1 ОБЩАЯ ЧАСТЬ

* 1. Актуальность проекта

Главной задачей диплома является разработка веб приложения для проведения вебинаров**, и развертывание приложения на сервере. Приложение нацелено на привлечение клиентов для основного продукта. Основные элементы вебинара, это общение с аудиторией и сбор данных для связи с потенциальными клиентами в будущем.**

* 1. Сетевые технологии
     1. Краткие сведения о веб приложениях

Основное отличие веб приложения от сайта, это генерация контента в зависимости от пользовательского ввода (реактивность). Веб приложение часто связывается с сервером для получения и отправки данных. Это могут быть чаты, социальные сети и т.п.

Типичное веб-приложение состоит из HTML-верстки, CSS-стилей и JavaScript-кода, который позволяет достичь максимального уровня интерактивности и отзывчивости, кода выполняющего внутреннюю логику и базы данных.

Есть несколько подходов к архитектуре веб приложения. Выбор может варьироваться в зависимости от масштаба и сложности приложения.

Монолитная архитектура – это архитектурный подход, в котором вся основная логика приложения собрана в одном месте. Монолитное приложение состоит из однослойного объединения разных компонент в одно целое. Клиент получает только готовые страницы.

Сервис-ориентированная архитектура (SOA) – это полная противоположность монолитной архитектуры. Это модульный подход к разработке программного обеспечения, базирующийся на обеспечении удаленного использования по стандартизированным протоколам распределённых, слабо связанных легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами.

REST (Representational state transfer) – является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Бессерверная архитектура – Новая разновидность SOA, отличается тем, что обеспечивает работу облачных сервисов по требованию. Это означает, что бессерверные функции запускаются только в момент фиксации определенного события.

1.2.2 Этапы разработки

Разработка технического задания – составление документации. на этом этапе нужно описать как будет выглядеть и работать приложение.

Создание дизайна и навигации – прорисовка дизайн макетов, разработка UI и UX дизайна, логотипа, структуры навигации и т.д.

Верстка – описание страницы на языке разметки в соответствии с дизайн макетом.

Программирование – оживление верстки с помощью JavaScript, написание внутренней логики.

Тестирование – то процесс исследования ПО с целью выявления ошибок и определения соответствия между реальным и ожидаемым поведением ПО.

Развертывание – запуск приложения на рабочем сервере, настройка конфигурации сервера, базы данных, сторонних сервисов.

1.2.3 Основные сервисы веб прложения

Frontend (фронтенд) – это сервис предоставляющий пользователю интерфейс через который он может общаться с базой другими сервисами веб приложения. Основной функционал фронтенда состоит в отправке запросов по заранее определенным стандартам (интерфейсам, протоколам) к сервисам, отвечающим за обработку данных.

Backend (бэкенд) – это может быть один монолитный сервис или кластер сервисов отвечающих за внутреннюю логику. Внутренняя логика – обработка данных, поступающих с внешних ресурсов, работа с базой(-ами) данных, управление другими сервисами.

База данных – группированный набор однородной информации. В контексте применяется для хранения наборов данных авторизации, токенов.

SMTP server (почтовый сервер) – необходим для отправки писем по электронной почте.

Proxy server (сервер переадресации) – может служить для перенаправления трафика основываясь на заголовках пакета данных, либо для распределения нагрузки на сервера.

* 1. Описание языка программирования
     1. Общие сведения

Для написания полноценного веб-приложения не хватит какой-то одной технологии, поэтому существует понятие стек технологий, которое в полной мере охватывает инструменты, использованные при разработке.

Стек технологий — это набор технологий, на основе которых разрабатывается веб приложение. Самый распространенный стек – это HTLM, CSS, JavaScript (далее js), PHP, MySQL. При чем первые 3 присутствуют в каждом стеке, либо получаются при компиляции проекта. Так, например в React (библиотеке для js), для разметки страницы используются файлы-шаблоны с расширением «.jsx», которые в последствии компилируются в .html и .js файлы. Или препроцессоры для CSS, такие как SASS, SCSS, LESS, которые компилируются в CSS файлы.

Для своего проекта я взял архитектурный подход REST, но веб приложение не соответствует всем критериям REST. Приложение я разбил на 3 части; это пользовательский интерфейс (Frontend), внутренняя логика (Backend) и База данных. Так же я обращаюсь к сторонним сервисам, в частности это почтовый сервер и доменный сервер.

Мой стек состоит из Angular 11 для фронтенда (HTML, CSS, js), NestJs для бэкенда (внутренняя логика), MySQL – база данных. Для развертывания я использовал Docker. Почтовый сервер и доменное имя я купил на dreamhost.com, Сервер для развертывания и базу данных приобрел на digitalocean.com

1.3.2 REST архитектура

REST (Representational state transfer) – это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

Как было упомянуто выше – REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Как это работает.

Отсутствие дополнительных внутренних прослоек означает передачу данных в том же виде, что и сами данные. Т.е. мы не заворачиваем данные в XML, как это делает SOAP и XML-RPC, не используем AMF, как это делает Flash и т.д.

Каждая единица информации однозначно определяется URL – это значит, что URL по сути является первичным ключом для единицы данных. К примеру, путь до 10 квартиры в 3-ем доме будет выглядеть как /house/3/apartment/10.

Управление информацией сервиса – целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же HTTP. Так вот, для HTTP действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия CRUD (Create-Read-Updtae-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST.

