Learn How to Combine Existing React Hooks into New Combined Hooks

Peter Kellner
DEVELOPER, CONSULTANT AND AUTHOR

@pkellner   linkedin.com/in/peterkellner99   ReactAtScale.com

Hooks bring state and lifecycle to React Functional Components.

Hooks easy to use without complex usage patterns.

Complex problems leads to high number of hooks being used.

Demonstration of a routing page for our app.

Three Most Common Hooks

- useState
- useRef
- useEffect

Four More Hooks

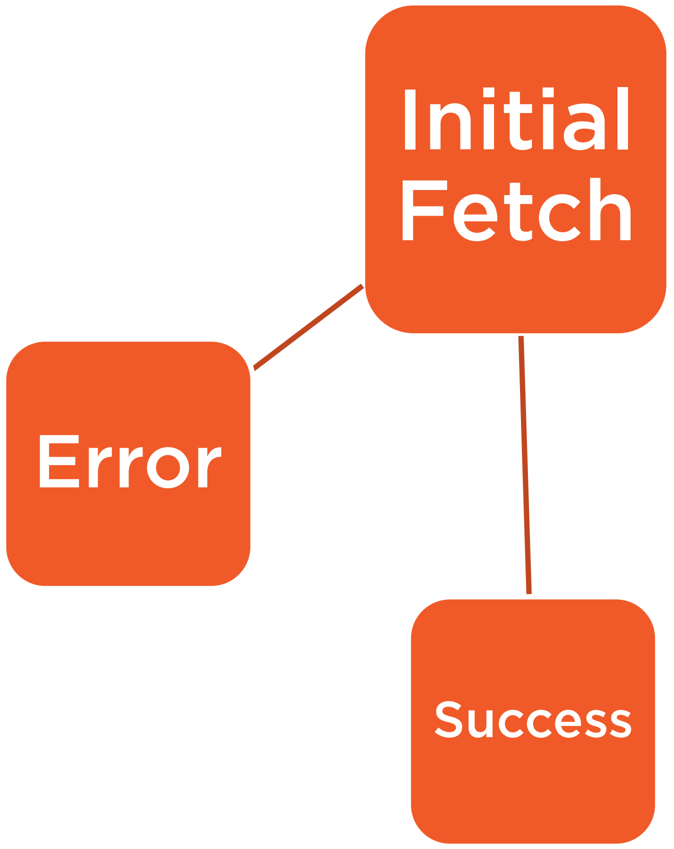
- useContext
- useReducer
- useCallback
- useMemo

REST

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services

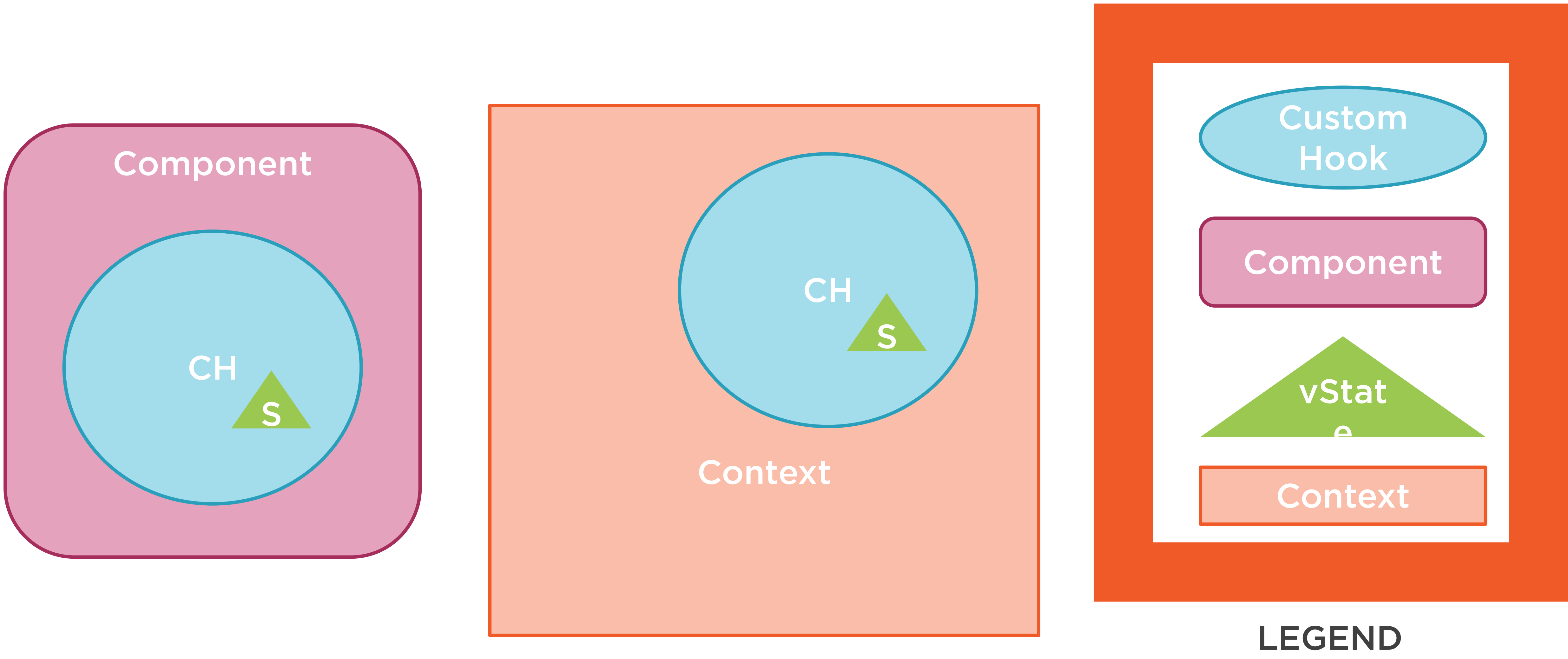
HTTP VERB	URL Endpoint
GET	/api/speakers
PUT	/api/speakers/$ID

