Student Guide: Deploying React Apps to Azure App Service

TL;DR - What You Need to Know

If you built a React app for class and now need to deploy it to Azure, you probably need to add an Express server first. Here's why and how.

Important: This guide is about deploying your **frontend React app**. Your **backend Azure Function App** is deployed separately and doesn't need Express.

Understanding Your Project Structure

Your project has two main parts:

```
AI_EventPlanner/
├── frontend/           # React app (needs Express for Azure deployment)
│   ├── src/            # Your React components, pages, etc.
│   ├── public/         # Static assets
│   └── package.json    # React dependencies
└── backend/            # Azure Function App (separate deployment)
    ├── EventFunction/  # Your API endpoints
    ├── host.json       # Function App configuration
    └── package.json    # Function dependencies
```

Two Different Servers, Two Different Purposes:

1. **Express Server (Frontend)** - *This is what you need to add*
    - Serves your React app's HTML, CSS, and JavaScript files
    - Handles React Router navigation
    - Required for Azure App Service deployment
2. **Azure Function App (Backend)** - *You already have this*
    - Provides API endpoints for your React app
    - Handles data processing, database operations, etc.

- Deployed separately to Azure Functions (not App Service)

Why Can't I Just Deploy My React App Directly?

Your React app (created with `create-react-app`) is designed for development. It creates static files (HTML, CSS, JavaScript) but doesn't include a web server for production. Azure App Service needs a server to handle web requests.

Note: This is different from your backend Azure Function App, which already has its own server infrastructure provided by Azure Functions.

What Happens Without Express for Your Frontend?

- Direct URLs don't work (like `yourapp.com/about`)
- Page refreshes return 404 errors
- React Router navigation breaks
- Can't serve static files properly on Azure

What Express Gives Your Frontend:

- Proper URL routing for React Router
- Web server that Azure App Service can run
- Serves your React build files correctly
- Handles all frontend routing needs

Your Backend Function App Already Handles:

- HTTP requests (GET and POST to `/api/EventFunction`)
- Basic request processing and logging
- JSON response formatting
- Development and production deployment setup

Current Function Behavior:
Your `EventFunction` currently accepts a `name` parameter and returns a personalized greeting. This is a foundational setup that can be extended to handle event data, database operations, AI processing, or any other server-side logic you need.

Quick Setup (5 Minutes)

Make sure you're in the frontend directory for all these steps!

1. Navigate to Frontend and Add Express

```
cd frontend
npm install express
```

2. Create server.js

Create a new file called `server.js` in your **frontend** folder (not the root, not in backend):

```
const express = require('express');
const path = require('path');
const app = express();
const port = process.env.PORT || 8080;

// Serve your React app's built files
app.use(express.static(path.join(__dirname, 'build')));

// Handle React Router - send all requests to React
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'build', 'index.html'));
});

app.listen(port, () => {
  console.log(`Frontend server running on port ${port}`);
});
```

3. Update package.json

In your **frontend/package.json**, change the scripts section:

Before:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  ...
}
```

After:

```
"scripts": {
  "start": "node server.js",
  "dev": "react-scripts start",
  "build": "react-scripts build",
  ...
}
```

4. Test It Works

```
# Make sure you're still in the frontend directory
npm run build
npm start
```

Visit `http://localhost:8080` - your React app should work!

Your backend Function App continues to run separately and doesn't need these changes.

Understanding the Development vs Production Flow

During Development (What You've Been Doing)

```
npm run dev  # Starts React development server on port 3000
```

- Uses `react-scripts start`
- Automatic reloading
- Development optimizations
- Only for your computer

For Production/Azure (What You Need Now)

```
npm run build  # Creates optimized files in 'build' folder
npm start      # Runs Express server on port 8080
```

- Uses `node server.js`
- Serves optimized, built files
- Production-ready
- What Azure will run

Common Student Questions

"Why didn't we do this in class?"

Because during development, the React development server (`npm run dev`) works perfectly for building and testing your app. You only need Express when deploying to a production environment like Azure App Service. Your Azure Function App backend was already production-ready.

"Will this break my development workflow?"

No! You can still use:

- `npm run dev` for development (React dev server on port 3000)
- `npm start` for production testing (Express server on port 8080)
- Your backend Function App runs independently on its own port

"Do I need to change my React code?"

No! Your React components, styles, and functionality stay exactly the same. You're just adding a server to serve your existing app. Your API calls to the backend Function App don't change either.

Example: Here's what a typical App.js looks like in this project:

```
import React, { useState } from 'react';

function App() {
  const [form, setForm] = useState({
    name: '',
    date: '',
    category: '',
    description: ''
  });
  const [submitted, setSubmitted] = useState(false);

  const categories = [
    'Conference',
    'Workshop',
    'Meetup',
    'Webinar',
    'Party',
    'Sports',
```

```
  'Other'
];

const handleChange = (e) => {
  setForm({ ...form, [e.target.name]: e.target.value });
};

const handleSubmit = (e) => {
  e.preventDefault();
  setSubmitted(true);
  // Here you would send the form data to your backend Function App API
};

return (
  <div style={{ maxWidth: 500, margin: '2rem auto', padding: '2rem', bo
    <h1>AI Event Planner</h1>
    <p>Welcome to your event planning app!</p>
    <h2>Schedule an Event</h2>
    <form onSubmit={handleSubmit}>
      <div style={{ marginBottom: 12 }}>
        <label>Event Name:<br />
          <input type="text" name="name" value={form.name} onChange={ha
        </label>
      </div>
      <div style={{ marginBottom: 12 }}>
        <label>Date & Time:<br />
          <input type="datetime-local" name="date" value={form.date} or
        </label>
      </div>
      <div style={{ marginBottom: 12 }}>
        <label>Category:<br />
          <select name="category" value={form.category} onChange={handl
            <option value="" disabled>Select category</option>
            {categories.map(cat => <option key={cat} value={cat}>{cat}<
          </select>
        </label>
      </div>
      <div style={{ marginBottom: 12 }}>
        <label>Description:<br />
          <textarea name="description" value={form.description} onChang
        </label>
      </div>
      <button type="submit" style={{ padding: '0.5rem 1.5rem', backgrou
    </form>
```

```
      {submitted && (
        <div style={{ marginTop: 20, color: 'green' }}>
          <strong>Event scheduled!</strong>
          <div>Name: {form.name}</div>
          <div>Date: {form.date}</div>
          <div>Category: {form.category}</div>
          <div>Description: {form.description}</div>
        </div>
      )}
    </div>
  );
}

export default App;
```

This React code works perfectly as-is! Adding Express doesn't change any of this - Express just serves this React app to users.

Connecting to Your Backend Function App:

Your backend has an Azure Function called `EventFunction` that currently accepts a name parameter. When you're ready to connect this React app to your backend, you'd modify the `handleSubmit` function like this:

```
const handleSubmit = async (e) => {
  e.preventDefault();

  try {
    // Call your Azure Function App API
    const response = await fetch('https://your-function-app.azurewebsites
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        name: form.name,
        date: form.date,
        category: form.category,
        description: form.description
      })
    });

    if (response.ok) {
      const result = await response.text();
```

```
      console.log('Function response:', result);
      setSubmitted(true);
    } else {
      console.error('Failed to submit event');
    }
  } catch (error) {
    console.error('Error submitting event:', error);
  }
};
```

Your current Function varies depending on how it was created, but typically looks like one of these:

Option 1: JavaScript Function (index.js)

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');
    const name = (req.query.name || (req.body && req.body.name));
    const responseMessage = name
        ? `Hello, ${name}. This HTTP triggered function executed successf
        : "This HTTP triggered function executed successfully. Pass a nam

    context.res = {
        body: responseMessage
    };
}
```

Option 2: Python Function (__init__.py)

```
import azure.functions as func
import logging

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')
```

```
    if name:
        return func.HttpResponse(f"Hello, {name}. This HTTP triggered fur
    else:
        return func.HttpResponse(
             "This HTTP triggered function executed successfully. Pass a r
             status_code=200
        )
```

Option 3: C# Function
Your function might be in a .cs file with C# code.

The important thing is that regardless of the language, your Function App is ready to handle HTTP requests and can be extended to process event data from your React frontend.

But even with API calls, your React code stays in the frontend, and Express just serves it!

"What if I don't want APIs or dynamic features in the frontend?"

You still need Express for Azure App Service deployment. Even if your frontend never adds its own APIs, Express is required to:

- Handle React Router properly
- Serve your static files on Azure
- Prevent 404 errors on page refreshes

Your backend Function App handles all your actual API logic and will continue to work separately.

"How does this work with my backend Function App?"