Чтобы распределенная система считалась сконструированной по REST архитектуре, необходимо, чтобы она удовлетворяла следующим критериям:

* Client-Server. Система должна быть разделена на клиентов и на серверов.
* Stateless. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и если необходимо, идентификации клиента.
* Cache․ Каждый ответ должен быть отмечен является ли он кэшируемым или нет.
* Uniform Interface. Универсальный интерфейс между компонентами системы
* Layered System. В REST допускается разделить систему на иерархию слоев, но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например, если вы вызывайте службу PayPal а он в свою очередь вызывает службу Visa, вы о вызове службы Visa ничего не должны знать.
* Code-On-Demand. В REST позволяется загрузка и выполнение кода или программы на стороне клиента.

1.3.3 Angular 11

Angular это платформа и фреймворк для разработки одностраничных приложений с использованием TypeScript и HTML. Он включает в себя ядро и опциональные TypeScript библиотеки, которые можно импортировать в приложение. Angular приложение состоит из компонентов Angular которые организованы в модули NgModules. NgModules собирают связанный код в функциональные наборы; Приложение Angular определяется набором модулей NgModules. В приложении всегда есть как минимум корневой модуль, обеспечивающий начальную загрузку, и, как правило, гораздо больше функциональных модулей.

* Компоненты определяют шаблоны (виды, представления), которые представляют собой наборы элементов экрана, которые Angular может выбирать и изменять в соответствии с логикой и данными программы.
* Компоненты используют службы, которые предоставляют определенные функции, не связанные напрямую с представлениями. Поставщики услуг могут быть введены в компоненты как зависимости, что сделает код модульным, многоразовым и эффективным.

Модули, компоненты и сервисы – это классы, использующие декораторы. Одно из важнейших понятий Angular – Декоратор.  Это функции, которые изменяют классы JavaScript. Angular определяет ряд декораторов, которые прикрепляют определенные виды метаданных к классам, чтобы система знала, что означают эти классы и как они должны работать.

* Метаданные для класса компонента связывают его с шаблоном, определяющим представление. Шаблон сочетает в себе обычный HTML с директивами Angular и разметкой привязки, которые позволяют Angular изменять HTML перед его отображением.
* Метаданные для класса обслуживания предоставляют информацию, необходимую Angular, чтобы сделать ее доступной для компонентов посредством внедрения зависимостей (DI).

1.3.3.1 Установка

Для разработки Angular приложений необходимо настроить среду разработки.

Первое – установка Node.js и пакетного менеджера по вкусу (npm, yarn)

Для установки Node.js на платформу с операционной системой Windows необходимо скачать и запустить пакет установки с официального сайта «https://nodejs.org/en/download/». Нужно обратить внимание на версию Node, так как Angular требует наличия активной LTS или обслуживаемой LTS-версии. Посмотреть статус версий Node можно по этой ссылке «https://nodejs.org/en/about/releases/».

Пакетный менеджер npm предоставляется вместе с пакетом Node.js.

Основное – установка Angular CLI (command line interface). Для этого в командной строке необходимо выполнить команду «npm install -g @angular/cli».

После установки уже можно создать новый проект командой

«ng new [название проекта]».

Для запуска тестового сервера необходимо выполнить команду «ng serve» либо «npm run start» находясь в директории проекта. Запущенный сервер доступен по адресу «localhost:4200» или «127.0.0.1:4200».

Страница полученная после выполнения всех вышеперечисленных действий приведена на рисунке 1.3.1.

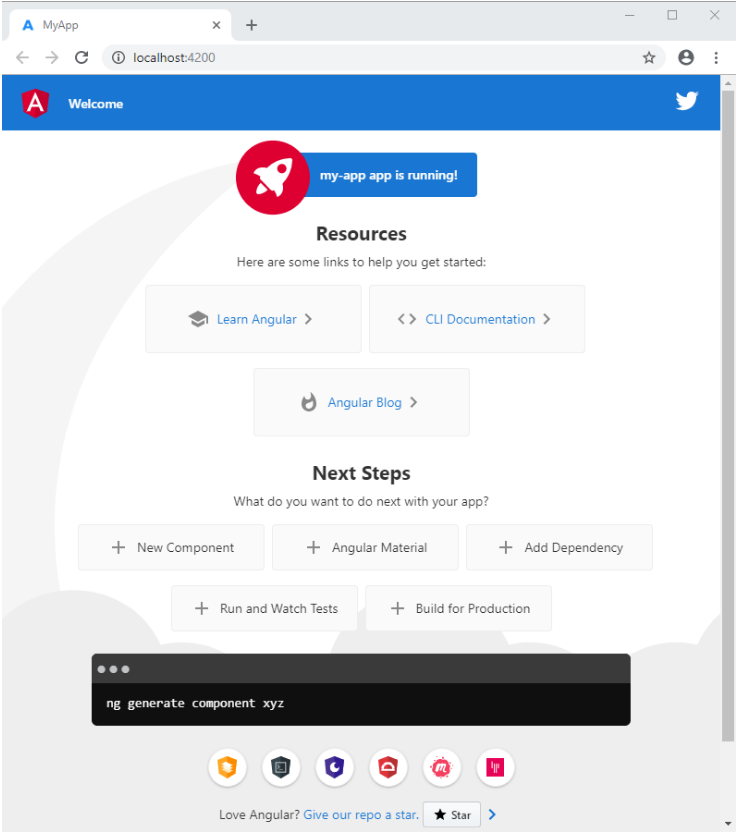


Рисунок 1.3.1 – автоматически сгенерированная начальная страница Angular.

