**КТ № 5. КТ № 5. Nodemon. Асинхронность в Node.js.**

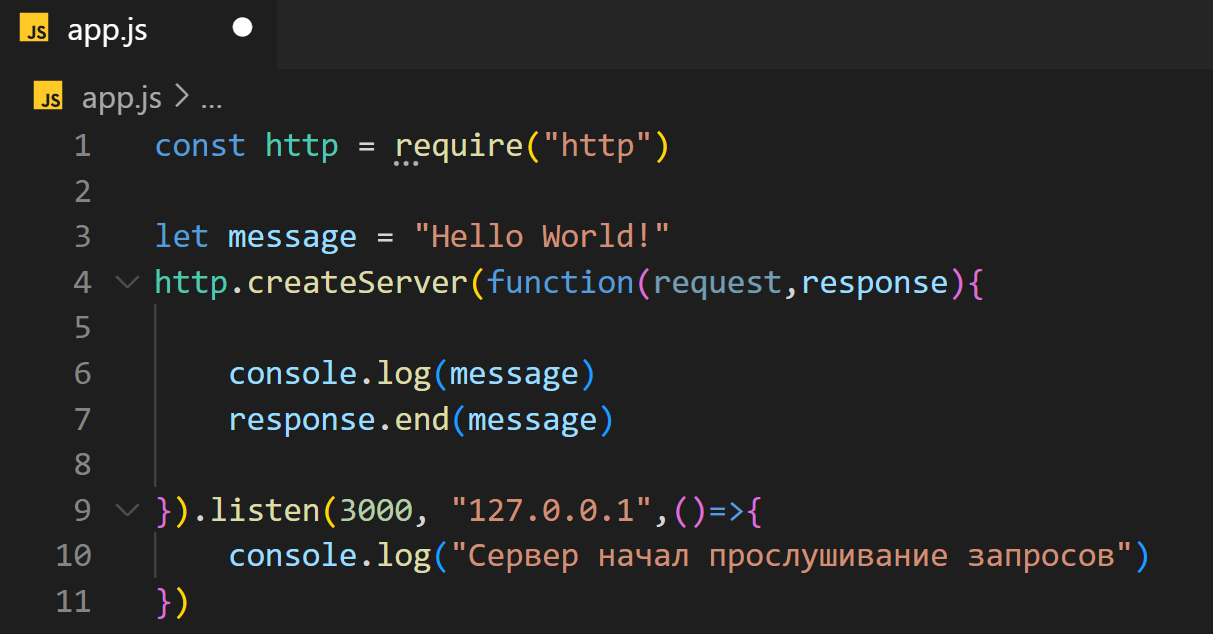
**Время на выполнение практической работы:** 4 часа.

**Теоретические сведения к выполнению практической работы:**

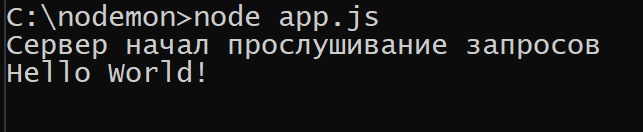
**Nodemon**

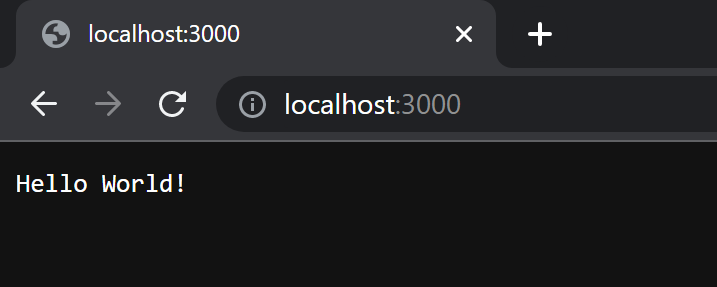
В процессе разработки может потребоваться необходимость внести изменения в уже запущенный проект. В файле app.js определен следующий код:

В процессе разработки может потребоваться необходимость внести изменения в уже запущенный проект. В файле app.js определен следующий код:



Запустим сервер с помощью команды node app.js, и при обращении пользователя по адресу http://localhost:3000/ браузер пользователя отобразит строку "Hello World!". Одновременно строка выводится на консоль.