- **Frontend (React + Express)**: Deployed to Azure App Service - serves your website
- **Backend (Function App)**: Deployed to Azure Functions - provides your API endpoints
- **They work together**: Your React app makes API calls to your Function App URLs

Next Steps - Complete Deployment Process

Once you have Express set up and working locally, follow these detailed steps to deploy to Azure App Service.

Remember: You're deploying two separate things:

- **Frontend**: React app with Express → Azure App Service (this guide)
- **Backend**: Function App → Azure Functions (separate process)

Prerequisites for Deployment

1. **Azure CLI** installed and logged in
2. **Node.js** and **npm** installed
3. **Azure subscription** with an App Service created
4. **Express server** configured (you just did this!)

Step 1: Build Your React App

Navigate to your frontend directory and build the React app:

Windows (PowerShell/Command Prompt)

```
cd frontend
npm install
npm run build
```

Mac/Linux (Terminal)

```
cd frontend
npm install
npm run build
```

Step 2: Update Your server.js for Deployment

Your `server.js` should serve files from the correct location for Azure deployment. Update it to:

```
const express = require('express');
const path = require('path');
const app = express();
const port = process.env.PORT || 8080;

// Serve static files from the current directory (for flat deployment str
app.use(express.static(__dirname));

// For any other route, serve index.html (for React Router)
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'index.html'));
});
```

```
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

Step 3: Create Deployment Package

Create a minimal deployment package with only the necessary files:

Windows (PowerShell)

```
# Make sure you're in the frontend directory
cd frontend

# Create deployment directory
New-Item -ItemType Directory -Path "deploy-package" -Force

# Copy essential files
Copy-Item "build\index.html" -Destination "deploy-package\"
Copy-Item "build\static" -Destination "deploy-package\" -Recurse
Copy-Item "build\asset-manifest.json" -Destination "deploy-package\"
Copy-Item "package.json" -Destination "deploy-package\"
Copy-Item "server.js" -Destination "deploy-package\"
```

Mac/Linux (Terminal)

```
# Make sure you're in the frontend directory
cd frontend

# Create deployment directory
mkdir -p deploy-package

# Copy essential files
cp build/index.html deploy-package/
cp -r build/static deploy-package/
cp build/asset-manifest.json deploy-package/
cp package.json deploy-package/
cp server.js deploy-package/
```

Step 4: Install Production Dependencies

Navigate to the deployment directory and install only production dependencies:

Windows & Mac/Linux

```
cd deploy-package
npm install --omit=dev
cd ..
```

Why this step is important:

- Azure App Service will run `npm install` during deployment
- We only need production dependencies (like Express) on the server
- This reduces deployment size and potential conflicts
- Development dependencies (like React scripts) are not needed in production

Step 5: Install Archiver Package

Install the archiver package to create cross-platform compatible zip files:

Windows & Mac/Linux

```
npm install archiver
```

Step 6: Create Deployment Script

Create a Node.js script to generate a zip file with Unix-style paths (this fixes Windows deployment issues):

Windows & Mac/Linux

Create `create-deployment-zip.js` in your frontend directory:

```
const fs = require('fs');
const path = require('path');
const archiver = require('archiver');

async function createDeploymentZip() {
    const output = fs.createWriteStream('deploy-app.zip');
    const archive = archiver('zip', {
        zlib: { level: 9 } // Maximum compression
    });
```

```
// Listen for all archive data to be written
output.on('close', function() {
    console.log(`Archive created: ${archive.pointer()} total bytes`);
    console.log('Zip file created successfully with Unix-style paths!
});

// Handle errors
archive.on('error', function(err) {
    throw err;
});

// Pipe archive data to the file
archive.pipe(output);

// Add files from deploy-package directory
const deployDir = './deploy-package';

// Force Unix-style paths by converting backslashes to forward slashe
const toUnixPath = (filePath) => filePath.replace(/\\/g, '/');

// Function to add all files in a directory recursively with Unix pat
function addDirectoryFiles(dirPath, basePath = '') {
    const files = fs.readdirSync(dirPath);

    for (const file of files) {
        const fullPath = path.join(dirPath, file);
        const relativePath = basePath ? `${basePath}/${file}` : file;
        const unixPath = toUnixPath(relativePath);

        const stat = fs.statSync(fullPath);
        if (stat.isDirectory()) {
            // Recursively add directory contents
            addDirectoryFiles(fullPath, relativePath);
        } else {
            // Add individual file with Unix path
            archive.file(fullPath, { name: unixPath });
            console.log(`Added: ${unixPath}`);
        }
    }
}