1.3.3.1 Модули

Angular NgModules отличаются от модулей JavaScript (ES2015) и дополняют их. NgModule объявляет контекст компиляции для набора компонентов, который предназначен для домена приложения, рабочего процесса или тесно связанного набора возможностей. NgModule может связывать свои компоненты со связанным кодом, например службами, для формирования функциональных единиц.

Каждое приложение Angular имеет корневой модуль с условным названием AppModule, который обеспечивает механизм начальной загрузки, запускающий приложение. Приложение обычно содержит множество функциональных модулей.

Подобно модулям JavaScript, NgModules может импортировать функции из других модулей NgModules и разрешать экспорт своих собственных функций и их использование другими модулями NgModules. Например, чтобы использовать службу маршрутизатора в своем приложении, мы импортируем [Router](https://angular.io/api/router/Router) файл NgModule.

Организация кода в отдельные функциональные модули помогает в управлении разработкой сложных приложений и в проектировании с возможностью повторного использования. Кроме того, этот метод позволяет использовать отложенную загрузку, то есть загрузку модулей по запросу, чтобы минимизировать объем кода, который необходимо загружать при запуске.

Метаданные NgModule.

NgModule определяется классом с декоратором. Наиболее важные свойства заключаются в следующем.

@[NgModule](https://angular.io/api/core/NgModule)(…)

* declarations: [Компоненты](https://angular.io/guide/architecture-components) , директивы и каналы, принадлежащие этому модулю.
* exports: Подмножество объявлений, которые должны быть видимыми и используемыми в шаблонах компонентов других модулей.
* imports: Другие модули, экспортированные классы которых необходимы для шаблонов компонентов, объявленных в этом NgModule.
* providers: Провайдеры [сервисов,](https://angular.io/guide/architecture-services) которые этот NgModule вносит в глобальный набор сервисов; они становятся доступными во всех частях приложения.
* bootstrap: Главное представление приложения, называемое корневым компонентом, в котором размещены все остальные представления приложения. Только корневой модуль NgModule должен устанавливать bootstrap свойство.

Простое определение NgModule представлено на рисунке 1.3.2

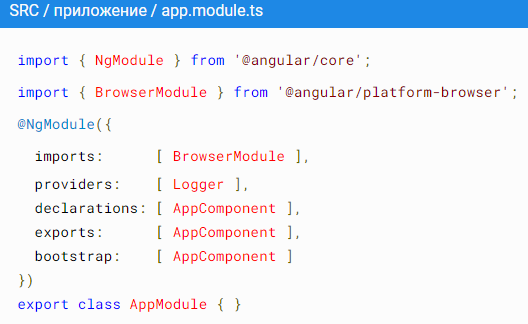


Рисунок 1.3.2 – Простое определение NgModule.

1.3.3.2 Компоненты

Каждое приложение Angular имеет по крайней мере один компонент, корневой компонент, который соединяет иерархию компонентов с объектной моделью документа страницы (DOM). Каждый компонент определяет класс, содержащий данные и логику приложения, и связанный с шаблоном HTML, который определяет представление, которое будет отображаться в целевой среде.

Декоратор идентифицирует класс непосредственно под ним в качестве компонента, а также предоставляет шаблон и соответствующие метаданные уникальные для каждого компонента.

Метаданные Component.

@[Component](https://angular.io/api/core/Component)(…)

* selector: CSS селектор для компонента.
* template/templateUrl: шаблон HTML, который компонент использует для отображения информации. В большинстве случаев этот шаблон представляет собой отдельный HTML-файл.
* styles/stylesUrl: стили для шаблона компонента. В большинстве случаев стили для шаблонa компонента определяются в отдельном файле.

Простое определение Компонента представлено на рисунке 1.3.3

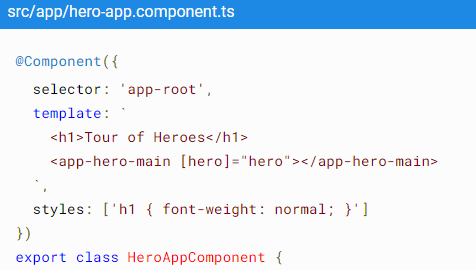


Рисунок 1.3.3 - Простое определение Компонента.

1.3.3.3 Службы

Для данных или логики, которые не связаны с конкретным представлением и которые необходимо использовать в разных компонентах, можно создать класс обслуживания. Определению класса обслуживания непосредственно предшествует декоратор «@[Injectable](https://angular.io/api/core/Injectable)()». Декоратор предоставляет метаданные, которые позволяют другим поставщикам внедрять класс обслуживания в качестве зависимостей в классе.

Внедрение зависимостей (DI) позволяет сохранять классы компонентов компактными и эффективными. Они не извлекают данные с сервера, не проверяют ввод данных пользователем и не регистрируются непосредственно в консоли; они делегируют такие задачи службам.

Метаданные декоратора Injectable.

Для служб не обязательно определять метаданные.

* providedIn: определяет место ввода сервиса в приложение.

Простое определение Сервиса представлено на рисунке 1.3.4.

