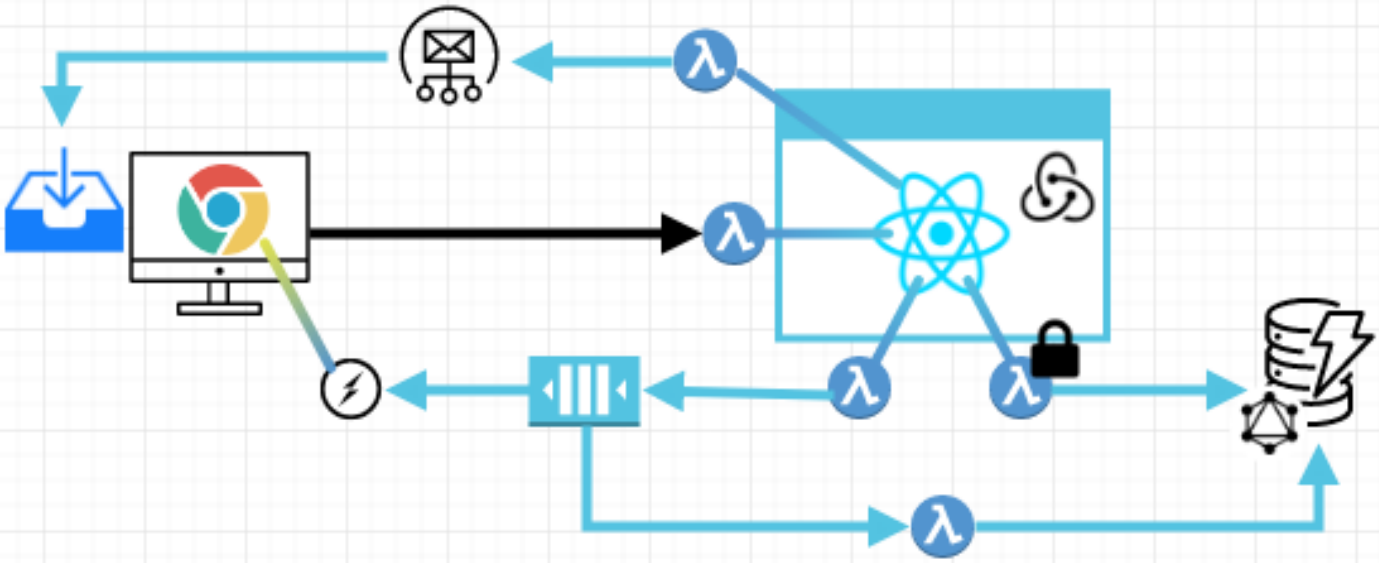
Then we feed these `Date`-objects into the `utcstring`. We get the string of the corresponding time in UTC. Since we also stored our `visittimestamp` in UTC, they are comparable.

This is it! Our first full-stack app. Here's the full source code of the program (`src/index.tsx`).

Listing 2.17: The full source code (src/index.tsx)

```
import React from 'react';
import { GraphQLString } from 'graphql';
import { Query } from 'react-apollo';
import { DataLayer, Environment, Entry, IsomorphicApp,
  Middleware, Route, serviceWithDataLayer, update,
  WebApp, withDataLayer, withIsomorphicState
} from 'infrastructure-components';

const setDate = (d, hours) => (
  new Date(d.getFullYear(), d.getMonth(), d.getDate(), hours)
);

const pad = (n) => n<10 ? '0'+n : n;
const utcstring = (d) => (
  d.getUTCFullYear()
  + "-" + pad(d.getUTCMonth()+1)
  + "-" + pad(d.getUTCDate())
  + "-" + pad(d.getUTCHours())
);
export default (
  <IsomorphicApp
    stackName = 'visit-count'
    buildPath = 'build'
    assetsPath = 'assets'
    region='eu-west-1'>

    <Environment name='dev'/>
```

```
    <DataLayer id='datalayer'>
      <Entry
        id='visitentry'
        primaryKey='visittimestamp'
        data={{ visitcount: GraphQLString }}
      />

      <WebApp id="main" path="*" method="GET">

        <Route
          path='/'
          name='React-Architect'
          render={withDataLayer((props) => (
            <div>
              <Query {...props.getEntryScanQuery('visitentry', {
                visittimestamp: [
                  utcstring(setDate(new Date(), 0)),
                  utcstring(setDate(new Date(), 23))
                ]
              })} >{
                ({loading, data, error}) => (
                  loading && <div>Calculating...</div>
                ) || (
                  data && <div>Total visitors today: {
                    data['scan_visitentry_visittimestamp'].reduce(
                      (total, entry) => total + parseInt(entry.visitcount), 0
                    )
                  }</div>
                ) || (
                  <div>Error loading data</div>
                )
              }</Query>
            </div>
          ))}>

          <Middleware callback={serviceWithDataLayer(
            async function (dataLayer, request, response, next) {

              await update(
                dataLayer.client,
                dataLayer.updateEntryQuery('visitentry', data => ({
                  visittimestamp: utcstring(new Date()),
                  visitcount: data.visitcount ?
                    parseInt(data.visitcount) + 1 : 1
                }))
              );
              next();
            }
          )}/>

        </Route>
      </WebApp>
    </DataLayer>
  </IsomorphicApp>
);
```



3. How Much Configuration Do You Need?

Having your app in the hands of its users is the primary goal of software development. If it is not, your app delivers no revenue. It does not make your users' lives easier. It provides no feedback about how useful it is.

This applies in general. Regardless of who the users are. It does not matter whether your app is for everybody. It does not matter whether it is for the employees in a certain company. It does not matter if this is your company and the users are your colleagues. It does not even matter if the app is just for you.

Your app delivers no value at all until it is in the hands of the users.

Getting the app into their hands is up to us developers! And we need to do this fast. Nowadays, time-to-market is not only important. It is paramount. Especially, when you work in a startup-environment. But it applies to big enterprises alike. When somebody thinks of a good idea, the first two questions always are: "how much will it cost?" and "when will it be ready?"

If you fail at either one point, you may lose the business opportunity.

From your users' perspective, you need to answer these questions from end to end. This means, your users do not care whether you are the fastest developer on earth if you then lose a lot of time for handing over the app to your users. The users do not care if you can show them how the app runs on your computer.

Your users do not have access to your computer... I hope. You need to have your app running in an environment dedicated to its use. An environment that your users can access. Like a

web-server with a domain. Usually, this is the "production", "release", or "live" environment. The name is arbitrary. What matters is the purpose of this environment.

You need to have this environment, even if the only user is you. It is an environment with a version of your app that is not under development. Because source code under development usually does not work. At some times, it crashes, because there is a syntactical problem. At other times, it does not produce the intended effects. It may store the wrong data. It may miss calling an API. And so on. The bigger the change is that you're working on, the longer it takes you to build a working version.

If you work in a larger organization, chances are good that you have multiple environments. You have development, testing, staging, and production environments. Each environment serves a specific purpose. Each environment has its users. For instance, the users of your testing environments are the testers of your app.

You need to set up each of the environments. You have to configure it. If you have no environments, you have no users. You do not even have testers!

And you have to deploy your software in it. If you are not able to deploy your app in the environments, you have no users, either.

The more environments you have, the more often you'll need to deploy your app. The faster you develop features of your app, the more often you'll need to deploy it. Remember the importance of time-to-market.

Deployments are a core activity in a software's way from development to use. You'll need to do them frequently. And you need to execute them reliably.

The best you can do with repetitive tasks that you need to execute reliably is to automate them. Computers are predestined for repetitive tasks. Humans make mistakes from time to time. When every detail matters, automation increases speed and reliability.

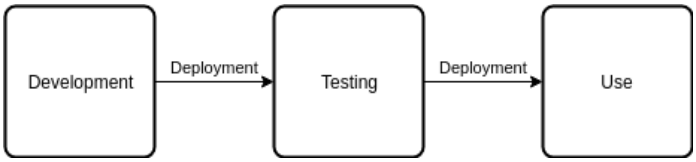


Figure 3.1: The traditional path of software from development to use

This chapter covers how you set up your projects and configure your environments.

3.1 Is Create-React-App A Dead End?

The default way of setting up a React project is Facebook's `create-react-app` script. `create-react-app` makes it easy to start developing a React app. It does not require you

Figure 3.2: Is create-react-app a dead end?

to set up your project manually. You don't need to first learn how to setup Typescript, Webpack, and Babel. `create-react-app` lets you concentrate on the development of your components. You can start coding right away. You can start your app locally. And you can build a deployable package of your app.

There's just one problem: **How do you deploy your app?**

`create-react-app` leaves you unsupported when it comes to deploying your app to a server.

"Just put the deployable package to your server with FTP!" you say. While this works for single-page-apps, it does not for a full-stack app that consists of anything more than some static files.

Even in the case of a single-page app, the problem is: it is a manual task. You need to make sure you copy the right files from your computer. You need to make sure you put the files into the correct folder at the server. These tasks are not complex. But they demand diligence. They are repetitive.

These are things where people make mistakes. It is not surprising that the cause of serious problems lies in mistakes made in such everyday tasks.

"Just write a script that triggers copying the files automatically. You can put it into your Webpack configuration," you say? I'd recommend the same. Don't forget to put the server's FTP credentials at a secure place. Don't forget to specify the FTP-server. Then, `eject` `create-react-app` and add your `bash`-script. **You're leaving the easy create-react-app**

road now...

"Do we need a web-server? Why don't we host our app on the cloud (for instance on AWS)?"

Amazon supports serving single-page apps directly from their Simple Storage Service (S3). Once configured, you can upload your bundled React app and a simple HTML file. S3 then provides a technical URL (`http://yourbucket.s3-website-us-east-1.amazonaws.com`) serving this HTML file that loads and renders your React app.

Not too bad. And the setup of this one service is not too complex either. But once you want to use a proper domain, like `www.your-domain.com`, you have to configure some other AWS resources, too:

- IAM (access management)
- CloudFront (content delivery-service)
- Route53 (custom domain setup)
- Certificate Manager (ssl-certificate setup)

Amazon provides extensive documentation. There are many tutorials on this topic, too. **But you're leaving the easy create-react-app road for sure...**

"Let's use infrastructure-as-a-service", you say? It is a powerful tool to set up cloud infrastructures, you say?

Infrastructure-as-a-Service (IaaS) has changed the way we deal with computing resources. IaaS allows us to outsource the provision and maintenance of infrastructure. We don't have to buy or lease hardware anymore. We don't have to care about operating system updates. We can scale the resources we need from almost nothing to almost everything in a split second.

When you use IaaS, you need to specify:

- the infrastructure you use
- how the infrastructure behaves
- how the components connect
- how the infrastructure connects to the application that runs on it

And you have to specify all these things to the very detail!

The following code snippet illustrates the configuration of infrastructure. Note: this is an excerpt. It is not even close to a complete configuration.

You're leaving the easy create-react-app road now...

Listing 3.1: Excerpt of a serverless.yml

```
provider:
 region: eu-west-1
 stackName: ${self:service.name}-${self:provider.stage, env:STAGE, 'dev'}
 environment:
  STAGE: ${self:provider.stage, env:STAGE, 'dev'}
  DOMAIN_URL: { "Fn::Join" : ["", [" https://#{ApiGatewayRestApi}", ".execute-
    api.eu-west-1.amazonaws.com/${self:provider.stage, env:STAGE, 'dev'}" ] ]
     }
  TABLE_NAME: ${self:service}-${self:provider.stage, env:STAGE, 'dev'}-data-
    layer
 iamRoleStatements:
  - Effect: Allow
   Action:
    - s3:Get*
    - s3:List*
   Resource: "*"
 stage: dev
 stage_path: dev
 name: aws
 runtime: nodejs8.10

functions:
 server:
  handler: build/server/server.default
  events:
   - http: ANY /
   - http: 'ANY {proxy+}'
   - cors: true

resources:
 Resources:
  StaticBucket:
   Type: AWS::S3::Bucket
   Properties:
    BucketName: ${self:provider.staticBucket}
    AccessControl: PublicRead
    WebsiteConfiguration:
     IndexDocument: index.html
    CorsConfiguration:
     CorsRules:
      - AllowedMethods:
        - GET
       AllowedOrigins:
        - "*"
       AllowedHeaders:
        - "*"
```

3.2 Configuration Is Not a Developer's Task

Maybe configuration is not supposed to work the `create-react-app` way? The developers are responsible for writing the code that runs on the infrastructure. Why don't we put more power in their hands and hand over responsibility for the infrastructure, too?

In fact, there are benefits, when developers can deploy applications themselves. Rather than sending it off to someone else for deployment. Once set up, such a streamlined deployment process comes in handy for many reasons:

- time-to-market
- reduced cost of overhead
- fewer errors caused by hand-overs and split responsibilities

But the setup of an environment that employs IaaS is not meant to be a developer's task. IaaS does not reduce the extent of expertise you need to have in your project team. IaaS requires you to know everything about infrastructure. It doesn't hurt to be an expert in different fields. But how many fields can you be a real expert in? Developers need to be real experts in their domain, already. We expect them to understand user stories. They cope with the functional logic that resides within the business domain. We want them to think about edge cases that may occur in the business process.

"But configuring IaaS is like programming. Developers work with configuration files all the time!" you say?

Today, developers get indoctrinated in writing well-structured code and adhere to best practices. We want them to write reusable and maintainable code. As you can imagine, configurations comprise many, many lines of definitions. They have a structure... yes. But some definitions spread across distant lines and even files. It feels like the long-forgotten `goto` instruction that we used in the 1970s.

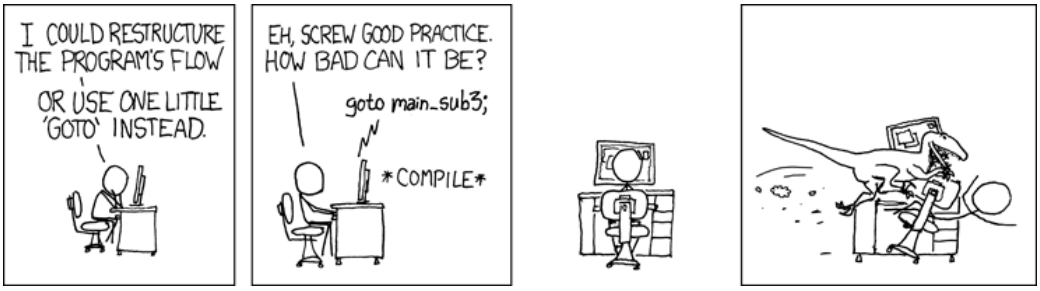


Figure 3.3: The danger of definitions spread across files. Source: https://xkcd.com/292/

Let's have a look at the configuration of a web-app that does not use `create-react-app`, should we? You build and deploy your React-app with scripts like `npm run build` and `npm run deploy`. When we search the `package.json` file for the scripts' definitions, I'll bet we find something like this:

Listing 3.2: Scripts section of a package.json

```
"scripts": {
 "build": "webpack --config webpack.config.js",
 "deploy": "npm run build && sls deploy && s3sync"
}
```

It may not look exactly alike. But, you specify the steps that run sequentially to build and deploy the app. Your configuration works in an imperative way. In imperative programming, you produce side-effects. A side-effect is a change of the state at a global (or at least modular) level. You continue to change the state until you reached the desired state.

The problem with side-effects is that keeping track of the global state is cumbersome and error-prone. You'll never know whether a variable still has the value it had before. You may have changed it!

"If it is imperative, what's the state then," you ask? The state is your filesystem! This is the worst global state you can think of. Your filesystem is not even limited to a certain process, program, or even operating system (if you have multiple installed).

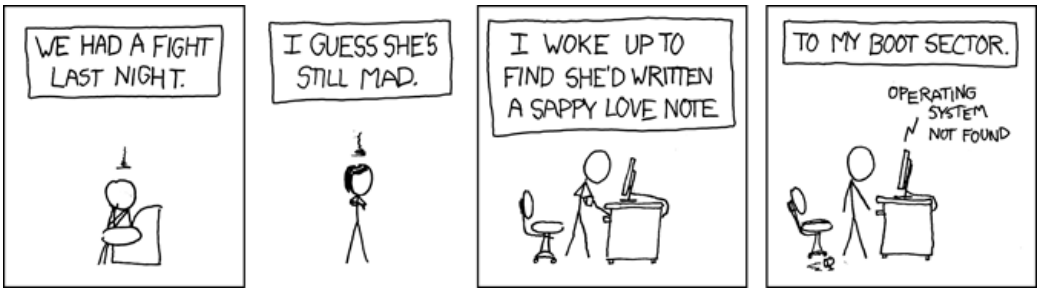


Figure 3.4: If you have security issues, your filesystem may in fact be: globally accessible! ;-),

Source: https://xkcd.com/340/

Let me further illustrate what's wrong with an imperative configuration. Your web-app most likely has an `index.html` that looks like this:

Listing 3.3: A simple index.html

```
<html>
 <body>
  <div id="root"></div>
  <script src="some/path/app.bundle.js"></script>
 </body>
</html>
```

Look at the `src` of the `<script/>`. This script loads our React-app. Provided our React-app can be found at `some/path/app.bundle.js` once it has been deployed.

So, what do you have to do to make your the script find your React-app?

First, your `build`-script calls Webpack (or another bundler). Webpack uses the file `webpack.config.js`. In the `output`-object, we see that the `publicPath` and the `filename` specify the path to our React-app.

Listing 3.4: `output`-object of the `webpack.config.js`

```
output: {
  filename: 'app.bundle.js',
  path: path.resolve(__dirname, './build/client'),
  publicPath: "/some/path/"
}
```

If it's that easy, why are you making such a fuss about this? ... Because it is not! In fact, Webpack creates the `app.bundle.js` and puts it onto the folder `build/client` as specified in `output.path`. Webpack does not deploy our app.

The deployment is the next step: `sls deploy`. This is a script of the Serverless-library that takes the `serverless.yml` as its input. In the respective section of this configuration, we specify to use an S3 (AWS Simple Storage Service) bucket to host our `index.html`.

Listing 3.5: Excerpt of the `servereless.yml`

```
WebAppS3Bucket:
 Type: AWS::S3::Bucket
 Properties:
  BucketName: our-bucket
  WebsiteConfiguration:
   IndexDocument: index.html
```

But the only thing we can do is specifying a bucket name: `our-bucket`. How does it know where to put our `app.bundle.js`?

Well, it doesn't. We have to do it on our own. In our `s3sync-script`. This finally connects all our loose ends.

It (a) takes the files from `./build/client` — the folder where Webpack put our `app.bundle.js` — and (b) uploads it to `our-bucket` — the bucket that Serverless created for us — (c) into the folder `/some/path` — the path where the `<script/>` of the `index.html` looks for the app.

Listing 3.6: The `s3sync-script`

```
s3NodeClient.uploadDir({
 localDir: "./build/client",
 s3Params: {
  Bucket: "our-bucket",
  Key: "some/path"
 }
});
```

This is side-effects at their worst! We only covered a single one aspect in this example: loading the React-app in a script. It covers four(!) different configuration files that all must fit together. If we changed something in one place, we would need to change it in another place, too. This is error-prone!

Further, you need to configure the entry-point file, the domain name, and many more aspects. As a result, your configuration files grow pretty fast and become muddled.

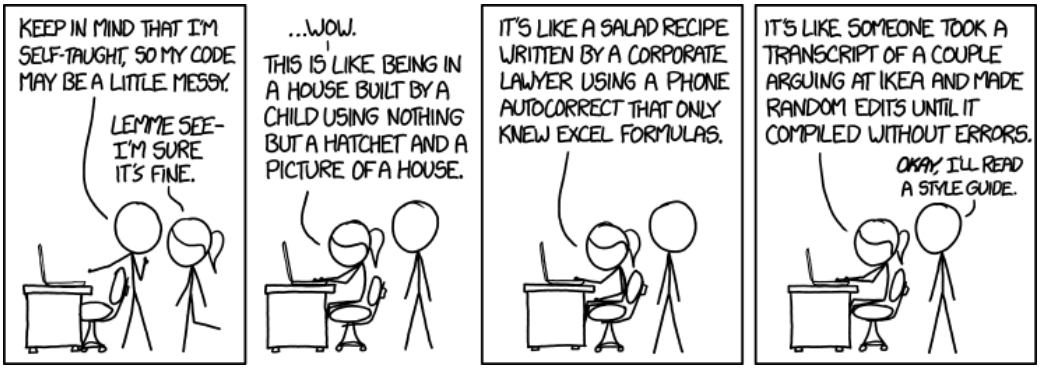


Figure 3.5: Some feedback on muddled code

3.3 Why Don't You Configure Your Infrastructure With React?

React is a great library to develop web-applications with. It lets you structure your code into declarative components that reduce side-effects. But why do you keep the configuration of your Serverless web-application outside of React?

React is declarative

In declarative programming, you concentrate on the "what". Not the "how". For instance, the following snippet tells your app to serve certain pages at the given route-paths.

Listing 3.7: Declarative Programming

```
const App = () => {
 return <BrowserRouter>
  <Route path="/" render={() => <Home />} />
  <Route path="/dashboard" render={() => <Dashboard />} />
 </BrowserRouter>
}
```

With React, you don't need to specify the steps the app needs to execute to actually serve the pages when the user opens a path.

You leave these *"how it is done"*-things up to the component. Regardless of whether you developed the component yourself or whether you got it from a public repository (like `npm` or `yarn`). Even if the component has not been developed at all! You can even use a *"yet-to-be-developed"*-component that *"simulates"* the result you are looking for and replace it with the real component later.

IN CONGRESS, JULY 4, 1776.

A DECLARATION

BY THE REPRESENTATIVES OF THE

UNITED STATES OF AMERICA,

IN GENERAL CONGRESS ASSEMBLED.

Figure 3.6: The most famous declaration! It describes the "What" — Independence. Not the "How" — The American Revolutionary War.

React components reduce side-effects

In React, what exactly a component does depends only on the properties you pass into it (e.g. serving a path or rendering a component). The component does not depend on some outside state that is cumbersome to keep track of.

Components of the same kind with the same properties always produce the same result. This makes understanding a React application easy.

React components are functions

Functions serve a specific purpose. A function encapsulates all the things required for achieving its purpose. It hides its internal complexity. Functions expose as parameters the things that are important for the developer. They must serve specific use cases. But likewise, they must be abstract to a certain degree, so you can reuse them.

How would functions that create the infrastructure for us look like?

A function that creates our infrastructure for us assures that definitions fit together. You don't have configurations spread across distant files anymore. This function hides the internal complexity of the infrastructure configuration. It works on an abstract level.

What is a useful degree of abstraction when configuring a React project?

On the one hand, the level of abstraction must be meaningful for developers to work with. On the other hand, someone must implement these configuration-producing functions. These remain to be the system engineers with their deep knowledge of infrastructure and configurations.

3.4 The Common Language of Developers and System Engineers

Why do we write user stories?

"Because we want to focus on the needs of the user!" says the product owner.

"I can build my campaign on it," says the marketing manager.

"It is because the business people are not able to specify technical specifications!" says the developer.

All these statements apply. User stories provide a functional overview of what the software is about.

User stories are not supposed to replace the more detailed elaborations of the user interface, the campaign, or the requirements. But user stories are a starting point for collaboration between specialists of different domains!

User interface designers can sketch the user flow. Marketing managers can build their campaigns on it. Developers can derive technical specifications.

Have you ever experienced the struggle of specialists of different domains to talk about the same thing?...

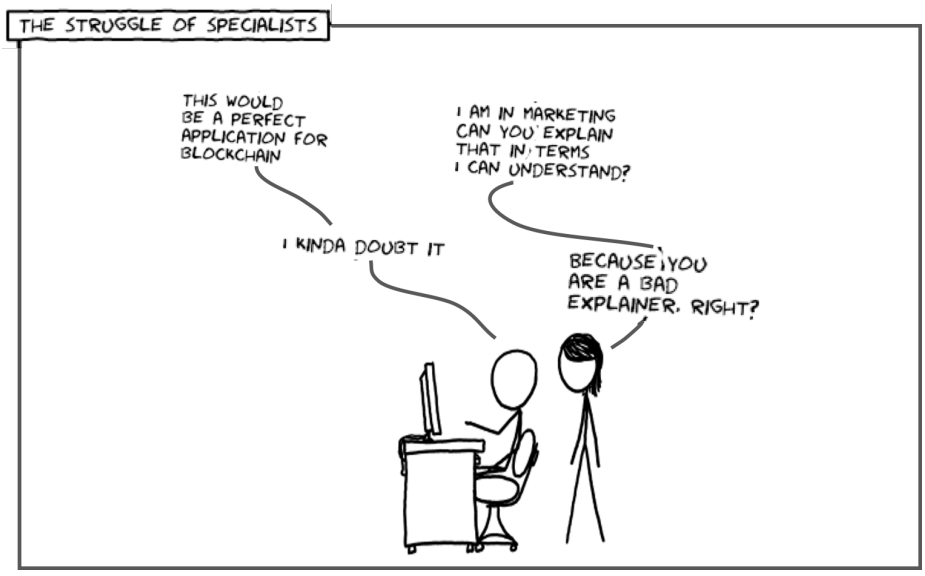


Figure 3.7: Two specialists talking about the same thing... more or less

User stories have proved useful as a tool for communication. Because user stories use a language shared by the different specialists.

But a software development project requires many specialists from different domains to collaborate. User stories serve as a tool for communication between business experts and developers. They do not help universally.

How do developers and system engineers communicate?

Before the developers can start implementing a single user story, the technical infrastructure needs to be set up. This is the task of system engineers. They provision hardware. They install operating systems. They configure all the infrastructure the code is supposed to run on.

This infrastructure is not homogeneous. There are application servers. There are databases. There are load balancers. There are technical registries. There are even serverless computation resources.

System engineers are specialists in the configuration of all these components. But it has nothing to do with user stories. You can't derive how to configure your infrastructure from a user story like:

"As a reader of a blog-post, I want to see other posts the author wrote."

We need another tool for collaboration. What is the shared understanding between developers and system engineers?

Developers write the software running on different infrastructure components. From the given story, they can derive:

- a front-end (something to display the blog-post and the other posts)
- a data source (there seem to be other posts)
- connecting both: (when displaying a blog-post, we know the author. We need to search the data storage for posts of this author)

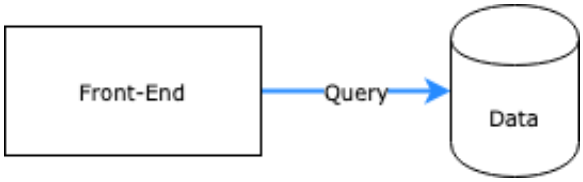


Figure 3.8: What the developer can derive from the user story

With this information, a system engineer could already work. Because we identified the required components and how they need to interact. However, at this level of detail, almost any infrastructure would do! We could even work with a mainframe application and a 3270 terminal interface. The 3270 terminals run directly on the mainframe that stores the data.

Figure 3.9: The IBM 3270 terminal interface.

Maybe this is not what our product owner had in mind?! We need a little more specificity. That's why non-functional requirements are important. A non-functional requirement specifies criteria that judge the operation of a system. Rather than its specific behaviors.

The non-functional requirement of "easy to use" may sound generic, too. But it complements the user story about "a reader of a blog-post". Someone may provide a more detailed picture of the infrastructure components. This someone needs to know what "a reader of a blog-post may find easy to use". And she needs to know what the capabilities of technologies are.

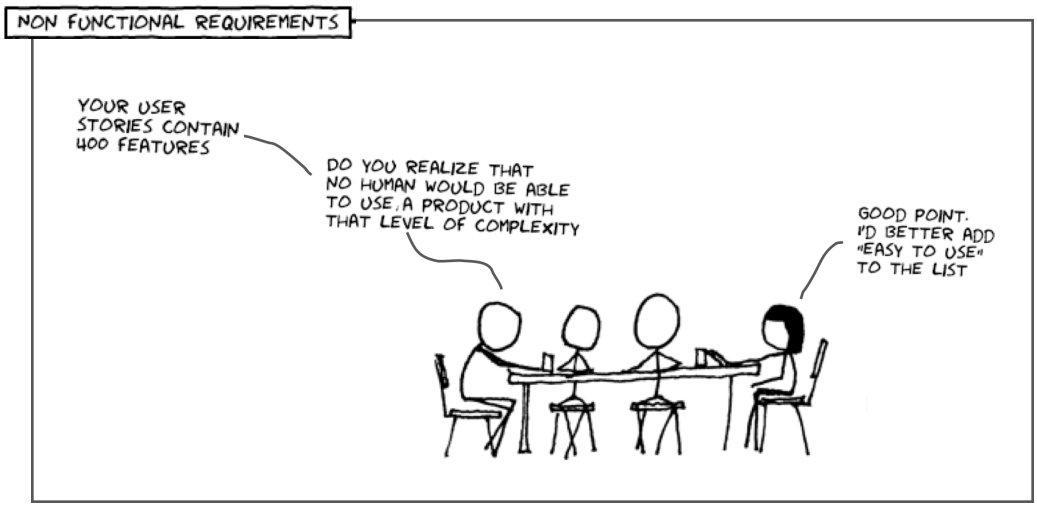


Figure 3.10: Non-functional requirements.

This someone is the developer. We expect developers to understand user stories. A developer may derive "blog-post users use the web". And she knows about the capabilities of technologies. Such as "HTML is a means to create easy to use web-interfaces", "JavaScript lets you query data from web-interfaces", and "the best queryable data source is a database".

With the additional knowledge of "readers and authors do not share a local computer", the developer derives "the database must be accessible for readers and authors. It must reside at the server-side". The developer can draw the following picture:

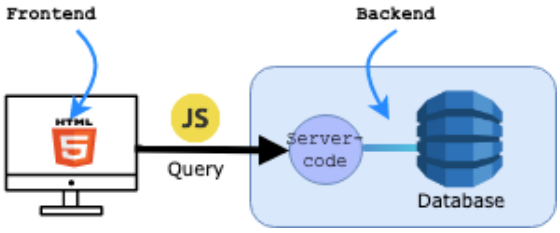


Figure 3.11: What the developer can derive from the NFR

When the system engineer looks at this picture, she says: "Why didn't you say that you want a web-app with a server-side API and a database? I can create such an architecture for you".

A Common Language

The developer does not need to know how to configure a web-server. She does not need to know how to set up a database. But she can identify the infrastructure components, how they interact and the software stack she wants to use. This is an infrastructure architecture.

System engineers require a description of the components and how they are supposed to

interact. This is the infrastructure architecture, too.

Infrastructure architectures are the common language of developers and system engineers.

User stories enable business experts and developers to collaborate. Infrastructure architectures enable developers and system engineers to collaborate.

"The whole example is constructed! Even system engineers know that blogging is a phenomenon on the web", you say?

You are right. But a lot of software gets developed for specific users with specific problems. Not every piece of software gets solved with the same technology.

We want our system engineers to be technical specialists. They must know all the details of many technologies. They don't have the time to understand the users. It is not their task.

Likewise, we want our developers to cope with the functional logic that resides within the business domain. Being a technical expert who can use a programming language is not enough anymore! Developers don't have the time to know all the details of the technical configuration. it is not their task (...anymore).

With the increasing complexity of software projects, we need more specialists in software development teams. Whenever specialists collaborate, they require some common language.

Infrastructure architectures serve as the common language of developers and system engineers.

3.5 The Architecture-as-a-Function Paradigm

The infrastructure architecture resides at a high level of abstraction. You don't need to understand every detail of its configuration. Yet, you can understand how it works. You can verify whether an architecture provides all the components you need to run your software on. You can verify whether an architecture satisfies the non-functional requirements.

Most developers know at least the basic architectures used in their domain. Even if they do not. It is easier to learn the characteristics of architectures than it is to learn how to write its configuration. And from a developer's point of view, an understanding of how the architecture works is more important than the ability to configure it.

You can define architectures in general terms. You can use simple phrases like: "a single-page React app", "a service-oriented React app", or "an isomorphic React app".

We want to configure our infrastructure with React. We want to use declarative components that describe the "what". Not the "how". And we want to define the infrastructure in terms of the architecture. Because this is what makes sense to us developers, yet it enables the system engineers to create the configurations.

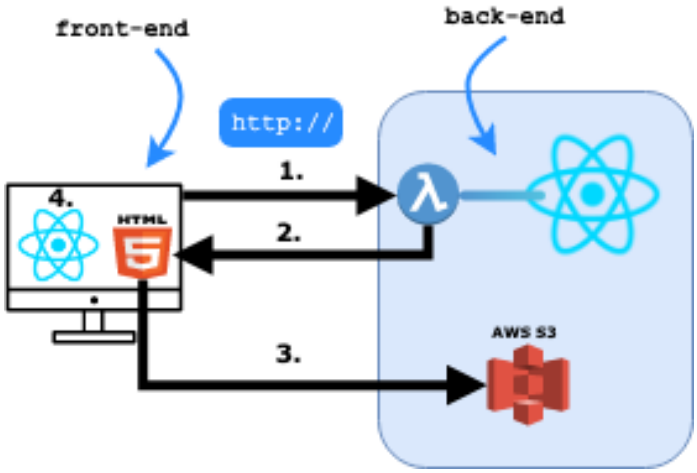


Figure 3.12: The architecture of a Serverless isomorphic React app

Then we want to tell our app simply to be a `<SinglePageApp/>`, `<ServiceOrientedApp/>`, or an `<IsomorphicApp/>`. That's it!

We want to reduce side-effects. So we may want to provide as properties to the React components the things that matter from an architectural perspective. We may want to specify the routes the app serves. We don't want to specify how it serves them.

You don't want to specify that your service is accessible through the internet. Of course, it is.

You don't want to specify that the `index.html` loads the script that contains your React app. What else should it load?

You don't want to... I think you see my point.

In this book, we use the library Infrastructure-Components.

Infrastructure-components are simple React-components that let you specify the architecture of your app.

Infrastructure-Components are functions, like any other React-component. A function serves a very specific purpose. The component `<SinglePageApp/>` does one thing: create a single-page app. You don't need to care about how it works internally. You just work with its returned output.

> Infrastructure-Components are open-source. Have a look at the source code. Especially, if you are interested in how they work internally.

Let's have a brief look at Infrastructure-Components in action. The following code-snippet is a complete Serverless single-page React app:

Listing 3.8: The source code of a complete Serverless single-page React app

```
/** src/index.tsx */
import React from 'react';

import {
 Environment,
 Route,
 SinglePageApp
} from "infrastructure-components";

export default (
 <SinglePageApp
  stackName = "example"
  buildPath = 'build'
  region='us-east-1'>

  <Environment
   name="prod"
   domain="www.infrastructure-components.com"
  />
  <Route
   path='/'
   name='Infrastructure-Components'
   render={ (props)=> (
    <div>Hello Infrastructure-Components!</div>
   )}
  />
 </SinglePageApp>
);
```

In this app, you:

- define the architecture of your app by the `<SinglePageApp/>` component
- specify a run-time `<Environment/>` that you can start locally or deploy
- simply connect your domain with the environment.
- serve a visible page through the `<Route/>`-component

We will cover all these things in much more detail in the following chapters of this book. Before that, we have a look at how to set up an Infrastructure-Components-based project.

3.6 Which Architecture Is Right For My Project?

Do you remember? We need to get our app into the hands of its users. And we need to do this fast!

We have solved the inability to deploy our app in chapter 2. But there's an even bigger problem. This is us. Us developers! Because we tend to over-engineer.

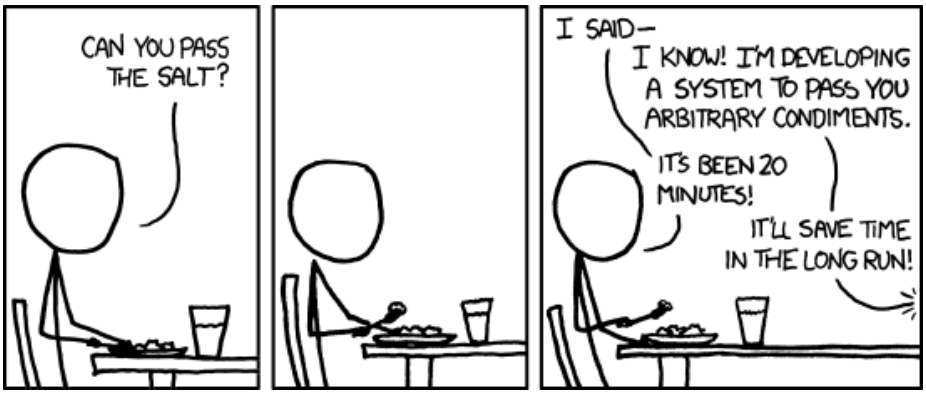


Figure 3.13: Over-engineering in practice, https://xkcd.com/974/

We developers over-engineer for a good reason. We aim to avoid technical debt.

Technical debt is the implied costs of rework caused by choosing a short-cut solution now instead of using a better approach that would take longer.

Let's consider the following example from chapter 1. You want to output today's date as a string with its first and last hour (0 and 23).

Pretty simple. Have a look at the following source code.

Listing 3.9: The easy way

```
/** THE EASY WAY */
const today = new Date();
console.log("today is from "
  + today.getFullYear()
  + "-" + today.getMonth()
  + "-" + today.getDate()
  + "-" + 0
  + " to "
  + today.getFullYear()
  + "-" + today.getMonth()
  + "-" + today.getDate()
  + "-" + 23
);
```

Alternatively, we could write the code in an (over-?) engineered way. We would put all things into separate functions.

Listing 3.10: The (over-?) engineered way

```
/** THE (OVER-?) ENGINEERED WAY */
const dateToString = (d) => (
  d.getFullYear()
  + "-" + d.getMonth()
  + "-" + d.getDate()
  + "-" + d.getHours()
);

const setDate = (d, hours) => (
  new Date(d.getFullYear(), d.getMonth(), d.getDate(), hours)
);

console.log("today is from "
  + dateToString(setDate(new Date(), 0))
  + " to "
  + dateToString(setDate(new Date(), 23))
);
```

Which solution is better? The easy way is shorter. The (over-?) engineered way is longer and it creates more `Date`-objects during run-time. Is it overkill? Both solutions produce the same output (by the way, today is October 15th, 2019): `today is from 2019-9-15-0 to 2019-9-15-23`

Wait! This is wrong. We have October! Not September.

