Основано на:

<https://www.youtube.com/watch?v=nY0R7pslbCI>

В этом тексте рассказывается о том, как объединить приложение Netstress с приложением React.js в одном моно-репозитории. Автор объясняет, что обычно для таких случаев рекомендуется использовать фреймворк Next.js, который предоставляет поддержку как для фронтенда, так и для бэкенда. Однако, если требуется использовать Nest.js в качестве бэкенд-фреймворка, то необходимо разделить приложение на две части и запустить два сервера для работы с ними.

[[ 0:0 ]](https://youtu.be/nY0R7pslbCI?t=0)

**Combining Netstress and React.js into a Mono Repo**

In this text, the author discusses the possibility of combining a Netstress application with a React.js application into a single mono repo. They explain that if you want to create a full-stack application using JavaScript or TypeScript on both the front end and back end, you can use React.js for the front end. They mention that most people in this use case would consider using Next.js, which is a meta framework for React that also serves as a backend framework, allowing you to create your own API routes and do server-side rendering and static site generation. However, the author also mentions Nest.js and its benefits. They explain that while Next.js offers minimal backend support and leaves the rest of the architecture up to you, Nest.js has first-class support for things like GraphQL, authentication, authorization, database connections, Swagger docs, web sockets, and event-driven architectures. They suggest that if you choose Nest.js as your primary backend framework, you would effectively have two applications, Nest.js for the backend and Next.js for the front end, which can be a valuable combination.

[[ 2:8 ]](https://youtu.be/nY0R7pslbCI?t=128)

**Creating a Monorepo with Nest.js and React**

Both Nest.js and Next.js are server frameworks, which means that you would typically need to spin up two different servers to run your application. However, if you don't need server-side rendering or static site generation, you can just use React on its own and have your Next.js application serve it up in production. In this video, the goal is to create a monorepo that combines both frameworks into a single running server. To get started, create a directory for your application and install the Turbo dependency. Then, add a "workspaces" field to your package.json file, specifying the path pattern where your workspaces will be located. In this case, the applications will be in an "apps" folder. Next, use the Nest.js CLI to create a new project in the "apps" folder. Finally, you can continue building your project with both frameworks in the monorepo.

[[ 4:9 ]](https://youtu.be/nY0R7pslbCI?t=249)

**Creating React and Nest applications, setting up TurboRepo**

To start with, we have already installed the necessary packages. On the left side, we can see that our application has been created. Now, let's create our client application in the terminal. For the React application, we will use Veet. We can use the command "npm create Vat latest" followed by the project name, which in this case is "client". Make sure to select React and TypeScript. However, note that Veet does not automatically install the packages, so we need to run "npm install" at the root of our project. It is important to install the dependencies at the root level, as the workspaces feature will look at all the individual package.json files in both the API and client folders. At this point, we have two applications - the Nest API and the React client. Now, we need to figure out how to work with both applications simultaneously. This is where TurboRepo comes in. TurboRepo is a tool that helps set up a monorepo quickly. We will create a new file called "turbo.json" at the root and add some initial configuration. You can pause the screen and copy the configuration from the video. The "turbo.json" file defines the task dependency graph of our project. Within the "pipeline" object, we can add tasks that TurboRepo will run across our different applications based on matching npm scripts in the workspace packages.

[[ 6:15 ]](https://youtu.be/nY0R7pslbCI?t=375)

**Running applications using npm runDev and Turbo run Dev**

В клиентском приложении у нас есть файл package.json, в котором мы можем запустить приложение с помощью команды npm runDev. В API также есть файл package.json со своими сценариями, включая Dev, который мы переименовали в start Dev. Мы также добавили флаг preserve watch output, чтобы сохранить вывод в терминале при изменении кода. В корневом package.json мы добавили скрипт Dev, который запускает команду turbo run Dev. Если мы запустим команду npm run Dev в терминале, то увидим, что она фактически запускает turbo run Dev, который выполняет задачу Dev для каждого приложения, используя соответствующие команды из их package.json. Таким образом, оба наших приложения запускаются одновременно, и мы можем проверить их работу, открывая их в браузере по соответствующим адресам.