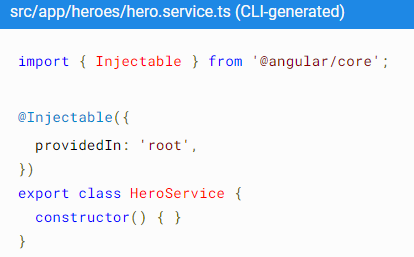


Рисунок 1.3.4 - Простое определение Сервиса

1.3.3.4 Шаблоны

Шаблон сочетает HTML с разметкой Angular, которая может изменять элементы HTML перед их отображением. Директивы шаблона обеспечивают логику программы, а разметка привязки связывает данные вашего приложения и DOM. Есть два типа привязки данных:

* Привязка событий позволяет приложению реагировать на ввод данных пользователем в целевой среде, обновляя данные приложения.
* Привязка свойств позволяет интерполировать значения, вычисленные на основе данных приложения, в HTML.

Перед отображением представления Angular оценивает директивы и разрешает синтаксис привязки в шаблоне для изменения элементов HTML и DOM в соответствии с данными и логикой вашей программы. Angular поддерживает двустороннюю привязку данных, что означает, что изменения в DOM, такие как выбор пользователя, также отражаются в данных программы.

1.3.3.5 Маршрутизация

Angular [Router](https://angular.io/api/router/Router)NgModule предоставляет службу, которая позволяет определять путь навигации между различными состояниями приложения и просматривать иерархии в вашем приложении. Он основан на знакомых соглашениях о навигации в браузере:

* Введите URL-адрес в адресную строку, и браузер перейдет на соответствующую страницу.
* Щелкните по ссылки на странице, и браузер перейдет на новую страницу.
* Нажмите кнопки браузера «Назад» и «Вперед», и браузер будет перемещаться вперед и назад по истории просмотренных вами страниц.

Маршрутизатор сопоставляет URL-подобные пути с представлениями, а не со страницами. Когда пользователь выполняет действие, такое как щелчок по ссылке, которое загружает новую страницу в браузере, маршрутизатор перехватывает поведение браузера и показывает или скрывает иерархии представлений.

Если маршрутизатор определяет, что текущее состояние приложения требует определенных функций, а определяющий его модуль не загружен, маршрутизатор может выполнить отложенную загрузку модуля по запросу.

Маршрутизатор интерпретирует URL-адрес ссылки в соответствии с правилами навигации по представлениям приложения и состоянием данных. Есть возможность переходить к новым представлениям, когда пользователь нажимает кнопку или выбирает из выпадающего списка, или в ответ на какой-либо другой стимул из любого источника. Маршрутизатор регистрирует активность в истории браузера, поэтому кнопки «назад» и «вперед» также работают.

Чтобы определить правила навигации, необходимо связать пути навигации с компонентами. Путь использует синтаксис, подобный URL-адресу, который интегрирует данные программы, почти так же, как синтаксис шаблона интегрирует представления с данными вашей программы. Затем можно применить логику программы, чтобы выбрать, какие представления отображать или скрывать в ответ на ввод данных пользователем и разработанные правила доступа.

1.3.4 NestJs

Nest (NestJS) – это платформа для создания эффективных масштабируемых серверных приложений [Node.js.](https://nodejs.org/)Он использует прогрессивный JavaScript, построен с использованием [TypeScript](http://www.typescriptlang.org/) и полностью поддерживает его (но при этом позволяет разработчикам кодировать на чистом JavaScript) и сочетает в себе элементы ООП (объектно-ориентированное программирование), FP (функциональное программирование) и FRP (функциональное реактивное программирование).

Под капотом Nest использует надежные фреймворки HTTP-серверов, такие как [Express](https://expressjs.com/) (по умолчанию), и при желании также может быть настроен на использование [Fastify](https://github.com/fastify/fastify) !

Nest обеспечивает уровень абстракции выше этих общих фреймворков Node.js (Express / Fastify), но также предоставляет их API напрямую разработчику.

Nest предоставляет готовую архитектуру приложений, которая позволяет разработчикам и командам создавать хорошо тестируемые, масштабируемые, слабо связанные и легко обслуживаемые приложения. Архитектура в значительной степени вдохновлена ​​Angular.

1.3.4.1 Установка

Для разработки Nest.js приложений необходимо настроить среду разработки.

Первое – установка Node.js и пакетного менеджера по вкусу (npm, yarn)

Для установки Node.js на платформу с операционной системой Windows необходимо скачать и запустить пакет установки с официального сайта «https://nodejs.org/en/download/».

Пакетный менеджер npm предоставляется вместе с пакетом Node.js.

Основное – установка Nest CLI (command line interface). Для этого в командной строке необходимо выполнить команду «npm i -g @nestjs/cli».

После установки уже можно создать новый проект командой

«nest new [название проекта]».

Для запуска тестового сервера необходимо выполнить команду «nest start» либо «npm run start» находясь в директории проекта. Запущенный сервер доступен по адресу «localhost:3000» или «127.0.0.1:3000».

1.3.4.2 Контроллер

Контроллеры отвечают за обработку входящих запросов и возврат ответов клиенту.

Цель контроллера - получать конкретные запросы для приложения. Механизм маршрутизации контролирует, какой контроллер принимает какие запросы. Часто у каждого контроллера есть более одного маршрута, и разные маршруты могут выполнять разные действия.

Для создания базового контроллера используются классы и декораторы. Декораторы связывают классы с необходимыми метаданными и позволяют Nest создавать карту маршрутизации (связывать запросы с соответствующими контроллерами).

@Controller()декоратор, требуется для определения базового контроллера.

Метадата Этого декоратора это префикс пути маршрута. Использование префикса пути в @Controller() декораторе позволяет легко сгруппировывать наборы связанных маршрутов и минимизировать повторяющийся код.