State Transitions for REST GET

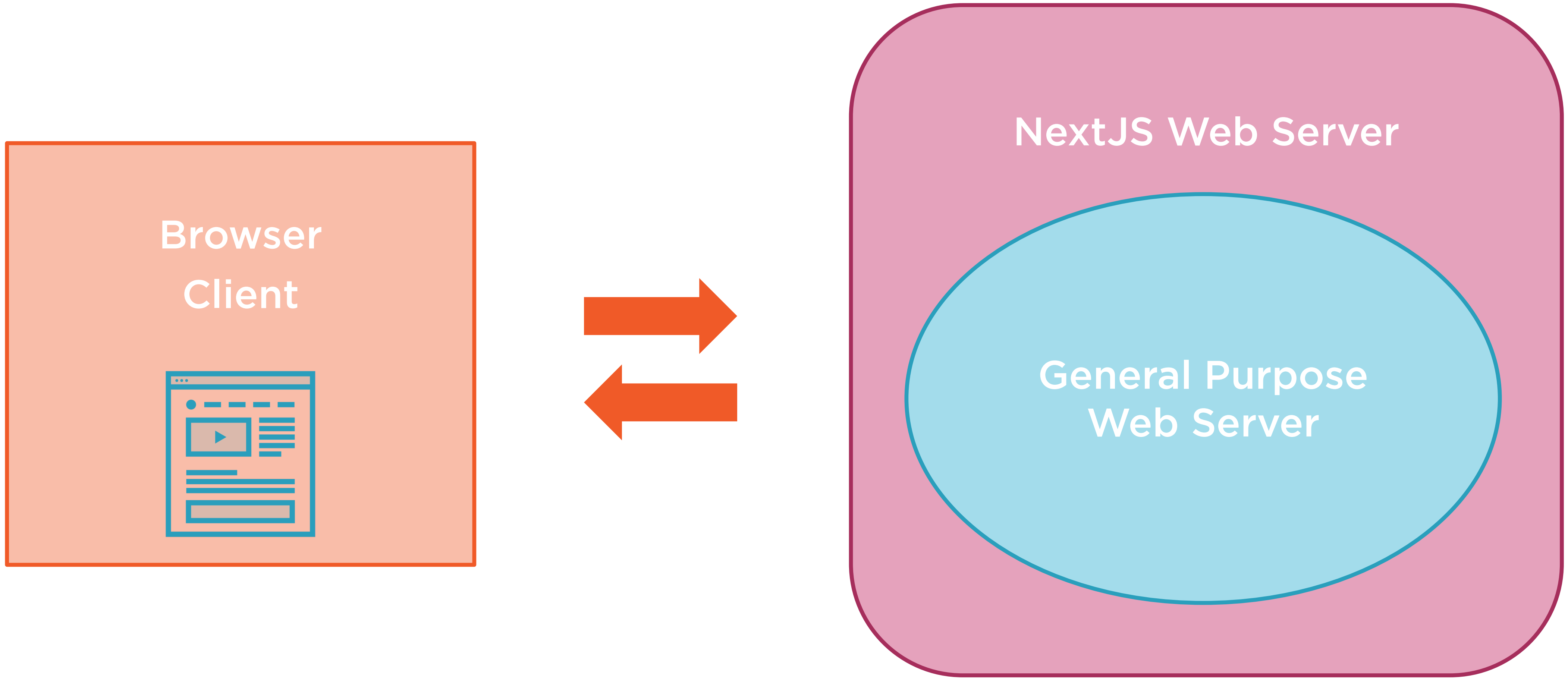


isLoading	hasErrored	errorMessage	data
TRUE	FALSE	“”	[]
FALSE	TRUE	ERROR	[]
FALSE	FALSE	“”	[..,..,..,..]