// Add all files from the deploy-package directory
addDirectoryFiles(deployDir);
```

```
    // Finalize the archive
    await archive.finalize();
}

createDeploymentZip().catch(console.error);
```

Step 7: Generate Deployment Zip

Run the script to create the deployment zip:

Windows & Mac/Linux

```
node create-deployment-zip.js
```

This will create `deploy-app.zip` with proper Unix-style paths that work on Azure Linux App Service.

Step 8: Deploy to Azure

First, check your Azure resources:

List Resource Groups:

```
az group list --output table
```

List Web Apps in Resource Group:

```
az webapp list --resource-group YOUR_RESOURCE_GROUP_NAME --output table
```

Deploy the Application

Replace `YOUR_RESOURCE_GROUP_NAME` and `YOUR_APP_NAME` with your actual values:

Windows & Mac/Linux

```
az webapp deploy --resource-group YOUR_RESOURCE_GROUP_NAME --name YOUR_AI
```

Example:

```
az webapp deploy --resource-group EventPlannerRG --name eventplannerfront
```

Step 9: Verify Deployment

Check Deployment Status

```
az webapp log deployment show --resource-group YOUR_RESOURCE_GROUP_NAME -
```

View Application Logs

```
az webapp log tail --resource-group YOUR_RESOURCE_GROUP_NAME --name YOUR_
```

Test Your Application

Visit your app URL: `https://YOUR_APP_NAME.azurewebsites.net`

Step 10: Cleanup (Optional)

Remove temporary files after successful deployment:

Windows (PowerShell)

```
Remove-Item "create-deployment-zip.js"
Remove-Item "deploy-app.zip"
Remove-Item "deploy-package" -Recurse
```

Mac/Linux (Terminal)

```
rm create-deployment-zip.js
rm deploy-app.zip
rm -rf deploy-package
```

Need Help?

App doesn't work after adding Express?

- Make sure you ran `npm run build` first **in the frontend directory**
- Check that `server.js` is in the **frontend** folder (same folder as frontend/package.json)
- Verify Express is in your **frontend/package.json** dependencies
- Make sure you're testing at `http://localhost:8080` (not 3000)

Getting deployment errors?

- Follow the complete deployment guide for cross-platform zip creation
- Check Azure logs for specific error messages
- Ensure your Azure App Service is configured for Node.js
- Remember: you're only deploying the **frontend** to App Service

Backend Function App issues?

- Your Function App has its own deployment process (separate from this guide)
- Function Apps are deployed to Azure Functions, not App Service
- Make sure your React app's API calls point to the correct Function App URL

Comprehensive Troubleshooting

Express and Setup Issues

1. **"Cannot find module 'express"' Error**
    - **Problem**: Express not installed or not in dependencies
    - **Solution**: Run `npm install express` in your frontend directory
    - **Check**: Verify Express appears in your `package.json` dependencies
2. **"npm start" Runs React Dev Server Instead of Express**
    - **Problem**: Scripts in `package.json` not updated
    - **Solution**: Update your start script to `"start": "node server.js"`
    - **Alternative**: Use `"dev": "react-scripts start"` for development
3. **Server.js Not Found During Deployment**
    - **Problem**: Missing `server.js` file
    - **Solution**: Follow the Quick Start section above to create `server.js`
    - **Check**: Ensure `server.js` is in your frontend root directory

Deployment Issues

4. **Rsync Errors on Azure Linux**
   - **Problem**: Windows-created zip files use backslashes in paths
   - **Solution**: Use the Node.js archiver script above to create Unix-compatible paths
5. **404 Errors for React Routes**
   - **Problem**: Server not configured to serve index.html for all routes
   - **Solution**: Ensure your server.js has the `app.get('*', ...)` route handler
6. **Static Files Not Loading**
   - **Problem**: Incorrect static file serving configuration
   - **Solution**: Use `app.use(express.static(__dirname))` for flat deployment structure
7. **Resource Group or App Not Found**
   - **Problem**: Incorrect resource names
   - **Solution**: Use `az group list` and `az webapp list` to find correct names
8. **Port Configuration Issues**
   - **Problem**: App not responding on Azure
   - **Solution**: Ensure your server uses `process.env.PORT || 8080`
9. **"Build" Directory Not Found**
   - **Problem**: Forgot to run `npm run build` before deployment
   - **Solution**: Always run `npm run build` first to create the build directory

Backend Function App Issues

10. **API Calls Failing**
    - **Problem**: Incorrect Function App URL or authentication
    - **Solution**: Check your Function App URL and ensure it's publicly accessible
    - **Check**: Test your Function App endpoint directly in a browser
11. **CORS Errors**
    - **Problem**: Frontend and backend on different domains
    - **Solution**: Configure CORS in your Function App settings
    - **Azure Portal**: Go to Function App → CORS → Add your frontend domain

Environment Variables

If your app uses environment variables, configure them in Azure:

```
az webapp config appsettings set --resource-group YOUR_RESOURCE_GROUP_NAM
```

Complete Example Workflows

For Students Starting Without Express

If you're starting with just a React app (no Express), here's the complete workflow:

Windows

```
# Navigate to your React app directory
cd C:\path\to\your\project\frontend

# Step 1: Add Express
npm install express

# Step 2: Create server.js (see Quick Start section for content)
# Create the server.js file with the content from Step 2 above

# Step 3: Update package.json scripts
# Edit package.json to change "start": "react-scripts start" to "start":

# Step 4: Test locally
npm run build
npm start
# Visit http://localhost:8080 to verify it works

# Step 5: Create deployment package
New-Item -ItemType Directory -Path "deploy-package" -Force
Copy-Item "build\index.html" -Destination "deploy-package\"
Copy-Item "build\static" -Destination "deploy-package\" -Recurse
Copy-Item "build\asset-manifest.json" -Destination "deploy-package\"
Copy-Item "package.json" -Destination "deploy-package\"
Copy-Item "server.js" -Destination "deploy-package\"

# Step 6: Install production dependencies
cd deploy-package
```

```
npm install --omit=dev
cd ..

# Step 7: Install archiver and create zip script
npm install archiver
# Create create-deployment-zip.js (see Step 6 above)
node create-deployment-zip.js

# Step 8: Deploy to Azure
az webapp deploy --resource-group YOUR_RESOURCE_GROUP_NAME --name YOUR_AI

# Step 9: Check deployment
az webapp log deployment show --resource-group YOUR_RESOURCE_GROUP_NAME -
```

Mac/Linux

```
# Navigate to your React app directory
cd /path/to/your/project/frontend

# Step 1: Add Express
npm install express

# Step 2: Create server.js (see Quick Start section for content)
# Create the server.js file with the content from Step 2 above

# Step 3: Update package.json scripts
# Edit package.json to change "start": "react-scripts start" to "start":

# Step 4: Test locally
npm run build
npm start
# Visit http://localhost:8080 to verify it works

# Step 5: Create deployment package
mkdir -p deploy-package
cp build/index.html deploy-package/
cp -r build/static deploy-package/
cp build/asset-manifest.json deploy-package/
cp package.json deploy-package/
cp server.js deploy-package/

# Step 6: Install production dependencies
cd deploy-package
```

```
npm install --omit=dev
cd ..

# Step 7: Install archiver and create zip script
npm install archiver
# Create create-deployment-zip.js (see Step 6 above)
node create-deployment-zip.js

# Step 8: Deploy to Azure
az webapp deploy --resource-group YOUR_RESOURCE_GROUP_NAME --name YOUR_AI

# Step 9: Check deployment
az webapp log deployment show --resource-group YOUR_RESOURCE_GROUP_NAME -
```

Best Practices

1. **Always use the archiver script** to avoid path separator issues
2. **Test locally first** with `npm start` before deploying
3. **Keep deployment package minimal** - only include production files
4. **Check logs immediately** after deployment to catch issues early
5. **Use environment variables** for configuration instead of hardcoding values
6. **Separate frontend and backend deployments** - they're different services
7. **Keep your Function App and App Service in the same resource group** for easier management

Architecture Overview

After deployment, your application works like this:

```
User's Browser
     ↓
Azure App Service (Frontend - React + Express)
     ↓ (API calls)
Azure Functions (Backend - Function App)
     ↓
Database/Other Services
```

1. **User visits your website** → Azure App Service serves your React app
2. **React app needs data** → Makes API calls to Azure Function App
3. **Function App processes requests** → Returns data to React app
4. **React app displays results** → User sees the updated UI