Простой контроллер представлен на рисунке 1.3.5

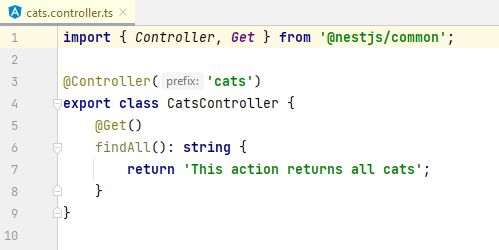


Рисунок 1.3.5 – простой Nest контроллер.

@Get() Метод запроса HTTP декоратор до findAll()метода говорит Nest создать обработчик для конкретной конечной точки для HTTP - запросов. Конечная точка соответствует методу HTTP-запроса (в данном случае GET) и пути маршрута. Путь маршрута для обработчика определяется путем конкатенации (необязательного) префикса, объявленного для контроллера, и любого пути, указанного в декораторе запроса. Т.е. «GET /cats/».

1.3.4.3 Провайдеры

Провайдеры – это простые классы JavaScript, объявленные как providers в [модуле](https://docs.nestjs.com/modules). Единственное отличие — это использование @Injactable() декоратора.

В Nest, благодаря возможностям TypeScript, чрезвычайно легко управлять зависимостями, потому что они внедряются только по типу. Например для внедрения сервиса CatService необходимо написать лишь одну строку в конструкторе класса в который мы внедряем сервис: «constructor(private catsService: CatsService) {}».

У провайдеров обычно есть время жизни («область действия»), синхронизированное с жизненным циклом приложения. Когда приложение загружается, каждая зависимость должна быть внедрена, и поэтому должен быть создан экземпляр каждого провайдера. Точно так же, когда приложение завершает работу, каждый провайдер будет уничтожен.

Чтобы указать, что поставщик является необязательным, используется @Optional() декоратор в подписи конструктора.

1.3.4.3 Модули

Модуль - это класс, помеченный @Module() декоратором. @Module() Декоратор предоставляет метаданные , которые Nest использует для организации структуры приложения.

Каждое приложение имеет по крайней мере один модуль, корневой модуль. Корневой модуль является отправной точкой, которую Nest использует для построения графа приложения - внутренней структуры данных, которую Nest использует для разрешения взаимосвязей и зависимостей между модулями и поставщиками. Хотя теоретически очень маленькие приложения могут иметь только корневой модуль, это не типичный случай.

@Module() Декоратор принимает один объект, свойство которого описывает модуль:

* providers: провайдеры, экземпляры которых будут созданы инжектором Nest и которые могут использоваться как минимум в этом модуле.
* controllers: набор контроллеров, определенных в этом модуле, которые должны быть созданы
* imports: список импортированных модулей, экспортирующих поставщиков, которые требуются в этом модуле
* exports: подмножество тех providers, которые предоставляются этим модулем и должны быть доступны в других модулях, которые импортируют этот модуль.

По умолчанию модуль инкапсулирует провайдеров. Это означает, что невозможно внедрить поставщиков, которые не являются непосредственно частью текущего модуля и не экспортируются из импортированных модулей.

В Nest модули по умолчанию являются одиночными, поэтому можно легко использовать один и тот же экземпляр любого провайдера для нескольких модулей. Каждый модуль автоматически становится совместно используемым модулем. После создания его можно повторно использовать в любом модуле.

Простое представление модуля предоставлено на рисунке 1.3.6

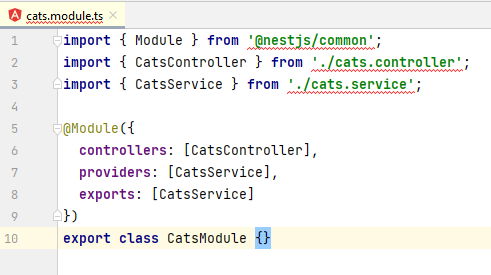


Рисунок 1.3.6 – Простой Nest модуль

Любой модуль, который импортирует, CatsModule имеет доступ к CatsService и будет использовать один и тот же экземпляр со всеми другими модулями, которые его импортируют.

В отличие от Nest, [Angular](https://angular.io/) providers регистрируются в глобальной области видимости. После определения они доступны повсюду. Однако Nest инкапсулирует поставщиков внутри области видимости модуля. Что бы сделать модуль глобальным нужно добавить @Global() декоратор. @Global() Декоратор делает модуль глобальной области видимости. Глобальные модули должны регистрироваться только один раз, обычно корневым или основным модулем.

Модульная система Nest включает мощную функцию, называемую динамическими модулями. Эта функция позволяет легко создавать настраиваемые модули, которые могут динамически регистрировать и настраивать поставщиков.  пример определения динамического модуля представлен на рисунке 1.3.7.

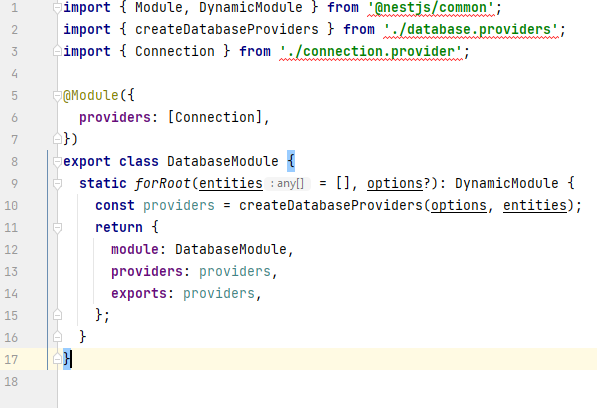


Рисунок 1.3.7 – динамический Nest модуль

Чтобы импортировать и настроить динамический модуль нужно в целевом модуле в imports написать «DatabaseModule.forRoot([данные которые принимает функция])».