[[ 8:19 ]](https://youtu.be/nY0R7pslbCI?t=499)

**Local development setup vs production setup: different servers running side by side. Proxying requests from client to server.**

В локальной разработке наша настройка отличается от настройки в продакшн, когда мы фактически развернули наше приложение вместе. Таким образом, у нас фактически есть два разных сервера, работающих параллельно. Мы используем webpack для запуска nest.js, а для клиентской части у нас есть Veet, который запускает наше react-приложение. Оба этих сервера ожидают изменений в коде и автоматически обновляют приложение. Однако в нашей продакшн среде это не работает так. У нас будет один сервер, который будет обслуживать сборку нашего Nest-приложения, и отдельную сборку для нашего react-приложения. Сборка нашего Nest-приложения просто запускается через node, она не требует сервера webpack, и она будет способна обслуживать клиентскую сборку в браузере. У нас также будет API, которое будет включено в наше Nest-приложение, чтобы клиент мог делать запросы к нему. В локальной разработке, если у нас есть два отдельных сервера, как мы перенаправляем запросы от клиента к серверной части? Например, если здесь работает сервер, и клиент делает запрос к /API, как он знает, что нужно перенаправить его в наше Nest-приложение, если они работают на разных серверах? К счастью, это то, что поддерживает Veet, поэтому давайте настроим это и посмотрим на наш vconfig. Мы просто добавим некоторую конфигурацию сервера, которая говорит: "если вы видите этот путь /API, убедитесь, что он перенаправляется на localhost:3000, который является нашим локальным Nest-приложением". Это означает, что каждый запрос, который мы пытаемся сделать к нашему Nest-приложению, должен иметь эту базу /API. Я также настрою наше Nest-приложение.

[[ 10:24 ]](https://youtu.be/nY0R7pslbCI?t=624)

**Setting Global Prefix for API Routes in Nest.js and Making a Request in React**

В файле main.ts мы добавляем глобальный префикс API для всех маршрутов, чтобы они начинались с /API. Затем мы переходим к файлу app.tsx в нашем клиентском приложении React. Мы очищаем код и добавляем состояние greeting с помощью useState. Затем мы используем useEffect, чтобы сделать запрос к /API и получить приветствие. Мы устанавливаем полученное приветствие в состояние greeting и отображаем его на странице. Теперь, если вы запустите приложение, оно должно отображать "Hello, world", полученное с нашего Nest API.

[[ 12:28 ]](https://youtu.be/nY0R7pslbCI?t=748)

**Proxying for local development and building applications together**

В продакшене прокси не нужна, так как запросы отправляются напрямую на тот же сервер. Прокси используется только для локальной разработки. Когда разработка завершена и готова к развертыванию новая функция, нужно подумать о том, как собрать оба приложения вместе. Аналогично тому, как мы запускали оба приложения вместе для разработки, мы также можем собрать оба приложения одновременно. В файле turbo.json мы добавим ключ build в нашу конвейерную систему. У нас уже должен быть скрипт сборки для нашего API в файле package.json API, и также должен быть скрипт сборки для клиента в файле package.json клиента, который использует TSC и Veet build. Мы также должны добавить несколько дополнительных настроек. Мы указываем путь к папке dist в каждом приложении, так как после сборки в каждом из них будет папка dist. Затем depends on - это просто граф зависимостей, когда вы запускаете npm run build, что от этого зависит. Вы можете перечислить другие скрипты, например, тестирование или линтинг, если у вас есть такие скрипты настроены. В нашем случае это просто сборка.

[[ 14:31 ]](https://youtu.be/nY0R7pslbCI?t=871)

**Configuring Turbo for Dependency Building and Testing**

В этом тексте рассказывается о конфигурации сборки проекта с использованием пакета Turbo. Символ "картошка" указывает, что задача сборки зависит от пакетов и их зависимостей, и перед сборкой приложения необходимо сначала собрать эти зависимости. Для тестирования этой конфигурации добавляется команда "turbo run build" в файл package.json, которая вызывает сборку во всех приложениях. При повторном запуске сборки происходит кэширование, если код не изменился. В результате сборки ожидается появление папки "disk" в приложении API.

[[ 16:36 ]](https://youtu.be/nY0R7pslbCI?t=996)

**Serving Static Files in Nest.js**

В данном тексте рассказывается о том, что в проекте есть папка "client", в которой находятся выходные файлы. Эти файлы используются для кэширования и повторного использования уже созданных данных. Для создания сборки проекта используется команда, которая выполняется параллельно. Однако, в продакшене приложение Nest будет использовать сборку клиентской части для отображения в браузере. Для этого используется модуль server static, который нужно установить и настроить. Для установки нового пакета в приложении используется команда "npm install --workspace", а для настройки модуля server static добавляется импорт в файл app module и указывается путь к сборке клиентской части.

[[ 18:40 ]](https://youtu.be/nY0R7pslbCI?t=1120)

**Building and Starting the React and Nest.js Application**

At this point, we need to rebuild our application and start our API in production mode. In the package.json file for the API, there is a "start prod" script that runs the application. We will add a similar script in the root package.json file that runs the API from the root. After making these changes, we can run "npm run build" to pull in the changes and then "npm start" to start the Nest API server in production mode and serve the client build. The application should successfully start and serve the React static build. The single running Nest server serves the React client build to the browser. The browser displays the "hello world" response from the server. This setup provides an end-to-end full stack setup for React and Nest.js.