As we know, React is a JavaScript-based library that does not have the ability to make HTTP requests; thus, we need to use third-party libraries to achieve this.

There are plenty of libraries available to make HTTP calls into React apps. A few of them are listed below.

* Axios
* Fetch
* Superagent
* React-axios
* Use-http
* React-request

We will use the Axios HTTP library to make HTTP Get REST API call in this example tutorial.

**YouTube Video**

**5. Prerequisites**

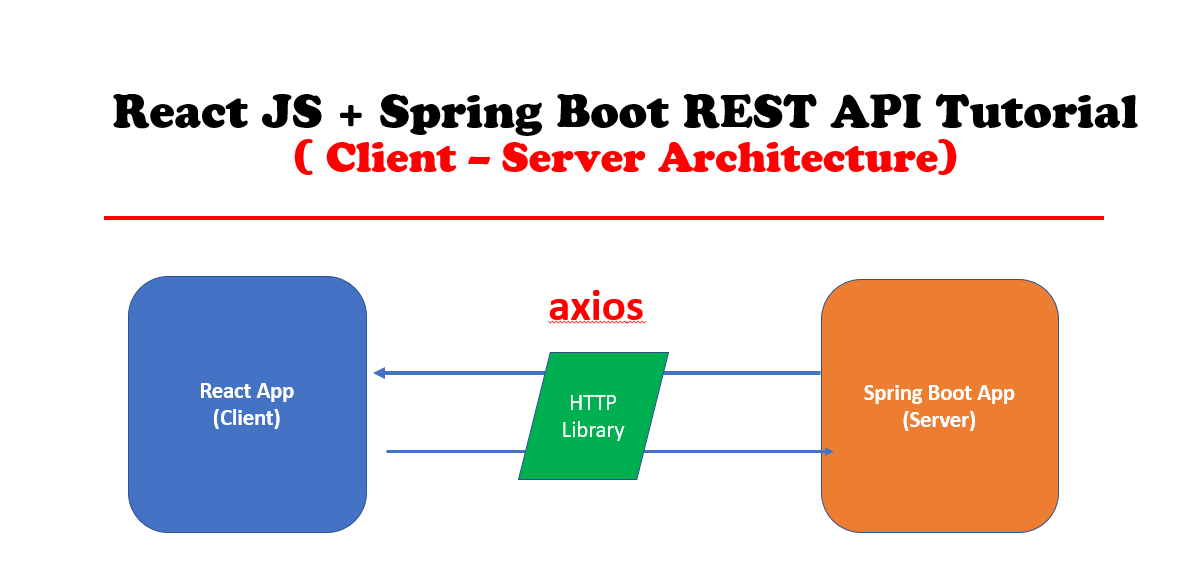
* Basic familiarity with HTML & CSS
* Basic knowledge of JavaScript and programming
* Spring Boot Basics
* ReactJS basics
* Node.js and npm installed globally

**What we will build?**

We will build two projects:

1. sprintboot-backend (server) – To develop REST API
2. react-frontend (client) – Consume REST API

**Client-Server Architecture**

**[](https://1.bp.blogspot.com/-oOrve_ySuU0/XwB-Yqy6BaI/AAAAAAAAH94/NhHO7m7gxXIGWyQTjzvMovKcA8kG78QqgCLcBGAsYHQ/s1600/reactjs-springboot-rest-api.png)**

**1. Develop Spring Boot Backend Application**

We will use [**Spring Data JPA**](https://www.javaguides.net/p/spring-data-jpa-tutorial.html) to develop the repository layer and we use the H2 in-memory database to store the data.

**1. Create a Spring Boot Application**

There are many ways to create a Spring Boot application. You can refer to below articles to create a Spring Boot application.

>> [**Create Spring Boot Project With Spring Initializer**](https://www.javaguides.net/2019/04/create-spring-boot-project-with-spring-initializer.html)  
>> [**Create Spring Boot Project in Spring Tool Suite [STS]**](https://www.javaguides.net/2019/04/create-spring-boot-project-in-spring-tool-suite.html)

**2. Add maven dependencies**

<?xml version="1.0" encoding="UTF-8"?>

<project

xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.0.RELEASE</version>

<relativePath/>

<!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>springboot-backend</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springboot-backend</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**Create JPA Entity - User.java**

Create a new package called *model* inside *net.javaguides.springboot* package and then create the *User* class inside the *model* package with the following contents -

package net.javaguides.springboot.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private long id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

private String email;

public User() {

}

public User(String firstName, String lastName, String email) {

super();

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

**Create Spring Data JPA Repository - UserRepository.java**

Create a new package called *repository* inside *net.javaguides.springboot* package and then create the following interface inside the *repository* package -

package net.javaguides.springboot.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import net.javaguides.springboot.model.User;

@Repository

public interface UserRepository extends JpaRepository<User, Long>{

}

**Spring Controller with REST API - /api/users**

Let's create a *UserController* class and add the following code to it:

package net.javaguides.springboot.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import net.javaguides.springboot.model.User;

import net.javaguides.springboot.repository.UserRepository;

@CrossOrigin(origins = "http://localhost:3000")

@RestController

@RequestMapping("api/")

public class UserController {

@Autowired

private UserRepository userRepository;

@GetMapping("users")

public List < User > getUsers() {

return this.userRepository.findAll();

}

}

Note that we have added the below line of code to avoid CORS issues:

@CrossOrigin(origins = "http://localhost:3000")

**Run Spring Boot Application and Test Rest API**

Let's insert a few records in the *users* table while application startup.