1.3.4.4 Экраны

Экран - это класс, помеченный @Injectable()декоратором. Охранники должны реализовать CanActivateинтерфейс.

У экранов одна единственная ответственность. Они определяют, будет ли обработан данный запрос обработчиком маршрута или нет, в зависимости от определенных условий (например, разрешений, ролей, списков контроля доступа и т. Д.), Присутствующих во время выполнения. Это часто называют авторизацией. Авторизация (и ее родственник, аутентификация , с которой она обычно взаимодействует) обычно обрабатывается [промежуточным программным обеспечением](https://docs.nestjs.com/middleware) в традиционных приложениях Express. Промежуточное ПО - прекрасный выбор для аутентификации, поскольку такие вещи, как проверка токена и присоединение свойств к request объекту, не сильно связаны с конкретным контекстом маршрута (и его метаданными).

Простой пример реализации экрана аутентификации представлен на рисунке 1.3.8

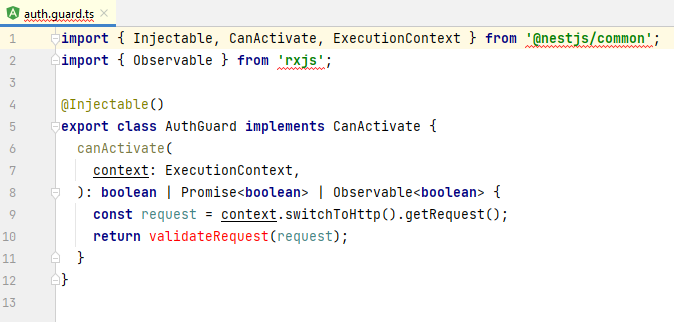


Рисунок 1.3.8 – экран аутентификации Nest.

Логика внутри validateRequest() функции может быть настолько простой или сложной, насколько это необходимо.

Каждый экран должен выполнять canActivate() функцию. Эта функция должна возвращать логическое значение, указывающее, разрешен ли текущий запрос или нет. Он может возвращать ответ синхронно или асинхронно (через Promiseили Observable). Nest использует возвращаемое значение для управления следующим действием:

* если он вернется true, запрос будет обработан.
* если он вернется false, Nest отклонит запрос.

1.3.4.5 Перехватчики

Перехватчик - это класс, помеченный @Injectable() декоратором. Перехватчики должны реализовывать NestInterceptor интерфейс.

Перехватчики обладают набором полезных возможностей, вдохновленных техникой аспектно-ориентированного программирования (АОП). Они позволяют:

* привязать дополнительную логику до / после выполнения метода
* преобразовать результат, возвращенный функцией
* преобразовать исключение, созданное функцией
* расширить базовое поведение функции
* полностью переопределить функцию в зависимости от конкретных условий (например, для целей кеширования).

Каждый перехватчик реализует intercept() метод, который принимает два аргумента. Первый - это ExecutionContext инстанс (точно такой же объект, что и у экранов). ExecutionContext наследуется от ArgumentsHost.

Второй аргумент – это CallHandler. CallHandlerИнтерфейс реализует handle() метод, который можно использовать для вызова метода обработчика маршрута в какой - то момент в вашей перехватчика. Если не вызывать handle() метод в своей реализации intercept()метода, метод обработчика маршрута вообще не будет выполняться.

Простой пример перехватчика представлен на рисунке 1.3.9

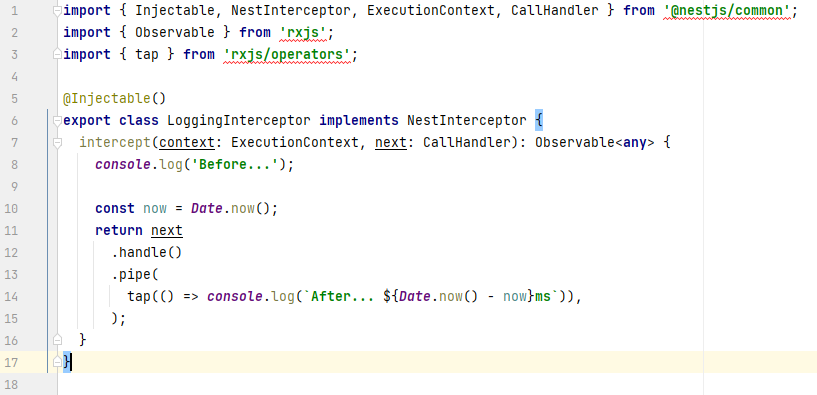


Рисунок 1.3.9 – Nest перехватчик

1.3.5 TypeScript

Весь мой код написан на TypeScript так как он является основным языком как для Angular, так и для NestJs.

По определению «TypeScript – это JavaScript для разработки масштабного приложения».

TypeScript – это строго типизированный объектно-ориентированный компилируемый язык. Он был разработан Андерсом Хейлсбергом (разработчиком C #) в Microsoft. TypeScript – это и язык, и набор инструментов. TypeScript - это типизированный надмножество JavaScript, скомпилированного в JavaScript. Другими словами, TypeScript – это JavaScript плюс некоторые дополнительные функции.

TypeScript превосходит свои другие аналоги, такие как языки программирования CoffeeScript и Dart, в том смысле, что TypeScript является расширенным JavaScript. Напротив, такие языки, как Dart и CoffeeScript, сами по себе являются новыми языками и требуют среды исполнения, зависящей от языка.