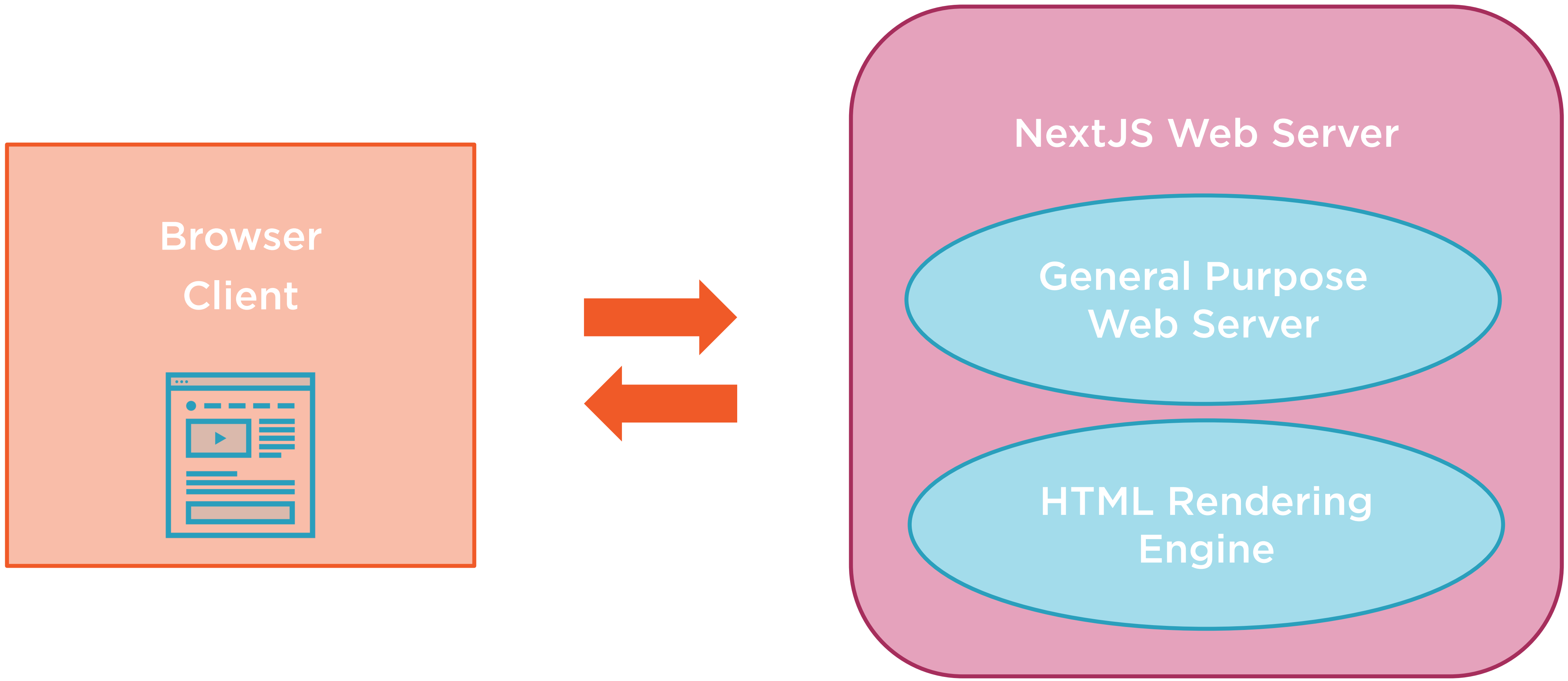
Components With Only Custom Hooks



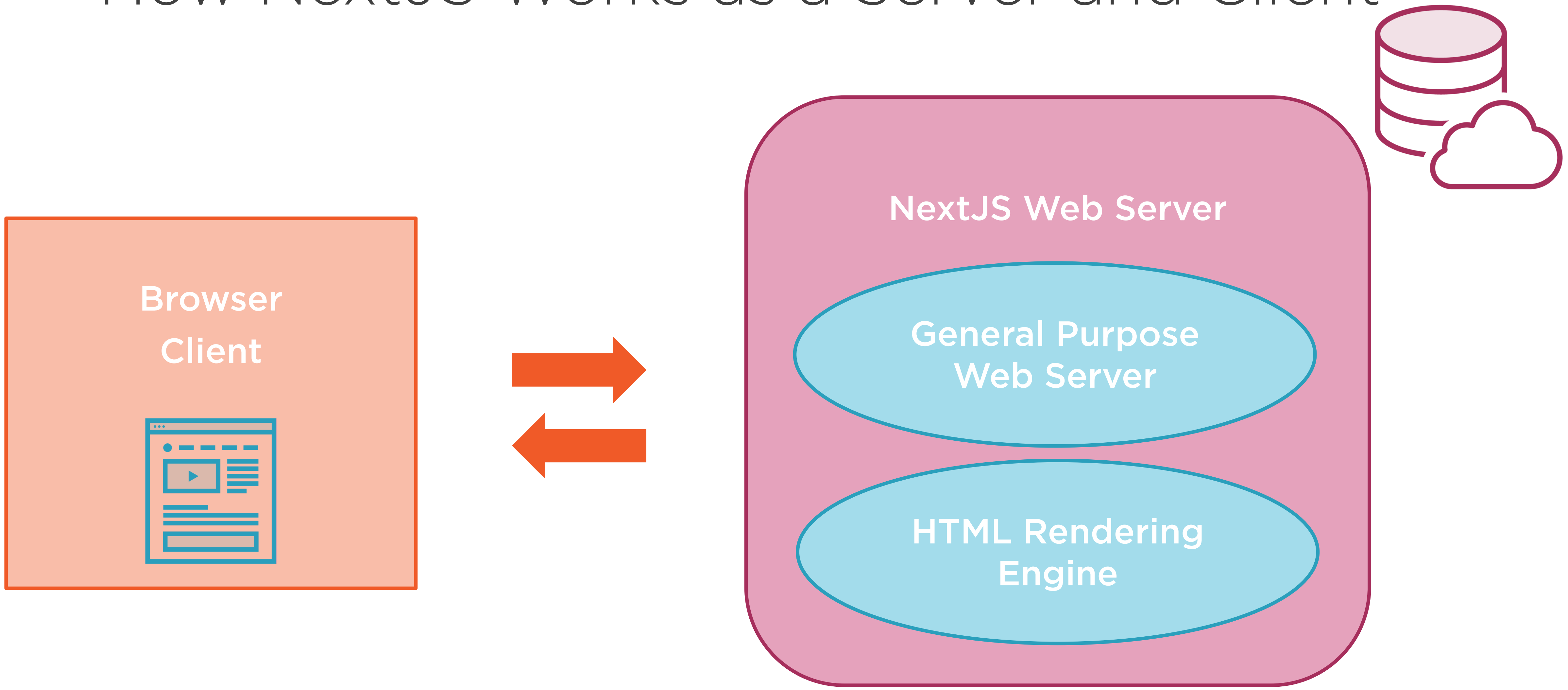
How NextJS Works as a Server and Client



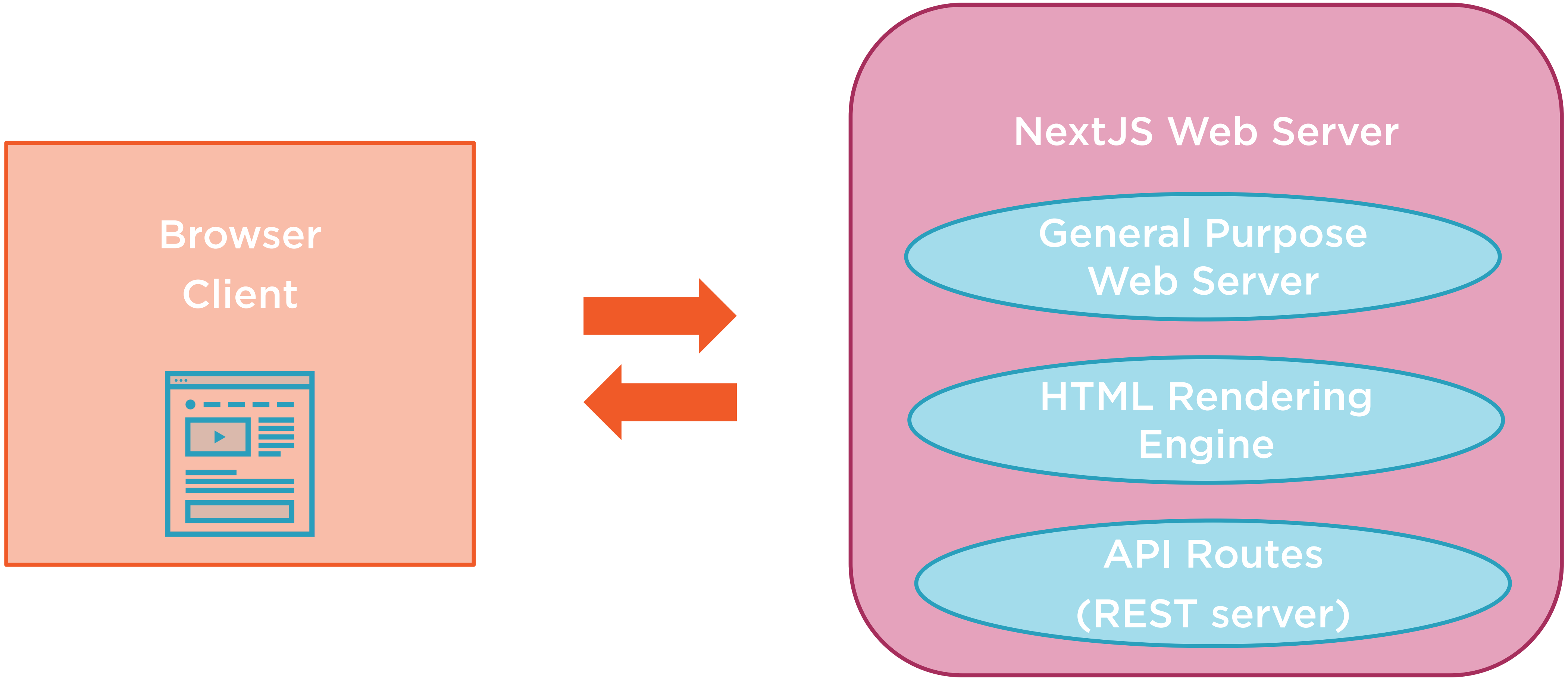
How NextJS Works as a Server and Client



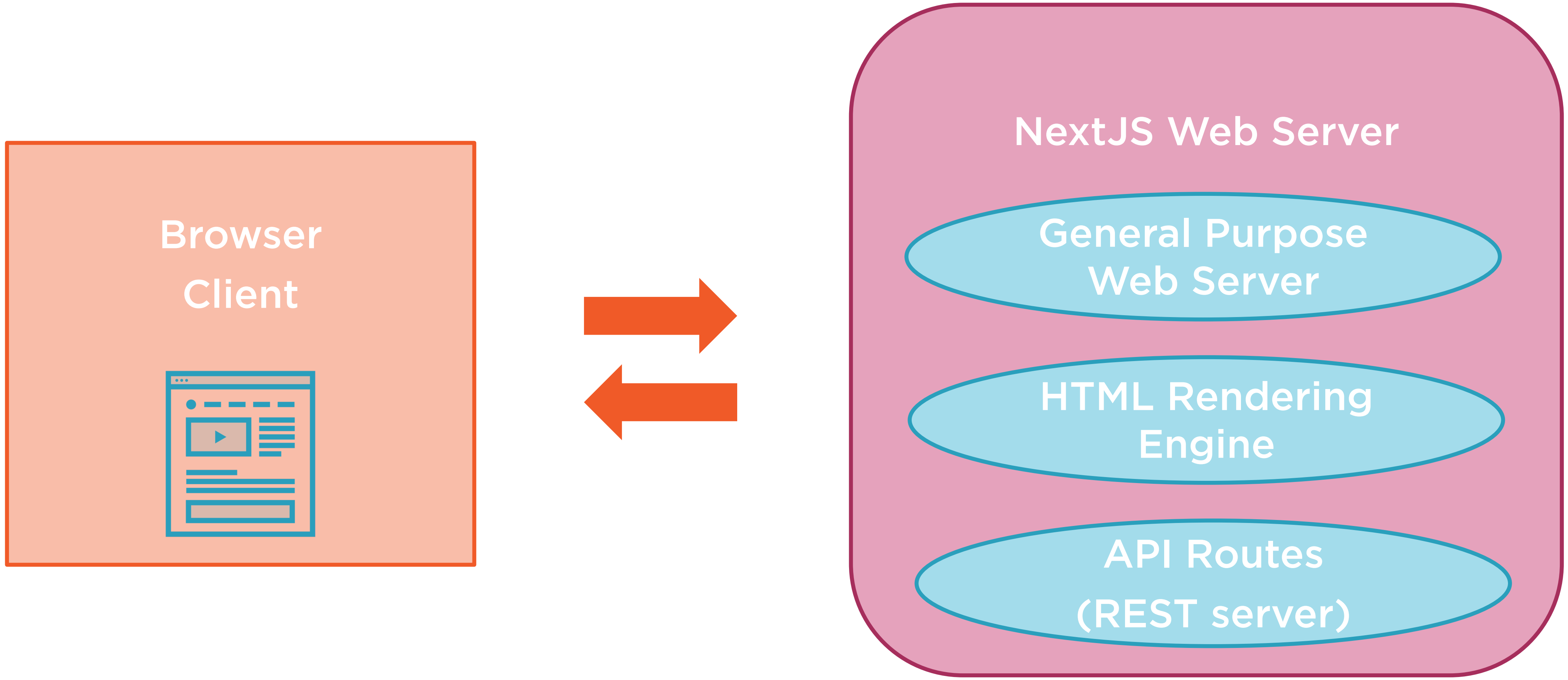
How NextJS Works as a Server and Client



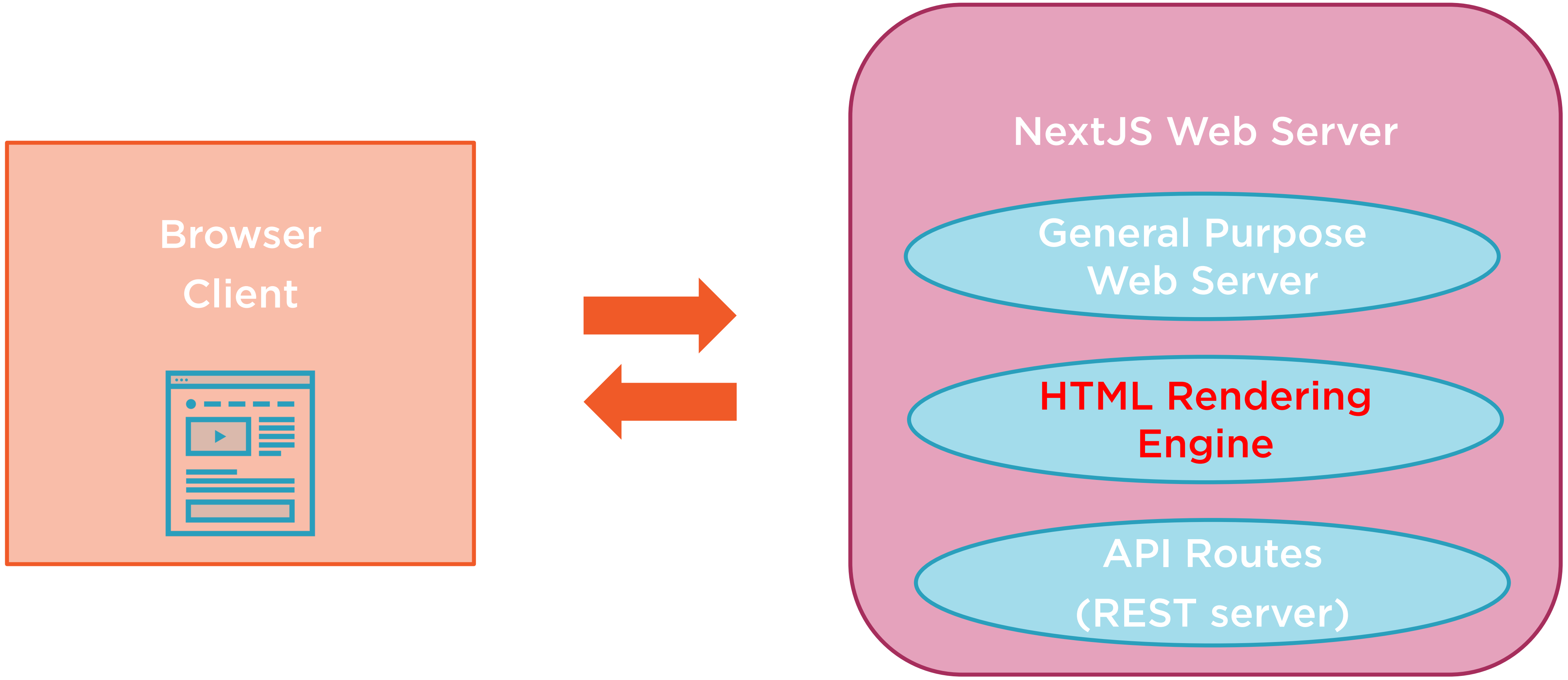
How NextJS Works as a Server and Client



How NextJS Works as a Server and Client