The reason is: the `getMonth()`-function of the JavaScript `Date`-object starts counting at `0`. So, we need to add `1` to it.

In the "engineered way"-code, we can correct the result with a single change: `parseInt(d.getMonth()+1)`.

In the "easy way"-code, we need to correct the error in two places. `parseInt(today.getMonth()+1)`. This is more work. And this is a source of errors. We might easily miss changing it in one place. We get a completely different result. Such as `today is from 2019-10-15-0 to 2019-9-15-23` in case we changed only the first occurrence.

This additional work and this source of errors are our technical debt! It is little. Yes. But it is a little example, too.

Technical debt can be compared to monetary debt. If technical debt is not repaid, it makes changes harder to implement later on. Whoever has to change something has to deal with the imperfect concepts you codified on the first occasion. The debt keeps increasing over time. It accumulates interest.

If you ignore technical debt long enough, you can go technically bankrupt. This is the case when your code does not allow you changing it anymore without breaking it.

In our example, what if we wanted to output the first and the last hour of our local day in another timezone. Let's assume we are in New York City. It is 7 a.m. (Eastern Standard Time, -5 to UTC). What is our New York City day in UTC?

We could simply replace the called functions with their `UTC`-versions, for instance: `d.getUTCDate()` replaces `d.getDate()`.

The engineered code does not require many changes.

Listing 3.11: The (no longer over-) engineered way

```
/** THE (NO LONGER OVER-) ENGINEERED WAY */
const dateToString = (d) => (
  d.getUTCFullYear()
  + "-" + parseInt(d.getUTCMonth()+1)
  + "-" + d.getUTCDate()
  + "-" + d.getUTCHours()
);

const setDate = (d, hours) => (
  new Date(d.getFullYear(), d.getMonth(), d.getDate(), hours)
);

console.log("today is from "
  + dateToString(setDate(new Date(), 0))
  + " to "
  + dateToString(setDate(new Date(), 23))
);
```

Due to our technical debt, in the easy-way-code, we need to apply the change at more places than in the engineered example.

Listing 3.12: The changed easy way

```
/** THE EASY WAY */
const today = new Date();
console.log("today is from "
  + today.getUTCFullYear()
  + "-" + parseInt(today.getUTCMonth()+1)
  + "-" + today.getUTCDate()
  + "-" + 0
  + " to "
  + today.getUTCFullYear()
  + "-" + parseInt(today.getUTCMonth()+1)
  + "-" + today.getUTCDate()
  + "-" + 23
);
```

But if we look at the result, we see that both outputs differ!

```
today is from 2019-10-15-0 to 2019-10-15-23
today is from 2019-10-15-5 to 2019-10-16-4
```

The easy-way-code does not output our local day as it is in UTC. Because it simply takes the date and hard-codes the hours. We need to add more code to make it work. We have to pay interest on our technical debt.

Have a look at the following example! ... Do you see where I struggled?

Listing 3.13: The easy way with interest in technical debt

```
1  /** THE EASY WAY WITH INTEREST ON TECHNICAL DEBT */
2  const today = new Date();
3  console.log("today is from "
4    + today.getUTCFullYear()
5    + "-" + today.getUTCMonth()
6    + "-" + today.getUTCDate()
7    + "-" + 5
8    + " to "
9    + today.getUTCFullYear()
10   + "-" + parseInt(today.getUTCMonth()+1)
11   + "-" + parseInt(today.getUTCDate()+(today.getHours()+5 > 23 ? 1 : 0))
12   + "-" + 4
13 );
```

What did I do here? (Besides reintroducing the old bug of having missed adding `+1` to the month... funny fact: This is a real mistake I made. I think I copied the code fourth and back too often...)

Since we are in NYC, I can hard-code the hours of the UTC-day (`5` to `4`). But depending on the current hour of the day, it may already be tomorrow in UTC! In that case, we add `1` to the date.

This code works for the current day - that is October, 15th. But what if it was October, 31st? We would even need to add another month. What about December 31st? We would even need to add another year.

This is too much for me right now! I don't have the time... I have to write a book... It is not too bad if the result is wrong for 5 hours at 12 days a year, is it?

Did I just go technically bankrupt?

The technical debt did not only produce more work in the first step. It added up in the second step. It even made me introduce a bug! Even further, the "easy" solution only works in NYC. It does not work in any other timezone. I am not even talking about daylight saving time...

Too much technical debt will prevent features and bug fixes from shipping in a reasonable amount of time.

If technical debt is that bad, how can we call good engineering, that prevents technical debt, be "over"-engineering?

The problem with code is that you never know which parts you may need to change in the future. You would need to have a magic crystal ball. Both, the requirements and the technology change at a speed that does not allow you to foresee the future.

When we first wrote the code example above, how could we have predicted that we want to calculate our local NYC days in another timezone? We couldn't! At this time, our solution was over-engineered. We were lucky it came in handy.

What if that new requirement wouldn't have been added? We would have spent quite some time (and thus money) on a solution we did not need.

What if you don't understand the problem to solve good enough yet? While you're programming, you are learning. It can take some time before you understand what the best solution is. Sometimes, you first have to provide a proof-of-concept. It might become part of a solution later. It might be a bunch of hooey, as well.

Spending time on something you don't need is like gambling. You give some time now for the chance of winning some time back later.

Sometimes, you'll win. Sometimes, you'll lose the bet. You end up with code you don't need. But this code is part of your app now. You have to live with it. You need to take care of it. It is another form of technical debt. Because the code is unnecessary for the current purpose of your app. And you don't know whether it serves a future purpose or not.

No matter what you do, you'll end up with technical debt. There's no way we can prevent it. But there are means to control it.

First, there's a best practice in software development: "You ain't gonna need it" (YAGNI). This practice says that you should not add anything to your code until you need it. You implement the code you need to solve your problem at hand. You don't implement now what you (might!) need later.

We'll incur technical debt! But it allows us to deliver software faster. We get our app into the hands of the users.

Second, there's another best practice: "Don't repeat yourself" (DRY). This practice aims to reduce the repetition of code. Rather than duplicating code, you replace it with an abstraction (such as a function).

This is your chance to reduce technical debt in a meaningful way. You learned you need this piece of code. The more you need it, the more time you invest in it. When you reuse it

again, take the time it would take you to implement the code from scratch. You have this time because the chance to reuse some existing code just saved you some time. Look at the code. If necessary, improve it or even refactor it. But only invest the time you saved.

By paying down your technical debt that way, you make sure you invest the time where it matters. Further, you can argue that you worked on the feature (or user story or whatever) you promised to work on. The invested work added to that feature.

You don't have to ask for or claim time for refactoring. Your business will always ask you why refactoring is necessary right now: *"It does work right now. Doesn't it?"* If it does, spending time (thus money) on it does not repay. If it doesn't, you are to blame for writing code that does not work.

"But if we don't refactor the code now, implementing features in future will require more time," you say? The answer most likely is: *"You were lazy at the time you implemented it. You took a shortcut. And now we need to pay for implementing it again?"*. You're not gonna win this blame game. It is no fun, either.

...very well. But how does all this help me to decide which architecture is right for my project?

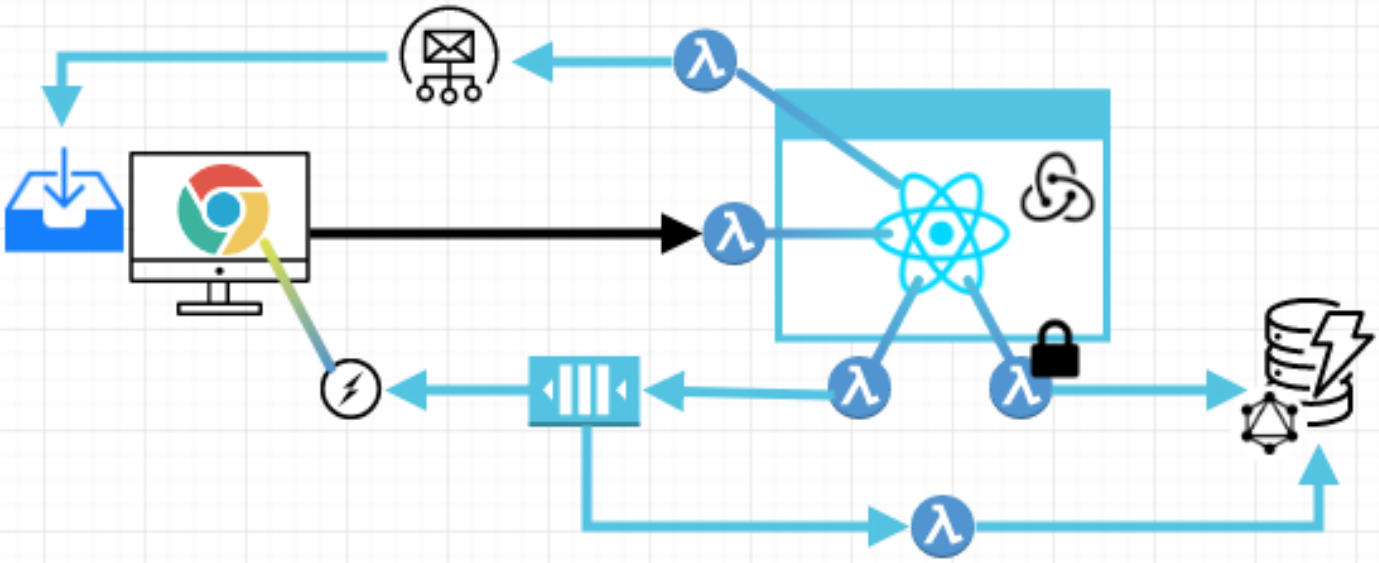
The answer is: Do not over-engineer. Use the simplest architecture solving your problem at hand. If the next problem requires a more complex architecture, you simply change your architecture.

Figure 3.14: One does not simply change the architecture!

The architecture is the foundation of a project. It defines how you render your components (front-end or isomorphic). It specifies how you store and access your data. Whether you use a redux-store or a database. One does not simply change the architecture of a web-app!

Oh, yes. You do!

Remember, we don't configure our infrastructure manually anymore. We specify the architecture of our app through React-components. If we want to use another architecture, we simply replace the component.



4. Set Up An Infrastructure-Components-Based Project

4.1 Set up your development environment

Regardless of whether you use `create-react-app` or you create an infrastructure-components-based project, you'll need to set up your development environment.

A basic development environment needs to serve at least the following two purposes:

- Edit source code files
- Run scripts

While a simple text editor and a terminal console are all you really need, I would recommend using an integrated development environment (IDE).

Modern IDEs provide a lot of convenience functions. Code-highlighting, auto-complete, style checking, version control support, and more. There are free and commercial IDEs that specialize in JavaScript-based projects (for instance JetBrain's WebStorm).

Which IDE best servers your needs is up to you.

In order to run the scripts to build, start, and deploy your project, you'll need `Node.js` on your computer. `Node.js` is a JavaScript run-time environment.

The setup of `Node.js` depends on the operating system of your computer. You can download its source code or a pre-built installer from the official page.

`Node.js` ships with the `Node Package Manager` (`npm`). This tool manages the libraries you use in your project. It downloads them from a central repository.

Infrastructure-Components-based projects support running your project locally. This includes a database (DynamoDB). This requires the `Java 8 JDK`. Here's how you can install the JDK on your computer.

4.2 Create Project Files

Your React project consists of a set of files. You have different ways of creating or getting these files:

- Create a customized boilerplate project using the infrastructure-components-configurator
- Clone or download the code from a GitHub-repository
- Create the files manually

Creating project files manually is not the most convenient way. But let's start with this case. We use the chance to go through the files so that you get an understanding of what the files are for.

4.2.1 Create Files Manually

The following set of files is the minimum you need to build, start (locally), and deploy your app.

Listing 4.1: The minimum file structure of your project

```
project/
|- src/
|  |- index.tsx
|- .env
|- .gitignore
|- package.json
```

The `.env`-file specifies variables used during the build and deployment processes. For instance, the `.env`-file contains the credentials you need to deploy your project. It is good practice to exclude `.env`-files from being version-controlled. You don't put them into the repository.

In other settings, `.env`-files also contain values used at the run-time of your app, such as API paths. But specifying values in `.env`-files builds on side-effects. Infrastructure-Components aim to reduce such side-effects. They let you define such values as part of the `<Environment/>`-components instead.

The `.gitignore` file contains the files and folders that you don't want to put into a repository. For instance, it defines the `.env`-file. Further, it keeps our repository free from temporary and output files. If you have the source code of your project, you can always reproduce the output files.

The following snippet depicts a basic version of a `.gitignore`-file:

Listing 4.2: A basic version of the `.gitignore`-file

```
.infrastructure_temp/
*.env
.serverless
.dynamodb
.s3
container/
build/
dist/
node_modules/
v8-compile-cache-0/
```

The `package.json` describes your project from a technical perspective. It is the most important file of JavaScript-based projects. Your `package.json` is a dynamic file. Different scripts read from it and write into it. For instance, the command `npm install --save <name>` adds a package to your `dependencies`.

Documentation — package.json.

- The `name` field contains your package's name. It must be lowercase and one word. It may contain hyphens and underscores. The `name` is important if you plan to publish your project as a library to `npm`. Otherwise, it is arbitrary.
- If you plan to publish your project, the `version` field must be in form `x.x.x`
- The `scripts` section contains the definitions of the scripts that automate repetitive tasks, such as building, starting, and deploying your app.
- The `dependencies` section contains the libraries that your project needs to run. They become part of your app when it is deployed.
- The `devDependencies` section contains the libraries that your project uses during development and deployment. They are not a part of your app once it is deployed.

See the official npm page for complete documentation.

The following snippet depicts the `package.json` of an Infrastructure-Components-based project. We do not plan to publish our project at `npm`. Thus, the `name` and `version` contain arbitrary values. The `dependencies` contain the libraries that we'll use throughout the examples in this book. You can add further libraries once you need them.

The `devDependencies` contain all the libraries that we only require during development. The library `infrastructure-scripts` comprises the scripts required of building, starting, and deploying an Infrastructure-Components-based app. All the other libraries support starting and deploying a serverless app. The build script is the only script we need to specify initially. It runs the `scripts` command with three arguments. `"scripts .env build src/index.tsx"`.

Documentation — the `scripts` command.

The `scripts`-command is part of the library `infrastructure-scripts`.

- `.env` specifies the file with the environment variables to be used during deployment. Your credentials are the only values you should define here. Values required during the app's run-time become part of the `<Environment/>`-components.
- `build` is the command you want the script to run. We describe the different commands later in this chapter
- `src/index.tsx`is the relative path to your top-level source code file. This file can have another name or can be at another location.

Listing 4.3: The `package.json`

```
{
 "name": "my-app",
 "version": "0.0.1",
 "scripts": {
  "build": "scripts .env build src/index.tsx"
 }
 "dependencies": {
  "@apollo/react-hooks": "^3.1.3",
  "@babel/polyfill": "^7.8.0",
  "aws-sdk": "^2.601.0",
  "express": "^4.17.1",
  "graphql": "^14.5.8",
  "graphql-tag": "^2.10.1",
  "infrastructure-components": "^0.3.10",
  "isomorphic-fetch": "^2.2.1",
  "react": "^16.12.0",
  "react-apollo": "^2.5.8",
  "react-dom": "^16.12.0",
  "react-helmet": "^5.2.0",
  "react-router": "^5.1.2",
  "react-router-dom": "^5.1.2",
  "request": "^2.88.0",
  "serverless-http": "^2.1.0",
  "styled-components": "^4.1.3",
  "universal-cookie": "^4.0.1"
 },
 "devDependencies": {
  "infrastructure-scripts": "^0.3.11",
  "serverless-domain-manager": "^3.3.1",
  "serverless-dynamodb-local": "^0.2.38",
  "serverless-offline": "^4.9.4",
  "serverless-pseudo-parameters": "^2.5.0",
  "serverless-single-page-app-plugin": "^1.0.2"
 }
}
```

The file `src/index.tsx` is the main entry point of our app. You specify it in the scripts of the `package.json`-file. This file has a single top-level component. It exports as default the component that defines the infrastructure architecture of your app. E.g. `<SinglePageApp/>`, `<ServiceOrientedApp/>`, or `<IsomorphicApp/>`.

Infrastructure-Components support the following infrastructure architectures:

- Serverless single-page apps (`<SinglePageApp/>`)
- Serverless service-oriented apps (`<ServiceOrientedApp/>`)
- Serverless isomorphic apps (`<IsomorphicApp/>`)

Documentation — The top-level infrastructure architecture components.
The top-level infrastructure architecture components require the following properties.

- The `stackName` is the (arbitrary) name of your app. It supports only lower case characters and hyphens because the name serves as an identifier within AWS. Further, `stackNames` must be unique within an AWS account.
- The `buildPath` is the relative path to the folder within your project where the build-resources are put, e.g. `'build'`. You may want to add this name to your `.gitignore` file to keep your repository free from compiled files.
- The `region` is the AWS-region you want your infrastructure to reside after deployment, e.g. `'us-east-1'`.
- (**only** `<IsomorphicApp/>`) The `assetsPath` is the relative path to the folder where the app stores the bundled JavaScript-code at run-time, e.g. `'assets'`

Why do we specify paths? Aren't these side-effects?
Infrastructure-Components aim to reduce side-effects. You can create, start, and deploy your project without bothering with the specified paths (simply use the default values from the examples).

But you might want to add or use scripts and functions beyond the scope of Infrastructure-Components. Having control over certain settings makes life easier in this case.

Each top-level infrastructure architecture component accepts `<Environment/>`-components as its (direct) children. An `<Environment/>`-component specifies a run-time environment of your app. With environments, you can distinguish your development-environments from your production-environment.

Documentation — `<Environment/>`.

- The `name` is the name of the environment that serves as literal in scripts that work with a certain environment, like `deploy`. Thus, use short names, with no special characters other than hyphens.

In your project, you may need to specify values that depend on a certain environment. This is especially useful when you call third-party APIs in your app. When these APIs trigger specific actions, like buying a product, you don't want to call the real API while you develop your own app. Instead, you want to call a version of the API that is technically alike, but that does not trigger the action. Different URLs let you distinguish APIs. So, you'll want to specify different URLs of an API depending on the environment.

The default solution is to use environment variables. In other project settings, you specify the values in different `.env`-files. Then you make sure that you load the corresponding `.env`-file depending on whether you want to develop, test, or run your app productively. This side-effects-based approach is error-prone.

In Infrastructure-Components-based projects, you specify your environments as children of your top-level component. These let you also specify environment-specific values. They accept <EnvValue/>-components as their children, like this:

Listing 4.4: Specify a value specific to an environment

```
import { Environment, EnvValue } from 'infrastructure-components';

...

<Environment name="dev" >
  <EnvValue name="MY_VARIABLE" value="hello world" />
</Environment>
```

Documentation — <EnvValue/>.

- The `name` is the name of the environment-variable. You can access its value at run-time through `process.env.MY_VARIABLE`. Replace `MY_VARIABLE` with the specified name.
- The `value` is the string you specified. If you want to work with numbers or objects, you'll need to parse them at run-time.

4.2.2 Use Template/ GitHub

Creating the project files manually every time you start a new project would be quite cumbersome. Except for a few changes, they are always the same. So why don't we copy them?

For this purpose, I've created the following GitHub repositories. They contain the basic set of files. Each repository contains a basic example of the respective infrastructure architecture.

- Serverless Single-Page App
- Serverless Service-Oriented App
- Serverless Isomorphic App

You can download or fork these GitHub repositories.

4.2.3 The Infrastructure-Components-Configurator

When you download or clone a repository, you'll get the files with their specific names. But of course, you'll want to use your own custom project name.

You can get your customized boilerplate code from the Infrastructure-Components-configurator for free.

You need to select the architecture, specify a name for your React app (like "myapp"), and a

name of an environment, (e.g. "dev"). Then you can download and unpack your customized boilerplate code.

Whether you prefer to create the project files manually, clone a GitHub repository, or use the configurator does not matter.

4.3 Prepare Your AWS Account

In this book, we want to get your app in the hands of its users. For that, we need a hosting provider. We use Amazon Web Services (AWS). AWS offers reliable, scalable, and economical cloud computing services.

Currently, Amazon provides generous free tiers of its resources. However, Amazon may charge you for the resources you use. Please make sure to control the AWS resources you use.

If you don't have an AWS account, you can create it at https://aws.amazon.com.

Infrastructure-components aim to automate as many things as possible. However, there are a few things we need to do manually. Because these things require explicit confirmation.

- Create a technical user (with programmatic access / API-key)
- Register a custom domain (optional, only if you want to use your own domain)
- Verify an email address (optional, only if you want to send emails from your app)

4.3.1 Create A Technical User

Infrastructure-components require a technical user during the deployment of your app. It creates all the resources.

AWS provides an Identity and Access Management (IAM) tool. This controls who (in terms of persons) and what (in terms of scripts) has access to your account and the resources in it. The only thing an Infrastructure-Components-based project requires is a user with programmatic access.

Let's go through the steps of creating one.

Log in to your AWS Console and select **IAM** from the list of services.

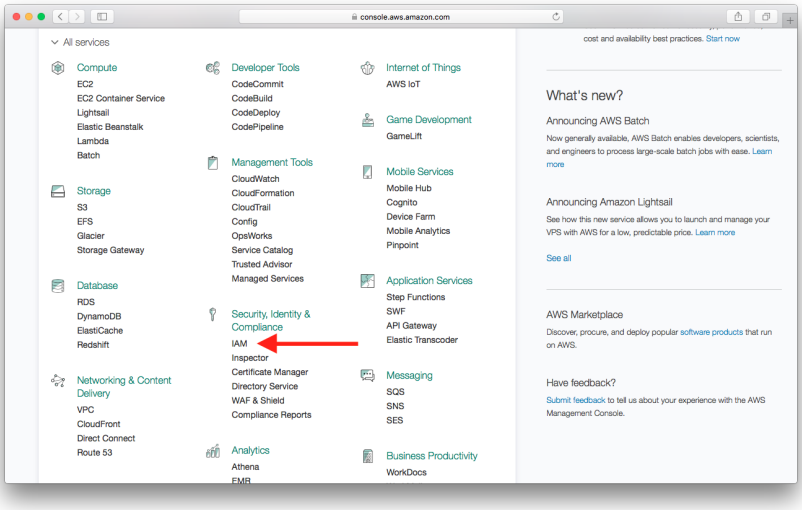


Figure 4.1: The list of AWS-services

At the IAM-page, Select **Users** from the menu.

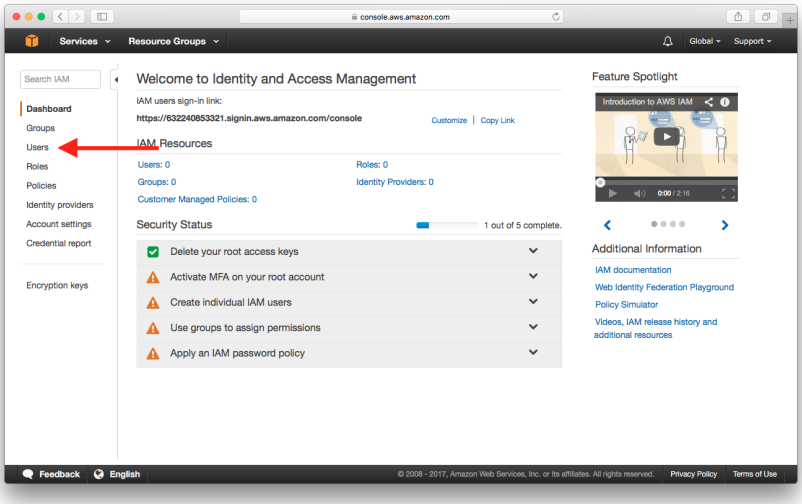


Figure 4.2: The IAM page

Select **Add User**.

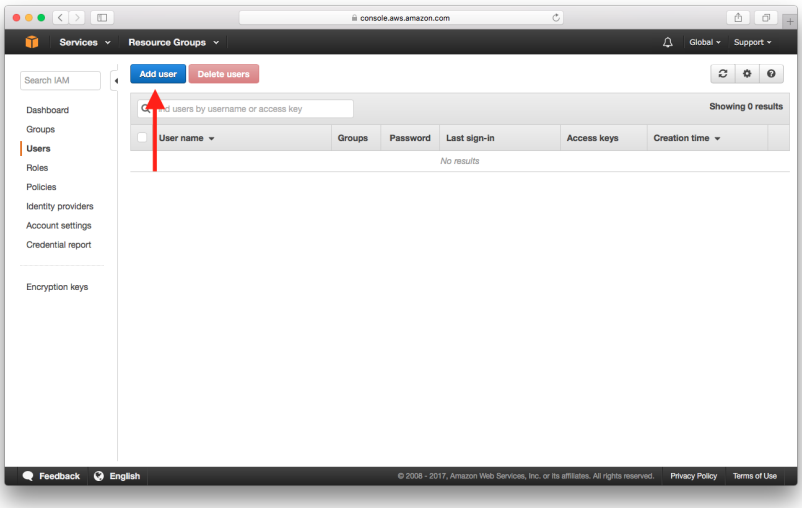


Figure 4.3: The list of users

In the new user dialog, enter a **User name** and check **Programmatic access**, then select **Next: Permissions**.

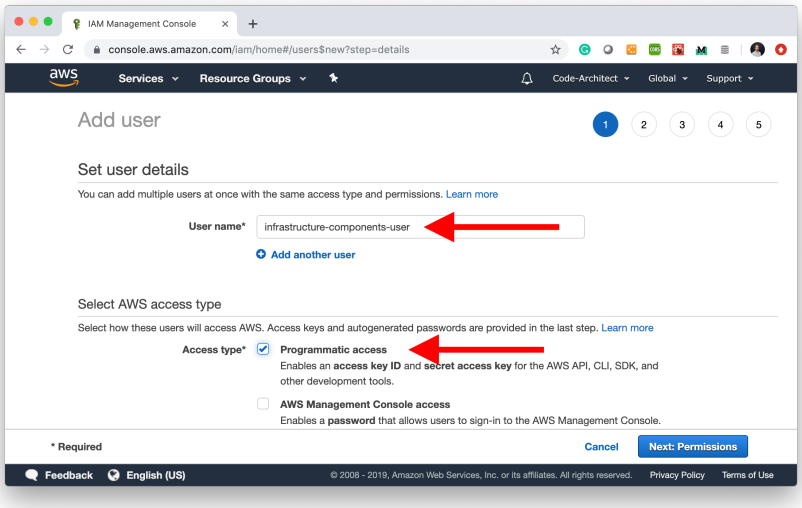


Figure 4.4: The new user dialog

Select **Attach existing policies directly** and click **Create Policy**

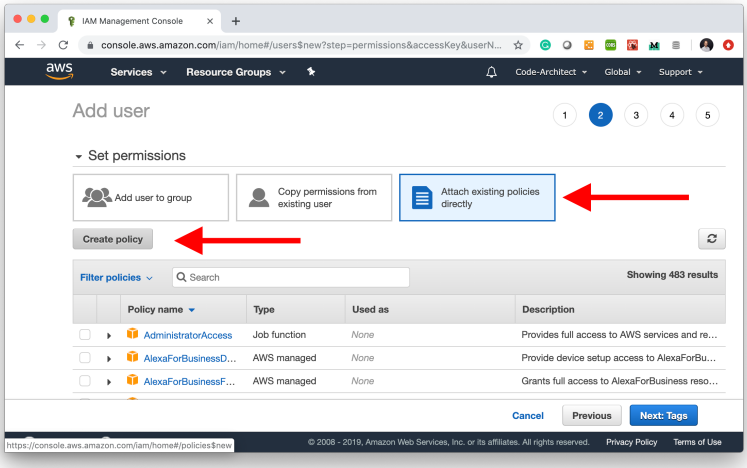


Figure 4.5: Attach policies

The policy creation dialog opens in a new browser window. Select the **JSON** tab.

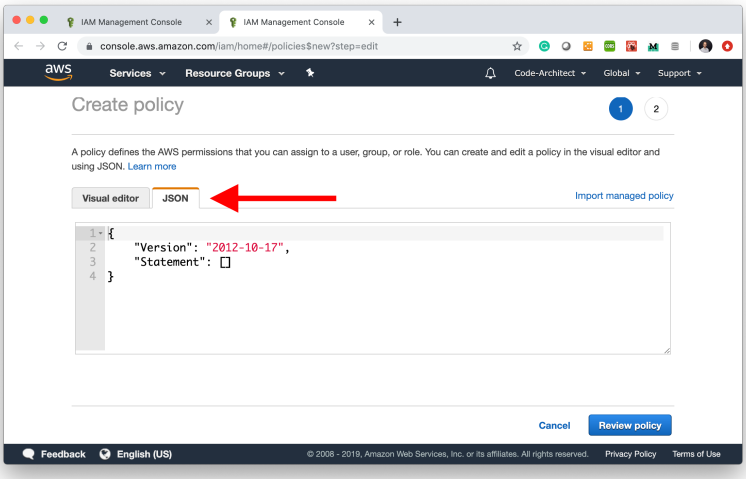


Figure 4.6: The create policy dialog

Enter the following policy into the field and click **Review Policy**.

Listing 4.5: The user policy

```
{
 "Statement": [
  {
   "Action": [
    "s3:*",
    "apigateway:*",
    "lambda:*",
    "logs:*",
    "cloudformation:*",
    "cloudfront:*",
    "acm:ListCertificates",
    "route53:ListHostedZones",
    "route53:ListResourceRecordSets",
    "route53:ChangeResourceRecordSets",
    "route53:GetChange",
    "iam:CreateRole",
    "iam:DeleteRole",
    "iam:DeleteRolePolicy",
    "iam:GetRole",
    "iam:PassRole",
    "iam:PutRolePolicy",
    "dynamodb:*",
    "execute-api:ManageConnections"
   ],
   "Effect": "Allow",
   "Resource": "*"
  }
 ],
 "Version": "2012-10-17"
}
```

In the review policy step, enter a **Policy Name** and an optional **Description**. Click **Create Policy**.

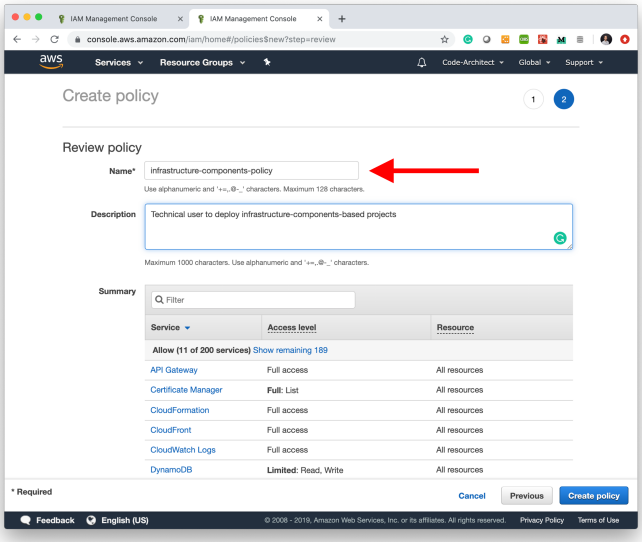


Figure 4.7: Review policy

Return to the browser window in which you started adding a user. The list of predefined AWS policies is pretty long. In order to find your newly created policy, you can filter the list by typing its name. Once you see it, select your policy.

Click on **Next: Tags**.

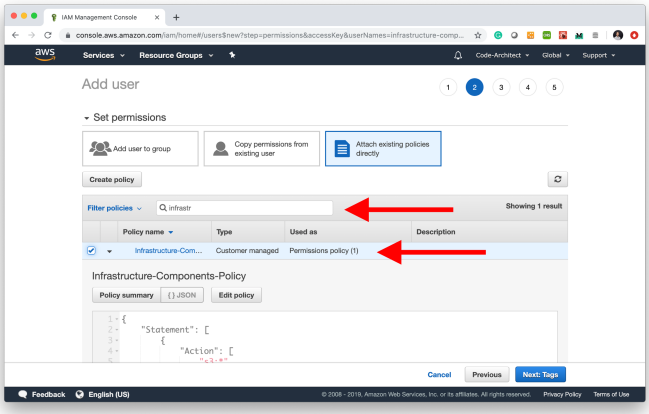


Figure 4.8: Select your policy

We don't need to add tags to our IAM user. Click **Next: Review**.

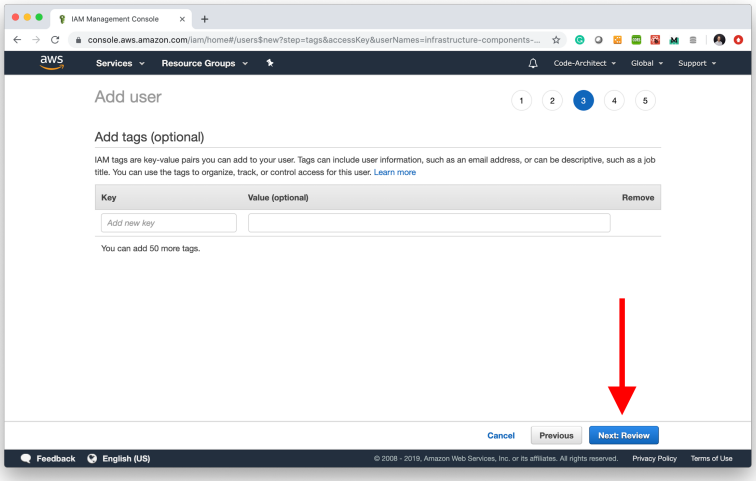


Figure 4.9: Add tags

In the user review page, select **Create user**.

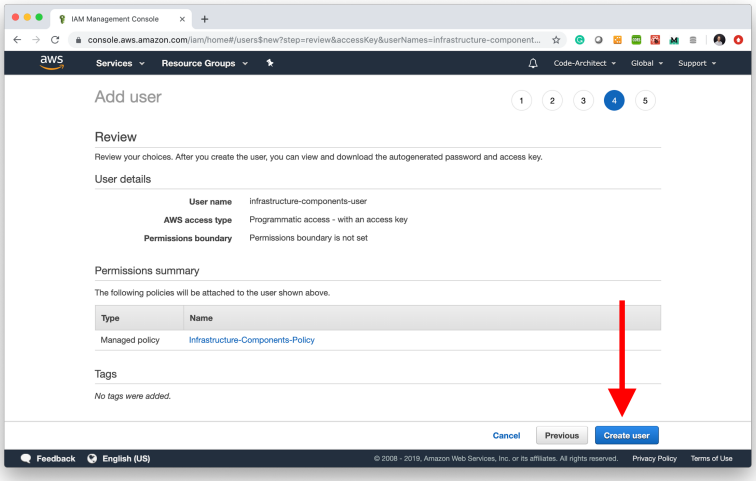


Figure 4.10: Review user

Once the user is created, you can get the credentials. Select **Show** to reveal the `secret access key`.

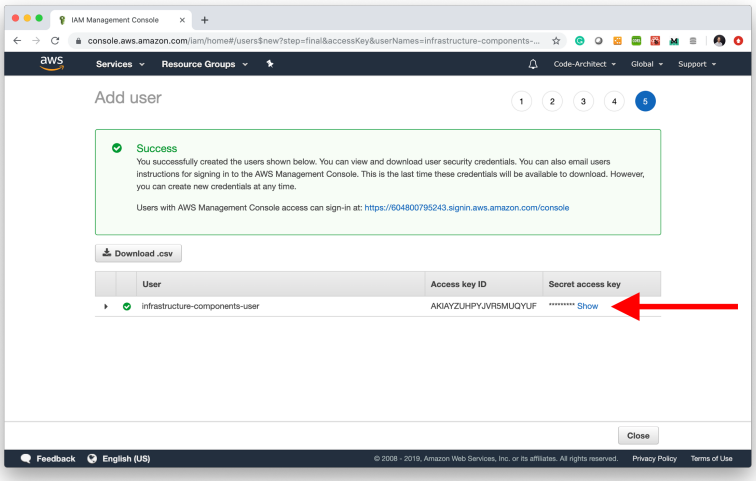


Figure 4.11: Get the credentials

Finally, put the `AWS key ID` and the `secret access key` into your `.env`-file at your project's root directory.

Listing 4.6: The `.env`-file with AWS credentials

```
1 AWS_ACCESS_KEY_ID=***...***
2 AWS_SECRET_ACCESS_KEY=***...***
```

Make sure that you treat your ID and Key securely.

With the credentials, this user logs into your account and deploys your app (automatically). You can use the same user for all of your projects.

During the run-time of your app, AWS uses a different user. This run-time user gets created automatically during the deployment. It has fewer access rights.

4.3.2 Register A Custom Domain

Once we deploy our React app to AWS, it will be available at a technical URL like `https://xxxxxxxxxx.execute-api.us-west-1.amazonaws.com` (isomorphic apps) or `http://example-bucket.s3-website.us-west-2.amazonaws.com` (single-page and service-oriented apps)

These URLs are not easy to remember and may not represent your brand (if you have one). Of course, having a web-app should come with an URL on a custom domain, like `www.my-web-app.com`.

The registration of a domain is optional. You don't need one to follow the content of this book. You can add a domain to your project at any later time.

A domain incurs annual costs, e.g. a `.com`-domain costs 12$/year.

Step 1: Register Domain

We start with the registry of a domain. Open Route53 — Domain Registration: `https://console.aws.amazon.com/route53/home?#DomainRegistration`. You'll see a form that asks you to enter a domain name and select the top-level-domain (e.g. `.com`).

