# Initial Setup

## 1. Setup Your Development Environment - Complete

* **Django Backend:**
  + Install Django and create a new Django project.
  + Set up a virtual environment for Python dependencies.
* **React Frontend:**
  + Use **create-react-app** to generate a new React application.
  + Configure proxy in **package.json** for local development to forward requests to the Django backend.
* **Docker:**
  + Install Docker Desktop for your operating system.
  + Create a **Dockerfile** for both Django and React.
  + Create a **docker-compose.yml** file to define services, networks, and volumes.

## 2. Develop Your Application – Will complete later

* **Django:**
  + Develop your models, views, and URLs.
  + Configure the Django REST Framework for API development if needed.
* **React:**
  + Develop your components, services, and state management using React.
  + Use Axios or Fetch API to communicate with the Django backend.

## 3. Dockerize Your Application – Need to redo – separate front/back end

* Write Dockerfile for both Django and React:
  + For Django, use the Python base image, copy your project, install dependencies, and run the Django server.
  + For React, use the Node base image, copy your project, install dependencies, and build your React app.
* In **docker-compose.yml**, define services for Django, React, and any other services like databases.

## 4. Testing Your Docker Containers

* Use **docker-compose up** to build and start your services.
* Test the application to ensure both Django and React are properly communicating.

## 5. Prepare for Deployment

* **Heroku:**
  + Sign up for a Heroku account if you haven't already.
  + Install the Heroku CLI.
  + Login to your Heroku account via CLI.
  + Create a new Heroku app.
* **Docker & Heroku:**
  + Configure **heroku.yml** for deploying Docker containers.
  + Push your Docker images to Heroku Container Registry.
  + Use Heroku's PostgreSQL addon for your database or configure another database service.

## 6. Deploy to Heroku

* Deploy your application using Heroku CLI.
* Release the containers to your Heroku application.
* Configure environment variables in Heroku.

## 7. Final Checks and Launch

* Ensure that all components are running correctly on Heroku.
* Perform final testing to check the functionality and performance of your application.
* Update your domain settings if you have a custom domain.

## Conclusion

This outline provides a broad overview of the steps involved in setting up and launching an application with Django, React, Heroku, and Docker. Each step involves several detailed actions, so it's important to consult official documentation or specific guides for each tool or service you're using.

# Setting Up Bootstrap in React

## Install Bootstrap:

* + Run **npm install bootstrap** to add Bootstrap to your project dependencies.
  + If you plan on using Bootstrap icons, also run **npm install react-bootstrap-icons**.

## Import Bootstrap CSS:

* + In your **src/index.js** or **src/App.js** file, add **import 'bootstrap/dist/css/bootstrap.min.css';** at the top to include Bootstrap's CSS.

## (Optional) Use React-Bootstrap:

* + For a more React-friendly experience with Bootstrap components, consider using React-Bootstrap.
  + Install React-Bootstrap by running **npm install react-bootstrap**.
  + Import specific components as needed from React-Bootstrap in your component files, for example, **import { Button } from 'react-bootstrap';**.

## Responsive Meta Tag:

* + Ensure your **public/index.html** includes the viewport meta tag in the **<head>** section for responsive design: **<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">**.

# Building Your First Component (Header)

## Component Structure:

* + Create a new file for your header component, e.g., **src/Components/Header.js**.
  + Define the structure of your header component using functional or class-based component syntax.

## Using Bootstrap in Your Component:

* + Use Bootstrap's navigation components and classes to structure your header, such as **navbar**, **navbar-brand**, **navbar-nav**, **nav-item**, and **nav-link**.
  + Consider incorporating a responsive design with collapsible navbars for mobile views using Bootstrap's responsive utilities.

## Customization and Styling:

* + To customize Bootstrap's default styles, consider using SASS. Install it with **npm install node-sass** and override Bootstrap's variables in custom SCSS files.
  + Alternatively, add custom CSS classes alongside Bootstrap's classes to adjust the styling as needed.

## Accessibility and SEO:

* + Ensure your header component is accessible, including proper ARIA roles and attributes where necessary.
  + Use semantic HTML elements (e.g., **<header>**, **<nav>**) to enhance SEO and accessibility.

## Testing:

* + Write unit tests for your header component to ensure it renders correctly and functions as expected. You can use Jest and React Testing Library, which come included with Create React App.
  + Perform cross-browser testing to ensure consistent behavior and appearance across different browsers and devices.

## Performance Optimization:

* + Lazy load images or components within your header if they are not critical to the initial page load.
  + Use React's **memo** for functional components or **PureComponent** for class components if your header doesn't rely on changing state or props.

## Integration and Review:

* + Integrate your header component into your app, typically by including it in your **App.js** file.
  + Conduct a code review with your team to ensure best practices and maintain consistency across the project.

**Final Steps**

* **Commit Changes**: Once your header component is complete and tested, commit your changes to your version control system.
* **Continuous Integration/Deployment (CI/CD)**: Ensure your CI/CD pipeline is set up to build, test, and deploy your application after changes are merged to your main branch.