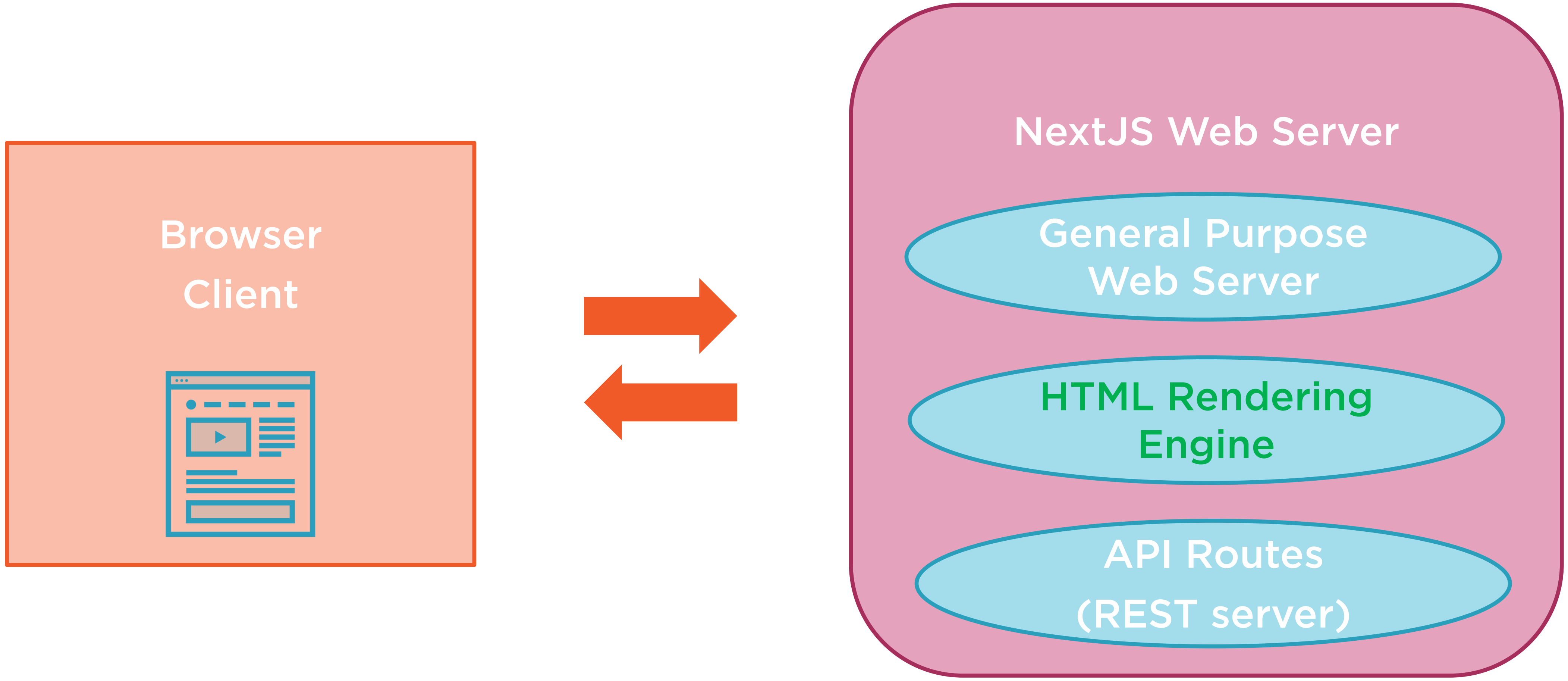


How NextJS Works as a Server and Client



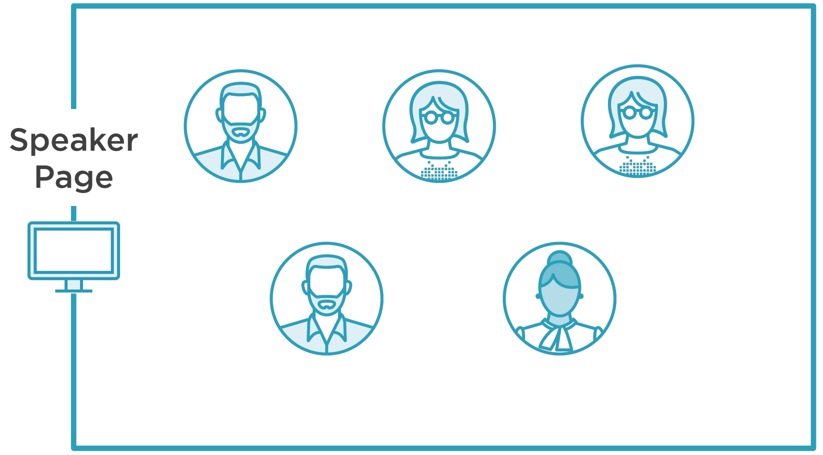
CPU Usage: 100%

How NextJS Works as a Server and Client



CPU Usage: 2%

Regeneration of Static Site for Changes



Regeneration of Static Site for Changes

Speaker
Page

Add new speaker to REST Service

Triggers NextJS site regeneration

New site available with new speaker

Regeneration of Static Site for Changes

Speaker