Преимущества использования TypeScript:

* Компиляция - JavaScript – это интерпретируемый язык. Следовательно, его необходимо запустить, чтобы проверить, что он действителен. Это означает, что вы пишете весь код только для того, чтобы он не работал, в случае ошибки. Следовательно, вам придется часами пытаться найти ошибки в коде. Транспилятор TypeScript предоставляет функцию проверки ошибок. TypeScript скомпилирует код и сгенерирует ошибки компиляции, если обнаружит какие-либо синтаксические ошибки. Это помогает выделить ошибки перед запуском скрипта.
* Сильная статическая типизация - JavaScript не является строго типизированным. TypeScript поставляется с дополнительной системой статической типизации и вывода типов через TLS (TypeScript Language Service). Тип переменной, объявленной без типа, может быть определен TLS на основе ее значения.
* TypeScript поддерживает определения типов для существующих библиотек JavaScript. Файл определения TypeScript (с расширением .d.ts ) предоставляет определение для внешних библиотек JavaScript. Следовательно, код TypeScript может содержать эти библиотеки.
* TypeScript поддерживает такие концепции объектно-ориентированного программирования, как классы, интерфейсы, наследование и т. Д.

Компоненты TypeScript.

В основе TypeScript есть следующие три компонента:

* Язык - он состоит из синтаксиса, ключевых слов и аннотаций типов.
* Компилятор TypeScript - компилятор TypeScript (tsc) преобразует инструкции, написанные на TypeScript, в его эквивалент JavaScript.
* Языковая служба TypeScript - «Языковая служба» предоставляет дополнительный уровень вокруг основного конвейера компилятора, который представляет собой приложения, подобные редактору. Языковая служба поддерживает общий набор типичных операций редактора, таких как завершение операторов, справка по подписи, форматирование и выделение кода, раскрашивание и т. Д.

TypeScript проверяет программу на наличие ошибок перед выполнением и делает это в зависимости от типов значений, это средство проверки статического типа.

1.3.5.1 JavaScript

JavaScript (также известный как ECMAScript) начал свою жизнь как простой язык сценариев для браузеров. В то время, когда он был изобретен, ожидалось, что он будет использоваться для коротких фрагментов кода, встроенных в веб-страницу - написание более нескольких десятков строк кода было бы несколько необычным. Из-за этого ранние веб-браузеры выполняли такой код довольно медленно. Однако со временем JS становился все более популярным, и веб-разработчики начали использовать его для создания интерактивных приложений.

Разработчики веб-браузеров отреагировали на это возросшее использование JS, оптимизировав свои механизмы выполнения (динамическая компиляция) и расширив возможности его использования (добавив API), что, в свою очередь, заставило веб-разработчиков использовать его еще больше. На современных веб-сайтах ваш браузер часто запускает приложения, содержащие сотни тысяч строк кода. Это долгий и постепенный рост «Интернета», который начинается с простой сети статических страниц и превращается в платформу для разнообразных приложений всех видов.

Более того, JS стал достаточно популярным, чтобы его можно было использовать вне контекста браузеров, например, при реализации серверов JS с использованием node.js. Природа JS «run anywhere» делает его привлекательным выбором для кроссплатформенной разработки.

Благодаря всему этому у нас есть язык, который был разработан для быстрого использования, а затем превратился в полноценный инструмент для написания приложений с миллионами строк. Это приводит к неожиданным последствиям. Например:

* Оператор равенства JavaScript (==) принудительно использует свои аргументы, что приводит к неожиданному поведению (рисунки 1.3.10, 1.3.11)
* JavaScript также позволяет получить доступ к свойствам, которых нет (рисунок 1.3.12)

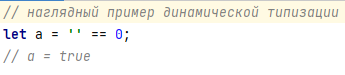


Рисунок 1.3.10 – наглядный пример проблем динамической типизации.

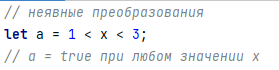


Рисунок 1.3.11 – неявные преобразования.

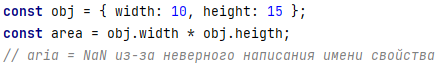


Рисунок 1.3.12 – обращение к несуществующему свойству.

1.3.5.2 Синтаксис

TypeScript – это язык, который является *надмножеством* JavaScript: поэтому синтаксис JS является допустимым TS. Синтаксис относится к способу написания текста для формирования программы. Поэтому тут разберем синтаксис JS и надстройки TS.

Рассмотрим базовые лексические структуры языка.

