

Networking

Many mobile apps need to load resources from a remote URL. You may want to make a POST request to a REST API, or you may need to fetch a chunk of static content from another server.

Using Fetch

React Native provides the [Fetch API](#) for your networking needs. Fetch will seem familiar if you have used XMLHttpRequest or other networking APIs before. You may refer to MDN's guide on [Using Fetch](#) for additional information.

Making requests

In order to fetch content from an arbitrary URL, you can pass the URL to fetch:

```
fetch('https://mywebsite.com/mydata.json');
```

Fetch also takes an optional second argument that allows you to customize the HTTP request. You may want to specify additional headers, or make a POST request:

```
fetch('https://mywebsite.com/endpoint/', {  
  method: 'POST',  
  headers: {  
    Accept: 'application/json',  
    'Content-Type': 'application/json',  
  },  
  body: JSON.stringify({  
    firstParam: 'yourValue',  
    secondParam: 'yourOtherValue',  
  }),  
});
```

Take a look at the [Fetch Request docs](#) for a full list of properties.

Handling the response

The above examples show how you can make a request. In many cases, you will want to do something with the response.

Networking is an inherently asynchronous operation. Fetch method will return a Promise that makes it straightforward to write code that works in an asynchronous manner:

```
const getMoviesFromApi = () => {
  return fetch('https://reactnative.dev/movies.json')
    .then(response => response.json())
    .then(json => {
      return json.movies;
    })
    .catch(error => {
      console.error(error);
    });
};
```

You can also use the `async / await` syntax in a React Native app:

```
const getMoviesFromApiAsync = async () => {
  try {
    const response = await fetch(
      'https://reactnative.dev/movies.json',
    );
    const json = await response.json();
    return json.movies;
  } catch (error) {
    console.error(error);
  }
};
```

Don't forget to catch any errors that may be thrown by `fetch`, otherwise they will be dropped silently.

TypeScript

JavaScript

Fetch Example



```
import React, {useEffect, useState} from 'react';
import {ActivityIndicator, FlatList, Text, View} from 'react-native';

type Movie = {
  id: string;
  title: string;
  releaseYear: string;
};

const App = () => {
  const [isLoading, setLoading] = useState(true);
  const [data, setData] = useState<Movie[]>([]);

  const getMovies = async () => {
    try {
      const response = await
fetch('https://reactnative.dev/movies.json');
      const json = await response.json();
      setData(json.movies);
    } catch (error) {
      console.error(error);
    } finally {
      setLoading(false);
    }
  };
};
```

Preview



My Device

iOS

Android

Web

By default, iOS 9.0 or later enforce App Transport Security (ATS). ATS requires any HTTP connection to use HTTPS. If you need to fetch from a cleartext URL (one that begins with `http`) you will first need to add an [ATS exception](#). If you know ahead of time what domains you will need access to, it is more secure to add exceptions only for those domains; if the domains are not known until runtime you can [disable ATS completely](#). Note however that from January 2017, Apple's App Store review will require [reasonable justification for disabling ATS](#). See [Apple's documentation](#) for more information.

On Android, as of API Level 28, clear text traffic is also blocked by default. This behaviour can be overridden by setting `android:usesCleartextTraffic` in the app manifest file.

Using Other Networking Libraries

The [XMLHttpRequest](#) API is built into React Native. This means that you can use third party libraries such as [frisbee](#) or [axios](#) that depend on it, or you can use the [XMLHttpRequest](#) API directly if you prefer.

```
const request = new XMLHttpRequest();
request.onreadystatechange = e => {
  if (request.readyState !== 4) {
    return;
  }

  if (request.status === 200) {
    console.log('success', request.responseText);
  } else {
    console.warn('error');
  }
};

request.open('GET', 'https://mywebsite.com/endpoint/');
request.send();
```

The security model for [XMLHttpRequest](#) is different than on web as there is no concept of [CORS](#) in native apps.

WebSocket Support

React Native also supports [WebSockets](#), a protocol which provides full-duplex communication channels over a single TCP connection.

```
const ws = new WebSocket('ws://host.com/path');

ws.onopen = () => {
  // connection opened
  ws.send('something'); // send a message
};

ws.onmessage = e => {
```

```
// a message was received
console.log(e.data);
};

ws.onerror = e => {
  // an error occurred
  console.log(e.message);
};

ws.onclose = e => {
  // connection closed
  console.log(e.code, e.reason);
};
```

Known Issues with fetch and cookie based authentication

The following options are currently not working with fetch

- `redirect:manual`
- `credentials:omit`
- Having same name headers on Android will result in only the latest one being present. A temporary solution can be found here: <https://github.com/facebook/react-native/issues/18837#issuecomment-398779994>.
- Cookie based authentication is currently unstable. You can view some of the issues raised here: <https://github.com/facebook/react-native/issues/23185>
- As a minimum on iOS, when redirected through a 302, if a Set-Cookie header is present, the cookie is not set properly. Since the redirect cannot be handled manually this might cause a scenario where infinite requests occur if the redirect is the result of an expired session.

Configuring NSURLSession on iOS

For some applications it may be appropriate to provide a custom `NSURLSessionConfiguration` for the underlying `NSURLSession` that is used for network

requests in a React Native application running on iOS. For instance, one may need to set a custom user agent string for all network requests coming from the app or supply `NSURLSession` with an ephemeral `NSURLSessionConfiguration`. The function `RCTSetCustomNSURLSessionConfigurationProvider` allows for such customization. Remember to add the following import to the file in which `RCTSetCustomNSURLSessionConfigurationProvider` will be called:

```
#import <React/RCTHTTPRequestHandler.h>
```


`RCTSetCustomNSURLSessionConfigurationProvider` should be called early in the application life cycle such that it is readily available when needed by React, for instance:

```
-(void)application:(__unused UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {

    // set RCTSetCustomNSURLSessionConfigurationProvider
    RCTSetCustomNSURLSessionConfigurationProvider(^NSURLSessionConfiguration *{
        NSURLSessionConfiguration *configuration = [NSURLSessionConfiguration
defaultSessionConfiguration];
        // configure the session
        return configuration;
    });

    // set up React
    _bridge = [[RCTBridge alloc] initWithDelegate:self launchOptions:launchOptions];
}
```

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**