Let's run this spring boot application from IDE -> Right-click -> Run As -> Java Application:

package net.javaguides.springboot;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import net.javaguides.springboot.model.User;

import net.javaguides.springboot.repository.UserRepository;

@SpringBootApplication

public class SpringbootBackendApplication implements CommandLineRunner {

public static void main(String[] args) {

SpringApplication.run(SpringbootBackendApplication.class, args);

}

@Autowired

private UserRepository userRepository;

@Override

public void run(String...args) throws Exception {

this.userRepository.save(new User("Ramesh", "Fadatare", "ramesh@gmail.com"));

this.userRepository.save(new User("Tom", "Cruise", "tom@gmail.com"));

this.userRepository.save(new User("Tony", "Stark", "tony@gmail.com"));

}

}

Hit this "[**http://localhost:8080/api/users**](http://localhost:8080/api/users)" link in a browser will popular list of users as JSON:

[{"id":1,"firstName":"Ramesh","lastName":"Fadatare","email":"ramesh@gmail.com"},{"id":2,"firstName":"Tom","lastName":"Cruise","email":"tom@gmail.com"},{"id":3,"firstName":"Tony","lastName":"Stark","email":"tony@gmail.com"}]

**Build React JS Frontend Application**

Let's go ahead and create a React application to consume */api/users* REST API.

**1 - Create a React UI with Create React App**

The**Create React App** CLI tool is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration.

To create a new app, you may choose one of the following methods:

**Using npx**

npx create-react-app react-frontend

**Using npm**

npm init react-app react-frontend

npm init is available in npm 6+

**Using Yarn**

yarn create react-app react-frontend

Running any of these commands will create a directory called *react-frontend* inside the current folder. Inside that directory, it will generate the initial project structure and install the transitive dependencies:

react-frontend

├── README.md

├── node\_modules

├── package.json

├── .gitignore

├── public

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

└── src

├── App.css

├── App.js

├── App.test.js

├── index.css

├── index.js

├── logo.svg

└── serviceWorker.js

Let's explore important files and folders of the react project.

For the project to build, these files must exist with exact filenames:

* public/index.html is the page template;
* src/index.js is the JavaScript entry point.

You can delete or rename the other files Let's quickly explore the project structure.

**package.json** - The package.json file contains all the required dependencies for our React JS project. Most importantly, you can check the current version of React that you are using. It has all the scripts to start, build, and eject our React app.

**public folder** - The public folder contains index.html. As react is used to build a single page application, we have this single HTML file to render all our components. Basically, it's an HTML template. It has a div element with id as root and all our components are rendered in this div with index.html as a single page for the complete react app.

**src folder**- In this folder, we have all the global javascript and CSS files. All the different components that we will be building, sit here.

**index.js** - This is the top renderer of your react app.

**node\_modules** - All the packages installed by NPM or Yarn will reside inside the node\_modules folder.

**App.js** - The App.js file contains the definition of our App component which actually gets rendered in the browser and this is the root component.

**2 - Adding Bootstrap in React Using NPM**

Open a new terminal window, navigate to your project's folder, and run the following command:

$ npm install bootstrap --save

After installing the bootstrap package, you will need to import it in your React app entry file.

Open the src/index.js file and add the following code:

import 'bootstrap/dist/css/bootstrap.min.css';

**src/index.js**

Here is the complete code for the index.js file:

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import \* as serviceWorker from './serviceWorker';

import 'bootstrap/dist/css/bootstrap.min.css';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

**3. React Service Component - REST API Call**

For our API calls, we will be using Axios. Below is the npm command to install Axios.

npm add axios

Below is the *UserService.js* service implementation to make our HTTP REST call via Axios.

Our backend User endpoint is available at [**http://localhost:8080/api/users**](http://localhost:8080/api/users).

import axios from 'axios'

const USERS\_REST\_API\_URL = 'http://localhost:8080/api/users';

class UserService {

getUsers(){

return axios.get(USERS\_REST\_API\_URL);

}

}

export default new UserService();

Make sure that you create an object of *UserService* class export it as:

export default new UserService();

**4. Develop a React Component**

Components are the building blocks of our whole react app. They are like functions that accept inputs in terms of props, state, and outputs a UI that is rendered in the browser. They are reusable and composable.