Page

Add new speaker to REST Service

Triggers NextJS site regeneration

New site available with new speaker

Regeneration of Static Site for Changes

Speaker
Page

Add new speaker to REST Service

Triggers NextJS site regeneration

New site available with new speaker

Demo

Build out a REST web service in our NextJS server that Implements GET and PUT only.

Server-side Rending Within Easy Reach

All data access isolated in a custom React hook

Node server can run JavaScript that is part of our NextJS app

userSpeakerDataManager Custom React Hook

```
import speakersReducer from './speakersReducer';
import axios from 'axios';
import { useEffect, useReducer } from 'react';

function useSpeakerDataManager() {
  const [{ isLoading, speakerList }, dispatch] =
    useReducer(speakersReducer, {
      isLoading: true,
      speakerList: [],
  });
```

useEffect Hook Skipped on First Page Render

```
      …
      useEffect(() => {
      const fetchData = async function () {
        let result = await axios.get('/api/speakers’);
        dispatch({ type: 'setSpeakerList', data: result.data });
      };
      fetchData();
      return () => {
        console.log('cleanup’);
      };
    }, []);
    return { isLoading, speakerList, toggleSpeakerFavorite };
  }

  export default useSpeakerDataManager;
```

100% of viewable HTML downloaded over one HTTP request.

Demo

Make changes to just our /pages/speakers.js, and /src/useSpeakerDataManager.js files to enable Server-side rendering in our React app

Takeaways

Consolidate loading and rendering data into React hooks

Extend custom React hooks to include updating

Integrated a REST service for reading and updating external data

Leveraged our learnings to update our app for near infinite scalability through server-side rendering and static site generation