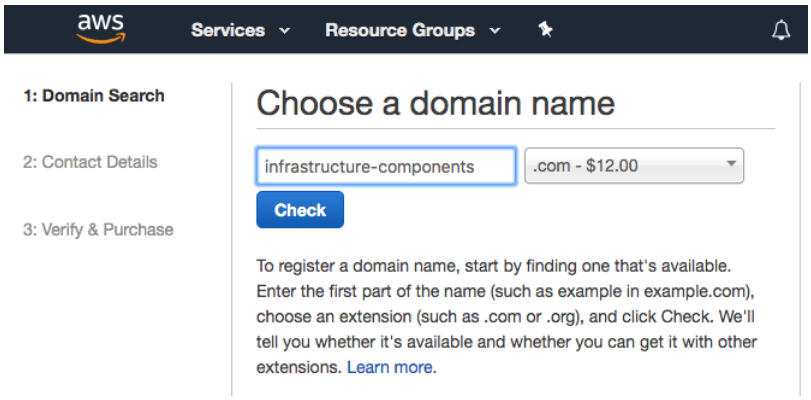


Figure 4.12: Choose a domain name

Enter the domain, you'd like to register (without any subdomain, as `www`), select the top-level in the dropdown, and click check. You'll get a list of available domains as depicted in the following image. Add the domain to your cart.

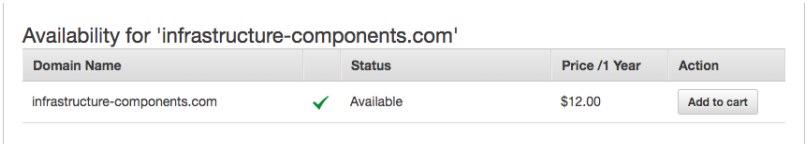


Figure 4.13: Check domain availability

Once your cart contains the domain, click continue. In the next screen, you need to add your contact details.

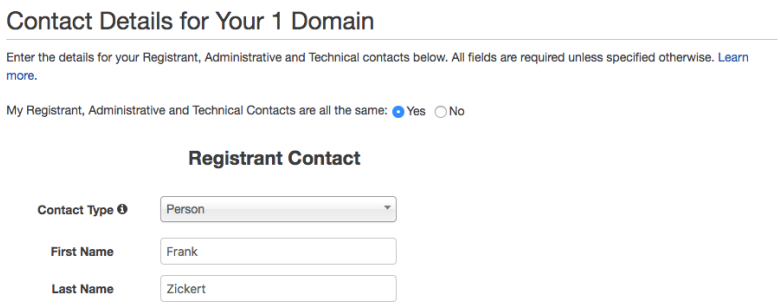


Figure 4.14: Enter contact details

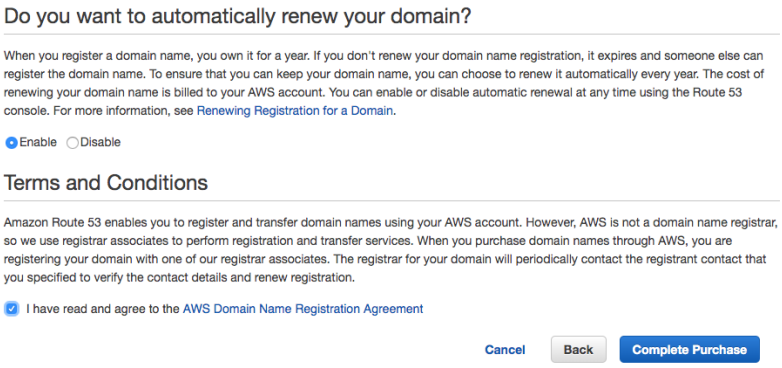


Figure 4.15: Renewal and Terms

On the last screen, you get a summary of your details. You can select whether your domain should be renewed automatically (otherwise it expires after a year) and you have to accept the terms and conditions. Click "Complete Purchase" and you're done!

The registration process may take some time. Amazon says that it takes up to three days. My experience is that it is done within 30 minutes. But be patient. If something does not work, Amazon will let you know.

Before you proceed with the next step, make sure your domain registration is completed. Go to the listings of your registered domains: `https://console.aws.amazon.com/route53/home?#DomainListing`. If you see it listed there, you're good to continue.

Step 2: Getting a certificate for your domain

These days, you need to serve your website over https (SSL-encrypted). Not only browsers warn your users if you don't support https. But AWS does not even support unencrypted websites anymore.

So, let's get a certificate. Once you registered your domain, re-

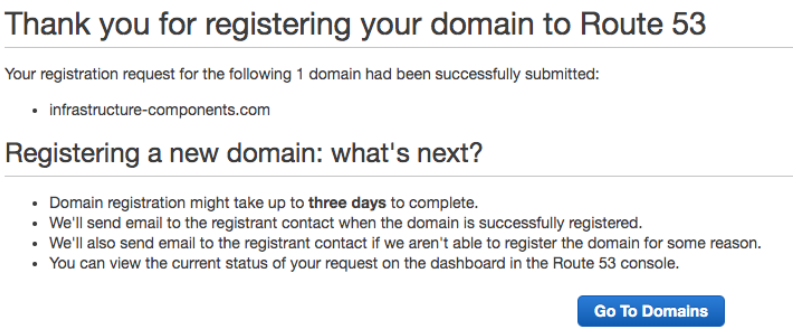


Figure 4.16: Done

quest a new certificate with the AWS Certificate Manager at `https://console.aws.amazon.com/acm/home?region=us-east-1#/`

You'll need to be in region us-east-1. This applies to the certificate, only. You can host your app in any of the other regions.

In the Certificate Manager, click the "Request a certificate" button. In the following dialog, select the option "Request a public certificate".

Then, enter the domain including the sub-domain (e.g. `www`) and the top-level domain (e.g. `.com`). If you plan to use different subdomains, you can add them here as well. You can also use wildcards, such as *. Proceed by clicking next.

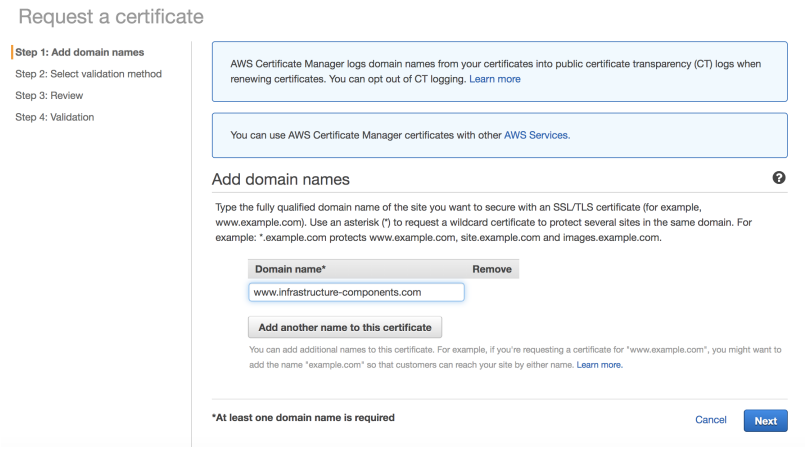


Figure 4.17: Enter domain

The next screen asks you to select the validation method. Select DNS validation as depicted in the following image.

In the summary, review your request and press "Confirm and request". The status is now Pending validation. Open the accordion for the details.

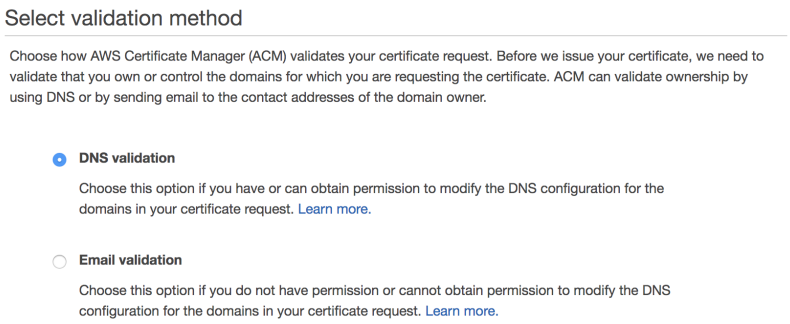


Figure 4.18: Select DNS validation

Click "Create record in Route53". Then, you'll need to wait up to 30 minutes. In my experience, it takes a few minutes, only.

In the final overview, you can see the ARN (the Amazon Resource Name) of your certificate. It looks like this: `arn:aws:acm:us-east-1:xxxxxxxxxxxx:certificate/xxxxxxx...xxxx`. Copy it. You'll need it in the next step.

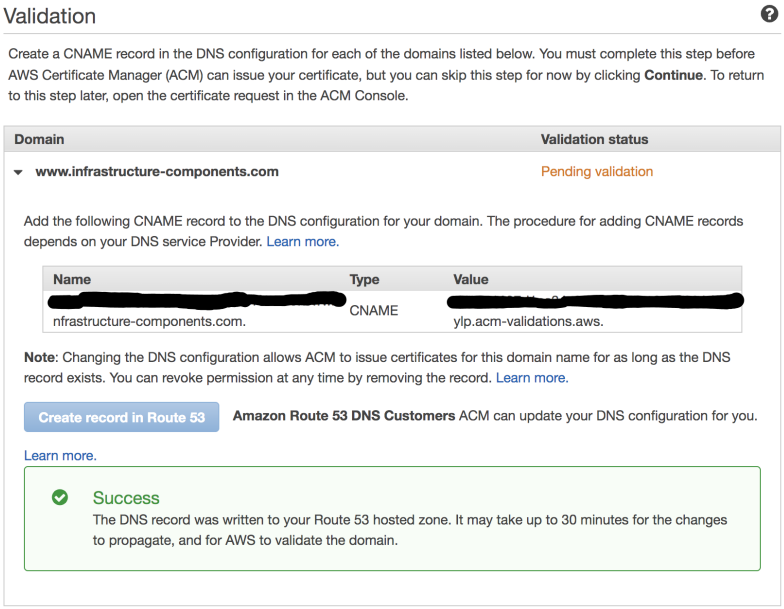


Figure 4.19: Get the certificate ARN

4.3.3 Verify An Email Address

In later chapter chapter 7, we will be sending emails.

The verification of an email address is not required for most of the examples

used in this book. But you'll need a verified email address to send emails using AWS SES.

In your AWS-console, open the SES menu:
`https://console.aws.amazon.com/ses/home?region=us-east-1` and select "Email Addresses".

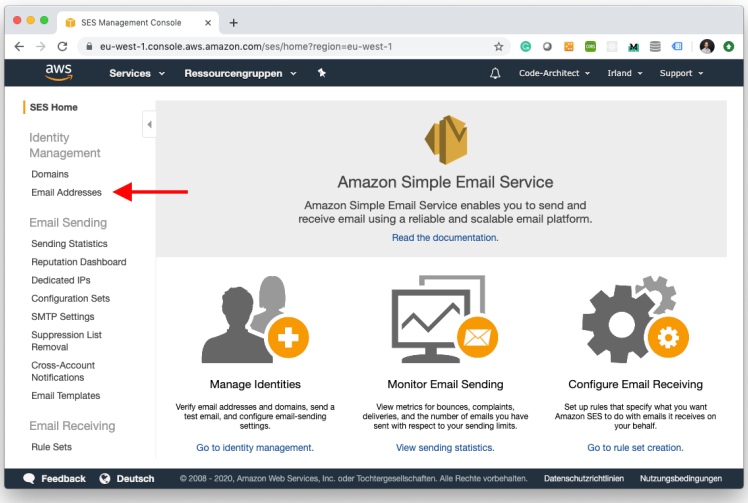


Figure 4.20: AWS SES

There, select "Verify a New Email Address".

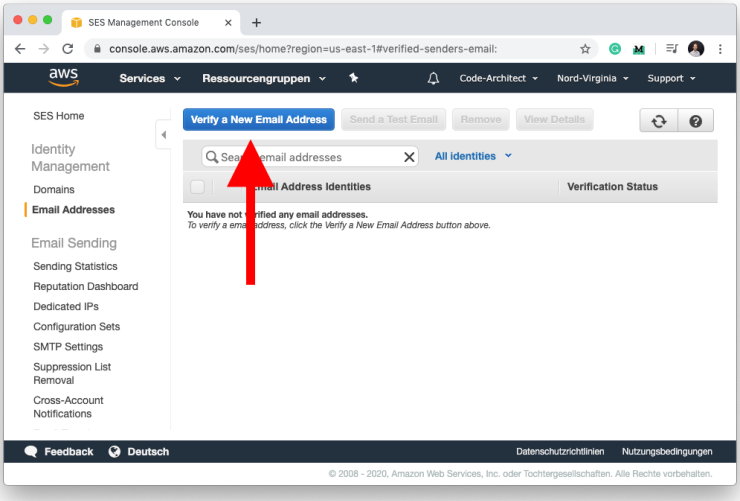


Figure 4.21: Verify email address

In the next step, enter and submit your email address.

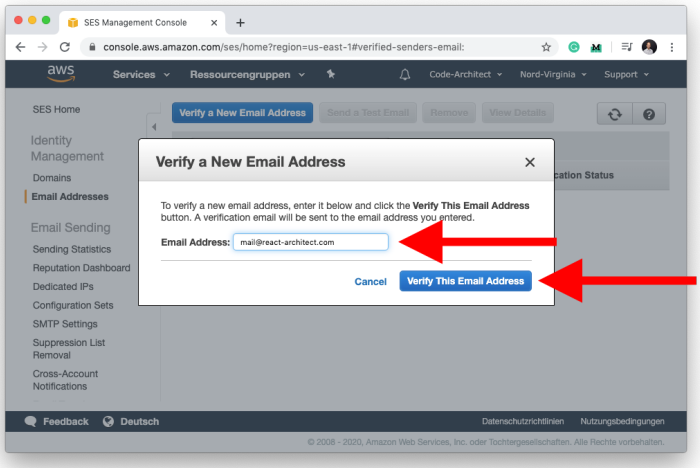


Figure 4.22: Enter email address

AWS sends a confirmation email to the entered address to verify that you are its owner.

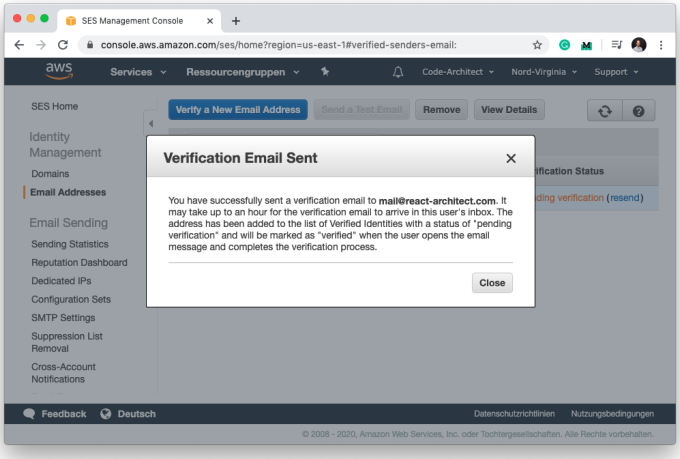


Figure 4.23: Email sent

The status of your email is pending until you click the link in the confirmation email.

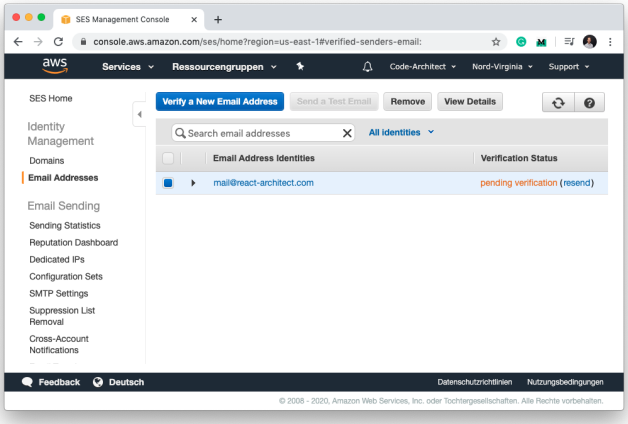


Figure 4.24: Pending status

In the meanwhile, look at your inbox. You should have received the email from AWS. Click the link in the email to verify your email address.

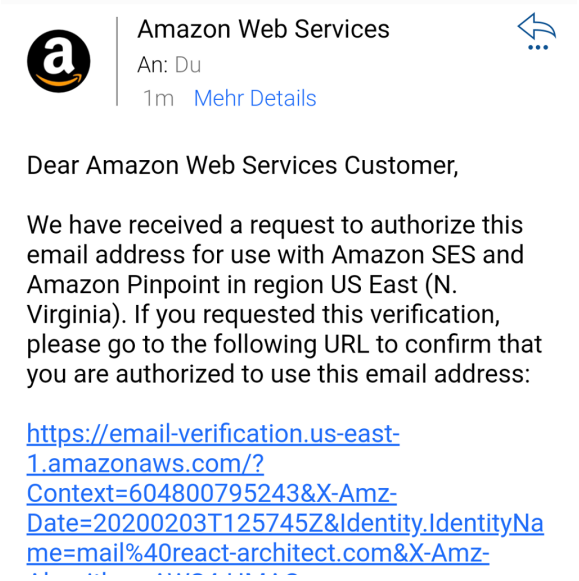


Figure 4.25: Confirmation email

Once verified, the address' verification status is "verified". Click the email address to see its details. There you can see the ARN that looks like this:
`arn:aws:ses:{your-region}:{your-account}:identity/{your-email-address}`

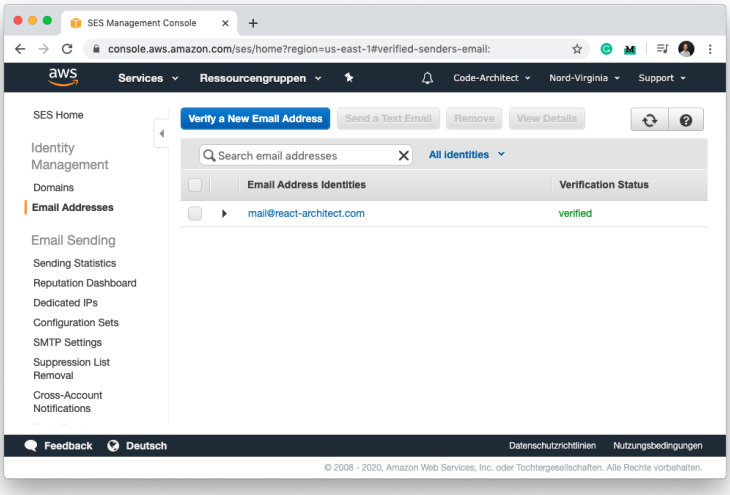


Figure 4.26: Get the ARN

Enter this ARN in the Resource-part of the `iamRoleStatements` that we provide as a property to `<ServiceOrientedApp />` or `<IsomorphicReactApp />`.

The `iamRoleStatements`-block adds custom rights to the runtime user of your app. The following statement allows the app to send emails (as HTML or plain text) from your verified email address.

Listing 4.7: Add the email

```
<IsomorphicApp
  stackName = "react-architect"
  buildPath = 'build'
  assetsPath = 'assets'
  region='us-east-1'
  iamRoleStatements={[
    {
      "Effect": "Allow",
      "Action": [
        '"ses:SendEmail"',
        '"ses:SendRawEmail"',
      ],
      "Resource": '"arn:aws:ses:us-east-1:xxxxxxxxxxx:identity/you@domain.
    com"',
    }
  ]}
/>
```

AWS checks your sending statistics. If you have too many rejections (invalid recipient addresses) or recipient complaints, your email address will be banned. Do not spam anyone!

4.4 Run The Scripts

Once we have set up the development environment, created the files, and have the credentials ready, we can develop, start (locally), and deploy our app with simple commands.

4.4.1 Install

The first thing we need to do is to install all the project dependencies. We use `npm` for that. Run the command `npm install` (or `npm i`) from the root of your project folder.

How to manage your dependencies with npm

`npm` provides short and long versions of its parameters. Both are equal. You can use `npm install` or `npm i`.

If you provide no further argument, as we do here, `npm` installs all the libraries you specified in the `dependencies` and `devDependencies` sections of your `package.json`.

If you want to install a single library, you can append it to the command:

```
1 npm install <name>
```

This command only installs the package with the specified name. It does not add it to the `dependencies` of your project. If you want to add it you need to specify

- `-S` or `--save` to add it to the `dependencies`
- `-D` or `--save-dev` to add it to the `devDependencies`

The following commands install `infrastructure-components` and add it to the `dependencies`:

```
1 npm install infrastructure-components --save
```

or

```
1 npm i infrastructure-components -S
```

Have a look at the official documentation for further details.

4.4.2 Build

The `npm run build` script checks your project for errors. Further, it analyzes your app's architecture and environments. You only need to run it at the beginning of the setup or when you add or change the architecture or an environment.

If you only change the code of your "normal" React components, you don't need to rerun `npm run build` again.

The build process adds further scripts to your `package.json`.

These scripts let you start your app offline and deploy it to AWS.

4.4.3 Start Your App Offline In Hot-Development Mode

The `build` script adds the following script to your `package.json`:

Listing 4.8: Script to run your app offline in hot-development mode

```
1 <your_app_name>: scripts <your_app_name> src/index.tsx
```

You can run it with `npm run <your_app_name>`.

If you defined a single-page or a service-oriented app architecture, `<your_app_name>` is the `stackname` of your `<SinglePageApp/>`-component. If you defined an isomorphic app, `<your_app_name>` is the `name` of the `<WebApp/>`-component.

Once you started the script, Wait until the console says that your app is running and open `localhost:3000` in your browser. You should see your app now. Changes to your source code become effective immediately in this mode. Just reload the page in the browser (F5).

This mode does not start your back-end resources. Use it to develop the user-interface of your app.

4.4.4 Start Your App Offline With Back-End Support

If you specified a `<ServiceOrientedApp/>` or an `<IsomorphicApp/>`, the `build`-script adds the following script for each `<Environment/>`-component to your `package.json`:

Listing 4.9: Script to run your app offline with back-end support

```
1 "start-<your_env_name>": "scripts .env start src/index.tsx <your-env-name>"
```

You can run it with `npm run start-<your_env_name>`.

This command starts the complete app locally. Including the back-end resources, such as API-endpoints and the database (if you have one). Only, sending emails does not work locally.

In the case of a `<ServiceOrientedApp/>`, your web-app runs on `localhost:3000`, and the services run on `localhost:3001`.

In the case of an `<IsomorphicApp/>`, the app and the services run on `localhost:3000`.

In both cases, the local DynamoDB database runs on `localhost:8000`

In this mode, changes do not apply instantly. You need to stop the stack (ctrl + c) and restart it.

Starting the whole software stack locally is only available for `<ServiceOrientedApp/>` and `<IsomorphicApp/>`-architectures. The simple reason is: a `<SinglePageApp/>` does not support any back-end resources.

4.4.5 Deployment

If you have specified your credentials in your `.env`-file, you can deploy your app to AWS with a single command: `npm run deploy-<your_env_name>`

Replace `<your_env_name>` with the name you specified in the `<Environment/>`-component. Infrastructure-Components support multiple environments per project. For instance, it is good practice to have a `dev`-environment and a `prod`-environment.

Once you started the `deploy`-script, it creates the whole infrastructure stack on your AWS account. You'll get back an URL like one of the following that now serves your app. Have a look at the output of your computer's console.

Listing 4.10: url of single page and service-oriented apps

```
1 https://<your_app_name>-<your_env_name>.s3.amazonaws.com
```

Listing 4.11: url of isomorphic apps

```
1 https://<something>.execute-api.<your_region>.amazonaws.com/<your_env_name>/
```

The deployment of the software stack may take some time (up to an hour for the first deployment). Subsequent deployments will be much faster.

4.4.6 Initialize Domain

If you registered a domain you need to connect it to one of your `<Environment />`s. You only need to do this once.

You can only connect a domain to an `<Environment />` if it has been deployed before!

If the environment has been deployed, you cna add two more properties to it: `domain` and `certArn`. Specify your full domain (including sub-domain and top-level-domain) and the ARN of your certificate as shown in the following code:

Listing 4.12: Example of an `<Environment />` with domain and certificate

```
/** src/index.tsx */
export default (
 <SinglePageApp
  stackName = "my-stackname"
  buildPath = 'build'
  region='eu-west-1'>

  <Environment
   name="prod"
   domain="www.infrastructure-components.com"
   certArn="arn:aws:acm:us-east-1:xxx:certificate/xxx"
  />
  <Route
   path='/'
   name='Infrastructure-Components'
   render={(props)=> <App />}
  />

 </SinglePageApp>
```

Since we changed an `<Environment />`, we need to run the `build`-script again: `npm run build`. The build-script adds another script to your `package.json`: `scripts .env domain src/index.tsx`. Run it with the command: `npm run domain-{your-env-name}`.

If you use a `<SinglePageApp />` or a `<ServiceOrientedApp />`, this command creates a CloudFront-Distribution in AWS. New domains may take up to 40 minutes to be initialized. In your development console, it shows `Serverless: Checking Stack update progress...` during that time.

The domain-initialization script adds an entry to your `.env-` file: `DOMAIN_{your-env-name}=TRUE`. This entry tells Infrastructure-Components to keep the domain the next time you deploy your app. **Make sure you keep this entry**. Make sure to have this entry in all the copies of this file. If you develop your app on different computers.

Once the script finishes, you have to redeploy your app. Run `npm run deploy-{your-env-name}`. You may notice that when you hit your address in a browser, it redirects to another URL rather than serving your website directly from the domain you specified. This is a temporary redirect that exists for about a day. AWS removes this redirect automatically.

Listing 4.13: Temporary redirect

```
https://{your_app_name}-{your-environment-name}.s3.amazonaws.com
```

When you attached a domain to an `<Environment />` of your app, you have some limitations regarding changing your app architecture. You can change it from `<SinglePageApp />` or a `<ServiceOrientedApp />` (or vice versa). But you cannot change it from or to `<IsomorphicApp />` anymore. Because an `<IsomorphicApp />` is handles domains completely differently. It uses AWS API Gateway. Whereas an `<SinglePageApp />` and an `<ServiceOrientedApp />` use AWS CloudFront. **If you want to apply any of the unsupported changes, you need to remove your app from AWS first**.

4.5 Remove An App From AWS

Infrastructure-Components use the Serverless-framework and AWS CloudFormation to deploy your app.

In the AWS console, open https://console.aws.amazon.com/cloudformation/homeCloudFormation. Here, you can see all deployed apps. Make sure you are in the right AWS region. Each deployed environment of your app is a separate CloudFormation stack.

If you want to delete an app, select it and click the "Delete" button.

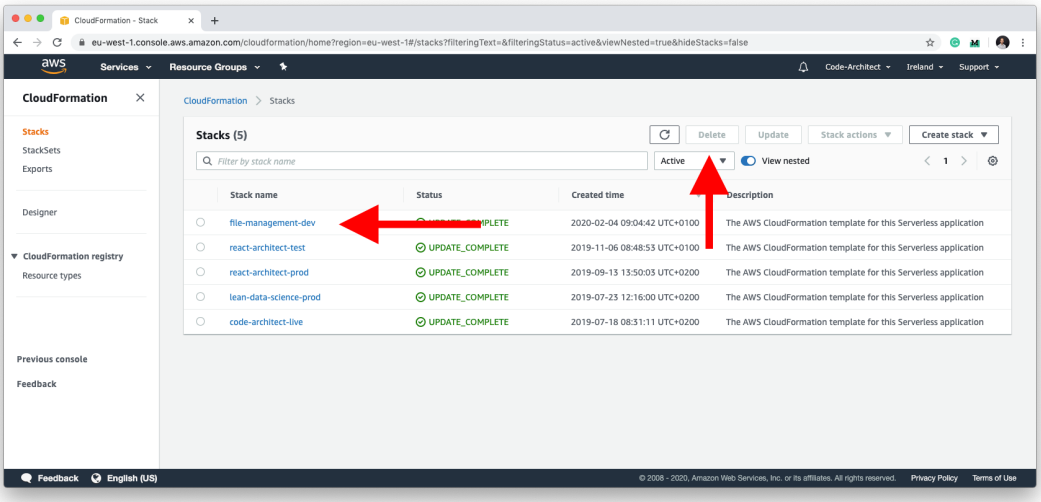


Figure 4.27: AWS CloudFormation

Confirm the deletion if you are sure.

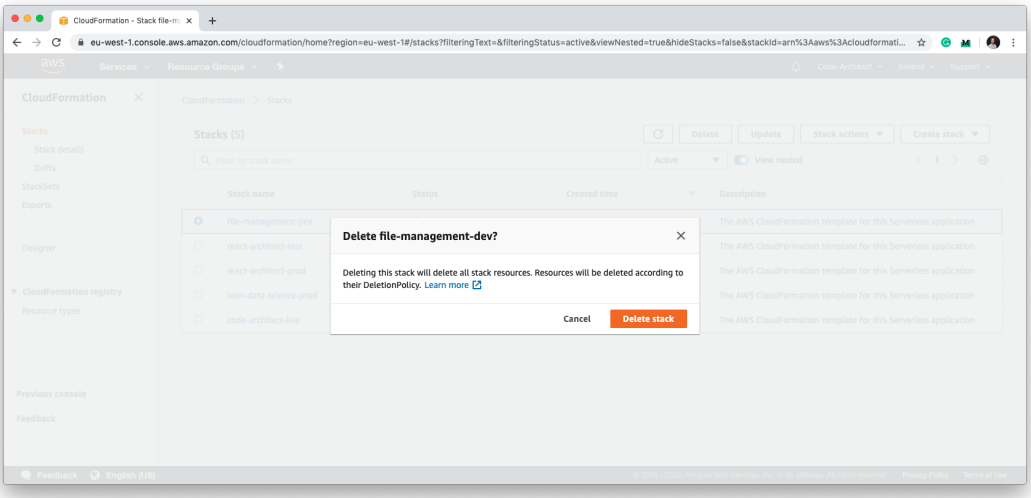


Figure 4.28: Confirm deletion

The deletion will take a while. If you connected a domain to the deployed environment, the deletion may take even more time. And, unfortunately, it will fail.

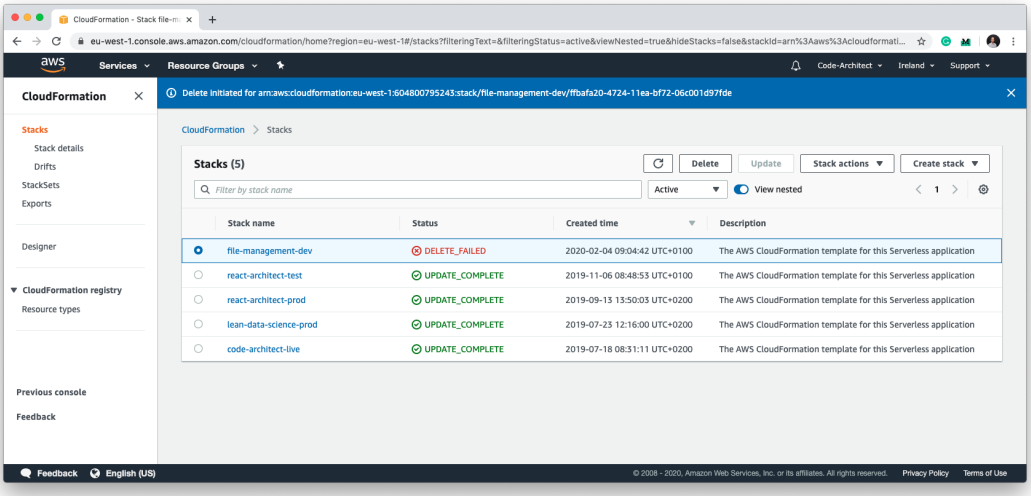


Figure 4.29: Failed deletion

The reason for the failure is the S3 buckets. S3-Buckets can't be deleted if they are not empty. But the S3-deployment bucket contains the files we require to manage the CloudFormation stack. Including deleting it.

Once the stack status says `DELETE_FAILED`, select and delete it again. This time, you get a list of all the resources that AWS could not delete automatically. Check them. We will delete them manually.

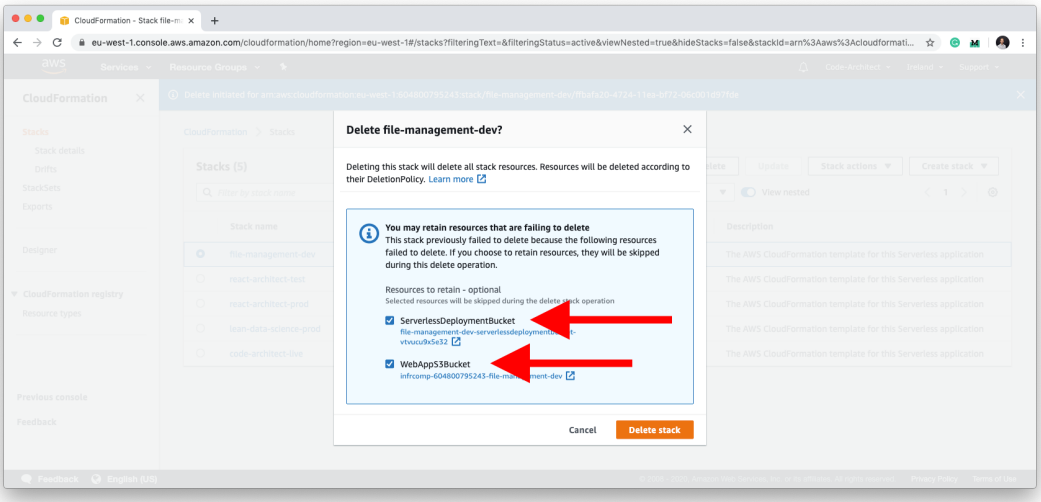


Figure 4.30: Failed deletion

Open the AWS S3 management console (https://s3.console.aws.amazon.com/s3/home). You'll get a list of all buckets in your account.

Select the main bucket and click "Delete". The main bucket has the name `infrcomp-{your-AWS-account}-{your-stackame}-{your-env-name}`.

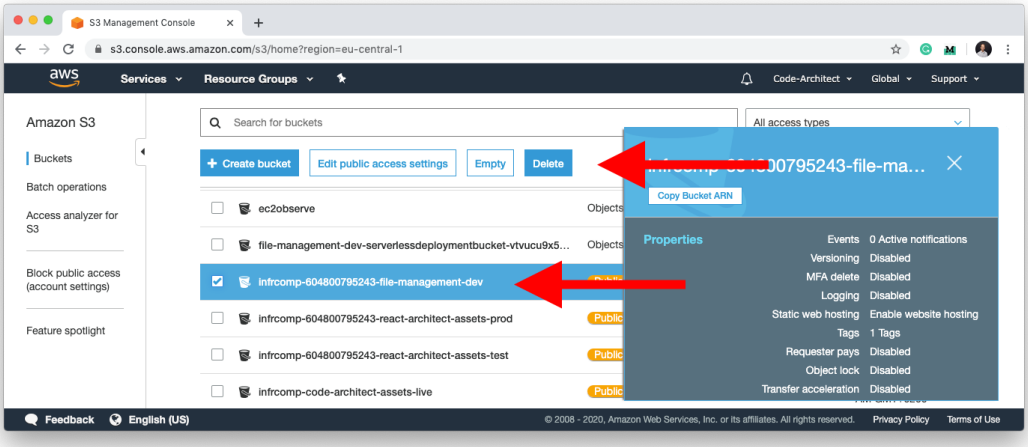


Figure 4.31: AWS S3 Console

You'll get a confirmation request. Copy the bucket's name into the field and proceed.

Do the same for the deployment-bucket of your app. It has the name `{your-stackame}-{your-env-name}-serverlessdeploymentbucket-{arbitrary-string}`.

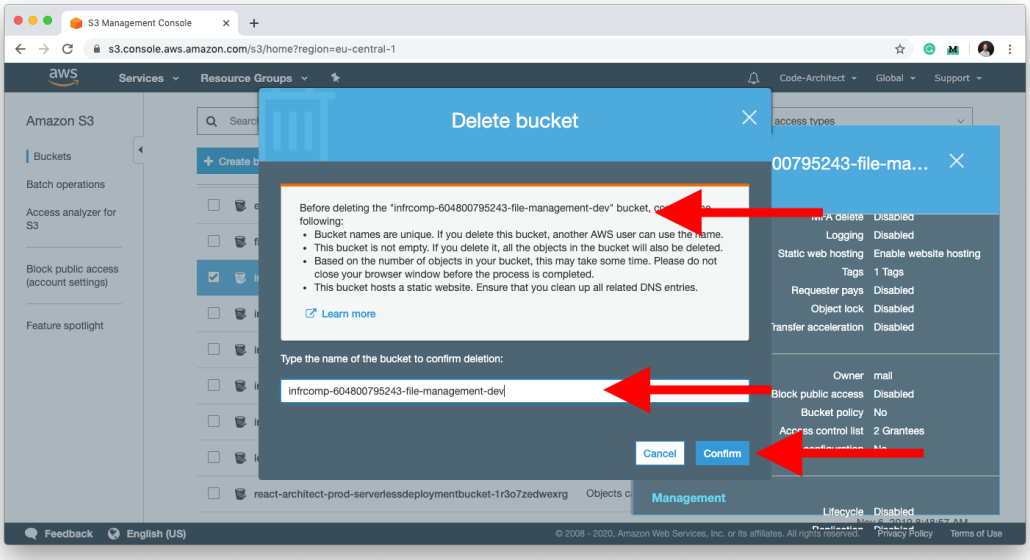


Figure 4.32: Confirm S3 deletion

If you connected the deleted environment with a domain, you need to edit your `.env`-file. Remove the `DOMAIN_{your-env-name}=TRUE` entry.