React components can be either a function component or a class component. In this example, we are going to use the class component.

Let's create a *UserComponent.js* file and add the following code to it.

import React from 'react';

import UserService from '../services/UserService';

class UserComponent extends React.Component {

constructor(props){

super(props)

this.state = {

users:[]

}

}

componentDidMount(){

UserService.getUsers().then((response) => {

this.setState({ users: response.data})

});

}

render (){

return (

<div>

<h1 className = "text-center"> Users List</h1>

<table className = "table table-striped">

<thead>

<tr>

<td> User Id</td>

<td> User First Name</td>

<td> User Last Name</td>

<td> User Email Id</td>

</tr>

</thead>

<tbody>

{

this.state.users.map(

user =>

<tr key = {user.id}>

<td> {user.id}</td>

<td> {user.firstName}</td>

<td> {user.lastName}</td>

<td> {user.email}</td>

</tr>

)

}

</tbody>

</table>

</div>

)

}

}

export default UserComponent

Let's understand the above code step by step.

* *constructor()* - The *constructor ()* is invoked before the component is mounted. In the constructor, we have declared our state variables and bind the different methods so that they are accessible from the state inside of the render() method.
* *componentDidMount()* - The *componentDidMount()* is called as soon as the component is mounted and ready.
* *render()* - The *render()* method is the most used lifecycle method. The *render()* method actually outputs HTML to the DOM.

We are using the **map operator** to loop over our user list and create the view like:

{

this.state.users.map(

user =>

<tr key = {user.id}>

<td> {user.id}</td>

<td> {user.firstName}</td>

<td> {user.lastName}</td>

<td> {user.email}</td>

</tr>

)

}

**5. App.js**

In the previous step, we have created *UserComponent* so let's go ahead add *UserComponent* to the *App* component:

import React from 'react';

import logo from './logo.svg';

import './App.css';

import UserComponent from './components/UserComponent';

function App() {

return (

<div className="App">

<UserComponent />

</div>

);

}

export default App;

**6. Run React App**

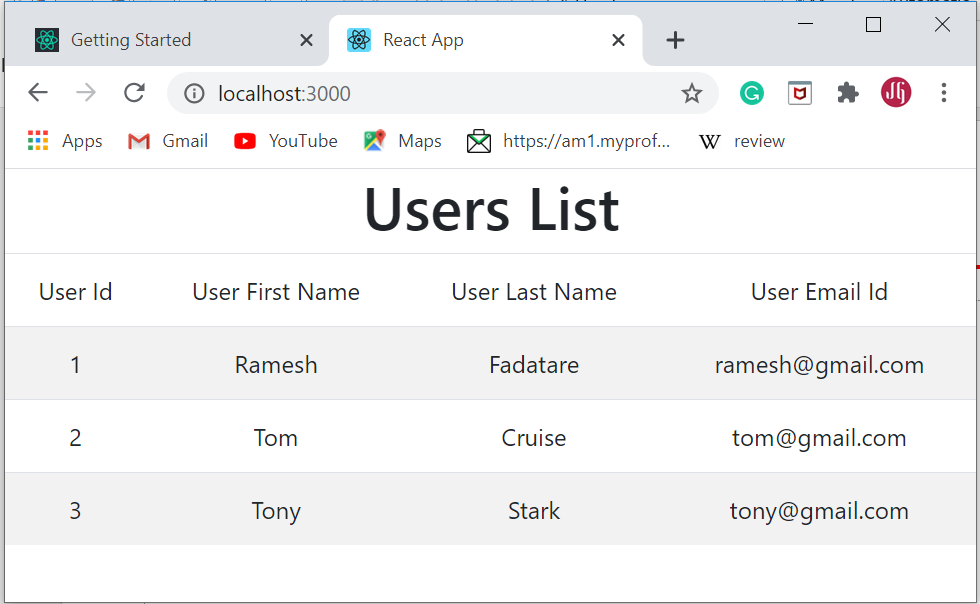
Use the below command to start the project:

npm start

Use yarn to start the project:

yarn start

Runs the app in development mode. Open [**http://localhost:3000**](http://localhost:3000/) to view it in the browser.

**[](https://1.bp.blogspot.com/-KtOlfdlC_7M/XwCAL2eSnyI/AAAAAAAAH-E/Kn0JuN7PFdkb_9N-9gNuZakX9ES2IYtLACLcBGAsYHQ/s1600/Screenshot%2B07-04-2020%2B14.40.08.png)**

**Conclusion**

In this tutorial, we have created a simple "single page application" using React as frontend and [**spring boot**](https://www.javaguides.net/p/spring-boot-tutorial.html)as backend. We have also seen how to integrate React frontend application with spring boot backend using Axios HTTP library.