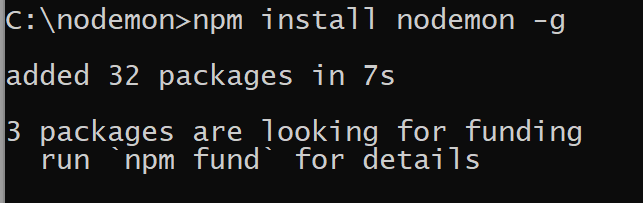


При этом сервер продолжает быть запущенным. И если мы изменим переменную message в файле app.js, то это никак не повлияет на работу сервера, и он будет продолжать отдавать клиенту строку "Hello World!".

В этом случае необходимо перезапустить сервер. Однако это не очень удобно, особенно когда необходимо часто делать различные изменения, тестировать выполнение. И в этом случае нам может помочь специальный инструмент nodemon.

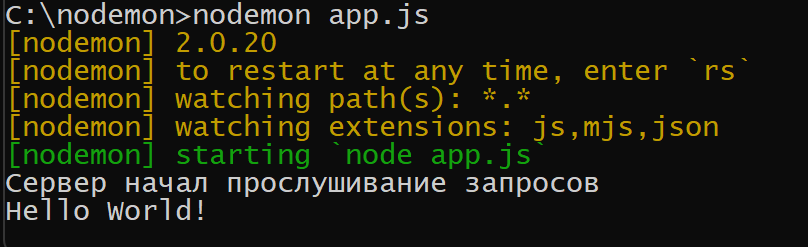
Установим nodemon в проект с помощью следующей команды:



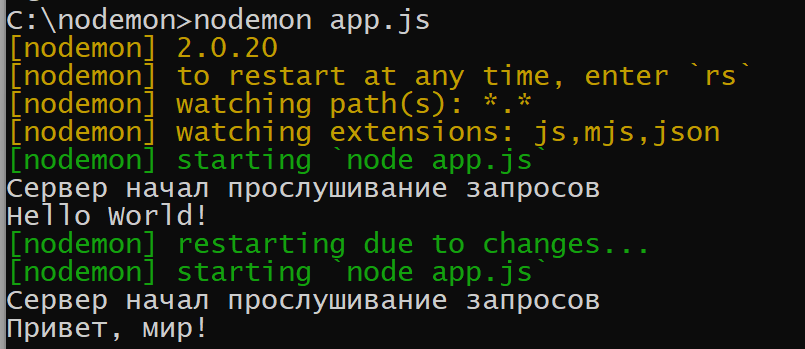


Флаг -g представляет сокращение от global и позволяет установить зависимость nodemon глобально для всех проектов на данной локальной машине.

После установки запустим файл app.js с помощью следующей команды:



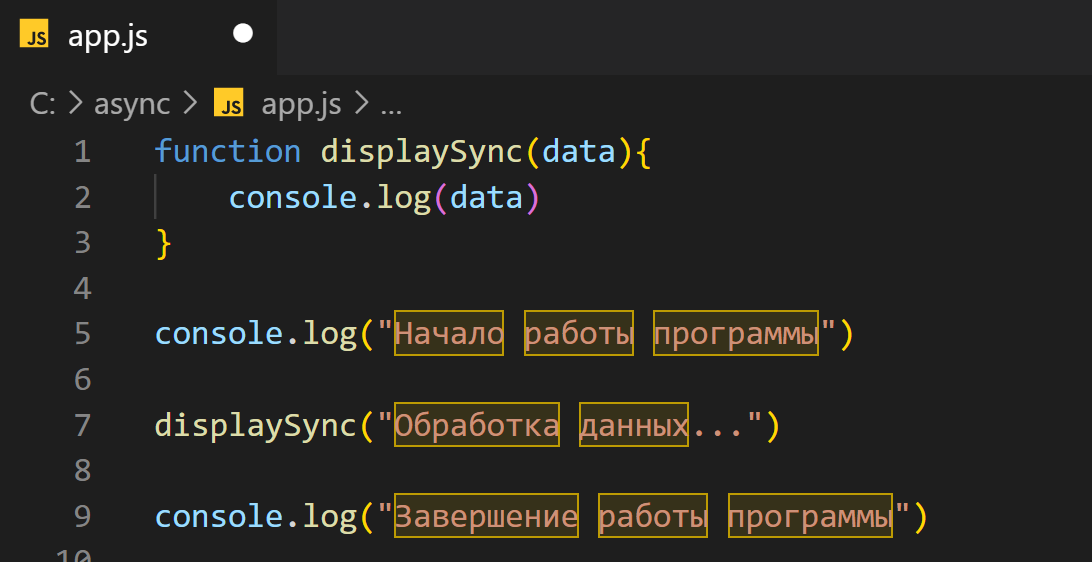
И если вдруг после запуска сервера мы изменим его код, например, поменяем переменную message с "Hello World!" на "Привет мир!", то сервер автоматически будет перезапущен:



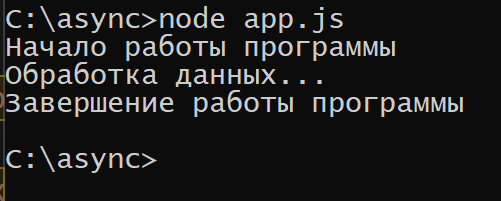
**Асинхронность в Node.js**

Асинхронность представляет возможность одновременно выполнять сразу несколько задач. Асинхронность играет большую роль в Node.js.

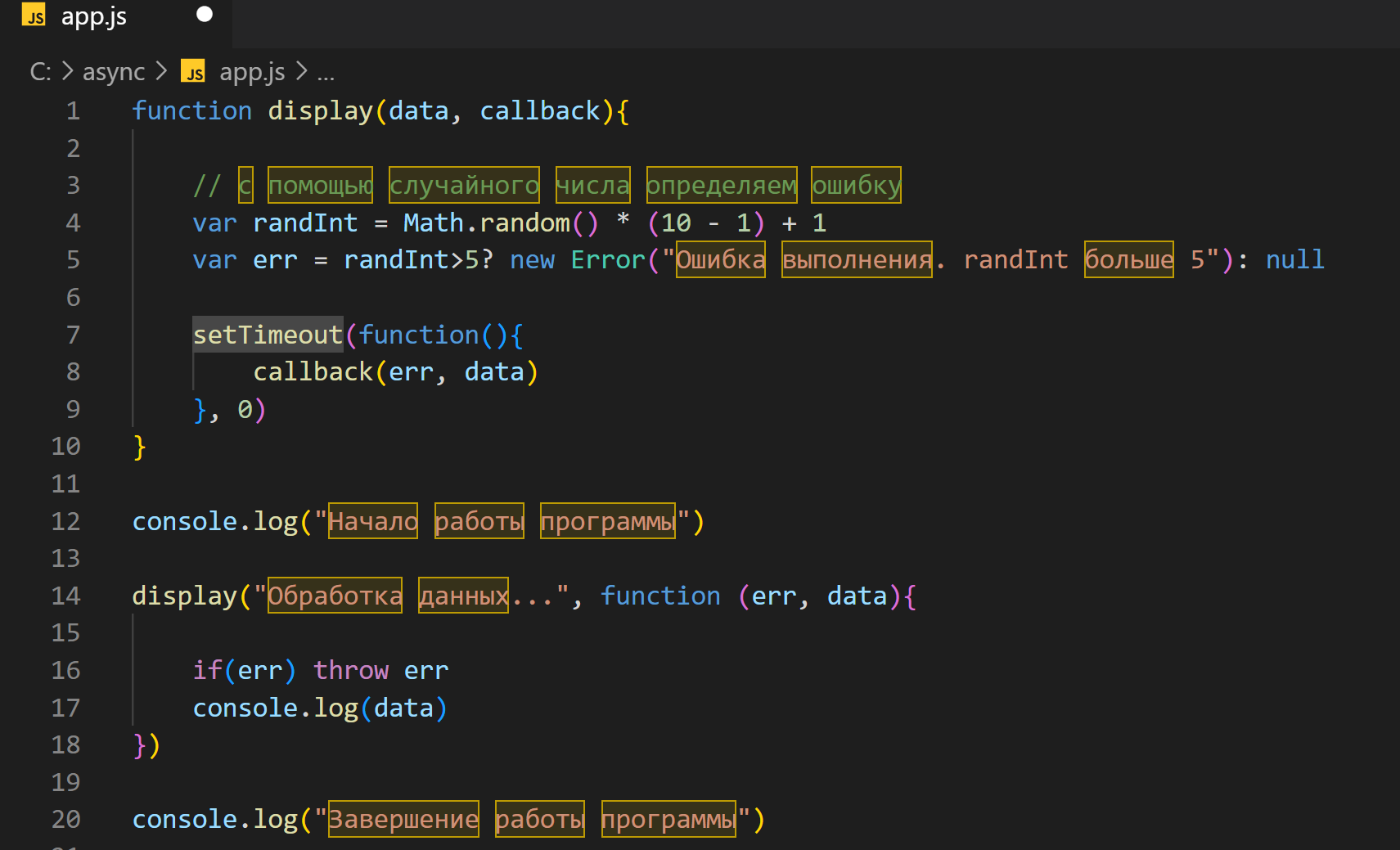
Например, допустим в файле приложения app.js у нас расположен следующий код:



Это стандартный синхронный код, все вызовы здесь выполняются последовательно, что мы можем увидеть, если мы запустим приложение:



Для рассмотрения асинхронности изменим код файла app.js следующим образом:



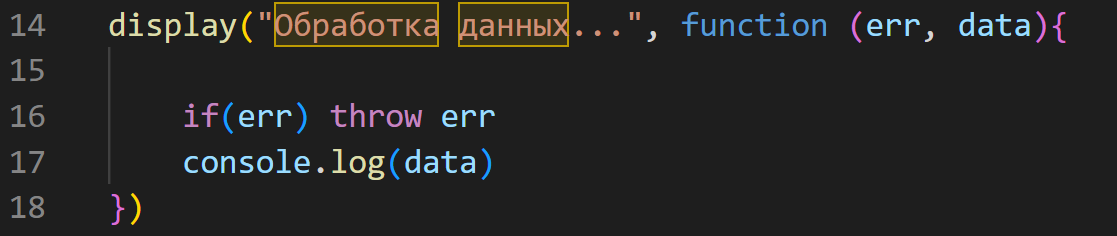
В начале также определяется функция display, но теперь кроме данных в качестве второго параметра она принимает функцию обратного вызова, которая и обрабатывает данные.

Эта функция callback принимает два параметра - информацию об ошибке и собственно данные. Это общая модель функций обратного вызова, которые передаются в асинхронные методы - первым идет параметр, представляющий ошибку, а второй - данные.

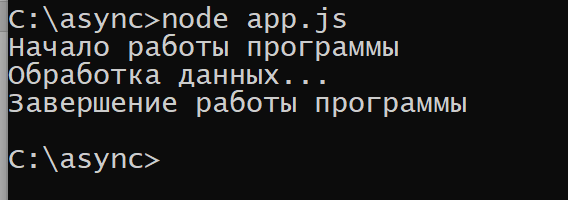
Для имитации ошибки используется случайное число: если оно больше 5, то создаем объект ошибки - объект Error, иначе же он равен null.

И последний важный момент - выполнение функции обратного вызова в функции setTimeout(). Это глобальная функция, которая принимает в качестве первого параметра функцию обратного вызова, а в качестве второго - промежуток, через который функция обратного вызова будет выполняться. Для нашей задачи вполне подойдет промежуток в 0 миллисекунд.

При вызове функции display в нее передается функция, которая в случае отсутствия ошибок просто выводит данные на консоль:



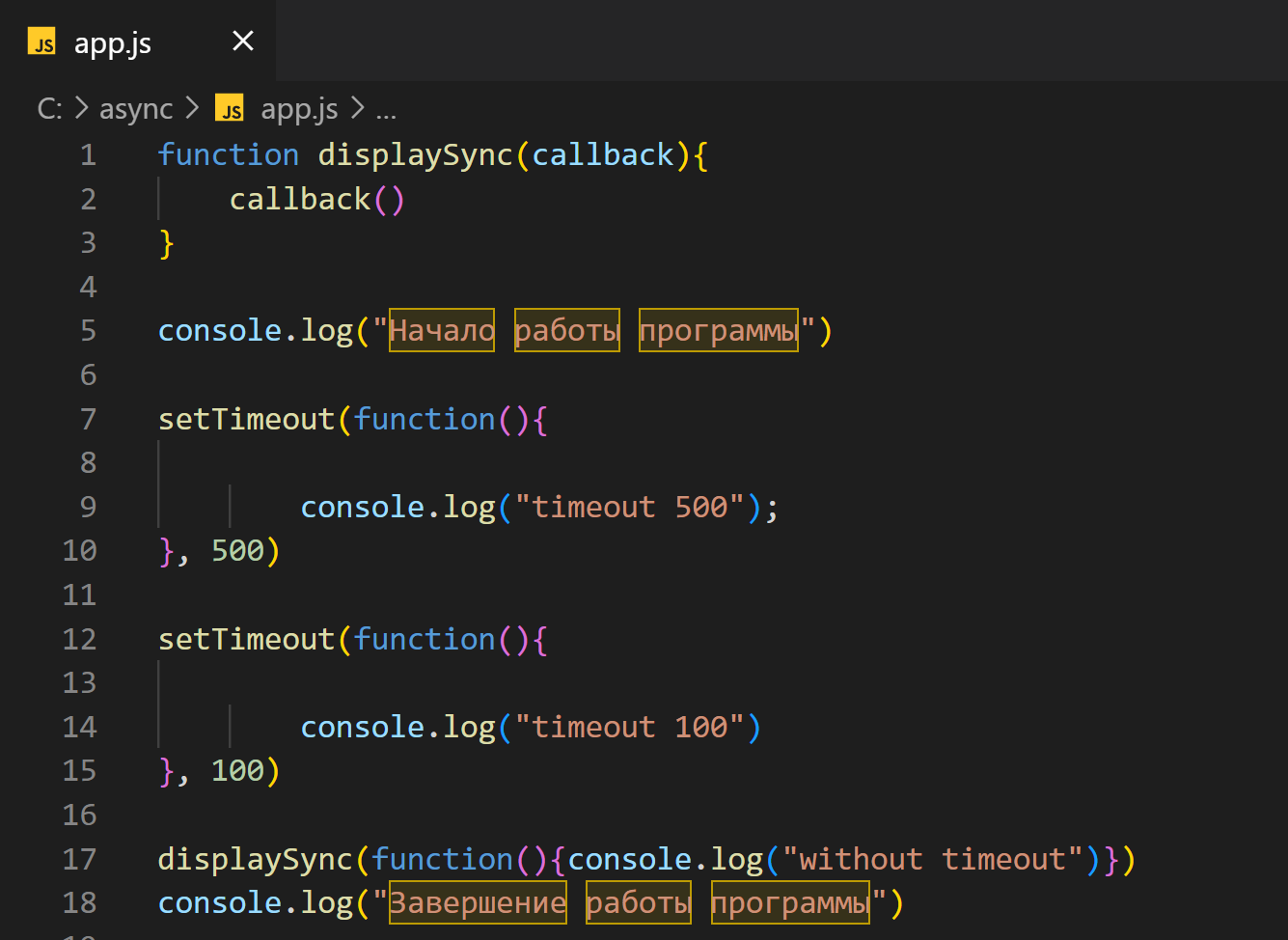
Теперь если мы запустим приложение, то увидим, следующую картину:



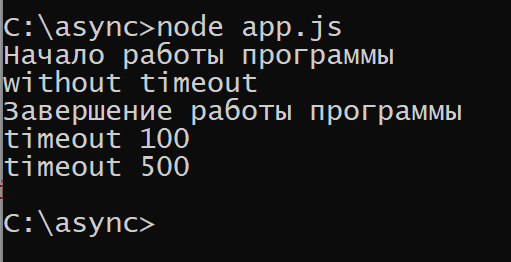
Несмотря на то, что в setTimeout передается промежуток 0, фактическое выполнение функции display завершается после всех остальных функций, которые определены в программе. В итоге выполнение на функции display не блокируется, а идет дальше. И это особенно актуально, если в приложении идет какая-либо функция ввода-вывода, например, чтения файла или взаимодействия с базой данных, выполнение которой может занять продолжительное время. То общее выполнение приложение не блокируется, а идет дальше.

Почему так происходит? Потому что все колбеки или функции обратного вызова в асинхронных функциях (в качестве таковой здесь используется функция setTimeout) помещаются в специальную очередь, и начинают выполняться после того, как все остальные синхронные вызовы в приложении завершат свою работу. Собственно поэтому выполнение колбека из функции setTimeout в примере выше происходит после выполнения вызова console.log("Завершение работы программы");. И стоит подчеркнуть, что в очередь колбеков переходит не функция, которая передается в display, а функция, которая передается в setTimeout.

Рассмотрим пример с двумя асинхронными вызовами:



Результат выполнения:



Несмотря на то, что в функцию display передается колбек, но эта функция с колбеком будет выполняться синхронно.

А колбеки из функций setTimeout будут выполняться только после всех остальных вызовов приложения.

**Этапы выполнения:**

1. ***Запустите файл со стандартным синхронным кодом;***
2. ***На каждом этапе изменения добавляйте в код комментарии;***
3. ***Измените код файла для рассмотрения асинхронности. Используйте функцию setTimeout();***
4. ***Поменяйте код файла, чтобы использовать два асинхронных вызова;***
5. ***Каждый этап продемонстрируйте необрезанным скриншотом;***
6. ***Перед каждым пунктом опишите свои действия;***
7. ***Сделайте вывод о проделанной работе.***