If you deployed your IsomorphicApp with a domain, go to API Gateway. https://console.aws.amazon.com/apigateway/main/apis Make sure to select the right region. Select the menu item Custom domain names. Delete the domain assignment by clicking the x in the top-right corner. Conditions for Domain. Not assigned to a resource. look at API-Gateway `https://eu-west-1.console.aws.amazon.com/apigateway/home?region=eu-west-1#/custom-domain-names` (serverless-domain-manager creates a cloudfront that is managed by AWS and cannot be seen by the user) CloudFront (may also use it)

If you want to delete all resources, don't forget to delete the certificate (in the Certificate Manager) and the technical deployment user (in IAM). However, you can reuse both for future projects.

4.6 Summary

Usually, app development involves a lot more configuration. We could have discussed `Webpack`, `Babel`, and the `Serverless-cli`. When we think about full-stack apps, these configurations get pretty large. And they are cumbersome to write and to manage!

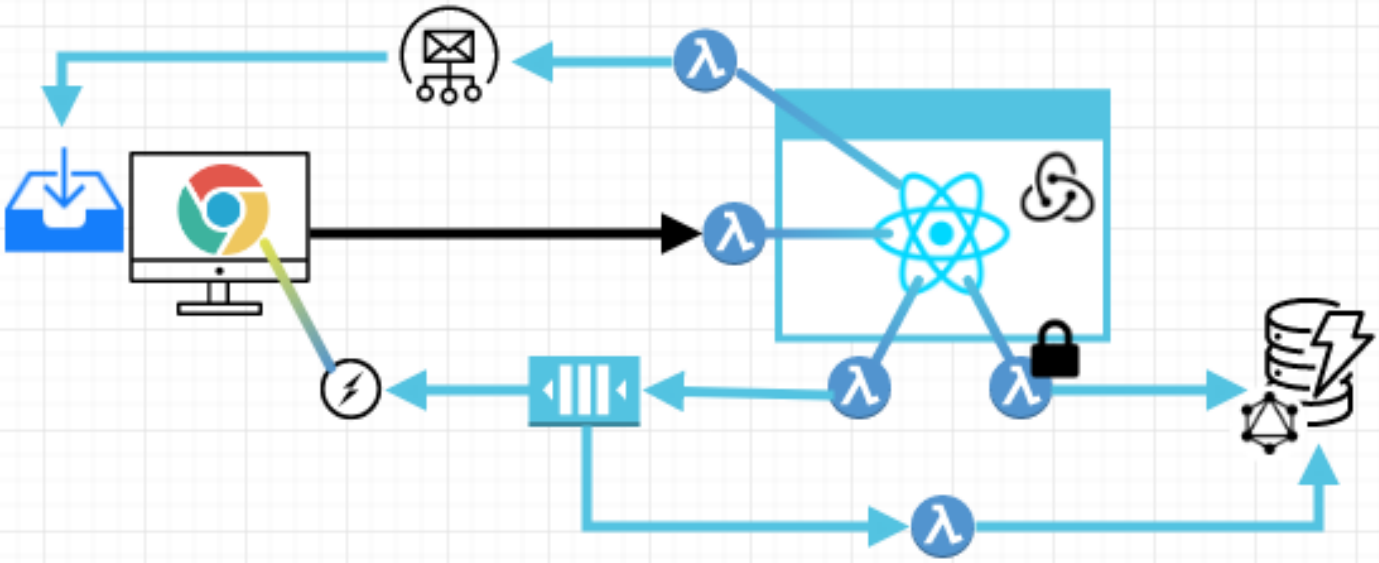
Rather than spending many pages on these infrastructure-related topics, we covered the approach we follow in this book: **Architecture-as-a-function**.

With the **architecture-as-a-function**-approach, you can get started easily. There are just the following steps:

- Once at all: Set up an AWS account and a user
- Once per development environment: Install `Node.js` and `Java JDK`
- Once per project: Create project files (manually, clone repository, or use configurator)

During the development, scripts automate all the repetitive tasks. Just like with `create-react-app`, we can do the following actions with a single one command:

- install dependencies (`npm i`)
- run the app locally in hot-development-mode (`npm run <your_app_name>`)
- run the whole software stack locally (`npm run start-<your_env_name>`)
- deploy the app to AWS (`npm run deploy-<your_env_name>`)



5. Serverless Single-Page React App (SSPRA)

There's a lot of information in the term "Serverless Single-Page React App". Let's look at what it tells us first.

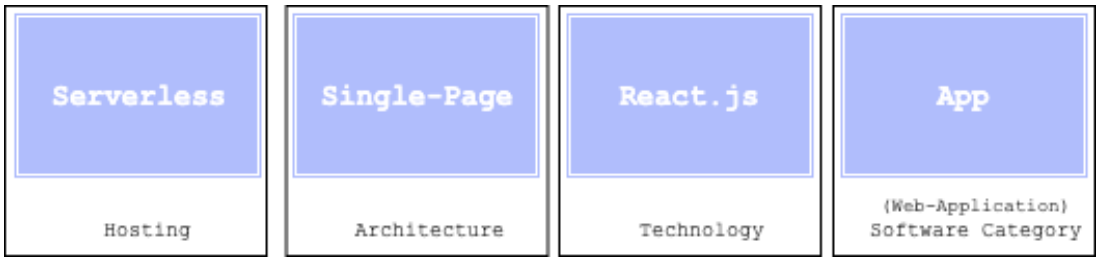


Figure 5.1: What the term "Serverless Single-Page React App" tells us

In our context, an *"app"* is short for web-application. It specifies the category of our software. A web-application interacts with the user dynamically. It avoids interruptions of the user experience between successive pages. This way, it distinguishes from web-sites. It rather appears to be a desktop application.

React.js is the technology we use. React is a JavaScript library that creates and manipulates an HTML5-compatible Document Object Model (DOM). React allows the developer to write code in a declarative way as if the whole page is rendered on each change. Technically, React renders only the sub-components that change. It uses a virtual DOM. That is an in-memory data-structure cache. React computes the differences to the real DOM and updates it efficiently.

"Single-Page" is the architecture of our app. It implies that our whole app consists of a single HTML page. Rather than loading entire new pages from a server, it rewrites the current page.

Serverless computing is a cloud-computing execution model. In this model, the cloud provider does not provide dedicated (real or virtual) servers. But the provider manages the allocation of resources dynamically. When required, your code runs on myriads of servers concurrently. If not required, your code runs on a single one server or not at all. Big enterprises appreciate the ability to scale up. From the perspective of a solo-developer, the ability to scale down to zero provides a huge value. If your service isn't used, it incurs very little (a few cents or bucks) or no costs.

5.1 The Architecture Of An SSPRA

The following image depicts the infrastructure architecture of a Serverless single-page app.

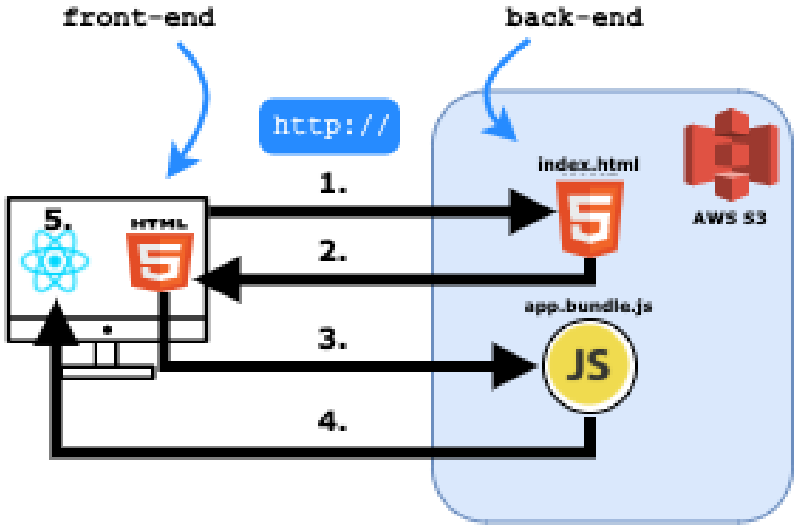


Figure 5.2: The architecture of a Serverless Single-Page React App

Like any web-application, web-site, or web-service, the life cycle of the Serverless Single-Page React App starts with an HTTP(s)-request (**step 1**). The user's browser sends a `GET`-request to the server hosting your app. This server is the *AWS Simple Storage Service (S3)*.

The address on S3 points to an HTML-file. Usually, its name is `index.html`.

> In this book, we use AWS cloud services. But Serverless Single-Page React Apps and all the other architectures we'll cover in this book are not specific to this provider.

How does AWS S3 work?
S3 manages data through an object storage architecture. Object storage (also known as object-based storage) is a computer data storage architecture that manages data as objects. Each object includes the data itself, a variable amount of metadata, and a globally unique identifier. The unique identifier allows S3 to manage physical hardware, data replication, and data distribution at object-level granularity. At the same time, the identifier provides fast and direct access from anywhere.

S3 provides scalability, high availability, and low latency.

In **step 2**, S3 sends the content of the `index.html` in its HTTP-response to the browser. It looks like this:

Listing 5.1: The `index.html` S3 sends to the browser

```
1 <html>
2  <body>
3   <div id="root"></div>
4   <script src="some/path/app.bundle.js"></script>
5  </body>
6 </html>
```

The `index.html` is pretty basic. But it contains all we need. We have an empty `<div/>`-element. Thus, when the browser displays the response, we see a blank screen. For now. And we load a script from `"some/path/app.bundle.js"`. This is the relative path to the code of our React app on S3.

The browser calls this path (**step 3**) and gets the code of our React web-app as the response (**step 4**) from S3.

In **step 5**, the React magic happens. React renders the DOM and puts it into the empty `<div/>`-element. The user can see and interact with the web-app now. Foremost, React.js is a user interface library.

Everything from now on runs in the browser. All the logic resides within the downloaded code. This includes other resources, such as styling (CSS). The web-app does not reload (from the server) at any point in the process anymore. The web-app may communicate with APIs to load and send data. But the web-apps code does not change anymore.

Single-Page React apps use the location hash (e.g. `www.my-domain.com/#/`) to provide the perception and navigability of separate logical pages in the application. But it does not transfer the control to any other page or resource anymore.

Why don't we put the code of the React web-app directly into the `index.html`?
The JavaScript code can grow pretty big (several megabytes). Depending on the user's internet connection, it can take a while to load the code. If it was included in the `index.html`, the browser would not receive any response to its first request to the server for that time. The browser might time-out before loading the web-app.

Loading the code separately does not increase the download speed. But the browser gets a response quickly. It won't time-out and won't show an error message.

With this understanding of a Serverless Single-Page React App, we can conclude on some things. The initial request (step 1) will be answered pretty fast. We just load a static resource from S3. Unbeatable! But the user does not yet see something.

First, the browser has to process the HTML, which is pretty fast, too. It is a very simple HTML. But loading the script may break the deal here. It pretty much depends on the size of our app and the internet connection speed. Decisive factors of the app size are the resources and assets, such as images or videos.

You can't increase the overall downloading speed. But you can improve the user experience. You can add some content to the `index.html`, such as a CSS-based loading animation. And you can split your `app.bundle.js` into multiple pieces and thus reduce the initial package size.

A Serverless Single-Page React App works with static files. It serves the same code and assets to each user. Regardless of who your user is and regardless of any other contextual information, such as the time of the day or your current state of business (e.g. you ran out of stock). But static code does not imply static content.

Your React app consists of JavaScript code running at the client-side. This is a powerful tool. You can load individual user data and any other contextual data dynamically in your code.

What about search engine optimization (SEO)? Before we answer this question, we shed some light on how a search engine works. Search engines have an index of all the content on the web. You may know an index from a book. It is a list of the keywords and a reference where to find it. If someone searches for a keyword, the search engine looks into its index and provides the list where the keyword appeared.

In general, search engines use web-scraping bots. A bot is not a person but a software. It fetches the content of your web-app by performing automated HTTP-requests (e.g. `GET`). And this is a problem. Because our Single-Page App returns our basic `index.html` as the response. This does not contain our content. It is our client-based JavaScript code that adds all the content we want the bot to put into the search engine index. Thus, the bot needs to run our JavaScript code to access our content.

Reportedly, Google can process client-side JavaScript. But other search engines may not (be able to) do it. Each search engine applies different algorithms to build and update its index. I would not dare say I know how these work in detail.

My best advice is to add relevant keywords to the static `index.html`.

5.2 Create, Start, and Deploy An SSPRA

It's time to get our hands on the code!

In this chapter, we create a file management system with React. We iteratively do this and adhere to the YAGNI-principle. We don't implement something unless we need it.

A Serverless Single-Page React app is sufficient to start with. Create the project files we mentioned in 4.2. The easiest way is to use the infrastructure-components configurator or to clone the empty project template. You'll find the complete source code of this chapter in this repository.

When you start with the Serverless Single-Page React App template, your `package.json` should look like the following code listing.

Listing 5.2: The `package.json` of a Serverless Single-Page React App

```
{
 "name": "file-management",
 "version": "0.0.1",
 "description": "",
 "scripts": {
  "build": "scripts .env build src/index.tsx"
 },
 "license": "MIT",
 "dependencies": {
  "infrastructure-components": "^0.3.10",
  "react": "^16.12.0",
  "react-dom": "^16.12.0"
 },
 "devDependencies": {
  "infrastructure-scripts": "^0.3.11",
  "serverless-domain-manager": "^3.3.1",
  "serverless-pseudo-parameters": "^2.5.0",
  "serverless-single-page-app-plugin": "^1.0.2"
 }
}
```

Don't forget to install the dependencies (`npm install`) before you start.

The following code listing depicts your `src/index.tsx`.

Listing 5.3: The source code of a basic serverless single-page React app

```
import React from 'react';

import {
 Environment,
 Route,
 SinglePageApp
} from "infrastructure-components";

export default (
 <SinglePageApp
  stackName = "file-management"
  buildPath = 'build'
  region='us-east-1'>

  <Environment name="dev"/>
  <Route
   path='/'
   name='Infrastructure-Components'
   render={ (props)=> (
    <div>Hello Infrastructure-Components!</div>
   )}
  />
 </SinglePageApp>
);
```

The `<SinglePageApp/>`-component takes only a few parameters. It takes the properties of a top-level architecture component (see: 4.3). The `stackName` is the (arbitrary) name of our app. This must be unique across your AWS account. Once the `src/index.tsx`-file is ready, you can build your project initially (`npm run build`).

The build-step adds a command to your `package.json` to start your single-page app in hot-development mode.

Just to make sure it works, let's start it locally (`npm run file-management`). (replace "file-management" by the name of your single-page app). When you open `localhost:3000` in your browser, you should see the text "Hello Infrastructure-Components!". So far so good.

5.3 Serving Static Files

The easiest way of serving files is to refer to them by their URL. For instance, you'll find the `index.html` of your app at the root-path (`/`).

Put the URL into a link (`<a/>`-tag). If you want to force the browser to treat the referenced file as a download, you can add the `download` and the `target="_blank"` properties to the link.

Listing 5.4: Serving a file by reference

```
1 <a href="index.html" download target="_blank">Index</a>
```

The problem with serving files by their URL reference is "side-effects". How do you make sure the file exists at the specified path? You have to put the file there manually. Alternatively, you can write a deployment script. Anyway, it is a side-effect. In your React-component, you rely on someone or something else to make sure the file you're referencing is at the path you're expecting it to be.

Another way of serving a static file is to add it to your code and import it. This is as easy as serving it by reference. In the following code-listing, we import the logo of our app that we added to the `assets`-folder.

Listing 5.5: Serving a file by reference

```
1 import logo from '../assets/logo.png';
2 <a href={logo} download target="_blank">Logo</a>
```

This approach reduces side-effects. But depending on the file type, it adds the whole file to our React app. This includes whole images. Whenever a user opens your app, she downloads the whole code, including the imported files. This increases loading time and worsens the user experience. For now, we're going to accept this. But we'll mitigate this problem later.

We create a separate component to show a list of our files. The `src/file-list.tsx`. Let's create a FileList-component for that. The following code depicts this component.

Listing 5.6: A list of files

```
1  import React from 'react';
2  import logo from '../assets/logo.png';
3
4  export default function () {
5    return <ul>
6      <li>
7        <a href={logo} download target="_blank">Logo</a>
8      </li>
9      <li>
10       <a href="index.html" download target="_blank">Index</a>
11     </li>
12   </ul>
13 };
```

The next listing shows how we integrate the component into our `src/index.tsx`.

We `export default` a function in `src/file-list.tsx`. Thus, we need to import the default from the module. Here's a good reference of the JavaScript import statement.

Listing 5.7: Integration of the `FileList` into the `src/index.tsx`

```
1  import React from 'react';
2
3  import {
4    Environment,
5    Route,
6    SinglePageApp
7  } from "infrastructure-components";
8
9  import FileList from './file-list';
10
11 export default (
12   <SinglePageApp
13     stackName = "file-management"
14     buildPath = 'build'
15     region='us-east-1'>
16
17     <Environment name="dev" />
18
19     <Route
20       path='/'
21       name='Infrastructure-Components'
22       render={(props) => <FileList/>}
23     />
24
25   </SinglePageApp>
26 );
```

The following image shows the list of files. If you click on a link, the browser should download the respective file.

5.4 Styling the entries

Modern websites and web-applications consist of three building blocks. HTML (HyperText Markup Language) defines the meaning and structure of web content. We use JavaScript to specify functionality and behavior. And there is the Cascading Stylesheet (CSS) language. It describes the style (that is the appearance) of the components. React combines HTML and JavaScript out of the box.

We could add a style to a component by pointing its `className`-property to a CSS-class we specify in separate CSS-files. While having all style-definitions in one place, this approach relies heavily on side-effects. Summarized in a global and hard to manage CSS-file. If you are a CSS pro, you can keep going this way. If you like. If not, I wouldn't recommend starting this way.

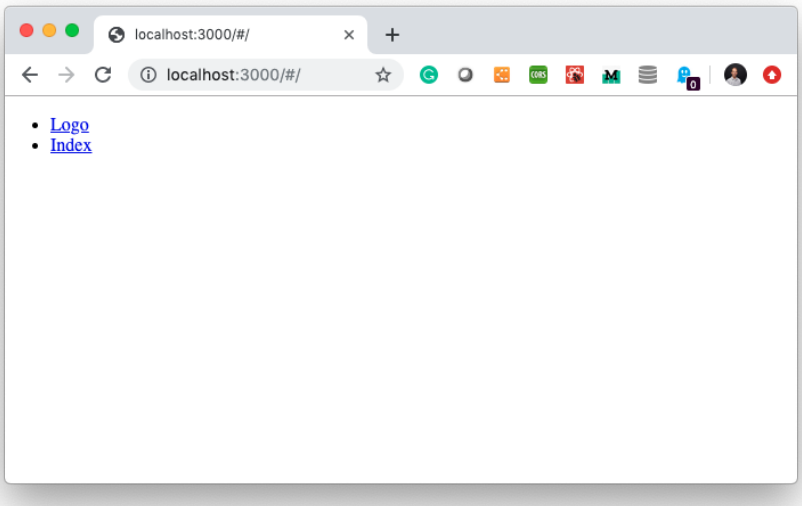


Figure 5.3: The list of files

We could also add styles by directly specifying it in a component's `style`-property. This is called an "Inline Style". Instead of a CSS string, you provide a JavaScript-object. Its keys are the `camelCased` versions of the names of the styles. Its values are the styles' values, usually strings. You apply a style to a component at the place of its use. You reduce side effects. But working with inline styles makes reuse cumbersome.

The third and most convenient way of managing style in React is with the help of the `styled-components`-library. It combines the advantages of CSS-files and inline style. Styled-Components lets you style your React components through local CSS literals. And it supports global themes. With styled-components, you can define the appearance of your components at the level of abstraction that best suits your problem at hand.

Let's have a look at our first styled component. We'll go through it right away. We skipped the parts of the code that did not change.

Listing 5.8: The first styled component in `src/file-list.tsx`

```
1  /* ...*/
2  import styled from 'styled-components'
3
4  const FileLink = styled.a`
5    background: gray;
6  `;
7
8  export default function () {
9    return <ul>
10     <li>
11       <FileLink href={logo} download target="_blank">Logo</FileLink>
12     </li>
13     <li>
14       <a href="index.html" download target="_blank">Index</a>
15     </li>
16   </ul>
17 };
```

We import the default module (`styled`) from the library in line 2.

> **Documentation — The `styled`-module.**
> The `styled`-module is the default export of the `styled-components`-library. This is a low-level factory that provides helper methods in the form of `styled.tagname`. The `tagname` is any valid HTML-tag. For instance, `styled.a` creates a link-component (`<a/>`). The `styled` and `styled.tagname`-functions take the CSS definitions in a template string (enclosed by backticks `` `...` ``). A template string is a multi-line string that allows embedded expressions. You embed an expression through a dollar character and a pair of curly brackets, like this: `${...}`.
>
> If you want to apply a style to an existing React component, you put it as an argument to `styled`. For example: `styled(MyComponent)`‘‘.
>
> See the official reference for more information.

We create a link-component (`<a/>`) (line 4) and apply a simple style to it (line 5). We specify the `background` to be `gray`.

At line 11, we replace the old `<a/>` by the `<FileLink/>`-component. This is nothing but an `<a/>` with a style. Thus, it still takes the `href`, `download`, and `target` properties.

I need to confess, I am not a designer. Yet, we can do better than a gray background alone. Let’s try.

Listing 5.9: Styling of file-entries in `src/file-list.tsx`

```
1  /* ...*/
2  const FileLink = styled.a`
3    display: block;
4    text-decoration: none;
5    color: black;
6    padding: 5px 0;
7    &:hover {
8      background: #CCC;
9    }
10 `;
11
12 const FileItem = styled.li`
13   border-top: 1px solid #888;
14   list-style-type: none;
15 `;
16
17 const FileEntry = (props) => <FileItem>
18   <FileLink download target="_blank" {...props}/>
19 </FileItem>;
20
21 export default function () {
22   return <ul>
23     <FileEntry href={logo}>Logo</FileEntry>
24     <FileEntry href="index.html">Index</FileEntry>
25   </ul>
26 };
```

In our improved `FileLink`, we apply a few styles. With `display: block;` (line 3), we set the formatting context. In a block formatting context, components are laid out one after the other, vertically. Each component's outer edges touch the edge of the containing block. Simply put: let the component take the whole line.

The `text-decoration: none;` (line 4) removes the underline a link usually has. The `color: black;` (line 5) sets the color of the font to black (usually blue for links). `padding: 5px;` sets the (inside) space area on all four sides of a component.

`&:hover { background: #CCC; }` sets the color of the link's background to a light gray when the user hovers over it with the mouse pointer.

Styling `<li/>`-components is as easy. Look at line 12. `<FileItem/>` is a styled `<li/>`. It has a thin (`1px`), gray (`#888`) border at the top (line 13). And we remove the dot a unnumbered list element usually has at line 14 (`list-style-type: none;`).

In our list, a file entry consists of an `<a/>` (now `<FileLink/>`)-component inside an `<li/>` (now `<FileItem/>`)-component. Rather than repeating this combination in the `<ul/>` over and over again, we replace it by a single component: the `<FileEntry/>` we define at line 17. This is not a styled component. But it is a simple functional React component. It

takes some arbitrary properties (`props`) as arguments and returns a `<FileLink/>` inside a `<FileItem/>`-component. There's a trick at line 18. The operator `...` "spreads" the `props`-object. It means that all the enumerable properties in `props` become discrete properties of `<FileLink/>`. In easy terms: we pass through the properties from one component to another.

It allows us to treat `<FileEntry/>`-components as links and put them directly into our `<ul/>`-component.

In the next code listing, we apply styles to our file-list and add a head to it.

Listing 5.10: Styling of the file-list in `src/file-list.tsx`

```
1  /* ...*/
2  const StyledList = styled.ul‘
3    margin: auto;
4    width: calc(100% - 20px);
5    padding-left: 0;
6  ‘;
7
8  const Head = styled.li‘
9    padding: 5px;
10   color: #888;
11   font-weight: bold;
12   list-style-type: none;
13 ‘;
14
15 export default function () {
16   return <StyledList>
17     <Head>Name</Head>
18     <FileEntry href={logo}>Logo</FileEntry>
19     <FileEntry href="index.html">Index</FileEntry>
20   </StyledList>
21 };
```

We want our `<StyledList/>`-component (it is a `<ul/>`) to fill the whole width of the browser except for a little space. We achieve this by setting the width to `100%` minus the space we want to leave altogether `20px` (line 4). (Note: there must be blanks before and after the "-"-sign). `margin: auto;` distributes the remaining (outer) space equally and therefore centers our list.

The other styles should be self-explanatory by now. Let's have a look at our app.

5.5 Advanced User Interaction

The visual and interactive capabilities of any React app (whether full-stack or not) go beyond applying CSS styles to the components. The drag and drop gesture serves as an

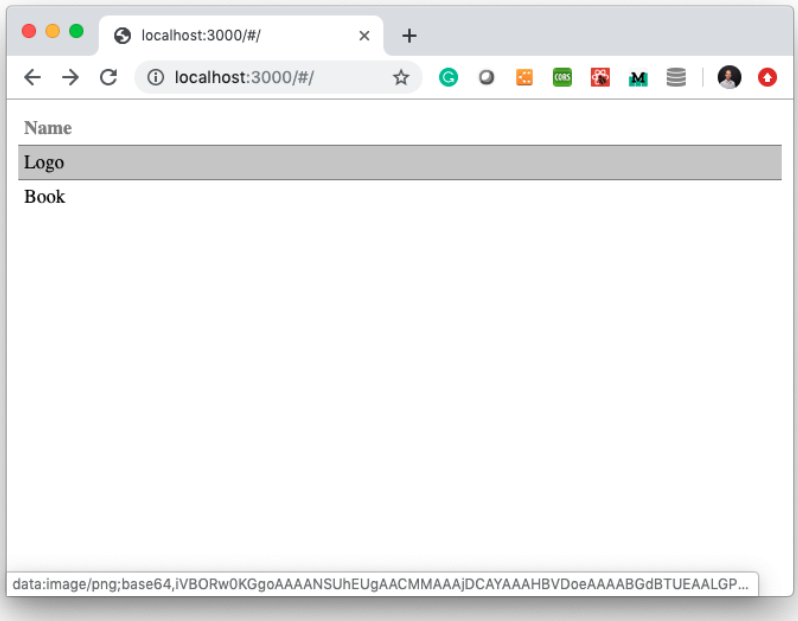


Figure 5.4: The styled list of files

example. Drag and drop is an intuitive way of moving and rearranging elements in Web and Mobile applications. It listens to pointer (mouse or touch) events, works with data, and changes the DOM.

Drag and drop is part of HTML5 (as described here). We could add this gesture to our app based on the low-level HTML5 API. However, there are quite some libraries out there that wrap this gesture into a more programmer-friendly, high-level API. `react-sortable-hoc` is a great example. It provides a set of higher-order React components to turn any list into an animated, touch-friendly, sortable list.

A higher-order React component (HOC) is an advanced technique in React for reusing component logic. Usually, React components transform properties into something visible. Like a link, a list, or any other HTML-element. A higher-order component adds some functionality to a component. It transforms a component into another component.

The higher-order component `react-sortable-hoc` adds drag and drop capabilities to our component.

First, we install the library. Stop the project (if it is running, use `ctrl + c`) and run the command `npm install react-sortable-hoc --save`.

But before we can add the drag and drop functionality to our list of files, we need to separate our layout and our data. Right now, our data (the name and the file-object are

mixed into the visual React-component). Each single file is hard-coded: `<FileEntry href="hard-coded-link">Hard-coded-name</FileEntry>`. This is not flexible or extendable.

The following excerpt depicts how we can separate layout and data.

Listing 5.11: Separating layout and data in the `src/file-list.tsx`

```
1  /* ...*/
2  export default function () {
3    const files = [
4      {
5        name: "Logo",
6        href: logo
7      }, {
8        name: "Index",
9        href: "index.html"
10     }
11   ];
12
13   return <StyledList>
14     <Head>Name</Head>
15     {
16       files.map((file, index) => (
17         <FileEntry key={`item-${index}`} href={file.href}>
18           {file.name}
19         </FileEntry>
20       ))
21     }
22   </StyledList>
23 }
```

Within our exported function, we define a constant: `files` (line 3). This is an array of JavaScript objects. Each object has two properties: `name` and `href`. This is the data we previously hard-coded within the `<FileEntry/>`-components.

In the code-block between the lines 15 and 21, we reintegrate the separated data (`files`-array) into the `<FileEntry/>`-components. The `files`-array provides the map-function. This function takes each of the array's items and transforms it according to the function that we provide as an argument and returns the results in a new array. It does not change the original array.

The function we provide as an argument starts at line 16. It is an anonymous function (without a name) and follows the arrow notation: `(arguments) => ("returned result")`. We use two arguments `file` and `index`. While the name we use for these two arguments is up to us, their positions matter. The first argument is the current item. This is our JavaScript-object. The second argument is the index of the current item.

Since we have two items in our `files`-array, the `map`-function calls the provided func-

tion two times. First with the arguments `file={name: "Logo", href: logo}` and `index=0`. Then with the arguments `file={name: "Index", href: "index.html"}` and `index=1`.

The function transforms these two arguments and returns a `<FileEntry/>`-component (lines 17-19). It specifies the `file`-object's `name` as the visible content and its `href` as the `<FileEntry/>`'s property of the ame name. Further, it specifies a string (`"item-filename"`) as the value of the `key` property. This has a technical reason. When we provide a code-block in JSX that returns an array (like the `map`-function does), then React requires us to provide a unique `key` property to each of the array's items.

If you run the app, you should not see any difference. But we now have all our data in an array that is free from any layout. We're now ready to add the drag and drop functionality through the `react-sortable-hoc` library.

Listing 5.12: Adding the drag and drop functionality in the `src/file-list.tsx`

```
1  /* ...*/
2  import {SortableContainer, SortableElement} from 'react-sortable-hoc';
3  const SortableFile = SortableElement(FileEntry);
4  const SortableList = SortableContainer(props => {
5    return (
6      <StyledList>
7        <Head>Name</Head>
8        {
9          props.files.map((file, index) => (
10           <SortableFile key={`item-${file.name}`} index={index} href={file.
     href}>
11             {file.name}
12           </SortableFile>
13         ))
14       }
15     </StyledList>
16   );
17 });
18
19 export default function () {
20   const files = [
21     {
22       name: "Logo",
23       href: logo
24     }, {
25       name: "Index",
26       href: "index.html"
27     }
28   ];
29
30   return <SortableList files={files}/>
31 };
```

React's key property

by Dr. Derek Austin

Any time you render a React component containing an array of React elements, each of the React elements must have a key property (or "key prop") associated with it, so that React can track the elements appropriately over time.

Examples of elements requiring a key property would be <li> list items or the <SortableFile> components in the example on page 104. But anytime a React component has more than one of the same JSX element, even elements, each one requires a unique key.

The key property is not the same as an API key or database key, like the AWS key previously mentioned. Instead, the React key property is a unique identifier used inside React.

Without the key property, you will end up with bugs in any dynamic list in React. To prevent these bugs, React may give you some warnings about missing or duplicate key properties:

Each child in a list should have a unique "key" prop.

- Meaning no key property was detected on list items in a component. Warning: Encountered two children with the same key.
- Meaning two or more list items in a component had the same key prop.

Why is the key property so important? When you give React an array of elements, React needs some sort of unique identifier to know which element is which.

React updates the display of its components using a reference to the React elements it was provided the last time it rendered this app component.

While rendering, React compares the previous group of React elements with the new elements just given to it. Based on this comparison, React updates the DOM accordingly.

Without a unique key property, React doesn't know whether you changed the order of the elements, removed an element, or maybe you added two elements but removed three.

When no key property is provided, React doesn't have any insight into what you did to this array of React elements this time compared to the last time it rendered the component.

So anytime you are rendering an array of React elements, you need to give it a key property so that React can determine whether elements were removed, added, or modified.

You always need to provide a unique key to prevent issues when working with lists of data inside React components.

Don't use an item's index as React's key property
by Dr. Derek Austin

You should not use the item's index as the React key property, because then React will not know which item actually changed if the list is reordered or if an item is removed.

The key property should be a unique identifier that does not change, but the item's index will change when a list is sorted.

The index is not appropriate to use as a key property, even though it will make the unique key warning go away, because it will cause bugs regarding focus and transitions.

Instead of using the index as a key property, use a unique identifier, like the unique id retrieved from a database.

```
1 {listItems.map((listItem) =>
2  <ListItem {...listItem}
3   key={listItem.id} />
4 )}
```

If there is no unique id, or you need to generate one dynamically, you can use a global counter variable.

```
1 let listItemCounter = 1;
2 function createNewListItem(text) {
3  return {
4   id: listItemCounter++,
5   text: ""
6  }
7 }
```

A more robust solution would be using the shortid.generate() function from the npm package shortid, which provides "short non-sequential url-friendly unique ids":

```
1 var shortid = require('shortid');
2 function createNewListItem(text) {
3  return {
4   id: shortid.generate(),
5   text: ""
6  }
7 }
```

At line 2, we import two higher-order components: the `SortableContainer` and the `SortableElement`.

The `SortableElement`-function makes a component draggable. At line 3, the

`SortableElement` takes the `FileEntry`-function as an argument and returns a similar function that is draggable. (Note: do not provide it as a rendered component with < and /> here but with its function name).

The `SortableContainer`-function prepares a parent-component to contain draggable children. We could pass the `StyledList`-function into it. As we did it before. But there's another way. Let's have a look.

As we mentioned before, a React function transforms properties into something visible, like this `const Component = props => <div/>`. A higher-order component accepts the name of the component (`Component`) or its body (`props => <div/>`). At line 5, we use the latter form. We take `props` and transform them into the `<StyledList />` with a `<Head/>` and the mapped `files`-array.

We only expect the `props` to contain the `files` (line 10). And the `files`-array is the property we provide when we return the rendered `<SortableList/>` in our exported function (line 31). The result is a concise exported function. All the drag-related logic is kept in the `<SortableList/>`-component.

At line 11, we add the `index`-property to the `<SortableFile/>`. This is the element's `sortableIndex` within the array required of the `react-sortable-hoc` library.

If you look at the app, you can now drag the two files. But once you drop them, they return to their original position. Of course, they do. Because our `files`-array does not change.

5.6 Local Component State

When you look at the reference of `react-sortable-hoc`, you see that the `SortableContainer` HOC adds the `onSortEnd`-property to our `<StyledList/>`. This property takes a callback function that is invoked when sorting ends. It receives the `oldIndex` and the `newIndex`.

But how do we change our `files`-array? It is an immutable `const`. But changing it to a mutable `var` would not help us. Because whenever React renders the component, it starts anew. It does not "remember" a variable's value. And unless React rerendered the component, we would not see any changes.

The solution is the `useState`-React hook (supported since `React 16.8.0`). Hooks offer a way to "attach" reusable behavior to a component. As such, hooks are similar to higher-order components. But while higher-order components need outside wrappers and thus, change your component hierarchy, hooks allow you to reuse logic without changing your component hierarchy.

The following code adds a state to our file-list.

Listing 5.13: Adding the `useState` hook to the `src/file-list.tsx`

```
1  import React, {useState} from 'react';
2  /* ...*/
3  export default function () {
4    const [files, setFiles] = useState([
5      {
6        name: "Logo",
7        href: logo
8      }, {
9        name: "Index",
10       href: "index.html"
11     }
12   ]);
13
14   return <SortableList files={files} onSortEnd={
15     ({oldIndex, newIndex}) => {
16       const removed = files.slice(0, oldIndex)
17         .concat(files.slice(oldIndex+1));
18
19       setFiles(
20         removed.slice(0,newIndex)
21           .concat([files[oldIndex]])
22           .concat(removed.slice(newIndex))
23       );
24     }
25   }/>
26 };
```

`useState` is a function. We import it from `react` at line 1 and we use it in line 4. It takes a single argument. This is the initial state. Therefore, we provide the same data we previously put into the `files`-array.

`useState` returns an array with two elements. The first element is the current state. Since we did not change the state, it is our initial state. This is our `files`-array. We use it as we did before. In line 14, we provide it as a property to the `<SortableList/>`-component.

What's new is the second element. This is a function that sets the state. We can use an arbitrary name. Let's call it `setFiles`. We use this function in the `onSortEnd`-function at line 19. We provide a new `files`-array with the updated order.

At line 16, we create a new (temporary) array without the dragged file. The array's `slice`-function returns a portion of an array. The two parameters specify the indexes of the `begin` and the `end` (end not included). The array's `concat`-function connects two arrays. So, we take the sub-array from the start to the dragged item (excluded) and connect it to the sub-array from after the dragged item to the end (`slice` called with a single argument at line 17)

We slice this temporary array from the start to the new position (line 20), connect it with

the dragged item (taken from the original `files`-array) (line 21), and connect it with the rest of the temporary array (line 22). This is the reordered `files`-array we provide to the `setFiles`-function.

Phew. That was a lot. But wait. What is this all about stateless, hierarchical, functional programming? In the end, we use a state! And I think there's an error at `line 4`. We declared the `files`-array as a `const`. But you can't change a `const`!

The `setFiles`-function does not change these values. It causes React to rerender the user interface with the new values. React replaces the old `<SortableList/>` with the new one. And the new `<SortableList/>` has its own, immutable state. There are no side-effects. You only need to consider the current state. Past states do not matter. How the state was created does not matter.

5.7 Serving Different Routes

A list of downloadable files is great. But what are files if you can't put them into folders? And of course, we want to address the different folders directly in the URL.

Before we integrate the folders into the program, we provide a style for them. Let's say we want to show the folders on top of the files. And for now, we don't need them to be draggable.

I believe the applied styles are self-explanatory. Except for the values (a different color), we use the same style-commands as we did before. But how we apply the styles differs.

What if you wanted the link to "Anything" to be red rather than black? While keeping the rest of the styles, of course. Copy-and-pasting the whole style is fast but it incurs technical debt. When you change the link's style the next time, you'll need to remember to change it in two places.

The better way is to inherit all styles from `<MyLink/>` and only change the color. You can call styled with an existing React component as an argument. You then add the applied style to the existing component and overwrite existing styles if they overlap.

This only works if you have a shared base-component under your control. In our example, this is `MyLink`. But what if you want to apply the same style to two different components. In React apps, for instance, you often work with `react-router-dom`. It provides the `<Link/>`-component. That is a convenient way to handle internal links. Basically, `<Link/>` is an HTML-link (`<a/>`). But you don't control it.

Listing 5.14: The style of folders

```
import styled, { css } from 'styled-components';

const Item = styled.li`
  border-top: 1px solid #888;
  list-style-type: none;
`;
const styledLink = css`
  display: block;
  text-decoration: none;
  color: black;
  padding: 5px;
`;
const FileLink = styled.a`
  ${styledLink}
  &:hover {
    background: #CCC;
  }
`;
const FolderLink = styled(Link)`
  ${styledLink}
  background: #FFE9A2;
  &:hover {
    background: #AAA;
  }
`;

const FileEntry = (props) => <Item>
  <FileLink download target="_blank" {...props}/>
</Item>;

const Folder = (props) => <Item>
  <FolderLink {...props}/>
</Item>;
```

Documentation — Link.

- the `pathname` is string representing the path to link to.
- `search` is a string representation of query parameters.
- `state` takes a JavaScript-object to persist to the location, e.g. `{fromDashboard:true}`.

You need a sharable style not fixed to a component yet. This is what the `css`-property is for. You import it from styled-components. You attach CSS definitions to it through a template string, too.

We move all our base CSS definitions into the `styledLink` constant. We can directly

integrate this into any other styled component. In our case, the expression is our `styledLink` constant we integrate into `MyLink` (line 16) and `MyInternalLink` (line 24).

We need to serve each folder as a `<Route/>`. Currently, our single one `<Route/>` is hard-coded. Since we don't know all the folders, we need to flexibly add the `<Route/>`-components. As we separated the data and the layout of our files, we can do it for the `<Route/>`s, too.

Documentation — <Route/>.

- the `path` of a route specifies the page's path in the URL.
- the `name` specifies the title of the page.
- the `render`-function lets you render your page's React-component with JavaScript.

The following code depicts, how we separate the data of the folders from the specification of the `<Route/>`s.

Listing 5.15: Separating the folder-data from the <Route/>-components

```
1  /** ... */
2  const folders = [
3    {
4      name: "Data",
5      path: "/"
6    }, {
7      name: "Documents",
8      path: "/documents"
9    }, {
10     name: "Images",
11     path: "/images"
12   },
13 ];
14
15 export default (
16   <SinglePageApp
17     stackName = "file-management"
18     buildPath = 'build'
19     region='us-east-1'>
20
21     <Environment name="dev" />
22     {
23       folders.map((folder, index)=> <Route
24         key={`folder-${folder.name}`}
25         path={folder.path}
26         name={folder.name}
27         render={(props) => <FileList />}
28       />)
29     }
30   </SinglePageApp>
31 );
```

At lines 2-13, we define an array of JavaScript-objects. Each object contains the name of the folder we want to show in our list and the path we want to serve the folder at. We specify "Data" as the name of the root-path ("/"). Also, note that all paths start with a leading "/".

Again, we use the `map`-function to add the `<Route/>`-components to the `<SinglePageApp/>`. At line 23, we map each folder in the `folders`-array to a `<Route/>`. We provide a `key` (line 24), the `path` (line 25), and the `name` of the `<Route/>` (line 26). Every `<Route/>` renders a plain `<FileList/>`. Don't we need to pass a folder's data to the `<FileList/>`-component, too? We could put it there directly. But we already provide it to the `<Route/>`-component. Putting it to the `<FileList/>` would duplicate our code. You would need to make sure that the data you put into the Route is the same that you put into the `<FileList/>`.

But there's another way of doing it. The `<FileList/>` is the child of the `<Route/>` that already has the data. If the `<FileList/>` needs the data, it can ask its parent for it. That way, we make sure that the `<FileList/>` within a `<Route/>` always works with the same data.

Further, the `<SinglePageApp/>` knows all its `<Routes/>`. Because it serves them. For the `<FileList/>` is also a (grand-) child of `<SinglePageApp/>`, it can ask the `<SinglePageApp/>` for it.

But rather than getting a reference to the component (either directly by providing it or indirectly by navigating through the component hierarchy), we access the data through HOCs provided by the respective parent components.

The `<SinglePageApp/>`-component (like the other top-level-architectural components, too) provides the `withRoutes`-HOC (import from `infrastructure-components`). And it gives you all the `routes`. It works similarly the `<SortableContainer/>`-HOC. You wrap the function into it to which you want to add the specific capability. In this case, the capability is to access all the `routes`.

the `withRoutes`-HOC adds the `routes`-property to the component it wraps. The `routes`-property is an Array of `IRoute`.

For the `routes`-property is an array, we can call its `map`-function. We map the `IRoute`-object of each route. We transform it into a `<Folder/>`-component. Again, we provide a unique `key`-property. The `to`-property describes the destination of the `<Link/>` (that we import from `react-router-dom`) (as `href` in `<a href="..."/>`. The destination is the `path` of the `IRoute` as we specified it in our `index.tsx`. Since the `<Folder/>`-passes its properties through to the `<FolderLink/>` that is a styled `<Link/>`, we can provide it directly to the `<Folder/>`-component

Finally, we provide the `name` of the `IRoute` as a child. This is what the `<Link/>` displays in the entry.

Refresh the browser window. We can see that the navigation bar shows the names of our `<SinglePageApp/>`‘s `<Route/>`-components.

Listing 5.16: The sortable list

```
/** ... */
import { withRoutes } from 'infrastructure-components';
import { Link } from 'react-router-dom';

const Folder = (props) => <FolderItem>
  <FolderLink {...props}/>
</FolderItem>;

const SortableList = withRoutes(SortableContainer(({items, routes}) => {

  return (
    <StyledList>
      <Head>Name</Head>
      {
        routes.map((route, index) => (
          <Folder key={'route-'+route.name} to={ route.path }>
          {
            /** display the name of the <Route /> component*/
            route.name
          }
          </Folder>
        ))
      }
      {
        items.map((file, index) => (
          <SortableFile
            key={`item-${file.name}`}
            index={index}
            href={file.href}
            name={file.name}/>
        ))
      }
    </StyledList>
  );
}));
```

We provide the list of the folders before the list of the files. Thus, they are rendered in that order, placing the folders to the top of the list. As the following image depicts.

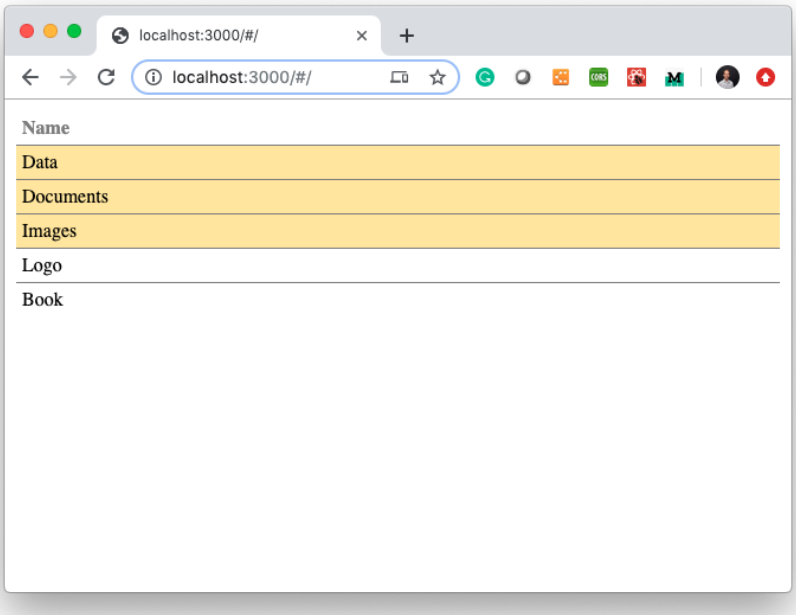


Figure 5.5: The navigation bar shows the links to the routes.

Documentation — IRoute.

- the `path` is the relative path of the route, e.g. "/" for the root, or "/something"
- the `method` defines the HTTP-method of the route, e.g. get, post, ...
- `middlewares` is an array of callbacks to be used of a route before handing over to the "*"-callback
- the `component` (optional) is a function (`(props: any) => ReactNode`). It takes any `props` to be passed to the component to be rendered and returns the rendered ReactNode of the route, e.g. (`<TestPage ...props/>`)
- `component` (optional) is the component (function) to render. Use either `component` or `component`.
- the `name` is the displayed name of the route
- `exact` specifies whether an exact route matching is required
- the `customType` (optional) is a custom type you can use to distinguish different types of routes
- if `isSecured` (optional) is `true`, the route is secured and requires the user to login. I.e. wraps the route into `<ForceLogin />`

When you click a folder, you see the path in the browser address bar change. But the content does not change. Of course not. Because every path renders the same `<StyledList/>`-component. In the next step, we let the `StyledList/>` take its location into account. Let's first look at the code.

Listing 5.17: Using the location-property.

```
/** ... */
import { Link, withRouter } from 'react-router-dom';
const SortableList = withRouter(withRoutes(SortableContainer(
  ({items, routes, location}) => {
    return <StyledList>
      <Head>Name</Head>
      { routes.filter(route => route.path !== location.pathname && (
          route.path.startsWith(location.pathname) ||
          location.pathname.startsWith(route.path)
        )
      ).map((route, index) => (
        <Folder key={'route-'+route.name} to={ route.path }>
        {
          /** display the name of the <Route /> component*/
          location.pathname.startsWith(route.path) ? ".." : route.name
        }
        </Folder>
      )) }
      { files.filter(
         item => item.path == location.pathname
      ).map((file, index) => (
        <SortableFile key={`item-${file.name}`} index={index}
          href={file.href}
        >{file.name}</SortableFile>
      ))}
    </StyledList>
  }
)));

export default function () {
  const [files, setFiles] = useState([{
    name: "Index",
    path: "/",
    href: "index.html"
  },{
    name: "App",
    path: "/",
    href: "file-management.bundle.js"
  }, {
    name: "Logo",
    path: "/documents",
    href: logo
  }]);

  return <div>
    <SortableList distance={2} files={files} onSortEnd={/*...*/}/>
  </div>
};
```

Again, we use a HOC. The withRouter- HOC (import from `react-router-dom` at line 2) provides the `location`-property.

At line 3, we wrap our `<StyledList/>` into it and receive the `location`-property. This is a JavaScript object providing the `pathname` of the current directory. We use this in two places. When we render the files and when we render the folders.

Let's start with the files. You may have seen that we added another key to the files-data. The `path`. The `path` provides the folder that the file is in. We put the "index" and the "app"-file into the root folder ("/") and the "Logo" into the "/documents"-folder. The "Images"-folder remains empty.

At line 23, we call the `filter`-function of the `files`-array. The `filter`-function has the same structure as the `map`-function. It takes a function as its argument. That function provides the current item. The function must return a `Boolean` value. If the returned value is `true`, the item is kept in the resulting array. If it is `false`, the item is excluded. Like the `map`-function, `filter` does not change the original array. But it creates a new array that contains all the items for which we returned `true`.

At line 24, we provide a function as the argument to the `filter`-function. It verifies whether the `path` of the `file`-object equals the `pathname` of the current location. The result is an array of the files in the current folder. At line 25, we proceed with the result of the filter function and apply a `map`-function. Such a sequence of function calls is called chaining.

We apply a similar structure to the folders. We first filter the folders. First, we exclude the current folder (line 9, when the `route`'s `path` equals the current location `pathname`). And we exclude all folders unless the current folder is a subfolder (line 10, the current folder's `path` starts with the location's `pathname`, but can be longer) or a parent-folder (line 11, the location's `pathname` starts with the current folder, but can be longer).

Showing subfolders of the current folder makes sense, doesn't it? But why would we also show the parent-folder in the list of a folder? At line 17, we render the visible string. We apply the same logic here. When the current-folder is the parent (the location's `pathname` starts with the current folder, but may be longer), we display two dots: "..". This is our way back upwards in the folder hierarchy. It makes sense, too. Doesn't it?

There's something you may object to. We use the same logic twice: whether a folder is the parent of the current folder: at lines 11 and 17. We should not repeat ourselves that way. Because if we changed this logic later, we might miss changing it at both occasions. So, let's wrap it into a function.

Listing 5.18: Avoid repitition

```
1  const SortableList = withRouter(withRoutes(SortableContainer(
2    ({files, routes, location}) => {
3      const isParent = (parent, child) => child.startsWith(parent) && child.
       substr(parent.length+1).indexOf("/") < 0;
4
5      return (
6        <StyledList>
7          <Head>Name</Head>
8          {
9            routes.filter(route => route.path !== location.pathname && (
10               isParent(location.pathname, route.path) ||
11               isParent(route.path, location.pathname)
12             )
13           ).map((route, index) => (
14             <Folder key={'route-'+route.name} to={ route.path }>
15             {
16               isParent(route.path, location.pathname) ? ".." : route.name
17             }
18             </Folder>
19           ))
20         }
21         { /* ... */}
22       </StyledList>
23     );
24   }
25 )));
```

At line 3, we define the `isParent`-function. It takes a a potential parent and child path and checks whether that route is a parent of the location. We use it lines 10, 11, and 16.

The following image shows the documents-folder

How can a single-page app serve multiple routes?

A Single-Page React App uses the React `hash-router`. `react-router` uses a trick here. It adds the path of a route after a #. Usually, anything after the # is ignored. But `react-router` uses this information and provides it to your web-app. It does not work without the #. Because this would cause a respective request to S3. And S3 would not find a resource at the specified address. But this is information that the server ignores. It does not load another resource off the s3-bucket.

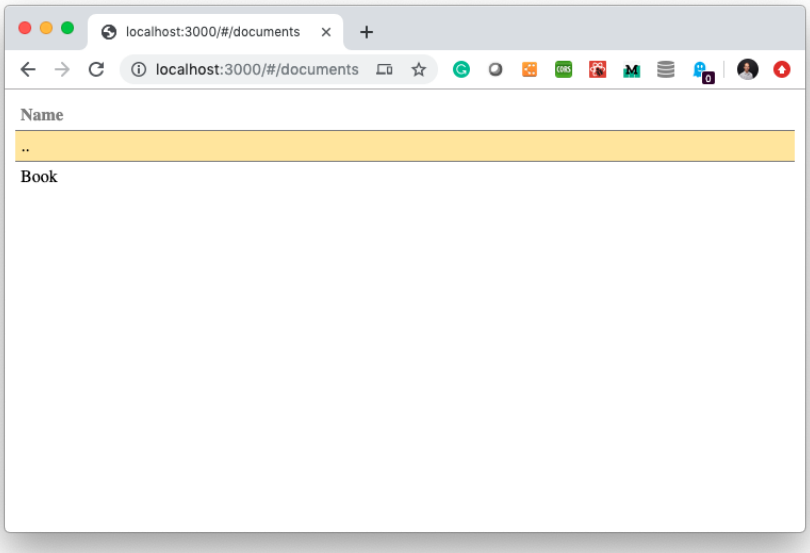


Figure 5.6: Folders with a file in it

5.8 Evaluation

We have covered many capabilities of a Serverless Single-Page React App in this section.

- Serving static files
- Styling components
- User interactivity
- Local state with the `useState` hook
- Serving different routes

Of course, there is much more you can do with a single-page app. But these fall into one of the covered categories. You can do animations (as a form of styling and user interactivity). You can use more complex local states, for instance with Redux.

But you may already have noticed the general limitation of a Serverless Single-Page React App. Even though its user interface appears quite dynamic, the app in its core is: static.

"Wait!" you may shout! *"You can call web-services from an SSPRA. You can dynamically load any resource you want!"* This is true, from a certain point of view. But the problem is: where does the web-service come from? I have not seen it in the architectural picture.

"“Services are not part of a React app. Just give it to the back-end team! They are in charge of the services.”

You can do this. But it would make the service something external you rely on. Not under

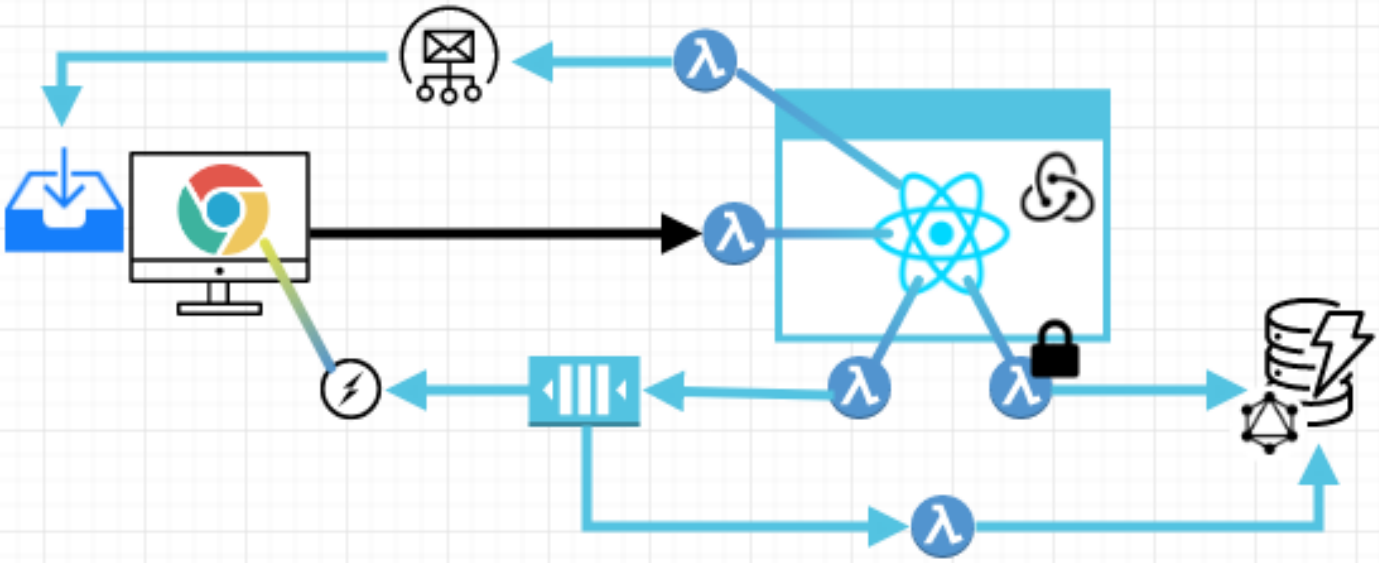
your control. But it is a small service. It is tightly connected to your app. Then it should be under your control, shouldn't it?

"“Don't make me angry! Just hand it over to the back-end team and give me the interface description. Seriously!” you say?

Unfortunately, there is no back-end team. If you want to be a full-stack developer, then it is you. You are in charge of providing the services.

"“Sigh.”

Don't worry. This book is about full-stack React development. Let's add back-end services to our app.



6. Serverless Service-Oriented React App (SSORA)

We already know React is a powerful user interface library. It supports anything you like. From rendering basic HTML elements to interactive and animated components. All these things happen in the browser. Your code runs in the browser. The data is in the browser.

Once downloaded, the code of your web-app does not change anymore. It is static. And if your app is a single-page React app, your data is static, too. We had a detailed look at the capabilities of a Serverless Single-Page React app. But we excluded calling remote web-services. Because services are not part of the React app running in the browser. Services run at the back-end side. They do not only change the architecture of your app. They change their capabilities.

Once you add services to your React app, it becomes a service-oriented app.

You may argue a back-end is separated from your React app for a reason. It requires different infrastructure, different technology, and different code.

If you're a big enterprise and you want to reuse the services, you should not attach them closely to your app. But when the services provide logic or data specific to your app, then they should be part of it, too.

Because web-services require coordinated interfaces. You have to make sure you send all the required data when calling a web-service. You have to work with the format of the response data. All the syntaxes have to fit.

Moreover, you have to make sure the web-service does what you think it does. The semantics have to fit, too. All the nitty-gritty details. For instance, do the numbers you receive from a

web-service include taxes? Does the web-service use a specific timezone or UTC? What does a "userId" really mean?

What happens if you need to change the web-service? The syntax might remain unchanged. But if the meaning of the web-service changes, your app must reflect this change, too. Your React app and the web-services are connected. If you like it or not. Your React app depends on its web-services.

When the web-services provide logic or data specific to your app, then they should be part of it, too. If you integrate your web-services into your React app, you can cross the boundaries of front-end and back-end. You don't need to structure your code based on the infrastructure it runs on. You can structure your code based on the functions it provides and the purpose it serves.

Let's have a look at what this means.

6.1 The Architecture Of An SSORA

The following image depicts the infrastructure architecture of a Serverless service-oriented React app.

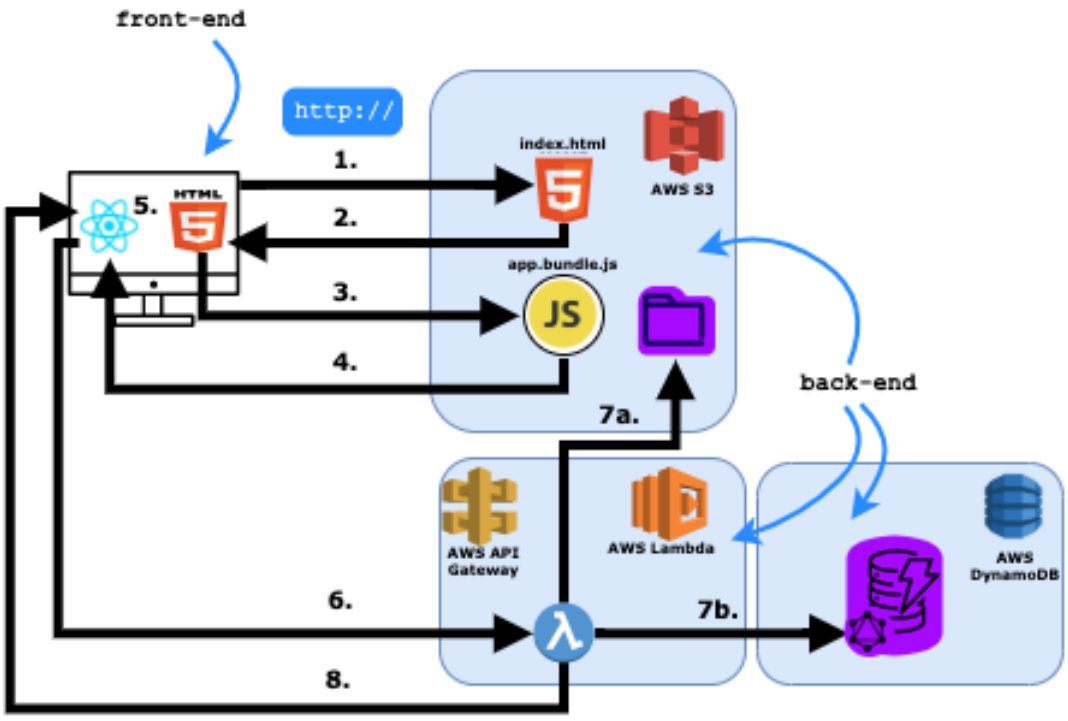


Figure 6.1: The architecture of a Serverless service-oriented React app

At first sight, you may see similarities to figure 5.1. And you are right. In its core, a

service-oriented app is a single-page app. Steps 1 to 5 are the same. The first request (step 1) gets an HTML as response (step 2). The HTML loads the JavaScript code of the React app (steps 3 and 4) and executes it in the browser (step 5).

The user can see and interact with the web-app now. And if you integrated any web-service calls, they can run from now on. They can run during the initial render-process. They can run as a response to user interaction. For instance, when the user clicks a button.

Step 6 is such a web-service call. It is an HTTP-request. Unlike the previous requests, it does not address a file on S3. But it submits to and requests data from a web-service. The AWS API-Gateway acts as the "front door" for applications to access data, business logic, or functionality from your backend services. It connects a stable URL with a computing resource our web-service runs on.

The computing resource is AWS Lambda. Lambda is a virtual environment your code runs in. You don't need to provision or manage servers. Because Lambda scales up and down dynamically.

Do not confuse AWS Lambda and a JavaScript lambda function. AWS Lambda is an infrastructure service providing Serverless computing resources. In JavaScript, a lambda function is an anonymous function. That is a function not specified with a name but directly at the place of its use.

Whenever a client (your React app in the browser) connects (sending an HTTP-request and waiting for the response) to the AWS API Gateway, it starts a new Lambda resource.

The code running on Lambda is under your control. This is a dramatic contrast to the code of your React app that runs in the user's browser. The user can inspect and change the code running in the browser. Easily! It is their browser. But neither can they inspect nor can they change the code running on Lambda. Users can only see what their browsers send to the web-service and the response they get.

We use this characteristic of server-side code in the steps 7a and 7b. In step 7a, we access the S3 file storage. If we only read from S3 (download), we could allow the clients to use the direct way as we did before. But what if we want to allow the user to upload a file? We may want to ensure the file not contain any forbidden content. We may want to check for the authentication of the user. If we put the file-uploading code (and writing permissions) into the client, a malicious user could remove our security checks from the code and only execute the file upload.

In step 7b, we access a DynamoDB database. Whenever we read or write the database, we need to make sure the code is trustworthy. If we opened up direct database access to a client, any user could retrieve all secret data. Such as the data of other users.

How does AWS Lambda work?

AWS Lambda is a serverless computing resource. You don't have to provide all the hardware required to run a back-end service. AWS takes over the whole capacity and load management. AWS Lambda is serverless in the sense that it only provides computing capacity when it is required. Lambda instances are added and removed dynamically. When a new instance handles its first request, the response time increases, which is called a cold start.

A Lambda instance does not run in idle mode for a long time. It keeps running for some limited time. This is not a fixed value and it may change. I've found references that state anything from between 5 minutes to up to an hour. Allegedly, AWS even analyzes the load behavior of your service to decide what idle time is best. An idle instance speeds up processing subsequent calls because it omits a cold-start.

A cold start may require a few seconds. Cold starts may sound problematic. But they are actually a good thing.

In the one extreme, when you have almost no users, every single user hits a cold start. The user experience may suffer. But! You completely save yourself to pay for an up and running server. Because Lambda is billed-per-execution. You only pay for the resources you use. And if you don't use any, you don't pay anything. By contrast, a real server would omit the cold starts. But you would pay for a server running idle all the time.

In another extreme, you have the same number of visits all the time. Lambda would start a single one or a few instances and keep them up all the time. Your users would experience almost no cold start. Just as good as if you had a real server.

In the third extreme, you have periods of no users and periods of myriads of users. Say you report the current events during a baseball game. When the match is ongoing there are many visitors requesting data. If there is no match, no one visits the page. With dedicated servers, you would need to provide as many servers as required to serve the peak! Otherwise, your page would not be available when users want to see it. But all these servers run idle when there is no match. With Lambda, you experience cold starts at the beginning of the match. While it runs, the frequent requests of the users keep the instances up and running. No cold starts. Once the match is over, the instances shut down and do not produce any more expenses.

DynamoDB is a NoSQL (Not only SQL) database. Infrastructure-components provide a GraphQL interface. GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. It is an efficient, robust, and flexible tool. It allows clients to define the structure of the data required, and the same structure of the data is returned from the server. Therefore, the requestor of data can specify

the format of the response.

Both steps, 7a and 7b, are optional. You may, as well, simply execute some business-critical code. Would you trust your user to run the price-calculation and billing code on their machine? Savvy users would set the price to zero. Without running trustworthy code, you would not know anything about this happening.

How does AWS DynamoDB work?
AWS DynamoDB is a managed NoSQL database service. It supports key-value and document data structures. These data structures allow DynamoDB to deliver high speed, independent of its size. It does not matter whether you have a hundred, a thousand, or a million entries in your table.

This fast access depends on the use of keys. Unlike SQL, DynamoDB does not support queries with complex conditions, joins, and subqueries. You have to pay special attention to your database design. Because if you don't have a key, you don't have fast access.

6.2 Create, Start, And Deploy An SSORA

We talked about the ability to change the architecture of your app when you need it. Let's do this!

While the SSPRA requires a very minimum of dependencies, the SSORA supports a back-end with a database. Because we want to proceed with our project, we add the other libraries manually. Make sure you have all the dependencies listed in 4.3. If not, add them and run `npm install`

There's only one thing you need to do in your code. Replace your top-level-component `<SinglePageApp/>` by `<ServiceOrientedApp/>` in `line 25`. Don't forget to import this component (`line 6`). All your components can stay unchanged.

You can still start your app locally in "hot-dev-mode" with `npm run file-management` (replace "file-management" with the `stackName` of your top-level component). You can still deploy your app with the command `npm run deploy-dev` (replace "dev" with the `name` of your `<Environment/>`).

You'll see no difference! Because there is no difference. We did not yet add any service or database. So your `<ServiceOrientedApp/>` remains to be a `<SinglePageApp/>`.

Listing 6.1: The `src/index.tsx` of your app as a `<ServiceOrientedApp/>`

```
import React from 'react';
import {
  Environment,
  Route,
  ServiceOrientedApp
} from "infrastructure-components";
import FileList from './file-list';
const folders = [
  {
    name: "Data",
    path: "/"
  }, {
    name: "Documents",
    path: "/documents"
  }, {
    name: "Images",
    path: "/images"
  },
];

export default (
  <ServiceOrientedApp
    stackName = "file-management"
    buildPath = 'build'
    region='us-east-1'>
    <Environment name="dev" />
    {
      folders.map((folder, index)=> <Route
        key={`folder-${folder.name}`}
        path={folder.path}
        name={folder.name}
        render={(props) => <FileList />}
      />)
    }
  </ServiceOrientedApp>
);
```

There's a little exception, though. If you run `npm run build` again, you'll notice that it adds another script to your `package.json`. This is `start-{your-env}` (again, replace `{your-env}` by your environment name).

`npm run start-{your-env}` starts your web-app, the services, and the database locally. Your web-app runs on `localhost:3000`. The services run on `localhost:3001`, the file-storage runs on `localhost:3002`, and the database runs on `localhost:8000`. However, in this mode, your app does not run in hot-development mode. So, if you change your code, you'll need to restart the app.

The database persists its state across starts. If you want to get a fresh (and empty) database, you'll need to delete the file `./.dynamodb/shared-local-instance.db`.

6.3 Advanced Styling

We start with an upload form the user can use to add files from her computer. There are two common types of upload forms. One, opening the system dialog requesting the user to select the files to be uploaded from her computer. And two, a dropzone. That is an area at the page where the user can drop files she started dragging from another place, like the file-explorer.

Let's create an upload form that supports both. We create a new file in our src-directory: `upload-form.tsx`. We start with our user-interface. As usual, let's first have a look at our code.

The advanced styling is not limited to service-oriented apps but can be used likewise in a single-page app. It is part of this section since the implementation of the function is part of the service-oriented app.

Listing 6.2: The `src/index.tsx` of your app as a `<ServiceOrientedApp/>`

```
import React from 'react';
import styled from 'styled-components';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'
import { faFileUpload } from '@fortawesome/free-solid-svg-icons'

const Dropzone = styled.div`
  margin: auto;
  width: calc(100% - 20px);
  padding: 10px 0;
  background-color: #fff;
  border: 2px dashed #888;
  display: flex;
  align-items: center;
  justify-content: center;
  flex-direction: column;
  font-size: 16px;
`;

export default function () {
  return <Dropzone>
    <FontAwesomeIcon icon={faFileUpload} size="4x" color="#888"/>
    <span>Upload Files</span>
  </Dropzone>
};
```

The first thing you'll notice are two new imports in lines 3 and 4. FontAwesome provides a large set of icons for free.

Before we continue, let's first install the libraries and add them to the project dependencies.

You can run the following command. This installs the latest versions of these libraries and adds them to your dependency list in your `package.json`.

Listing 6.3: Install the icon libraries

```
1 npm install --save @fortawesome/fontawesome-svg-core @fortawesome/free-solid-
      svg-icons @fortawesome/react-fontawesome
```

Alternatively, you can add the libraries manually to your `package.json` and run `npm install` afterwards. As of the day writing this book, the following versions are the latest.

Listing 6.4: The dependencies in the `package.json`

```
1 "@fortawesome/fontawesome-svg-core": "^1.2.19",
2 "@fortawesome/free-solid-svg-icons": "^5.9.0",
3 "@fortawesome/react-fontawesome": "^0.1.4",
```

If you want to look at the icons, have a look at the FontAwesome gallery.

The integration of icons is straight forward. FontAwesome provides a set of React-components. We imported the most important `FontAwesomeIcon`-component from `'@fortawesome/react-fontawesome'` at line 3.

Documentation — <FontAwesomeIcon/>.
You can find the complete documentation of this component at the official GitHub repository.

The component takes the following properties:

- the `icon` is the icon-object. You can import these from different libraries, e.g. `free-solid-svg-icons`.
- the `size` can be one of the following (from smaller to bigger): "xs", "sm", "lg", "2x", "3x" ...", "10x".
  `<FontAwesomeIcon/>`'s `size` property determines the icon size relative to the current font-size. This page provides more information on icon sizing.
- the `color` specifies the HTMl color of the icon.

At line 4, we import the `faFileUpload`-icon that we pass as property to the `<FontAwesomeIcon/>`-component at line 22. We wrap this icon and a brief explanatory text into a styled `div` we name `<Dropzone/>`.

Some of the `<Dropzone/>`'s stylings should be self-explanatory. Let's concentrate on the

new styles. This is the Flexbox we specify at line 12 by `display: flex;`. We refine it at lines 13-15. Flexbox is (in my opinion) the most powerful CSS style. The folks from css-tricks.com provide a complete and easy to follow guide on flexbox. So, I'll keep it short here.

The Flexbox is a flexible container of child elements. It supports different directions of laying out elements. In our case, this is vertically in a "column" (line15). We center each element along the main axis (`justify-content: center;`) and the cross axis (`align-items: center;`)

To summarize, we layout the elements below each other and centered.

There's another aspect you may wonder. How did I come up with the values of the `width` and `margin`? I chose them accordingly to the `<FileList/>`. I want the list of files and the drop zone area to close flush. The `<FileList/>` has an outer `margin` of `10px`. And so does the `<Dropzone/>`.

By now, you should shout "side-effects!" at me. And you're right. Let's remove this. We have already learned a way of sharing the style across different components. But if we introduce a new component every time we want to share a single style, we'll pretty soon end up in Styled-Component-Hell.

Styled-components provides a convenient way out. This is a theme. A theme lets us define names instead of styles.

Listing 6.5: Integration of the `<ThemeProvider/>`

```
1  import { ThemeProvider } from 'styled-components';
2
3  export const theme = {
4    /* the margin we keep to the frame of the browser window */
5    outerMargin: "10px",
6  };
7
8  export default (
9    <ServiceOrientedApp>
10   {
11     folders.map((folder, index)=> <Route
12       render={(props) => <ThemeProvider theme={theme}>
13         <FileList />
14         <UploadForm />
15       </ThemeProvider>
16       <Page>
17
18       </Page>}
19     />)
20   }
21   </ServiceOrientedApp>
22 );
```

Our theme is a simple JavaScript object. We define an arbitrary key, our `outerMargin`.

`<ThemeProvider>` is a wrapper component. This component provides a theme to all React components underneath it in the component hierarchy. All styled-components will have access to the provided theme, even when they are multiple levels deep.

The corresponding higher-order component is `withTheme` that we import from `'styled-components'`. We can use this higher-order component in the `upload-form.tsx` and the `file-list.tsx`. The `withTheme`-function adds the `theme`-object to the properties of a styled component. We can use the `theme` in a tagged string literal by providing a function in a code-block. That function receives the `theme` as a property. It contains the key `"outerMargin"`.

Listing 6.6: Using the theme

```
1 import styled, { withTheme } from 'styled-components';
2 /* ... */
3 const StyledList = withTheme(styled.ul`
4   margin: auto;
5   width: calc(100% - 2 * ${({theme}) => theme.outerMargin});
6   /* ... */
7 `);
```

In the `<Dropzone/>`'s style, we need to subtract another `2*2px` from the `width`. Because the `border` does not add to the `width`, but this is the visible edge of this component. For we use this value at two places, we introduce a constant for it.

Listing 6.7: The styled `<Dropzone/>`

```
1  import styled, { withTheme } from 'styled-components';
2  /* ... */
3  const Dropzone = (props) => {
4    const borderWidth = "2px";
5    const UploadFrame = withTheme(styled.div`
6      margin: auto;
7      width: calc(100% - 2 * ${props => props.theme.outerMargin} - 2 * ${
       borderWidth});
8      padding: 10px 0;
9      background-color: #fff;
10     border: ${borderWidth} dashed #888;
11     display: flex;
12     align-items: center;
13     justify-content: center;
14     flex-direction: column;
15     font-size: 16px;
16   `);
17
18   return <UploadFrame {...props}/>
19 };
```

Since the code of how we set the outer margin is the same, we could even put the whole CSS-function into the theme. We could also provide another function there. Having the raw value is all we need. And how to set a margin is nothing I want to abstract away. You need to make sure that you don't over-engineer here or end-up in abstraction-hell.

But putting a fixed value that we use more than once into a shared higher-order-component is always a good idea. And if this value concerns the styling, the way to go is a theme. We keep the values as local as possible.

Have a look at your app.

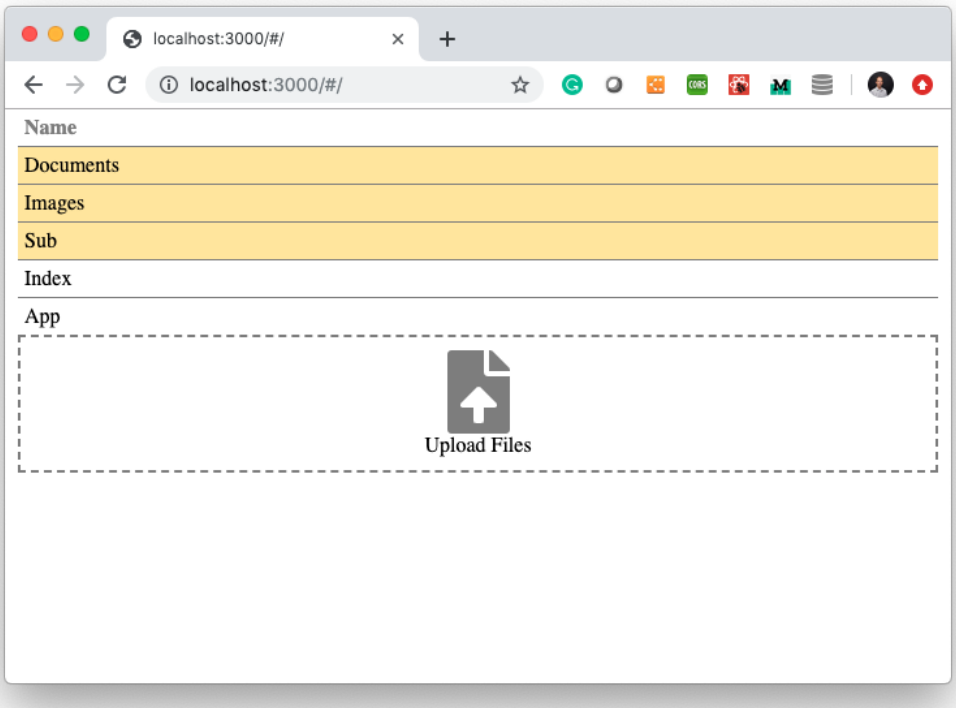


Figure 6.2: The design of the upload form

We have a nice user interface, now. But it doesn't do anything. Let's make the whole area clickable. When the user clicks, the browser's file-select dialog should open.

There are quite a few techniques for "customizing" the `<input type="file" />`-element. The problem with it is, that it comes with a button and a text by default. But we want to use our own `<Dropzone/>`-component.

The solution is the HTML-`<label/>`-element. The `<label/>`-element does not render anything special for the user. It provides better usability for mouse users. Because serves as a proxy for our `<input/>`.

Let's have a look.

Listing 6.8: The `navigation.tsx`, A basic component that adds a navigation bar placeholder

```
1  const Dropzone = (props) => {
2    const borderWidth = "2px";
3    const inputId = "uploadfile";
4
5    const UploadFrame = withTheme(styled.label`
6      margin: auto;
7      width: calc(100% - 2 * ${props => props.theme.outerMargin} - 2 * ${
       borderWidth});
8      padding: 10px 0;
9      background-color: #fff;
10     border: ${borderWidth} dashed #888;
11     display: flex;
12     align-items: center;
13     justify-content: center;
14     flex-direction: column;
15     font-size: 16px;
16
17     &:hover {
18       background-color: #ADA;
19       cursor: pointer;
20     }
21   `);
22
23   const UploadInput = styled.input`
24     display: none;
25   `;
26
27
28   return <UploadFrame {...props} htmlFor={inputId}>
29     <UploadInput
30       type="file"
31       name={inputId}
32       id={inputId}/>
33     {
34       props.children
35     }
36   </UploadFrame>
37 };
```

We changed the inner `<UploadFrame/>`-component. We replace the `<div/>` by a `<label/>` (line 5).

We provide the related `input` as a direct child of the `<label/>` in line 29. We connect the `<label/>` with the `<input/>` by providing the `htmlFor`-property of the `<label/>` (line 28). It matches the `id` of the `<input/>`-element (line 32). We use a `const` we define in line 3. Its value is arbitrary. The important thing is that it is unique across the elements we

use in the HTML-document. The HTML-property `for` in React is `htmlFor`. It has another name. Make sure you specify `"file"` as the type of the `<input/>`. This lets the `<input/>` open the file-select dialog rather than being a text-input.

We hide our `<input/>` through the CSS-style `display: none;`. And we specify a hover-effect for the `<UploadFrame/>`. We change the color to a light green and change the mouse pointer to a "hand" you are used to seeing when something is clickable.

Further, we insert all the properties of the parent into our `<UploadFrame/>`-component (through `{...props}`). This includes the children. But since we override the children property by specifying the `<UploadInput/>` as a child, we need to add the children defined outside again. We do this in line 35.

Let's have a look at our file-selection button in action.

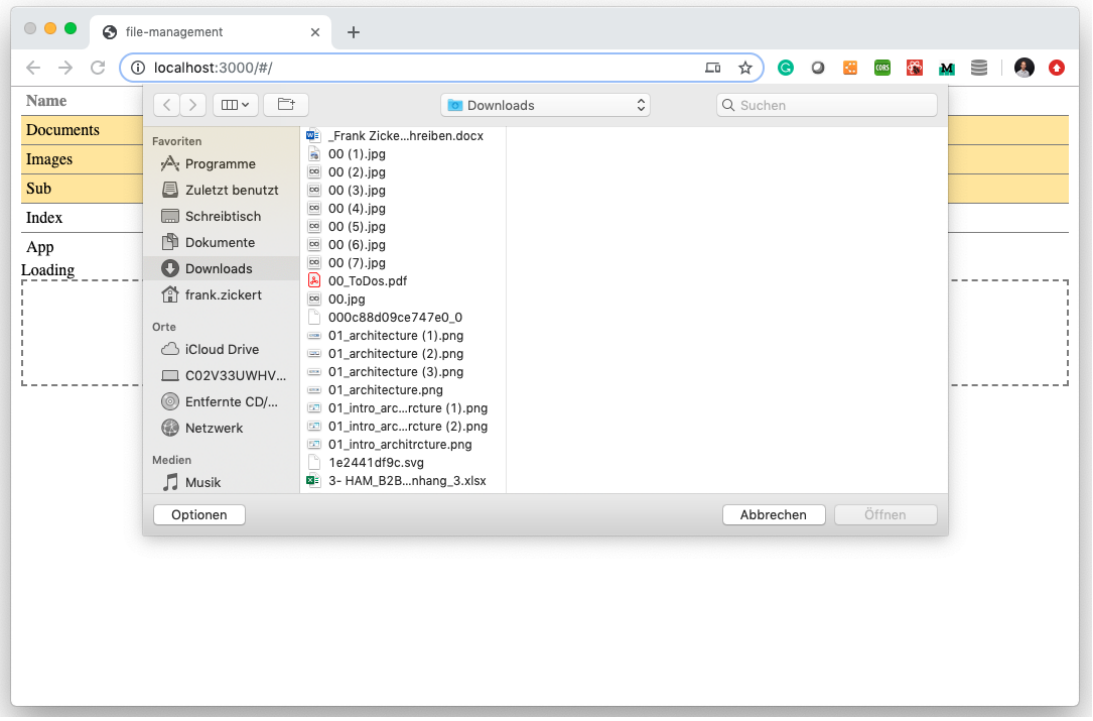


Figure 6.3: The file-selection

When you hover over the element, it becomes green. If you click on it, your system's file-dialog opens. But it does not do anything when you select a file. Yet.

6.4 File-Management

`<input/>`-elements provide the `onChange`-callback function. It gets called when the value of an element has been changed. Since we set the type of `<input/>` to `file`, the only

change possible is through the selection of a file in that dialog. The callback receives a single parameter: the event. This event has the `target`-attribute that contains an array of `files`.

Let's provide a function that takes the event and prints the `files`-array to the console. In line 12, we provide this function as the `onChange`-handler. We provide its declaration. (without passing arguments). We can do this because the `onFileSelected`-function has one parameter (the event). We could also write `onChange={event => onFileSelected(event)}`.

Listing 6.9: The onChange-callback

```
1  const Dropzone = (props) => {
2    /* ... */
3    const onFileSelected = event => {
4      console.log(event.target.files);
5    };
6
7    return <UploadFrame {...props} htmlFor={inputId} >
8      <UploadInput
9        type="file"
10       name={inputId}
11       id={inputId}
12       onChange={onFileSelected}/>
13     {
14       props.children
15     }
16   </UploadFrame>
17 };
```

Once you select a file, you can see some output as depicted in the following image. You can see we get the `lastModified`-timestamp and -datetime, as well as the `name`, the `size`, and the `type` of the file.

Figure 6.4: The output of the `onChange`-event

Uploading the file involves a series of technical steps. We need to load the binary data of the file and send it to a back-end service. This service must accept the file and write it to the S3 storage. Remember figure 7.1 that depicts the architecture of our service-oriented app. Before you start scrolling, here's an excerpt from the picture that shows the components we need to upload the file. It would be technically possible to upload the file directly to S3

from the browser. Since the browser is the user's machine, we would need to grant general access. Not only to our web-app. With a web-service in between, we run code we can trust. It lets us control (if we like) who may or may not upload files.

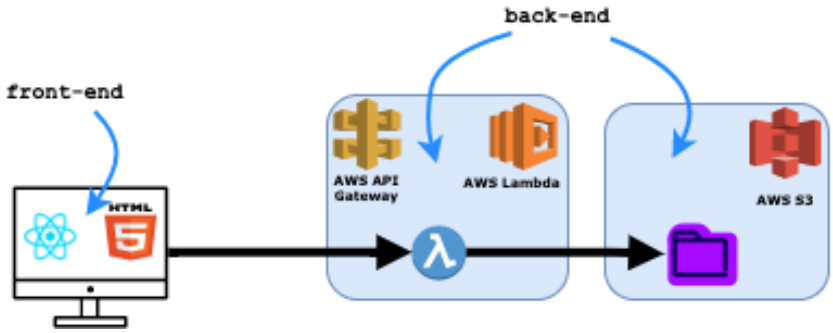


Figure 6.5: The upload components

Following the architecture-as-a-function approach, we do not implement all the technical steps ourselves. But we use the respective functions.

Infrastructure-components provide the `<Storage/>`-component. It takes an `id` and a `path` as properties. It does all the configuration for us.

Documentation — <Storage/>.

- the `id` is the (arbitrary) name of your storage. It serves as the identifier in your app and as the folder name at S3.
- the `path` is the relative URL of the endpoint.

Let's put all our code related to the file storage into a separate file: `file-storage.tsx`.

Listing 6.10: The FileStorage

```
1  import React from 'react';
2  import { Storage } from "infrastructure-components";
3
4  export const FILE_STORAGE_ID = "FILESTORAGE";
5
6  export default function () {
7    return <Storage
8      id={FILE_STORAGE_ID}
9      path="/filestorage"
10   />
11 }
```

At line 2, we import the `Storage/>`-component. We export a function as default that does nothing but returns the `<Storage/>` configured with its properties. It has an `id` (line 8) we specify in a `const` (line 4). And it has a `path` at line 9.

We can integrate the storage into our `src/index.tsx` easily. As usual, we omit the parts and details of the code that remained unchanged.

Listing 6.11: Integration of the FileStorage

```
1 import FileStorage from './file-storage';
2 export default (
3   <ServiceOrientedApp/>
4     /* ... */
5     <FileStorage/>
6     { folders.map((folder, index)=> <Route {/* ... */} />) }
7   </ServiceOrientedApp>
8 );
```

We import the `<FileStorage/>` at line 2 and add it as a direct child to the top-level `<ServiceOrientedApp/>`-component.

We're set up to work with files. Uploading is very easy. Because infrastructure-components provide a convenience function that does all the heavy lifting for us.

Documentation — The uploadFile-function.
The following snippet depicts the interface of the uploadFile-function

```
1 function uploadFile (
2   storageId: string,
3   prefix: string | undefined,
4   file: any,
5   onProgess: (uploaded: number) => Boolean,
6   onComplete: (uri: string) => void,
7   onError: (err: string) => void
8 )
```

- the `storageId` is the `id` of your `<Storage/>`
- the `prefix` is the relative path of the folder where to upload the file to. Separate subfolders with a "/". Leave empty ("") or `undefined` to upload the file into the root directory of your storage. If the folder does not yet exist, it gets created automatically on S3.
- the `file` is the JavaScript object you receive from the DOM-event, e.g. from `onChange (event) => event.target.files[0]`
- `onProgress` is a callback function. `uploadFile` uploads the file in blocks of about 100KB. After each uploaded block, the function is called. Return `true` to continue the upload or `false` to cancel it. The provided `uploaded`-parameter gives the total number of uploaded bytes thus far.
- `onComplete` is a callback function called when the upload completes. It provides back an URL of the file on S3.
- `onError` is a callback called if something went wrong. It contains the `error` message.

The `uploadFile`-function provides a convenient programming interface. But let's make it even simpler to use. Let's wrap it into our own `upload`-function we define in `file-storage.tsx`.

Listing 6.12: Integrating the FileStorage

```
1  import React from 'react';
2
3  import {
4    Storage,
5    uploadFile
6  } from "infrastructure-components";
7
8  export const FILE_STORAGE_ID = "FILESTORAGE";
9
10 export const upload = (prefix, file) => uploadFile(
11   FILE_STORAGE_ID,
12   prefix,
13   file,
14   //onProgess: (uploaded: number) => Boolean,
15   uploaded => {
16     const percent_done = Math.floor( ( uploaded / file.size ) * 100 );
17     console.log("Uploading File - " + percent_done + "%");
18     return true;
19   },
20   //onComplete: (uri: string) => void,
21   uri => console.log(uri),
22
23   //onError: (err: string) => void
24   err => console.log(err)
25
26 );
27
28 export default function () {
29   return <Storage
30     id={FILE_STORAGE_ID}
31     path="/filestorage"
32   />
33 }
```

We added the `upload`-function starting at line 10. This function takes a `prefix` (that is the `path` to the folder) and the `file`. We can specify the `storageId` with our `FILE_STORAGE_ID`. We further define basic versions of the callback-functions. At lines 15-19, we specify the `onProgress`-callback. It calculates and prints to the console the uploaded percentage. The file-object contains the total size. The uploaded-parameter gives us the uploaded bytes so far. The percentage is easy to calculate. We always return true. Because we don't want to cancel an upload once started.

We can integrate this upload function into our Upload-form. Let's go back to the

<Dropzone/> in our upload-form.tsx. In this function, we already have the file-object. But we don't have the folder-path. But wait, the withRouter-higher-order component we import from 'react-router-dom' gives us the current path: We can use it as the current folder.

We wrap the whole <Dopzone/>-component into this higher-order component and get access to the path. In the onFileSelected-function (line 9-11), we call the upload function we import from './file-storage' (line 2). We provide as arguments the path from the <Dropzone/>'s properties and the file-object from the onChange-event.

Listing 6.13: The upload file integration

```
1  /* ... */
2  import { upload } from './file-storage';
3
4  const Dropzone = withRouter((props) => {
5    /* ... */
6    const UploadFrame = withTheme(styled.label` /* ... */ `);
7    const UploadInput = styled.input` /* ... */`;
8
9    const onFileSelected = event => {
10     upload(props.location.pathname, event.target.files[0]);
11   };
12
13   return <UploadFrame {...props} htmlFor={inputId} >
14     <UploadInput type="file" name={inputId} id={inputId} onChange={
       onFileSelected}/>
15     { props.children }
16   </UploadFrame>
17 });
```

Let's have a look at the app in action. When we start the app with npm run start-dev, we start the whole software stack locally. Including the service and the storage. When you select a file to upload, you can see the messages in the browser's console. The link you get back after the download contains an address to your local storage (localhost:3002). Try opening it in another browser tab.

The local S3 storage runs at localhost:3002.

```
Uploading File - 32%                                                    file-storage.tsx:19
Uploading File - 64%                                                    file-storage.tsx:19
Uploading File - 97%                                                    file-storage.tsx:19
http://localhost:3002/file-management-dev/FILESTORAGE/images/carlos-muz file-storage.tsx:23
a-hpjSkU2UYSU-unsplash.jpg
```

Figure 6.6: The upload output

Uploading files is nice. Of course, we want to show them, too. Infrastructure components provides a ready-to-use component for that: <FilesList/>.

Documentation — The `<FilesList/>`-component.
The `<FilesList/>`-component is a wrapper function. It does not render the files on its own but provides the data. It wraps an internal API-call that requests the list of files on S3. The component requires the following parameters.

- the `storageId` is the `id` of your `<Storage/>`
- the `prefix` is the relative path of the folder whose content you want to get
- the `mode` is one of `LISTFILES_MODE.ALL`, `LISTFILES_MODE.FILES`, `LISTFILES_MODE.FOLDERS`. `ALL` lists all files in this folder and all of its subfolders. `FILES` lists only the files of this folder but not of its subfolders. `FOLDERS` provides a list of the direct subfolders.
- the `onSetRefetch?: (refetch: ()=> any) => void` takes a function that is called when this component mounts (i.e. rendered for the first time). It passes the `refetch`-function as an argument. calling `refetch` invalidates the results thus far and the component fetches the files again.

The `<FilesList/>`-component provides three elements to its direct children. You can use them through specifying a function.

- `loading` is a Boolean value indicating whether the API-request is currently running (`true`) or completed (`false`)
- `data` is `undefined` until the API-request resolves successfully. Then `data` contains a list of items. Each item describes a file. An item is a Javascript object with the filename (`file`), the link to the file on S3 (`url`), the timestamp of the file's last modification (`lastModified`), and the S3-internal path to the file (`itemKey`)
- the `error` is `undefined` unless the API-request fails. Then the `error` contains the error message.

Again, let's make the integration easier and create your own even-more-convenient-component in `file-storage.tsx`.

Listing 6.14: The FileStorageList

```
1  import React from 'react';
2  import {
3    FilesList,
4    LISTFILES_MODE,
5    Storage,
6    uploadFile
7  } from "infrastructure-components";
8  export const FILE_STORAGE_ID = "FILESTORAGE";
9  export const upload = (prefix, file) => uploadFile( /* ... */ );
10 export const FileStorageList = (props) => <FilesList
11   storageId="FILESTORAGE"
12   prefix={props.prefix}
13   mode={LISTFILES_MODE.FILES}>{props.children}</FilesList>;
```

We export the <FileStorageList/>-component. This is the pre-configured <FilesList/> (imported from 'infrastructure-components'). We specify the storageId and set the mode to LISTFILES_MODE.FILES. This limits the result to the files in the current directory. Further, we provide as arguments the prefix and the children we get from the properties.

The integration of the <FileStorageList/> is very easy now. In file-list.tsx, we add it to the returned result of the <SortableList/>:

Listing 6.15: Integration of the FileStorageList

```
1  /* ... */
2  import { FileStorageList } from './file-storage';
3
4  const SortableList = withRouter(withRoutes(SortableContainer(({files, routes,
       location}) => {
5
6    /* ... */
7    return (
8      <StyledList>
9        <Head>Name</Head>
10       { /*... */ }
11
12       <FileStorageList prefix={location.pathname}>{
13         ({loading, data, error}) => (
14           (loading && <div>Loading</div>) ||
15           (error && <div>{error}</div>) ||
16           data.map(
17             (item, index) => {
18               console.log(item);
19               return <SortableFile
20               key={`item-${item.file}`}
21               index={index}
22               href={item.url}>{item.file}</SortableFile>}
23           )
24         )
25       }</FileStorageList>
26     </StyledList>
27   );
28 }));
```

At line 2, we import the named <FileStorageList/>-component we just created. We add it as a child of the <StyledList/>-component and provide the location.pathname (we get through the withRouter-higher-order component) as the prefix.

At lines 13-24, we provide a function as the only child of <FileStorageList/>. This function receives three parameters that the <FilesStorageList/> provides to its children. These are loading, data, and error.

The following construct evaluates whether loading is a "truthy" value (lines 13-15). If it is, we render a `<div/>` saying "Loading". We connect the following parts with a logical OR (||). These don't get evaluated anymore (if `loading` is `true`). If `loading` is `false`, we next check for `error` to be "truthy". If it is, we render the error message. Finally, if neither `loading` nor `error` contains a truthy value, we take the `data` and transform each item into a `<SortableFile/>`-component. As we did it with our statically imported files before.

How to use truthy and falsy values in React

by Dr. Derek Austin

A common problem in React is conditionally displaying the text of a value only when that value exists, such as in the loading example given in this section.
There are multiple ways to solve this problem in JSX, including checking for the absence of null or undefined using the strict equality operator, `!==`, or the loose equality operator, `!=`:

```
1 if(loading !== undefined && loading !== null) { <div>Loading...</div> }
2 if(loading != null) { <div>Loading...</div> }
```

We can make that more concise by using the `&&` (logical AND) operator, which returns the value of second expression if the first expression evaluates to true:

```
1 loading != null && <div>Loading...</div>
```

To simplify further, we can leave out any equality operator. In that case, using `&&` will force the first expression to a Boolean value, either true or false, in a handy one-liner:

```
1 loading && <div>Loading...</div>
```

Values that evaluate to true in Boolean expressions are called truthy values, while values that evaluate to false are called falsy values.
The falsy values in JavaScript are 0, null, undefined, false, NaN, and the empty string "" – everything else is a truthy value, including empty objects and empty arrays [].
Just be careful using the truthy one-liner with numeric values, as the expression will not show for the value 0, because 0 is falsy.

```
1 count && <div>{count}</div>
2 props.count && <div>{props.count}</div>
```

For numeric values, a better solution is to use the typeof keyword or perform a null check:

```
1 typeof count === "number" && <div>{count}</div>
2 count !== undefined && count !== null && <div>{count}</div>
3 count != undefined && <div>{count}</div>
```

Now, our app does not only support uploading files, it also displays them. But there's a problem. If you upload a file, it is not directly shown in the list. You have to refresh the page in the browser first (F5). Of course, we want to display an uploaded file right away.

The `uploadFile`-function provides the `onComplete`-callback-function we can use as the respective trigger. But how do we trigger the re-rendering of another component? If we programmed with side-effects, we would get a reference to the other component and trigger it. But this is not a very React way.

But there's a way out. The `<FilesList/>`-component supports us here. It has the `onSetRefetch-property`. This is a property that expects a function. Once the `<FilesList/>` mounts in the component hierarchy (when it initializes) it calls the function you specify. And `<FilesList/>` passes another function as a parameter. This passed function triggers the `<FilesList/>` to invalidate its result thus far and fetch the data from the back-end again.

Listing 6.16: Definition of the onSetRefetch-property

```
1 onSetRefetch: (refetch: ()=> any) => void
```

But how can we pass the `refetch`-function we get through a callback in the `<FilesList/>` all the way up and down to the `uploadFile`-function?

We need to take the `refetch`, assign it to a variable, and call it when the `uploadFile` triggers it.

The following code does not work! But it illustrates the steps. At lines 6-8, we specify an inline function for the `onSetRefetch`-function. We put this into the `onUpload` variable we declare at line 2. When our `uploadFile`-function calls the `onComplete`-callback, we call the `onUpload`-function stored in the variable (line 13).

Listing 6.17: Definition of the onSetRefetch-property

```
1  /* ... */
2  var onUpload;
3
4  export const FileStorageList = (props) => <FilesList
5      { /* ... */ }
6      onSetRefetch={refetch => {
7        onUpload=refetch
8      }}
9    >{props.children}</FilesList>;
10
11 export const upload = (prefix, file, onUpload) => uploadFile(
12   /* ... */
13   //onComplete: (uri: string) => void,
14   uri => onUpload(),
15   /* ... */
16 );
```

When you run your app and upload a file, you'll get the following error:

Figure 6.7: The upload error

`onUpload` is `undefined`. Of course, it is. Because the `<FilesList/>` calls the function we provide to `onSetRefetch` at the time when it renders for the first time. We work with functional components here. This variable `onUpload` is a side-effect. At its worst.

But we already know a way of how to cope with that. We can use `useState`. But we can't replace `onUpload` by the `useState` declaration: `const [onUpload, setOnUpload] = useState(undefined)`. Because `useState` only works within a functional React-component.

The problem is that the component must be a (grand-)parent of both, the `<FilesList/>`-component and the `<Dropzone/>`-component (that calls the `uploadFile`-function). When we look at our component hierarchy it looks like this:

Listing 6.18: Relevant Hierarchy

```
<Page>
  <StyledList>
    <SortableList>
      <FileStorageList/>
    </SortableList>
  </StyledList>
  <UploadForm>
    <Dropzone/>
  </UploadForm>
</Page>
```

The first common parent is the `<Page/>`-component. This is rather at the top of our component hierarchy. And `<FileStorageList/>` and `<Dropzone/>` are at the very bottom. One way is to pass the functions as properties through the hierarchy. One-by-one. But this is not only very cumbersome. It would also tightly connect all the components in between with our `<FileStorage/>`-implementation.

But we already know a way out of this, too. This is a higher-order component. The only difference to the ones we used before is this time we write our own!

It is pretty simple. Because React provides Contexts. A context provides a way to pass data through the component tree without having to pass properties down manually at every level.

We start with creating the context in our `src/file-storage.tsx`. We provide an empty object as the default value.

Listing 6.19: Create context

```
1 const RefetchContext = React.createContext({});
```

Every Context object comes with a `<Provider/>` and a `<Consumer/>`-component. The `<Provider/>`-component accepts a `value`-property it passes down to any `<Consumer/>` that are descendants of this `<Provider/>`. One `<Provider/>` can be connected to many `<Consumers/>`.

We start with the `<Provider/>`. It is a functional React component. As such, we can use `useState` in it. And this is what we do.

Let's have a look.

Listing 6.20: Create Refetch Provider

```
1  export const RefetchProvider = (props) => {
2
3    const [refetch, setRefetch] = useState(undefined);
4
5    return <RefetchContext.Provider
6      value={{
7        refetch: refetch,
8        setRefetch: setRefetch
9      }}>{props.children}</RefetchContext.Provider>
10 };
```

At line 3, we create our state and its setter (`setRefetch`). The `<RefetchProvider/>`-component returns (thus renders) the `<Provider/>` of the created `RefetchContext` (line 5). We pass a JavaScript object to the `<Provider/>`'s `value`-property. This object contains the current state of `refetch` and the `setRefetch`-function (lines 7-8). At line 9, we let the `<RefetchProvider/>` include whatever children it receives.

The `<RefetchProvider/>`-component must be positioned at least high enough in the component hierarchy to be a (grand-)parent of both `<FileStorageList/>` and `<Dropzone/>`. Thus, at the level of the `<Page/>`-component.

In fact, we created our `<Page/>`-component to hold these types of components. It already wraps the `<ThemeProvider/>` of styled-components. Let's put the `<RefetchProvider/>` there, too.

The `<Consumer/>` of a Context can access the current `value`-property. Even if it is not a direct child. As long as it is a descendant of the `<Provder/>`. All `<Consumer/>`-components that are descendants of a respective `<Provider/>` re-render whenever the `<Provider/>`'s `value` changes.

Listing 6.21: Updated `page.tsx`

```
1  import React from 'react'
2
3  import { ThemeProvider } from 'styled-components';
4  import withStyledTheme from "./styled-theme";
5  import { RefetchProvider } from './file-storage';
6
7  export const Page = withStyledTheme(({styledTheme, children}) => {
8
9    return (
10     <ThemeProvider theme={styledTheme}>
11       <RefetchProvider>
12         {children}
13       </RefetchProvider>
14     </ThemeProvider>
15   );
16 });
17
18 export default Page;
```

Like the other higer-order components we used before, we want to have a wrapper function. We want to wrap it around a function and inject the respective property. Here's the code.

Listing 6.22: Context Consumer

```
1 export function withRefetch(Component) {
2   return function WrapperComponent(props) {
3     return <RefetchContext.Consumer>{
4         (context) => <Component {...props} refetch={context.refetch}/>
5     }</RefetchContext.Consumer>;
6   };
7 };
```

The `withRefetch`-function takes a component as an argument (line 1). And `withRefetch` returns a React component. This is the `<WrapperComponent/>` (line 2). Remember, React components are nothing but functions. Functions that take a property-objects and return another Component or HTML-element.

In our case, we return the `<Consumer/>` of our context (line 3). The `<Refetch.Consumer/>` provides the context to its child-function. We take this context and return the Component (line 4) we received as an argument of our `withRefetch`-function (line 1). That is whatever we put into our wrapper function.

When we return the component, we take all the properties from the `<WrapperComponent/>` and put it into the wrapped function. Further, we provide the `refetch`-property. This is the `refetch`-function we take from our Context. That is the same `refetch`-function we

control through the `useState` in the `<Provider/>`.

So whatever component (aka function) we put into the `withRefetch`-function, it returns another component (aka function) (line 2) that renders this component with the same properties and the additional `refetch` from our Context (line 4).

How does property forwarding work?
We pass our properties into the `<Component/>` using the Spread syntax. This syntax with three leading dots in front of an object (`...props`) expands the `props`-object into a list of key-value pairs. You don't need to know the keys.

Let's now integrate our new wrapper function. The `<Dropzone/>` requires the `refetch`-function. We integrate it in a similar way like our other higher-order components.

Listing 6.23: Updated Dropzone in upload-form.tsx

```
1 import { upload, withRefetch } from './file-storage';
2 const Dropzone = withRefetch(withRouter((props) => {
3   /* ... */
4   const onFileSelected = event => {
5     upload(props.location.pathname, event.target.files[0], props.refetch);
6   };
7
8   return <UploadFrame {/* ... */} />
9 }));
```

At line 1, we wrap the `<Dropzone/>`-component into the `withRefetch`-function. As we did it with the `withRouter`-function. This function injects the `refetch`-property we use at line 5.

In the last step, we do the same thing with the `setRefetch`-function. We create the following wrapper function and integrate it into the `<FileStorageList/>`-component that provides this function to the `<FilesList/>`.

When our UI creates the `<FilesList/>`, it calls the `setRefetch`-function that our Context transports from the low-level `<FilesStorageList/>`-component up to the `<RefetchProvider/>`. This calls the `setRefetch` of the `useState`-hook and updates the state. The `refetch`-constant now contains the function we can use to trigger the `<FilesList/>` to refetch the data from our storage.

During rendering, the `<ContextProvider/>` passes down the `refetch`-function to the `<Dropzone/>`. The `<Dropzone/>` uses it when it calls the `uploadFile`-function once a file-upload is complete. As a result, the `<FilesList/>` reloads the data and updates our list.

Listing 6.24: Updated Dropzone in upload-form.tsx

```
export function withSetRefetch(Component) {
  return function WrapperComponent(props) {
    return <RefetchContext.Consumer>{
      (context) => <Component {...props} setRefetch={context.setRefetch}/>
    }</RefetchContext.Consumer>;
  };
};

export const FileStorageList = withSetRefetch((props) => <FilesList
  storageId="FILESTORAGE"
  prefix={props.prefix}
  onSetRefetch={props.setRefetch}
  mode={LISTFILES_MODE.FILES}>{props.children}</FilesList>
);
```

We prepared the `<Dropzone/>` to initiate a file upload when you drop a file on it. Let's implement this feature. File drag and drop is part of the HTML-API. It is well documented.

Listing 6.25: The drag and drop integration

```
1 function UploadForm (props) {
2   return <Dropzone onDrop={dropHandler(props.location.pathname)} onDragOver={
      dragOverHandler}>
3     <FontAwesomeIcon icon={faFileUpload} size="4x" color="#888"/>
4     <span>Upload Files</span>
5   </Dropzone>
6 };
```

The integration of drag and drop events happens through the respective handlers we attach to the `<Dropzone/>`-component inside the `<UploadForm/>` (line 2). Note we provide function declarations in both cases. Even though we call the `dropHandler`-function (we implement this in a second) it is a function declaration. Because we do not explicitly take the parameters of this callback function. We use this form to pass the current location `pathname` to the handler.

When you look at the implementation of the `dropHandler`-function (line 13), you can see a new construct: `dropHandler = location => ev => {...}`. This is a curried function. Let's take it apart. `dropHandler` is a constant. It receives the anonymous function `location => {some result...}`. This function takes the `location` as a parameter. And it returns another function `ev => {some other result...}`. This other function is our basic event handler. It takes an event as a parameter and contains the code we want to run when the user drops a file onto the `<Dropzone/>`.

Listing 6.26: The drag and drop handlers

```
1  const uploadAll = (prefix, arr) => {
2    Object.values(arr)
3      .filter(item => item.kind === 'file')
4      .forEach(item => upload(prefix, item.getAsFile()));
5    return true;
6  }
7
8  const dragOverHandler = ev => {
9    // Prevent default behavior (Prevent file from being opened)
10   ev.preventDefault();
11 }
12
13 const dropHandler = location => ev => {
14   // Prevent default behavior (Prevent file from being opened)
15   ev.preventDefault();
16
17   if (ev.dataTransfer.items) {
18     uploadAll(location, ev.dataTransfer.items);
19   } else {
20     upload(location, ev.dataTransfer.files);
21   }
22 }
```

This new construct is function currying. We separate a function with multiple paramaters into a "chain" of functions you can call subsequently. During the definition of the event-handler (in the `<UploadForm/>` we call the first part: `dropHandler(props.location.pathname)`. We pass as an argument the `pathname`. It returns the event-handler function. When an event occurs, this function receives the event.

The body of the event-handler is quite simple. At line 15, we prevent the default behavior of dropping a file onto a browser tab. This would be opening the file. At line 17, we check whether the user drags a single file or multiple files. If we have multiple files, we upload them all through our `uploadAll`- function (lines 1-6). This function takes all files from the `ev.dataTransfer.items`-array, filters file items, and uploads them one-by-one.

If `ev.dataTransfer.items` is not defined. This is the case when the user drags a single file. We can access the `files`-object for it contains the file.

In the `dragOverHandler`, the only thing we do is to prevent the default behavior. Because we don't want to open the file.

6.5 Working With Services

*I thought we're creating a **service**-oriented app here? I don't see the service, yet! I'd expect some back-end here!* you say?

You're right. The back-end is well hidden in our `uploadFile`-function and in the `<FilesList/>`-component. Let's become more specific about the back-end services we use here. In this section, we supplement the files with some advanced, application-specific meta-data. We add the owner (the person who uploads the file) and a description.

Let's start with the user interface. When the user selects a file or drops it onto the `<Dropzone/>`, we do not start the upload right away. Instead, we display a form to allow the user to enter the username and a description of the file.

Listing 6.27: The `<Dropzone/>`-component

```
1  const Dropzone = withRouter((props) => {
2    const inputId = "uploadfile";
3    const UploadInput = styled.input`
4      display: none;
5    `;
6
7    const onFileSelected = event => {
8      props.selectFile({
9        pathname: props.location.pathname,
10       file: event.target.files[0]
11     });
12   };
13
14   return <LabelFrame {...props} htmlFor={inputId} >
15     {
16       props.active && <UploadInput
17         type="file"
18         name={inputId}
19         id={inputId}
20         onChange={onFileSelected}/>
21     }
22     {
23       props.children
24     }
25   </LabelFrame>
26 });
```

In our `<Dropzone/>`-component, we do not start the upload right when the user selects a file (lines 7-12). But we call the `props.selectFile`-function we expect in the component properties. We provide as an argument an object with the current `pathname` (taken from the `withRouter`-higher-order component) and the `file`-object the user selected.

Further, we introduce expect an `active`-property. Only if `props.active` is truthy, we

show the `<UploadInput/>`.

The main change happens in the `<UploadForm/>`. Let's have a look.

Listing 6.28: The `upload-form.tsx` With the extended form

```
function UploadForm (props) {
  const [selectedFile, selectFile] = useState(undefined);
  const [author, setAuthor] = useState("");
  const [description, setDescription] = useState("");

  return <Dropzone
    active={!selectedFile}
    selectFile={selectFile}
    onDrop={dropHandler(props.location.pathname, props.onUpload)}
    onDragOver={ev =>{ev.preventDefault()}}
  >{
    !selectedFile ? <React.Fragment>
        <FontAwesomeIcon icon={faFileUpload} size="4x" color="#888"/>
        <span >Upload Files</span>
    </React.Fragment> : <>
      <div>{selectedFile.file.name}</div>
      <input
        type="text"
        placeholder="Enter Author"
        value={author}
        onChange={event => setAuthor(event.target.value)}/>

      <input
        type="text"
        placeholder="Enter Description"
        value={description}
        onChange={event => setDescription(event.target.value)}/>

      <button onClick={()=>{
        upload(selectedFile.pathname, selectedFile.file, () => {
          props.refetch();
          selectFile(undefined);
        }, {
          author: author,
          description: description
        });
      }}>Upload now</button>
      <button onClick={(ev)=>{
        // prevent the label to trigger the file-selection right away
        ev.preventDefault();
        selectFile(undefined);
      }}>Cancel</button>
    </>
  }</Dropzone>
};
```

At line 3, we add another state. It holds the selected file. We pass the `selectFile`-function into the `<Dropzone/>` as the callback we just implemented. Thus, when the `<Dropzone/>` handles a file selection, it calls the `selectFile`-function and updates the component state. It writes the file data into `selectedFile`.

`selectedFile` serves us to specify the `active` property of the `<Dropzone/>`. It is `active` when there is no selected file. Thus, once the user selects a file, the `<UploadInput/>` within the `<Dropzone/>` disappears.

We provide a Boolean expression as the child of the `<Dropzone/>` (lines 12-43). Again, we check whether there is a selected file or not. If the user did not yet select a file, the condition `!selectedFile` is truthy and we render our `<FontAwesomeIcon/>` and `<span/>`.

But wait. We render them inside a `<React.Fragment/>` (line 12). So far, we rendered them as direct children to the `<Dropzone/>`. This is because a Boolean expression expects a single one child. Not a list of children. But if we wrapped the two children into a `<div/>`, we would lose our styling that the `<Dropzone/>` applies. This is the `<Dropzone/>` laying out its children vertically. The `<React.Fragment/>` groups a list of children without adding extra nodes to the DOM. It is the common pattern in React for a component to return multiple elements.

If `<React.Fragment/>` is too long for you. There is a shorter syntax you can use for declaring fragments. It looks like empty tags: `<> ... </>`. We use this in the case when there is a selected file (lines between 15 and 43). I do not suggest to mix the two styles in your code as we did here. But I wanted to demonstrate the two ways.

If there is a selected file, we show the `name` of the selected file (line 16) and a form with two input fields and two buttons (lines 17-42). We connect each input field with a state. This means, we take the value from the current state (lines 19 and 26). We start with an empty string when we initialize the state-hooks (line 3 and 4). Whenever the user types a letter (or deletes one), we listen on the `onChange`-event. Rather than showing the value directly within the field, we set the state (lines 21 and 27). The updated state triggers rerendering the input field with the new value. We use the React-rendering pipeline rather than manipulating HTML-elements through references.

This pattern further lets us access the current values of the two input fields at any time. We make use of this possibility in our "Upload now"-button. We specify a callback-function for the `onClick`-event of the button. The first thing we do, is we prevent any default action from being triggered. This default Action would be the `onClick` of the `<label/>` we have placed our buttons in. The `<label/>` is connected to the file-selection-dialog. But we don't want to open this again, when we click at one of our buttons.

In the "Upload now"-button, we provide the current values of the `author` and the `description` as additional data (lines 33-36). In the `onComplete`-handler, we trigger our list to refetch its data (line 31). And we reset the `selectedFile` by setting the state to

our intitial value `undefined` (line 32).

We also provide a button to cancel the upload of a file. Maybe the user selected the wrong one. In this case, we reset the selected file to `undefined` directly, without uploading it first (line 41).

We provide the additional data to the `upload`-function. Of course, we need to use it. We pass it through to the `uploadFile`-function (line 5). Have a look.

Listing 6.29: Complementinf the `upload`-function with additional data

```
1  export const upload = (prefix, file, onUpload, data) => uploadFile(
2    FILE_STORAGE_ID,
3    prefix,
4    file,
5    data,
6    //onProgess: (uploaded: number) => Boolean,
7    uploaded => {
8      const percent_done = Math.floor( ( uploaded / file.size ) * 100 );
9      console.log("Uploading File - " + percent_done + "%");
10     return true;
11   },
12   //onComplete: (uri: string) => void,
13   uri => onUpload(), //onUpload(uri),
14
15   //onError: (err: string) => void
16   err => console.log(err)
17
18 );
```

`uploadFile` sends the data to the back-end service alongside the file. Thus, we're ready to do something meaningful at the server-side.

In the `file-storage.tsx`, we can add a `<Middleware/>`-component as a child of the `<Storage/>`-component. A `<Middleware/>` takes a `callback`-function as a parameter. This function contains the code that runs at the server! It is an "Express.js"-middleware. If you are familiar with "Express.js", great. If not, no problem, we'll go through it right away.

Documentation — <Middleware/>-`callback`-property.

- The `request` contains all the data we get from the browser.
- The `response` is a preparation we can use to respond to the browser request.
- If we don't want to respond to the request directly, we can call the `next` function. This calls the next `<Middleware/>` or the `render` function of the `<Route/>` if there are no more `<Middleware/>`s (as in our example).

Listing 6.30: The `<Middleware/>` of the `<Storage/>`

```
import {
  Middleware,
  Storage,
  STORAGE_ACTION
} from "infrastructure-components";

/* ... */

export default function () {
  return <Storage
    id={FILE_STORAGE_ID}
    path="/filestorage"
  >
    <Middleware
      callback={ function (req, res, next) {
        const parsedBody = JSON.parse(req.body);
        res.locals = parsedBody.data;
        console.log("this is the service: ", parsedBody.data, parsedBody.
    action);
        if (parsedBody.action == STORAGE_ACTION.UPLOAD
          && parsedBody.data
          && parsedBody.data.author
          && parsedBody.data.author.toLowerCase() !== "frank") {
            return res.status(403).set({
              /* Required for CORS support to work */
              "Access-Control-Allow-Origin" : "*",
            }).send("not allowed");
          }

        return next();
      }}
    />
  </Storage>
};
```

In figure 6.9, you can see a Lambda function receiving the request of the client and sending it to S3. This Lambda is a `Middleware/>`.

`<Middlewares/>` run in a sequence. One-by-one. A `<Middleware/>` receives the `request`, the preliminary `response`, and a `next`-function. (There are also `error`-handlers that we'll cover later).

When we add another `<Middleware/>`, it runs before the default `<Middleware/>` of `uploadFile`. The following image depicts our back-end with two middlewares.

In our code, we implement a verification whether we want to upload the file at all, by

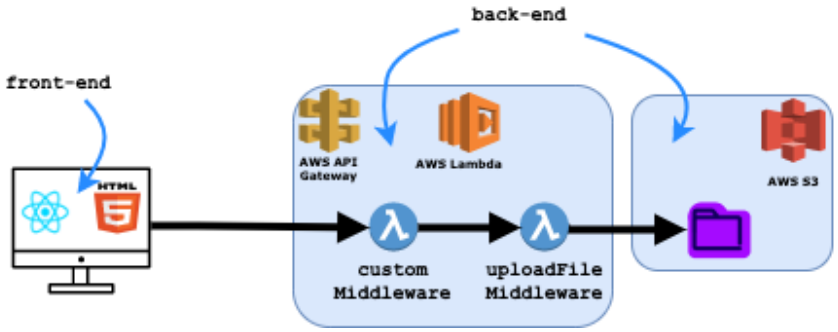


Figure 6.8: A back-end with two middlewares

proceeding with the `next()` (default) `<Middleware/>`. Or to return a response to the client and skipping the default behavior.

The `request`-object contains all the information provided by the client. This includes the `body` we can parse into a JavaScript-object (line 16). The `body` contains the data we sent alongside the file. And it includes the `action` of the call. We need this distinction, because the storage-service processes different requests. Serving the list and the upload requests. we can use the `STORAGE_ACTION` to distinguish both requests.

The `response` is the object we can be send back to the client. The following code sets the HTTP response code to 403 (meaning "Forbidden") (see HTTP response codes here. we set an HTTP header (lines 23-26) to allow CORS to work. CORS is the Cross-Origin-Resource-Sharing. If not enabled, the browser blocks requests that come from a different domain than the page. We need CORS to be set to `*` in order for our app to work locally in offline mode.

Each request can only be replied to with a single response. Thus, once you send a response, the browser succeeds. Even though your server code may continue. Whatever they do, they would not be able to tell the browser and thus the front-end about it.

> AWS Lambda proceeds with your code even if you sent a response. However, it does not start any new thread (e.g. using async/await). Lambda best practice is to send the response at the end of processing the request.

The `next`-function hands over to the next `<Middleware/>`. If you do not want further `<Middlewares/>` to be called, your function should `return` without calling next(). In this case, you'll need to make sure that you provide a response to the client. Otherwise it'll wait, and wait, and timeout after 30 seconds. Don't let your browser down waiting for you.

If you want to forward data from one `<Middleware/>` to the next `<Middleware/>`, you can use the `res.locals`-object. E.g. `res.locals = parsedBody.data;`.

The `uploadFile`-service includes the `res.locals`-object as `data` in the response. You can use it to return data to the client. If your `<Middleware/>` sends the response, you'll need to make sure that the response contains all the data you want it to contain.

In the code, we check whether the author of an uploaded file is me `"frank"`. Comparing strings in a case insensitive manner means to compare them without taking care of the uppercase and lowercase letters. To perform this operation the most preferred method is to use `toLowerCase()`-function. If `"frank"` is not the author, the default `<Middleware/>` handling the upload of the file does not run.

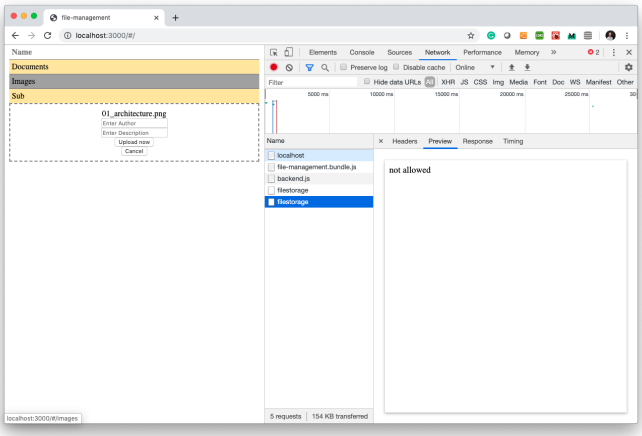


Figure 6.9: Forbidden request

Let's try this. In the first attempt, we try to upload a file without specifying the author. The first three requests belong to loading the page, including loading the list from the storage. The fourth request is our file upload. As you can see, it returns "not allowed". The string we sent in our response.

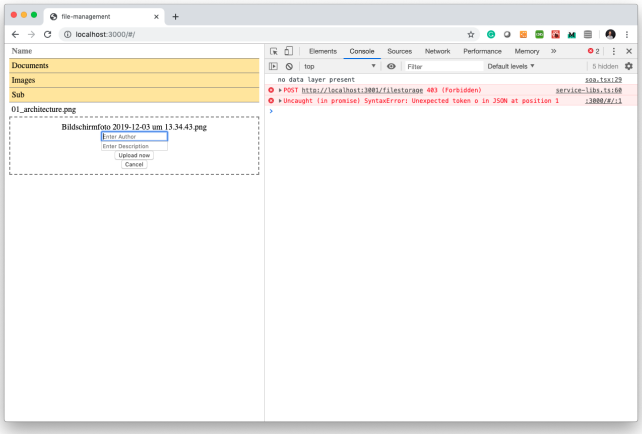


Figure 6.10: Console output

You can see that our user interface did not update! We did not implement a handler in case of an error. But we simply printed it to the console. The next picture shows the console output. As intended, it shows the error code 403 (Forbidden)

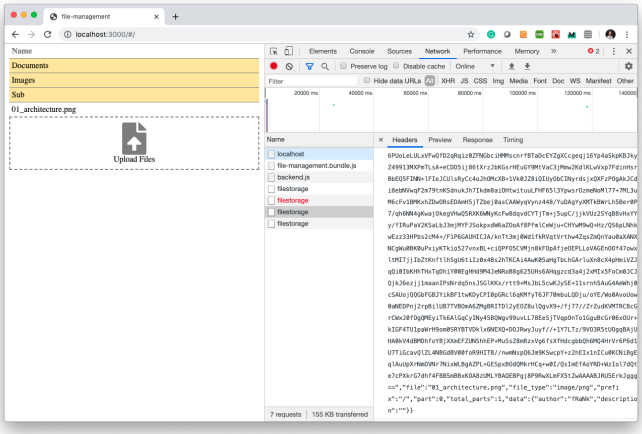


Figure 6.11: Allowed file upload

Finally, let's have a look at the console output of our server. You can see this in the console of your IDE. Or wherever you ran the `npm run start-dev` command from.

```
Serverless: POST /filestorage (λ: server)
this is the service:  { author: '', description: '' } UPLOAD

Serverless: POST /filestorage (λ: server)
this is the service:  { author: 'fRaNk', description: '' } UPLOAD
info: Stored object "FILESTORAGE/01_architecture.png_ICPART_0" in bucket "file-management-dev" successfully
info: management-dev/FILESTORAGE/01_architecture.png_ICPART_0 200 26ms 0b
info: management-dev/FILESTORAGE/01_architecture.png_ICPART_0 200 12ms 28.79kb
info: le-management-dev/FILESTORAGE/01_architecture.png_ICPART_0 204 3ms
info: Stored object "FILESTORAGE/01_architecture.png" in bucket "file-management-dev" successfully
info: management-dev/FILESTORAGE/01_architecture.png 200 5ms 0b
```

Figure 6.12: Server console output

The first two lines belong to our first request without an entered username. As you can see, the console statement prints `"this is the service"` and the data. But nothing happens afterwards.

In the second request, the `data` contains a valid author. The following logs come from the `uploadFile`-service. Apparently, our file upload was successful.

6.6 Using A Database

Our back-end receives the `data` from our front-end. This includes a `description`. Yet, we do not store it. This is a job for a database! Remember the architecture of our service-oriented app in figure 7.1. It includes a database.

The following image depicts the relevant parts of this architecture. By now, you may expect

`infrastructure-components` to provide a React-component that adds a database. Rather than forcing us to develop our own. And you're right.

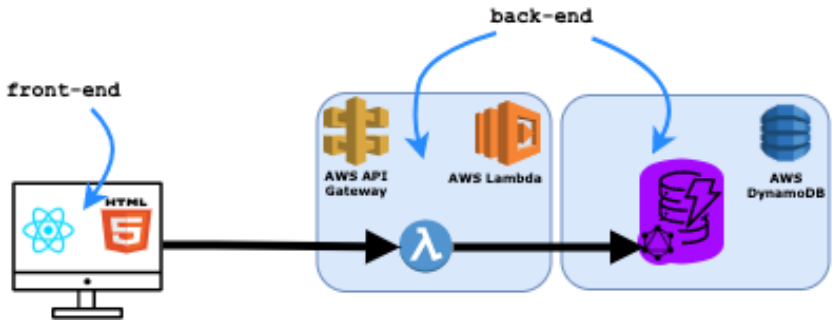


Figure 6.13: Integration of a database

We start with adding the required components to our `src/index.tsx`.

Listing 6.31: Adding a database

```
1  /* ... */
2  import {
3    DataLayer,
4    Environment,
5    Route,
6    ServiceOrientedApp
7  } from "infrastructure-components";
8
9  /* ... */
10 import FileMetaDataEntry from './file-meta-data-entry';
11
12 export default (
13   <ServiceOrientedApp>
14     <Environment name="dev" />
15     <FileStorage />
16     <DataLayer id="datalayer">
17       <FileMetaDataEntry />
18     </DataLayer>
19     {
20       folders.map((folder, index)=> <Route
21         render={(props) => <Page>{/* ... */}</Page>}
22       />)
23     }
24   </ServiceOrientedApp>
25 );
```

At line 16, We add the `<DataLayer/>` as a direct child of our top-level. This component adds all services and the database we need.

Documentation — <DataLayer/>.

- the `id` specifies the unqiue idenfifier. You can use an arbitrary name (lower case).

The <DataLayer />-component supports <Entry/>-components as direct children. An <Entry/> describes the type of items in your database. This is the structure of our data in the database. An <Entry/> is the abstract description of your data. It is not an instance. We refer to an instance of an <Entry/> as an item.

We add our customized <FileMetaDataEntry/>. We import it from file file-meta-data-entry.tsx.

Documentation — <Entry/>.

- the id specifies the unqiue idenfifier. You can use an arbitrary name (lower case).
- the primaryKey specifies the name of an attribute to serve as a key.
- the rangeKey specifies the name of an attribute to serve as another key.
- data is a JavaScript object. The keys of this object are arbitrary strings. These determine the fields of your data.

You can use any name except for some **DynamoDB keywords** you can find here. The list is quite long. Make sure you check the list before specifying your keys.

The combination of primaryKey and rangeKey describes a unique item in the database. There can't be a second item with the same values of these keys.

The following code depicts the implementation of our custom <Entry/>.

Listing 6.32: title

```
1  import React from 'react';
2  import { Entry } from "infrastructure-components";
3  import { GraphQLString } from 'graphql';
4
5  export const FILEMETADATA_ENTRYID = "filemetadata";
6
7  export default function FileMetaDataEntry (props) {
8    return <Entry
9      id={ FILEMETADATA_ENTRYID }
10     primaryKey="prefix"
11     rangeKey="filename"
12     data={{
13       author: GraphQLString,
14       description: GraphQLString,
15     }}
16   />
17 };
```

We use the prefix plus the filename as the unique key-combination. Thus, there can be no two files with the same path (prefix) and filename. This is reasonable, because it reflects our understanding of a file system.

The `<FileMetaDataEntry/>` is our database-side specification of the meta-data. We specify the following interface as the definition of the meta-data-object in our app.

Listing 6.33: The `IMetaData`-interface

```
1 export interface IMetaData {
2   prefix: string,
3   filename: string,
4   author: string,
5   description: string,
6 }
```

You can see, the two keys (`prefix` and `filename`) become parts of the object. They do not only serve as identifiers. They hold important data of our meta-data.

With this interface, we can use our IDE to verify our syntax and completeness of provided data. The following function writes an `IMetaData` to the database.

Listing 6.34: Writing an item to the database

```
1  import { mutate } from "infrastructure-components";
2
3  export async function addMetaData(dataLayer, data: IMetaData) {
4    await mutate(
5      dataLayer.client,
6      dataLayer.setEntryMutation(
7        FILEMETADATA_ENTRYID,
8        Object.assign({}, data, {
9          prefix: "pre_"+data.prefix,
10       })
11     )
12   );
13 };
```

The `mutate`-function (we import from `infrastructure-components` in line 1) changes or adds an item in/to the database.

Documentation — mutate.
`mutate` is an asynchronous function. It alters an item in your database.

- the `client` (first argument) specifies the GraphQl-client. When using infrastructure-components, you'll get this client from the `dataLayer`-object.
- the `mutation-command` (second argument) specifies the type and data of the database action you want to run. You create the `mutation-command` through the `dataLayer`-object helper functions, e.g. `setEntryMutation`

The `dataLayer`-object provides three helper functions to be used with the `mutate`-function.

- `setEntryMutation` adds an item to the database with the specified keys. It overwrites

an existing item with these keys, if it exists.
- `deleteEntryMutation` deletes an item from the database with the specified keys.
- `updateEntryQuery` adds an item to the database with the specified keys. It overwrites an existing item with these keys, if it exists. In the latter case, it provides to you the current values of the existing item.

Documentation — `setEntryMutation/deleteEntryMutation/updateEntryQuery`.
- the `entryId` (first argument) specifies the identifier of the `<Entry/>` the item belongs to.
- **ONLY** `setEntryMutation`: the `values`-object (second argument) specifies the data of the item. It must have all used keys (`primaryKey` and `rangeKey`) and all the data you want the item to have.
- **ONLY** `deleteEntryMutation`: the `values`-object (second argument) must specify the keys (`primaryKey` and `rangeKey`).
- **ONLY** `updateEntryQuery`: the `setValues`-function (second argument) receives as an argument the old data of the item (if it exists, otherwise it receives an empty object `{}`). This function must return an object that specifies the data of the item. It must have all used keys (`primaryKey` and `rangeKey`) and all the data you want the item to have.

In the code above, we wrap the `mutate`-function into our own, custom `async`-function. We take the `dataLayer` and the `IMetaData`-object as arguments. The `dataLayer` serves to provide the GraphQL-`client` and the `setEntryMutation`-function.

Usually, we would pass through the `data` as second argument to the `setEntryMutation`-function. But we change the data here. At lines 8-10, we create a new object (through `Object.assign`), pass the `data`-object into it, and overwrite the `prefix`-key. We attach the string `"pre_"` at the start of the `prefix`.

We do this because `prefix`es may start with a slash (`/`). But keys in our database must be valid strings. The fixed string `"pre_"` ensures our `prefix` to start with letters.

Our `addMetaData`-function is ready for use. Let's see it in action.

Listing 6.35: Accessing the database from the middleware

```
1  import { addMetaData } from './file-meta-data-entry';
2
3  /* ... */
4  <Middleware
5    callback={ serviceWithDataLayer(
6      async function (dataLayer, req, res, next) {
7        const parsedBody = JSON.parse(req.body);
8        res.locals = parsedBody.data;
9        console.log("calling data: ", parsedBody.data, parsedBody.action);
10
11       if (parsedBody.action == STORAGE_ACTION.UPLOAD
12         && parsedBody.data
13         && parsedBody.data.author
14         && parsedBody.data.author.toLowerCase() !== "frank") {
15           return res.status(403).set({
16             /* Required for CORS support to work */
17             "Access-Control-Allow-Origin" : "*",
18           }).send("not allowed");
19       }
20
21
22       await addMetaData(dataLayer, {
23           prefix: parsedBody.prefix,
24           filename: parsedBody.file,
25           author: parsedBody.data.author,
26           description: parsedBody.data.description,
27         }
28       );
29
30       return next();
31     }
32   )}
33 />
```

At line 5, we wrap our callback-function into the `serviceWithDataLayer`-HOC. It adds the `datalayer` to our function arguments (at the first position). At the lines 11-19, there is our username check. Right afterwards, if the username check did not return, we add the meta-data to our database (lines 22-28) by calling our convenience-function. We proceed with the `next` (the default `fileUpload`) middleware.

The order of our code is important here. First, we do the check for the username. This includes a check for `parsedBody.data` to be defined. Depending on the size of the file, uploads may get split into multiple parts. Thus, the service gets called multiple times. But `parsedBody.data` is only defined once for a file upload (at the last call). This check ensures we call `addMetaData` only once. Thus, we insert an item to database only once,

too.

Of course, we want our app to display the meta-data we inserted into the database. `infrastructure-components` build upon `react-apollo`. Thus, we can use `react-apollo`'s `<Query/>`-component. It integrates with GraphQL interfaces, such as the one our datalayer provides.

We create a convenience component to connect our `<FileMetaDataEntry/>` with the `<Query/>`-component. Have a look.

Listing 6.36: The MetaDataQuery

```
1  /* ... */
2  import { withDataLayer } from "infrastructure-components";
3  import { Query } from 'react-apollo';
4  export const FILEMETADATA_ENTRYID = "filemetadata";
5
6  /* ... */
7
8  const prePrefix = (prefix: string) => "pre_"+prefix;
9
10 export async function addMetaData(dataLayer, data: IMetaData) {
11   mutate(
12     dataLayer.client,
13     dataLayer.setEntryMutation(
14       FILEMETADATA_ENTRYID,
15       Object.assign({}, data, {
16         prefix: prePrefix(data.prefix),
17       })
18     )
19   );
20 };
21
22 export const MetaDataQuery = withDataLayer(props => (
23   <Query {...props.getEntryQuery(FILEMETADATA_ENTRYID, {
24     prefix: prePrefix(props.prefix),
25     filename: props.filename}
26   )} >{
27     ({data}, results) => (
28       <props.children
29         {...results}
30         fileMetaData={data ? data[`get_${FILEMETADATA_ENTRYID}`] : undefined}
31       />
32     )
33   }</Query>)
34 );
```

The `<MetaDataQuery/>`-component starts at line 22. We use the `withDataLayer`-function to insert the dataLayer-helper functions into the props. We directly return a <Query/>-component (line 23). Further, we expect our properties to contain all fields of the

IMetaData-interface. At the lines 23-26, we use the getEntryQuery-helper function. It builds a query for a single item whose prefix and filename match the values specifies within the properties (props.prefix and props.filename).

We added "pre_" to the prefix when we stored the meta-data. So, we need to do the same when retrieving the data. This is the same transformation. Thus, we add a respective function (line 8) and use it in the addMetaData function (line 16) and in our MetaDataQuery (line 24).

Our component returns a <Query/>-component that provides three parameters to its children: loading, data, and error. We "grab" data from it. The other two get collected in the results-object.

We pass the results-object (that contains loading and error) to the returned <props.children/>-component. Rather than explicitly forwarding each property separately, we use this pattern to focus on the data we're interested in.

The data-object contains the get_${FILEMETADATA_ENTRYID}-object. We provide this sub-object to the <Query/>-component (line 30).

The use of our convenience-<MetaDataQuery/>-component is straight forward. We use it in the <FileEntry/>-component. We can omit all the heavy loading now. We simply provide the prefix and the filename from our props.

Listing 6.37: Use of the <MetaDataQuery/>

```
1  import { MetaDataQuery } from './file-meta-data-entry';
2  const FileEntry = (props) => <Item>
3    <FileLink download target="_blank" {...props}/>
4    <MetaDataQuery prefix={props.prefix} filename={props.filename}>{
5      ({loading, fileMetaData, error}) => {
6        if (loading) {
7          return <div>...Loading...</div>;
8        }
9
10       if (fileMetaData) {
11         console.log(fileMetaData)
12         return <div>{fileMetaData.description} {fileMetaData.author}</div>
13       }
14
15       return <div>Error</div>
16     }
17   }</MetaDataQuery>
18 </Item>;
```

We changed the props a little bit. The <FileEntry/> took the filename as a child thus far. We did not care about its structure. But since we now want it to be a string (and not some arbitrary component), we take it as a named-property: filename.

We have to edit the use of `<FileEntry/>` accordingly

Listing 6.38: Edited use of `<FileEntry/>`

```
const SortableFile = SortableElement(FileEntry);
const SortableList = withRouter(withRoutes(SortableContainer(({files, routes,
    location}) => {
  /** ... */
  return (
    <StyledList>
      { /* .../*}
      {
        files.filter(
          item => item.path == location.pathname
        ).map((file, index) => (
          <SortableFile
            key={`item-${file.name}`}
            index={index}
            href={file.href}
            prefix={location.pathname}
            filename={file.name}/>
        ))
      }
    </StyledList>
  );
}));
```

We render the `<FileEntry/>` as a `<SortableFile/>` in the `<SortableList/>`. This component already accesses the `location.pathname` we use as the `prefix` and it gets the `file`-object including the `filename`. So, we add these two properties.

6.7 Evaluation

Regarded from a front-end perspective, a service-oriented App is still a single-page App. But once we see the app holistically, a service-oriented app comprises web-services, a file-storage, and a database. These are back-end resources. It uses the full stack.

Of course, you can add these back-end resources to any single-page-app. Even though the front-end code may not change, it makes the app a full-stack app. This app adheres to the JAM-stack principle. It is JavaScript, APIs, and Markup (HTML).

And we have seen a lot of the potential of a Serverless Service-Oriented React App in this chapter. We have covered the following topics:

- Uploading Files
- Serving Dynamic Files
- Create your own Higher-Order Component

- A custom back-end service
- A simple authentication mechanism (checking for the username)
- Working with a database

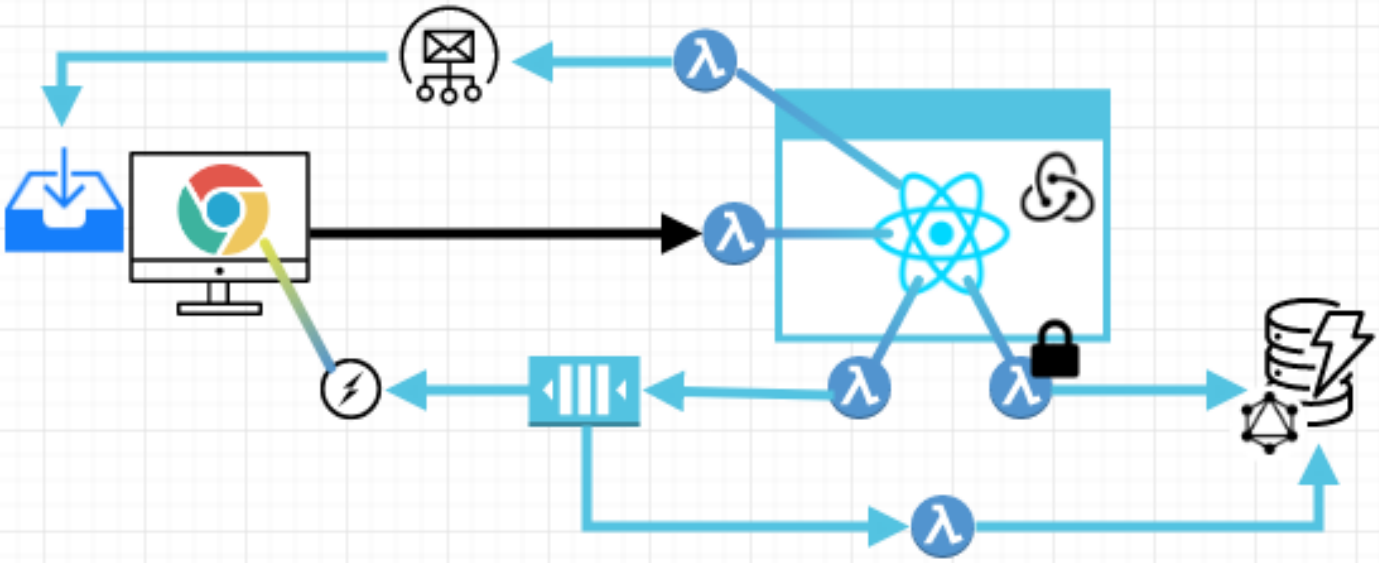
With the use of services and dynamic data storage (file-storage and a database), our app is dynamic. We integrated these services into our app. We have put quite some effort into reducing side-effects. And we have been quite successful doing it. If you look at the `file-meta-data-entry.tsx` or the `file-storage.tsx`, you can see these files comprise code running at the front-end and code running at the back-end. With an integrated full-stack app, we can structure our code by its purpose and not by its infrastructure.

That way, we are able to reduce side-effects and source code dependencies. Our modules (a source code file is a module) are loosely coupled and hide their internal (technical) logic from each other. We don't rely on global values that need to match across distant files.

If you would separate your source code based on its infrastructure (front-end, back-end services, database, etc.) you would need to make sure the paths, URLs, data structures, etc. fit together. Simple things like adding a constant value would require you to open different projects, edit different files just to enter the same source code. You'd run the risk of forgetting to edit the code at a required place. The result would be increased effort and risk.

Of course, there are situations when you may prefer separating your services from the rest of your code. And you are fine to do so. A service-oriented app is not limited to using its services. However, the integration comes with benefits regarding conciseness and source code structure. Even for a modular service-oriented app, where integration is optional.

In the next section, we'll look at a type of app that tightly couples front-end and back-end: the Serverless Isomorphic React App. For this type of app, integration is not an option. It is the building-block.



7. Serverless Isomorphic React App (SIRA)

Serverless Isomorphic React Apps differ from single-page and service-oriented apps in a fundamental aspect: the render process. Isomorphic Apps render at the server-side and at the client-side. Usually, the emphasis is put on the server-side-rendering (SSR).

Search-engine-optimization (SEO) is the main reason to use such an architecture.

While server-side rendering makes the job of search engines easier, the argument gets older. Today, Google is said to index JavaScript-based web apps as good as any classic HTML content. It is not much of a surprise. With web-apps becoming increasingly popular, missing all these contents would risk the usefulness of a search engine. Something Google won't want to risk. So it is in their own interest.

But if SEO is not a reason anymore, why should I bother spending all the work into rendering the app on the server-side?

One reason is personalization. A SPA provides the same app and the same content to everyone. Of course, it can load dynamic content from a service (making it an SOA). But the first response from the server is the same for every user. Because the server does not run any code before it responds. By contrast, an isomorphic app runs code at the server-side. Upon the first request, it can personalize the user experience.

Another reason is authentication. If you have secret content, you have to secure it. And this requires you to have full control of the running code. Of course, you can use services. But the ability to run your own code before the user receives the response from the server makes securing a route easy.

You can decide on whether your app should serve a route and respond to a request on the fly,

depending on the user. In a SPA setting, you have to serve the route first and then exclude the user from access the secret data afterward.

Why is Server-Side-Rendering SEO-relevant?

Search engines, such as Google, are able to provide search results because they have indexed the resources on the internet. They work like a big telephone book. Are you old enough to remember these books? I haven't seen one in years anymore.

Well, with all these websites out there, you can imagine how much work (even for automated bots) it is to create the index and to keep it up-to-date. These bots visit the websites to get their content. But they don't want to interact with it as a user would. So, why should it run the JavaScript code that is there for interactivity? It doesn't. But when you use React, your whole website is an empty HTML and relies upon JavaScript code to fill it with content. If the search engine does not run the JavaScript code, it cannot find your content and put it into its index.

But when your app returns rendered HTML upon the first request made to the server, it is easy for the search engine to find your content.

7.1 The Architecture of an SIRA

The following image depicts the infrastructure architecture of a Serverless Isomorphic React app.

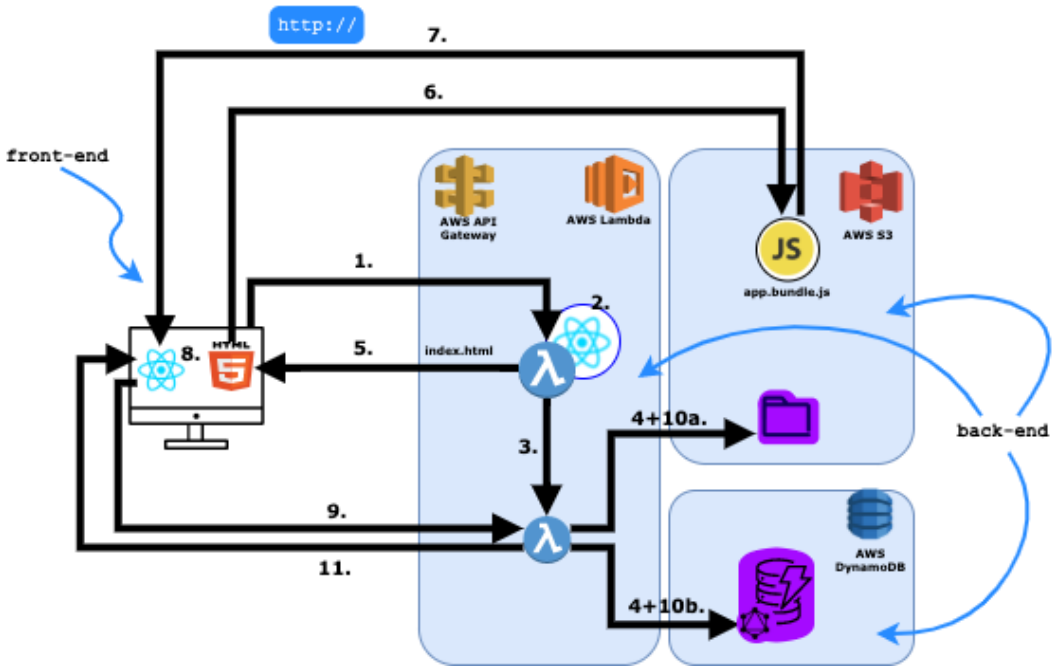


Figure 7.1: The architecture of a serverless isomorphic app

In comparison to a service-oriented app, an isomorphic app does not have any new infrastructure. There is the front-end, the services-layer including the API-Gateway and Lambda, and the data-layer comprising the S3 file storage and the DynamoDB database.

Both apps differ in how they use their resources. An isomorphic app creates the HTML dynamically in a Lambda function. Rather than loading a static HTML from S3. This allows processing the data sent in step 1 directly (step 2). React renders the HTML at the server-side. It can analyze the request. For instance, it can process the location or the authentication state of the user. At the server-side, the React app can use services (step 3) that may access the file storage (4a) and the database (4b).

The resulting HTML sent to the browser (step 5) can thus contain all the content and data. The first thing the user sees is not a blank page but the fully rendered app.

In step 6, the HTML requests the JavaScript code of the React app from S3 and receives it (step 7). In step 8, the browser runs the downloaded code and integrates it (called rehydration) into the downloaded HTML. From now on, the app behaves like any other SPA or SOA. You have the full interactivity on the client-side and can work with remote services (step 9) that access the file-storage (10a) and the database (10b) before they return the response (step 11).

There's a certain aspect important for Serverless Isomorphic React Apps. Since the first request is made to a Lambda function, it may involve a cold start and thus take longer. During the longer load time, your user would need to wait a little longer. This consumes the timing benefit you win by not downloading the whole app. But as we covered in the previous section, this applies to the first requests. Not on subsequent ones. A real server would omit these cold starts, but you would pay for a server twiddling its thumbs.

7.2 Create, Start, and Deploy An SIRA

Even though an isomorphic app differs from SPA and SOA, it is not difficult to use this kind of architecture. In fact, since it does not require any new infrastructure, the dependencies we have in the SOA-setting suffice for the isomorphic app, too.

Let's have a look at the code first.

Listing 7.1: The `src/index.tsx` of your app as an `<IsomorphicApp/>`

```
import React from 'react';
import "@babel/polyfill";
import {
  DataLayer,
  Environment,
  Route,
  IsomorphicApp,
  WebApp
} from "infrastructure-components";
import FileList from './file-list';
import UploadForm from './upload-form';
import Page from './page';
import FileStorage from './file-storage';
import FileMetaDataEntry from './file-meta-data-entry';

const folders = [ /* ... */ ];

export default (
  <IsomorphicApp
    stackName = "file-management"
    buildPath = 'build'
    assetsPath = 'assets'
    region='us-east-1'>

    <Environment name="dev" />
    <FileStorage />
    <DataLayer id="datalayer">
      <FileMetaDataEntry />
      <WebApp
        id="main"
        path="*"
        method="GET">
        {
          folders.map((folder, index)=> <Route
            key={`folder-${folder.name}`}
            path={folder.path}
            name={folder.name}
            render={(props) => <Page>
              <FileList/>
              <UploadForm/>
            </Page>}
          />)
        }
      </WebApp>
    </DataLayer>
  </IsomorphicApp>
);
```

In our code, there very few required changes. We need replace our top-level-component by `<IsomorphicApp/>` in `line 19`. This component takes an additional parameter `assetsPath`. This is the relative path to resources at runtime on your S3-storage.

Documentation — <IsomorphicApp/>.

- the `stackname` is the (arbitrary) name of your app. Please use only lower case characters and hyphens for the name serves as an identifier within AWS (when you deploy your app).
- the `buildPath` is the relative path to the folder where you want to put the build-resources. You may want to add this name to your `.gitignore` file to keep your repository free from compiled files.
- the `assetsPath` is the relative path to resources at runtime
- the `region` is the AWS-region you want your infrastructure to reside after deployment

The other major change is the introduction of the `<WebApp/>`-component.

The first thing to notice is that the `<WebApp/>` specifies a `path`. This field takes a regular expression (regex) specifying all the paths this `<WebApp/>` should serve. You can have multiple <WebApp/>-components in your isomorphic app. But make sure the paths are mutually exclusive.

The `<WebApp/>`-component needs to be a child of the `<DataLayer/>`-component in order to have access to the data-layer.

Documentation — <WebApp/>.

- the `id` is the (arbitrary) name of the web app. You can start a web-app locally in hot-dev mode: `npm run {id}`. Please use only lower case characters.
- the `path` specifies the relative paths that this app will serve, use "*" as a placeholder for “all”.
- the method is the HTTP-method that this app will work with, e.g. “GET”, “POST”, “PUT”, “DELETE”.

The last required change is to put the `<Route/>`-components into the `<WebApp/>` (lines 33-43).

Whenever we change the architecture of our app, we need to run the command `npm run build`. This applies here, too.

Since an `<IsomorphicApp/>` supports multiple `<WebApp/>`-components, our command for running an app in hot-dev-mode changed. You can run each `<webApp/>` locally in hot-dev-mode with the command `npm run main` (replace "main" with the `id` of your `<WebApp/>`-component).

The commands for starting the whole stack locally and for deploying it have not changed.

You can still start the whole software stack locally with `npm run start-{your-env}` (replace "dev" with the `name` of your `<Environment/>`).

And, you can still deploy your app with the command `npm run deploy-dev`. Pay attention to the URL you'll get when the deployment-script completes. It is different than the previous URL. The reason is that an isomorphic app points to an API-Gateway/Lambda-function and not to AWS S3.

7.3 Server-Side Rendering

The first thing you may see is while your app renders at the server-side is that it does not renders like the front-end does. The fontawsesome-icon seems to cover the whole screen. And the files do not appear in the list. But after a short time, everything jumps back to normal.

What you saw and what you still can see in the resource preview (within the developer tools of your browser. Click on Network, select the first entry (localhost) and click on preview). What you see is how the server rendered your app. The reason for everything switching back to normal after a short time is the effect when the browser downloaded and executed your app's code. This is what the client rendered.

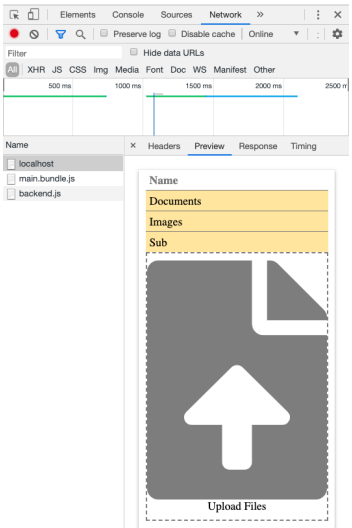


Figure 7.2: The app rendered at the server-side

Of course, we want our server to render everything correctly. We don't need server-side-rendering when it misses our data and the page does not look like the client renders it.

Don't worry. These things are easy to fix. Yet, they remind us to pay special attention to the fact that we're now rendering our app two times. At the server-side and at the client-side.

Font-awesome works with CSS-classes. For instance, the classes `svg-inline-fa`, `fa-w-16`, and `fa-lg` all have a critical role in controlling the size of the icon and the location within the DOM. But the server did not load the CSS-classes. We need to add two links to load the styles at the server-side.

Listing 7.2: Adding CSS to the `upload-form.tsx`

```
/* ... */
function UploadForm (props) {
  /* ... */
  return <Dropzone {/* ... */}>{
    !selectedFile ? <React.Fragment>
      <link rel="stylesheet"
       href="https://use.fontawesome.com/releases/v5.9.0/css/svg-with-js.css"
      ></link>
      <link rel="stylesheet"
       href="https://use.fontawesome.com/releases/v5.9.0/css/all.css"
      ></link>
      <FontAwesomeIcon icon={faFileUpload} size="4x" color="#888"/>
      <span >Upload Files</span>
    </React.Fragment> : <>
      {/* ... */}
    </>
  }</Dropzone>
};

export default withRefetch(withRouter(UploadForm));
```

The cause for the second thing we noticed not to work - the missing list of files - can be found in the `<StyledList/>`-component. We use the `useState`-hook there to keep the data of our list of files. We need the state in order to support drag and drop here. The problem is that React removes all states and data from the app when it renders it at the server-side. This includes the current value of the `useState`-hook.

But there's an easy solution, too. Infrastructure-components provide the `withIsomorphicState`-HOC. It adds the `useIsomorphicState`-function to the component properties.

Listing 7.3: Adding `withIsomorphicState` to the `<StlyedList/>-component`

```
1  /* ... */
2  import { /* ... */ withIsomorphicState } from 'infrastructure-components';
3
4  const StyledList = withIsomorphicState(withRouter((props) => {
5
6    const [fileList, setFiles] = props.useIsomorphicState("MYFILELIST", [ ]);
7    /* ... */
8
9    return <FileStorageList prefix={props./*location.*/pathname} >
10     { /* ... */}
11   </FileStorageList>;
12 }));
```

We replace the `useState` by the `props.useIsomorphicState` (line 6). The only difference is that `useIsomorphicState` takes two parameters instead of one. The first parameter is a string that needs to be unique across your app. The function uses this string to send the state from the server to the client. The second parameter is the initial state. This is the same as in `useState` If you restart your app, you can now see the app rendering all files and the content already at the server-side. Everything is included in the very first response of the server. The search engine bots can index all your content and the user can already see it while the browser loads the web app in the background.

Further, you can see the browser downloads the styles required of font-awesome.

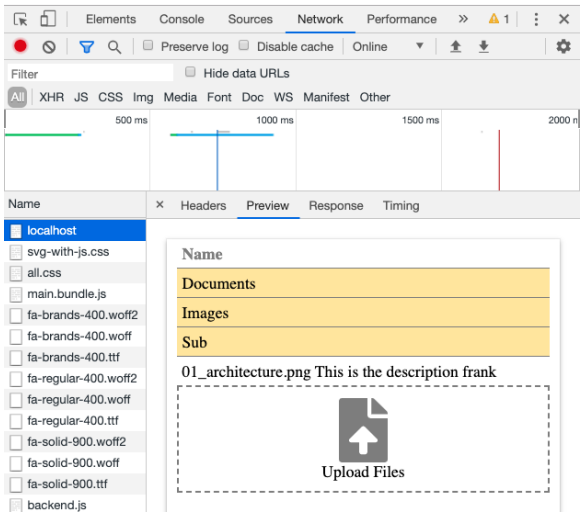


Figure 7.3: Fixed server-side rendering

7.4 Request Preprocessing

Until now, the folders are hard-coded in our `index.tsx`. We iterate through the list of folders and add respective `<Route/>`-components to our `<WebApp/>`. This is not very flexible. Because we need to know the folders before we create the `<WebApp/>`.

The major advantage of an isomorphic app is that it runs code before it sends the first response. It runs the React app's code. And it can even run code before the React app starts altogether. Let's use this capability of isomorphic apps to load the folders of our app dynamically.

Listing 7.4: The `src/index.tsx` with the dynamic route

```
import React from 'react';
import "@babel/polyfill";

import {
  DataLayer,
  Environment,
  IsomorphicApp,
  WebApp
} from "infrastructure-components";

import FileStorage from './file-storage';
import FileMetaDataEntry from './file-meta-data-entry';
import FolderRoute from './folder-page';

export default (
  <IsomorphicApp
    stackName = "file-management"
    buildPath = 'build'
    assetsPath = 'assets'
    region='us-east-1'>

    <Environment name="dev" />
    <FileStorage />
    <DataLayer id="datalayer">
      <FileMetaDataEntry />
      <WebApp
        id="main"
        path="*"
        method="GET">

        <FolderRoute/>

      </WebApp>
    </DataLayer>
  </IsomorphicApp>
);
```

We start with creating a more flexible `<FolderRoute/>`. We don't want to bloat our main component with the details. So we put all the code of the `<FolderRoute/>` into a separate file. This includes the `<Route/>`-component itself. So that we can simply import the `<FolderRoute/>` in our `index.tsx` (line13).

We use the `<FolderRoute/>` at line 31. By now, our `index.tsx` does not contain anything requiring much explanation anymore. But there's a thing worth mentioning. It's conciseness.

We can overview all the main aspects of our app. We can see that it is an usomorphic app with the name `file-management` and hosted in AWS region `us-east-1`. It has a single runtime environment (`dev`). It has a file storage. It has a data-layer with a single type of entries (`FileMetaDataEntry`). And the web app serves folders through a Route.

You don't need to look at configuration files in order to identify the components your app consists of. You see it at first sight. You have all the things at the top-level. If you want to see how things work in the detail, this file also gives you the clues where you may want to look at.

Let's look at our new `<FolderRoute/>`-component in the file `folder-page.tsx`.

Listing 7.5: The `src/index.tsx` of your app as a `<ServiceOrientedApp/>`

```
1  import React from 'react';
2  import { Middleware, Route } from 'infrastructure-components';
3
4  import FileList from './file-list';
5  import UploadForm from './upload-form';
6  import Page from './page';
7  import { withRouter } from 'react-router-dom';
8
9  const FolderPage = withRouter({location, ...props} => {
10   return <Page {...props}>
11     <FileList pathname={location.pathname} />
12     <UploadForm/>
13   </Page>;
14 });
15
16 export default function FolderRoute (props) {
17   return <Route
18     path="*"
19     name="Default"
20     component={FolderPage}
21   >
22     <Middleware callback={(req, res, next) => {
23       console.log("req: ", req.originalUrl);
24       next();
25     }}/>
26   </Route>
27 }
```

Our route does not serve an individual path, but we provide a placeholder for it: "*" (line 18). Thus, we let this route serve all URL-paths. We specify a component to be rendered (line 20). This `<FolderPage/>`-component renders our `<Page/>` with the `<FileList/>` and the `<UploadForm/>` (lines 9-14).

The new thing is that our `<Route/>` has a child-component now. A `<Middleware/>`. This `<Middleware/>` is the same component you can use in a `<Service/>` or the `<Storage/>`. It provides a callback function that takes the request, the prepared response, and the next-function as arguments.

Our middleware does not do very much. Except printing the `originalUrl` to the console (line 23). Let's have a look how that works out. This is the console output of the IDE. It contains the logs of the server.

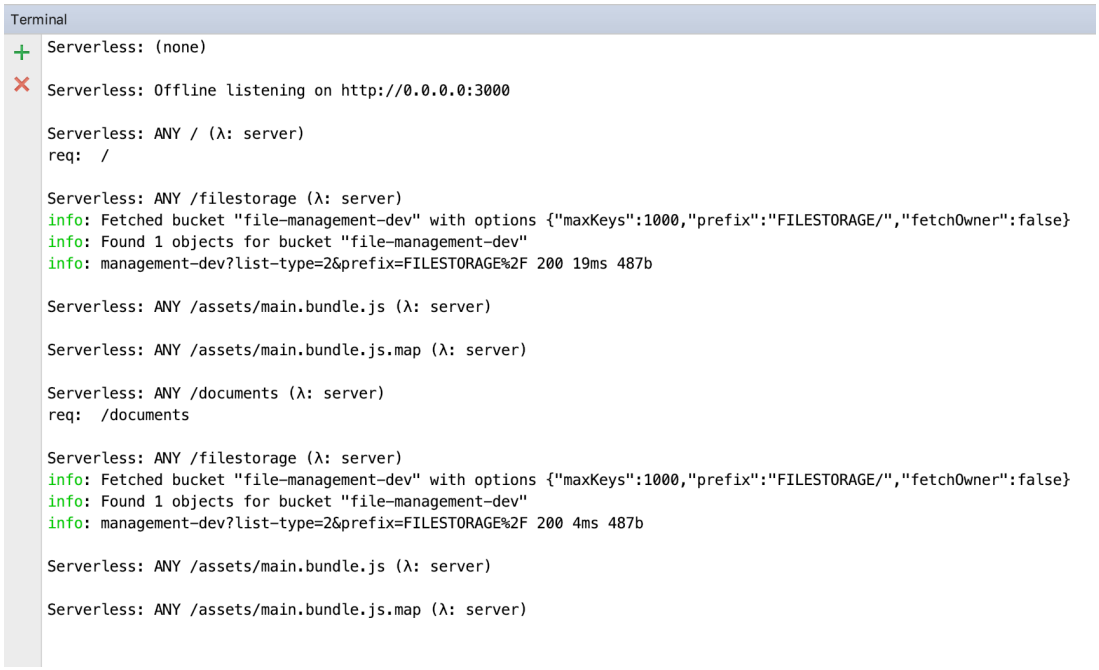


Figure 7.4: The output of the middleware running at the server-side

The logs show that I first opened: `localhost:3000`. The "req"-console.log shows "/". Afterwards, I opened `localhost:3000/Documents`. The "req"-console.log shows "/documents".

The `originalUrl` provides the relative path. This is the path to the folder we're interested in. This is the path we need to use when we render the `<FilesList/>` and its sub-folders. We come to that in a second. But first, we need to verify whether the path exists at all.

Listing 7.6: Showing only existing folders

```
1  import { Middleware, Route, LISTFILES_MODE,
2    serviceWithStorage } from 'infrastructure-components';
3  import { FILE_STORAGE_ID } from './file-storage';
4  /* ... */
5
6  export default function FolderRoute (props) {
7    return <Route {/* ... */}>
8      <Middleware callback={serviceWithStorage(
9        async (listFiles, req, res, next) => {
10         console.log("req: ", req.originalUrl);
11
12         res.locals.folders = await new Promise((resolve, reject) => {
13           listFiles (
14             FILE_STORAGE_ID, // storageId
15             req.originalUrl, // prefix
16             LISTFILES_MODE.FOLDERS, // listMode
17             {}, //data,
18             ({data, folders}) => {
19               console.log("my folders: ", folders);
20               resolve(folders);
21             },
22             (err: string) => reject(err),
23           );
24         });
25
26         next();
27       }
28     }/>
29   </Route>
30 }
```

We use the `listFiles`-function we add to the arguments through the `serviceWithStorage` wrapper (lines 8-9). We provide the `originalUrl` as the `prefix` to this function. If this prefix does not exist, we don't get any valid results here.

With our server-side rendering capability and some back-end code, we can process the browser requests dynamically.

7.5 Authentication

The ability to run server-side code before sending a response to a back-end request is the foundation of authentication. Because you need to be able to run code under your control. Code an adversary cannot manipulate or skip.

Further, you need your users to provide login credentials. Credentials identify the user.

They must ensure someone does not pretend to be a certain user if she is not.

Emails are great login credentials — much better than an arbitrary username. Users forget credentials but they rarely forget their email address. Whenever a user forgets her password, you can send her a new one. And you can verify an email address belongs to a certain user. Send a confirmation request to an entered email address. If she is able to confirm the request you make sure the user of your website has access to the email inbox.

Listing 7.7: The login page

```
import React, {useState} from 'react';
import { useHistory } from 'react-router-dom';
import { Route } from 'infrastructure-components';

const LoginPage = (props) => {
  const history = useHistory();
  const [credentials, setCredentials] = useState({email: "", password: ""});
  return <div>
    <input
      value={credentials.email}
      type="text"
      placeholder='Please enter your e-mail address'
      onChange={event => setCredentials({
        email: event.target.value,
        password: credentials.password
      })}/>
    <input
      value={credentials.password}
      type="password"
      placeholder='enter your password'
      onChange={event => setCredentials({
        email: credentials.email,
        password: event.target.value
      })}/>
    <button
      disabled={
        !(/^([A-Za-z0-9_\-.])+@([A-Za-z0-9_\-.])+\.([A-Za-z]{2,})$/.test(
          credentials.email))}
      onClick={()=>{
       if (credentials.password === "let-me-in")
        history.push("/");
      }}
    >Login</button>
  </div>
};

export default function (props) {
  return <Route path='/login' name='Login' render={() => <LoginPage />}/>
}
```

Let's add email authentication to our file management app. We only grant access to users who have registered with an email address.

We start with a login page. Our login page has two input fields and a button. One for the email address and one for the password. We control the values of these two fields with a React state hook.

Our login page is pretty basic. When the user clicks the button (lines 30-31), we check whether the entered email and password are correct. Further, we use a regular expression that verifies the format of the email (lines 27-28). If the format is not valid, we disable the button (line 26).

Once the email is valid and the user clicks the button, we check whether the password matches our secret string, "let-me-in" (line 30). If it does, we forward the user to our main page (`/`)! We use the `react-router-dom`'s `useHistory`-hook we initialize at line 6.

We export the corresponding `<Route/>` serving our login page (lines 37-39). Therefore, we can easily add the login page to our React app in the `index.tsx`-file by providing it as a child to the `<WebApp/>`-component.

It is important to specify the `<LoginRoute/>` before the `<FolderRoute/>` because the latter serves any path that could be a valid folder.

"You're kidding!"

Apparently, this is not very secure. Any email address works. Even `foo@fake.com`. The user can look at the code of our React app. They can easily find the string they need to enter.

This naive authentication mechanism uses the front-end, only. Most React apps don't need a back-end. But authentication is a good reason to implement one — you have to be able to trust the code verifying the login credentials.

Even worse. The user can call the folder-pages directly. Why would anybody use the login-page? Of course, we need to restrict access to secured routes.

Therefore, we need another process. Our process looks like this. The front-end sends the login (Identity) credentials in a request to the back-end. We will use cookies for that. Because the browser sends cookies automatically with each request. Thus, even if the user clicks the browser refresh button or types the URL of a page manually, the browser sends the credentials along.

The authentication runs as "trustworthy" code at the back-end (step 2). To verify the credentials, it uses a database. If the credentials are correct, the back-end sends the secured data to the front-end (step 3). If the authentication fails or if there are no credentials (for instance, the user is not logged in), then the authentication-layer redirects the user directly to our login-page (step 4).

Listing 7.8: The updated `index.tsx`

```
import React from 'react';
import "@babel/polyfill";

import {
  DataLayer,
  Environment,
  IsomorphicApp,
  WebApp
} from "infrastructure-components";

import FileStorage from './file-storage';
import FileMetaDataEntry from './file-meta-data-entry';
import FolderRoute from './folder-page';
import LoginRoute from './login-page';

export default (
  <IsomorphicApp
    stackName = "file-management"
    buildPath = 'build'
    assetsPath = 'assets'
    region='us-east-1'>

    <Environment name="dev" />
    <FileStorage />
    <DataLayer id="datalayer">
      <FileMetaDataEntry />
      <WebApp
        id="main"
        path="*"
        method="GET">

        <LoginRoute/>
        <FolderRoute/>
      </WebApp>
    </DataLayer>
  </IsomorphicApp>
);
```

When the user logs-in with new credentials (the registration), the authentication-layer sends an email to the specified address with a confirmation link. This link activates the user account and allows the user to access the secured pages.

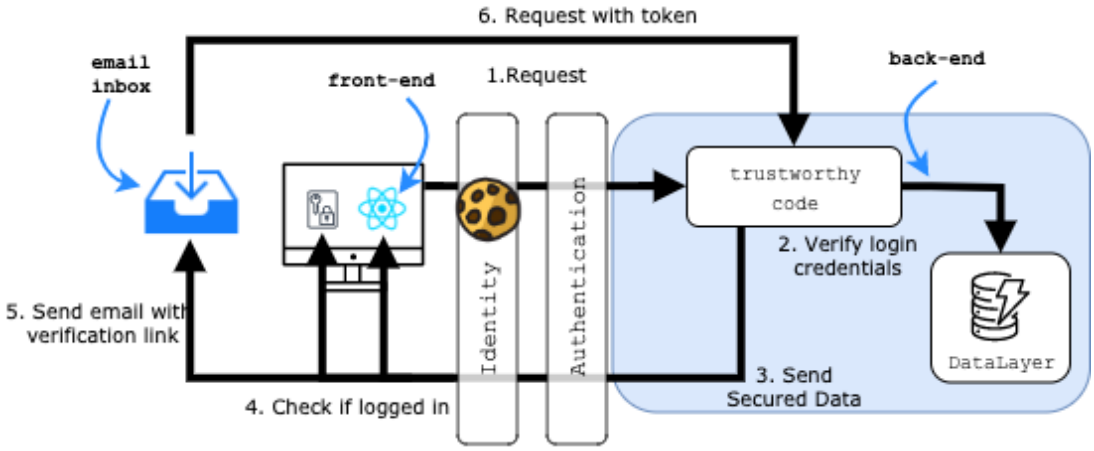


Figure 7.5: The authentication process

We start with securing our folder-route.

Listing 7.9: Securing the folder-page route

```
1  import { SecuredRoute } from 'infrastructure-components';
2  /* ...*/
3
4  export default function FolderRoute (props) {
5    return <SecuredRoute
6      path="*"
7      name="Default"
8      component={FolderPage}
9    >
10     <Middleware callback={/*...*/}/>
11   </SecuredRoute>
12 }
```

In the file `folder-page.tsx`, we use the `<SecuredRoute/>` we import from `infrastructure-components` (line 1).

The only difference is replacing the `<Route/>`-component with the `<SecuredRoute/>` (lines 5 and 11). The rest of the code remains unchanged

The `<SecuredRoute/>` works similar to the normal `<Route/>`. But it tells our app that this route has to be secured.

The `<SecuredRoute/>` needs to be a (grand-) child of an `<Authentication/>`-component. Because it is the `<Authentication/>`-component that ensures the user can access a `<SecuredRoute/>` only when she is logged-in. Otherwise, it redirects her to the `<LoginPage/>`.

We integrate the `<Authentication/>`-component in our `index.tsx`.

Listing 7.10: Adding the authentication-layer to the `index.tsx`

```
import React from 'react';
import "@babel/polyfill";

import {
  Authentication, AuthenticationProvider, DataLayer, Environment,
  Identity, IsomorphicApp, Middleware, Route, WebApp
} from "infrastructure-components";

import FileStorage from './file-storage';
import FileMetaDataEntry from './file-meta-data-entry';
import FolderRoute from './folder-page';
import LoginRoute from './login-page';
const SENDER_EMAIL = "mail@react-architect.com";

export default (
  <IsomorphicApp
    stackName = "file-management"
    buildPath = 'build'
    assetsPath = 'assets'
    region='us-east-1'
    iamRoleStatements={[{
      "Effect": "Allow",
      "Action": ['"ses:SendEmail"', '"ses:SendRawEmail"',],
      "Resource": `"arn:aws:ses:eu-west-1:xxxxxxxxxxx:identity/${
    SENDER_EMAIL}"`,
    }]}>

    <Environment name="dev" />
    <FileStorage />
    <DataLayer id="datalayer">
      <Identity >
        <Authentication
          id="emailauth"
          provider={AuthenticationProvider.EMAIL}
          loginUrl="/login"
          callbackUrl="/authentication"
          senderEmail={SENDER_EMAIL}
          getSubject={(recipient: string) => `Confirm Your Mail-Address`}
          getHtmlText={getHtmlText}>

          <FileMetaDataEntry />
          <WebApp id="main" path="*" method="GET">
            <LoginRoute/>
            <FolderRoute/>
          </WebApp>
        </Authentication>
      </Identity>
    </DataLayer>
  </IsomorphicApp>
);
```

First, we add an `<Identity/>`-component as a child to the `<DataLayer/>` (line 25). The `<Identity/>`-component stores an internal `id` of the user and stores it in a cookie as well as in the data-layer.

This internal `id` serves the `<Authentication/>`-layer to match the cookie data with a user and the corresponding credentials. Only if all these things fit together, we authenticate the user.

We add the `<Authentication/>`-component as a child of the `<Identity/>`-component.

Documentation — <Authentication/>.

- the `id` is the (arbitrary) name of the authentication.
- the `provider` specifies the kind of authentication we want to use. The object `AuthenticationProvider` provides the values of different providers.
- the `loginUrl` is the relative path to the login-page. This is the page an unauthorized user is redirected to.
- the `callbackUrl` is an internal relative path. This must be unused by all other routes.
- the `senderEmail` is the email address used to send the confirmation emails from. Make sure you verified this email in AWS.
- `getSubject=(recipient: string) => string` is a callback function. It receives the email address of the `recipient` as an argument. It must return a string used as the subject of the confirmation email.
- `getHtmlText=(recipient: string, url: string) => string` is a callback function. It receives the email address of the `recipient` and an URL as arguments. It must return a string used as the HTML-content of the confirmation email. When someone visits the `url`, the account of the user gets activated. Thus, the user and only the user should get access to this `url`. The best practice is to put the `url` into a link the user can click.

We configure our `<Authentication/>`-component to use email as the provider (line 33). We import the `AuthenticationProvider`-object at line 5.

At line 34, we specify the path to our login-page. At line 35, we specify an arbitrary internal URL ("/authentication"). This URL must not be used by any other route. But remember, our `<FolderRoute/>`-component serves all ("*") routes. This includes the "/authentication"-path.

We need to exclude this path from the `<FolderRoute/>`-component.

Listing 7.11: Exclude the "/authentication"-path in the `<FolderRoute/>`-component

```
export default function FolderRoute (props) {
  return <SecuredRoute
    path={/^((?!\/(?!(?:authentication|login|filestorage))([$|A-Za-z\-_]+)).)*$/}
    name="Default"
    component={FolderPage}
  >
    <Middleware/>
  </SecuredRoute>
```

The `path`-property of the `<Route/>` and the `<SecuredRoute/>`-component take a regular expression (regex). This is a pattern that includes or excludes certain strings.

The expression we specify for the `path` property accepts any string except for "/authentication", "/login", or "/filestorage". Thus, it excludes the paths used by other resouces.

Working with regular expressions is cumbersome if you're not an expert at it. Use a tool to verify the expression works as expected before you integrate it into your app. Tools, such as this regex tester can be of great help.

Next, we need to do is to connect our login-page with the authentication-layer.

Listing 7.12: Connecting the login-page with the authentication-layer

```
/* ... */
import { Route, withAuthCallback } from 'infrastructure-components';

const LoginPage = withAuthCallback(({authCallback, ...props}) => {
  /* ... */
  return <div>
    <input value={credentials.email} {/*...*/} />
    <input value={credentials.password} {/*...*/} />
    <button
      disabled={ !(
        /^([A-Za-z0-9_\-.])+@([A-Za-z0-9_\-.])+\.([A-Za-z]{2,})$/
        .test(credentials.email))
      }
      onClick={()=> authCallback(
        credentials.email,
        credentials.password,
        "/",
        err => console.log("error: ", err)
      )}
    >Login</button>
  </div>
});
```

In our `login-page.tsx`, we add the `authCallback`-function to the properties of the `<LoginPage/>`-component (line 4). We use the higher-order-component `withAuthCallback` that we import from `infrastructure-components` (line 2). This function works only if the component is a (grand-) child of the `<Authentication/>`-component (which it is).

In the button, we replace our hard-coded password verification with the `authCallback`-function (line 14). We provide the entered `email`, the `password`, and the `path` to our folder page ("/").

Let's take a look at our app. When we try to open our folder-page ("/"), the app recognizes that we're not logged in and redirects us to the login page. We enter our credentials. Since we're not yet registered, the app sends us a confirmation email with a verification link.

When we open the link, the app verifies the token and redirects us to our folder-page ("/"). We are now logged-in. When we click the logout-button, the app deletes the cookie and we return to our login page. From now on, we can log in directly with our credentials.

When you run your app offline, it won't send an email. The simple reason is the required email server does not work locally. But in this case, you can see the link in the console of your IDE.

```
Serverless: ANY /authentication (λ: server)
redirect to:  /?message=mailed&email=fzickert@googlemail.com
OFFLINE! Link to activate this user:  /authentication?confirmationtoken=603f84a5-47b3-4880-bfd7-12eda6b0f63c&email=fzickert@googlemail.com
```

Figure 7.6: The authentication token

Open the specified path (`http://localhost:3000/authentication?confirma...`) in the browser to activate your offline account. You should be redirected to `http://localhost:3000/login?message=verified`

Listing 7.13: Adding the logout button

```
1  import React from 'react';
2  import { userLogout } from 'infrastructure-components';
3  /* ... */
4
5  const FolderPage = withRouter(withIsomorphicState(withRequest(({request,
6    useIsomorphicState, location, ...props}) => {
7
8    return <Page {...props}>
9      <FileList pathname={location.pathname} />
10     <UploadForm/>
11     <button onClick={()=> userLogout("/login")}>Logout</button>
12   </Page>;
13 })));
```

Of course, we need to let the user log out again. For the `<FolderRoute/>` is the only

secured page, we integrate the button directly there. If you have multiple secured pages, you may want to add the logout button at a central place, such as the menu.

In the file `folder-page.tsx`, we already specify the `<FolderPage/>`-component. We add a simple button (line 11). In its `onClick`-callback, we call the `userLogout`-function we import from `infrastructure-components`. The argument we pass to this function is the page the user should be forwarded to. Since the user logs out, she should not remain at a secured page.

7.6 Sending emails

Emails may no longer be your favorite tool when you chat with your friends or colleagues. Emails do not only work well for authenticating your users. But emails also remain important to get in touch with your users.

- Emails enable you to send push messages to your users. Even at times when the user is not online. You have created something new? Let your users know!
- Emails persist. Your user can decide on keeping the information you've sent. She does not have to keep a browser tab open.
- You can verify email addresses. Send a confirmation request to an entered email address. Your user needs access to the inbox to confirm it!
- Emails are private. You decide whom to send an email to. You decide on the content of the email.

Most of these emails, you don't want to send manually. It is too much effort to send the same email over and over again. And sometimes you don't want your user to wait for you.

In the previous section, we already used email for authentication. In this section, we want to send the list of files in a folder when the user requests it.

Sending emails requires a back-end. Let's start with that.

We define a back-end by a `<Service/>`-component (line 8). This `<Service/>` requires an `id` that is unique across our app. We store the value in a constant (line 5) because we refer to later.

The <Service/> further listens to the local path "email" and the "POST" HTTP-method (line 8). We use a POST-service because we want the to provide the payload in the request body.

The `<Service/>` takes a `<Middleware/>`-component as a child (line 9). This has a `callback`-property. The callback is a function with two parameters: `req` (the browser request) and `res` (the prepared server response).

Listing 7.14: The email-sending service

```
import React from 'react';
import { LISTFILESMODE, Middleware, Service,
  serviceWithStorage} from 'infrastructure-components';
import {SENDER_EMAIL} from './index';
const EMAIL_SERVICE_ID = "emailservice";
export default function EmailService (props) {
  return <Service id={EMAIL_SERVICE_ID} path="/email" method="POST">
    <Middleware callback={serviceWithStorage(async (listF, req, res, next)=>{
      const {emailAddress, folder} = JSON.parse(req.body);
      const files = await new Promise((resolve, reject) => {
      listF (
          FILE_STORAGE_ID, //storageId: string,
          folder, //prefix: string,
          LISTFILES_MODE.FILES, //listMode: string,
          {}, //data: any,
          ({data, files}) => { resolve(files.map(item => item.file)) },
          (err: string) => { reject(err) },
        );
      });
      await new Promise(function (resolve, reject) {
        const AWS = require('aws-sdk');
        new AWS.SES({apiVersion: '2010-12-01'}).sendEmail({
          Destination: {
            BccAddresses: [], CcAddresses: [], ToAddresses: [emailAddress]},
          Message: {
            Body: {
              Html: { Charset: "UTF-8",
                Data:`Hey ${emailAddress}! Here are the files: ${files}`},
              Text: { Charset: "UTF-8",
                Data:`Hey ${emailAddress}! Here are the files: ${files}`
            },
            Subject: { Charset: 'UTF-8', Data: `Hello ${emailAddress}!`}
          },
          Source: SENDER_EMAIL, ReplyToAddresses: [SENDER_EMAIL],
        }).promise().then(data => {
          res.status(200)
            .set({"Access-Control-Allow-Origin": "*",}).send("ok");
          resolve();
        }).catch(err => {
          console.error(err, err.stack);
          res.status(500)
            .set({"Access-Control-Allow-Origin": "*",}).send("failed");
          reject(err);
        });
      });
    }}/>
  </Service>
}
```

We start by parsing the request body and obtaining the `emailAddress` and the current `folder` from it if they exist (line 9).

We use the current `folder` to get the list of files inside this folder. We use the `serviceWithStorage`-function. We wrap our callback into it (line 8) and get the listFiles (for brevity here: `listF`) as the first argument. This works similar to the `<FolderPage/>` where we load the folders. But this time, we set the mode to `LISTFILESMODE.FILES` to get the files (line 14). We set the current `folder` as the prefix and resolve the promise with the retrieved names of the `files` (line 16).

Sending emails may take some time. Therefore, we wrap it into a `Promise` (line 20) and await until it resolves. AWS provides access to its Simple Email Service (SES) through the AWS SDK we import at line 13.

At line 22, we configure the `AWS.SES` object with its current `apiVersion` and call its `sendMail`-function. This function takes all the data required of sending an email, like `Destination`-addresses, the `Message` with its `Body` and `Subject`, and the `Source`. The `Source` is our own email address we want to send the email from. We used the email address in our authentication-layer (in the `index.tsx`). Thus, we can import it from there (line 4).

We transform the `AWS.SES().sendEmail()`-result into a Promise (line 35). This lets us attach a success-handler (`then()`, lines 35–38) and an error-handler (`catch()`, lines 39–43).

Depending on the success of sending the email, we send the status 200 ("ok") or 500 ("failed") as a response to the browser.

Further, let's create a convenience function to help us calling the service. Infrastructure-components provide the `callService`-function.

Listing 7.15: The email-sending convenience-function

```
1  import { callService } from 'infrastructure-components';
2
3  export async function sendEmail (emailAddress: string, folder: string) {
4
5    return callService(EMAIL_SERVICE_ID, {
6      emailAddress: emailAddress,
7      folder: folder
8    }, (response) => {}, (error) => {
9      console.log("error: ", error)
10   });
11 }
```

We add the function `sendEmail`. It returns the result of the `callService`-function. Since this is a `Promise`, we also declare the `sendEmail`-function as `async`. We put the

`emailAddress` and the `folder` we receive as parameters into an object we provide as the second parameter. We send this object as the body to our back-end service. We provide the data at corresponding object keys. These keys are the values we expect to exist in the request body within our service.

We do the integration of the service in our `index.tsx`. In the following snippet, we skip most of the details.

Listing 7.16: Integration of the email-sending service

```
1  /* ... */
2  import FileListService from './file-list-service';
3
4  export const SENDER_EMAIL = "mail@react-architect.com";
5
6  export default (
7    <IsomorphicApp
8      iamRoleStatements={[{
9        "Effect": "Allow",
10       "Action": ['"ses:SendEmail"', '"ses:SendRawEmail"',],
11       "Resource":
12       `"arn:aws:ses:eu-west-1:xxxxxxxxxxx:identity/${SENDER_EMAIL}"`,
13     }]}>
14
15     <Environment name="dev" />
16     <FileStorage />
17     <DataLayer id="datalayer">
18       <Identity >
19         <Authentication>
20           <FileMetaDataEntry />
21           <FileListService/>
22
23           <WebApp>
24             <LoginRoute/>
25             <FolderRoute/>
26           </WebApp>
27         </Authentication>
28       </Identity>
29     </DataLayer>
30   </IsomorphicApp>
31 );
```

We import our service at line 2. We add it as a child of the `<Authentication/>`-component. Further note the `iamRoleStatements`-property we add to the `<IsomorphicApp/>`-component. If you did not add it for the authentication part, it is now time to do so. Sending emails requires the corresponding access rights.

In the next step, we add a button the `<FolderPage/>` to trigger the send-email-service. We add this next to the logout-button in the `<FolderPage/>`

Listing 7.17: Adding the "send content"-button

```
1  import React from 'react';
2  import { withUser } from 'infrastructure-components';
3  import { sendEmail } from './file-list-service';
4  /* ... */
5
6  const FolderPage = withUser(withRouter(withIsomorphicState(withRequest((
7    {userId, request, useIsomorphicState, location, ...props}) => {
8
9    return <Page {...props}>
10     <FileList pathname={location.pathname} />
11     <UploadForm/>
12     <button onClick={()=> userLogout("/login")}>Logout</button>
13     <button onClick={()=> sendEmail(userId, location.pathname) }>Send list</
       button>
14   </Page>;
15 })))));
```

The integration of the `button` (line 13) is simple. We call our convenience function `sendEmail` we import from the respective file (line 3). We need to provide the user's email address and the current folder.

We can get the email address through the `withUser` function that we wrap around the `<FolderPage/>`-component (line 6). It adds the `userId` to the properties. Since we use email addresses as a credential, this `userId` is the email address. The current folder is the `location.pathname` we already use to provide to the `<FileList/>`-component.

Unfortunately, sending emails does not work in the offline mode. In order to see your app, you need to deploy it. Run the command `npm run deploy-dev` (replace "dev" with the name of the `<Environment/>` you specified.

7.7 Evaluation

In this chapter, we had a closer look at isomorphic apps. They significantly differ from single-page and service-oriented apps. Because they render the app at the back-end and at the front-end side.

We have tapped the main characteristics of a Serverless Isomorphic React App in this chapter. We have covered the following topics:

- Server-Side Rendering
- Request Preprocessing
- Authentication
- Sending emails

With the ability to render your components at the back-end, you get a whole new dimension

of full-stack development. Because the borders of front-end and back-end start to blur.

In this book, we aimed to shed some light on the similarities and differences. You are now equipped to build and deploy full-stack React apps.

This is the end of this book. But it is just the start of full-stack development. And of course, a single book cannot cover all topics included in full-stack development.

In this book, we used libraries. Many libraries. The use of the `infrastructure-components`-library supported us in the configuration of our infrastructure. From a configuration perspective, it does not even matter if you use an SSPRA (`<SinglePageApp/>`), an SSORA (`<ServiceOrientedApp/>`), or an SIRA (`<IsomorphicApp/>`). That's why we spend quite some time to cover the implications when you use either one of the architectures.

About the Authors

Frank Zickert, Ph.D.

Frank Zickert has been working as an IT professional for 16 years. He studied Information Systems Development and earned his PhD in 2012 at Goethe University of Frankfurt.

Frank specializes in the technical conception and creation of software and system architectures. He works with Node.js, TypeScript, and React.

Dr. Derek Austin (Guest Author)

Derek Austin is a full-stack web developer and JavaScript software engineer. He writes on Medium.com and is the editor of Coding at Dawn.

Derek contributed two blog-post-sized sections.