* Символы: при написании программ на JavaScript используется набор символов Юникода. Юникод является надмножеством кодировок ASCII и Latin-1 и поддерживает практически все письменные языки, имеющиеся на планете. Стандарт ECMAScript 3 требует, чтобы реализации JavaScript обеспечивали поддержку стандарта Юникода версии 2.1 или выше, а стандарт ECMAScript 5 требует, чтобы реализации обеспечивали поддержку стандарта Юникода версии 3 или выше.
* Чувствительность к регистру: JavaScript – это язык, чувствительный к регистру символов. Это значит, что ключевые слова, имена переменных и функций и любые другие идентификаторы языка должны всегда содержать одинаковые наборы прописных и строчных букв.
* Пробелы, переводы строк и символы управления форматом: JavaScript игнорирует пробелы, которые могут присутствовать между лексемами в программе. Кроме того, JavaScript также по большей части игнорирует символы перевода строки. Поэтому пробелы и символы перевода строки могут без ограничений использоваться в исходных текстах программ для форматирования и придания им удобочитаемого внешнего вида.
* Необязательные точки с запятой: Как и в других языках программирования, для отделения инструкций друг от друга в языке JavaScript используется точка с запятой (;). Использование точек с запятой имеет важное значение для ясного выражения намерений программиста: без этого разделителя по ошибке можно принять конец одной инструкции за начало следующей и наоборот. Обычно в JavaScript точку с запятой между инструкциями можно не ставить, если они находятся в разных строках. (Точку с запятой можно также опустить в конце программы или если следующей лексемой в программе является закрывающая фигурная скобка }.) Многие программисты на JavaScript используют точки с запятой для явного обозначения концов инструкций, даже если в этом нет необходимости.
* Комментарии: JavaScript поддерживает два способа оформления комментариев. Любой текст между символами // и концом строки рассматривается как комментарий и игнорируется JavaScript. Любой текст между символами /\* и \*/ также рассматривается как комментарий. Эти комментарии могут состоять из нескольких строк, но не могут быть вложенными.
* Идентификаторы и зарезервированные слова: Идентификатор - это просто имя. В JavaScript идентификаторы выступают в качестве имен переменных и функций, а также меток некоторых циклов. Идентификаторы в JavaScript должны начинаться с буквы, с символа подчеркивания (\_) или знака доллара ($). Далее могут следовать любые буквы, цифры, символы подчеркивания или знаки доллара. JavaScript резервирует ряд идентификаторов, которые играют роль ключевых слов самого языка. Эти ключевые слова не могут служить идентификаторами в программах. JavaScript также резервирует некоторые ключевые слова, которые в настоящее время не являются частью языка, но которые могут войти в его состав в будущих версиях.
* Типы данных JS представлены на рисунке 1.3.13

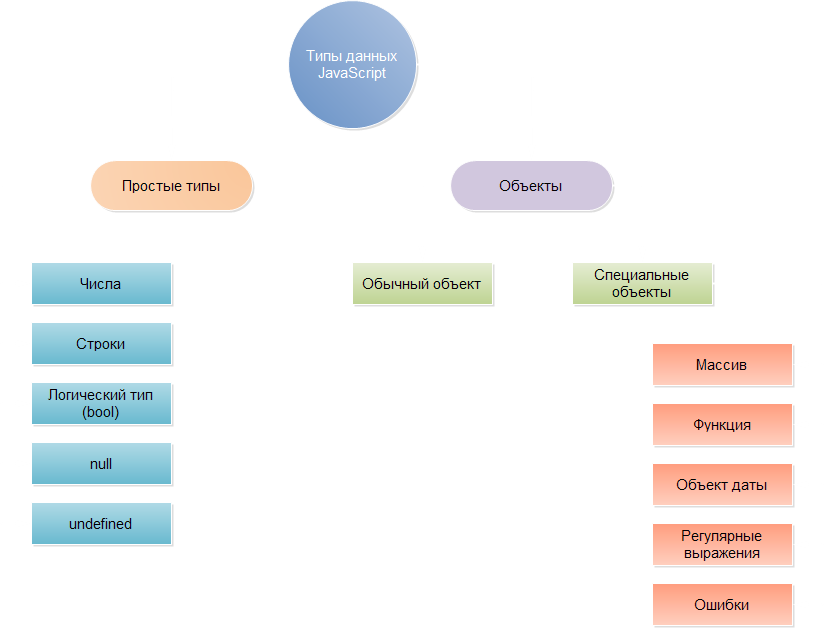


Рисунок 1.3.13 – типы данных JavaScript

1.3.6 Порядок выполнения функций событий

В скриптинге Unity есть некоторое количество функций события, которые исполняются в заранее заданном порядке по мере исполнения скрипта. Этот порядок исполнения описан ниже:

Редактор.

Reset: Reset (сброс) вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту и тогда, когда используется команда Reset.

Первая загрузка сцены.

Эти функции вызываются при запуске сцены (один раз для каждого объекта на сцене).

Awake: Эта функция всегда вызывается до любых функций Start и также после того, как префаб был вызван в сцену (если GameObject неактивен на момент старта, Awake не будет вызван, пока GameObject не будет активирован, или функция в каком-нибудь прикреплённом скрипте не вызовет Awake).   
OnEnable: (вызывается только если объект активен): Эта функция вызывается сразу после включения объекта. Это происходит при создании образца MonoBehaviour, например, при загрузке уровня или был вызван GameObject с компонентом скрипта.

OnLevelWasLoaded: This function is executed to inform the game that a new level

has been loaded.

Учтите, что для объектов, добавленных в сцену сразу, функции Awake и OnEnable для всех скриптов будут вызваны до вызова Start, Update и т.д. Естественно, для объектов вызванных во время игрового процесса такого не будет.

Перед первым обновлением кадра.

Start: Функция Start вызывается до обновления первого кадра(first frame) только если скрипт включен.

Для объектов добавленных на сцену, функция Start будет вызываться во всех скриптах до функции Update. Естественно, это не может быть обеспечено при создании объекта непосредственно во время игры.

Между кадрами.

OnApplicationPause: Эта функция вызывается в конце кадра, вовремя во время которого вызывается пауза, что эффективно между обычными обновлениями кадров. Один дополнительный кадр будет выдан после вызова OnApplicationPause, чтобы позволить игре отобразить графику, которая указывает на состояние паузы.

Порядок обновления.

Когда вы отслеживаете игровую логику и взаимодействия, анимации, позиции камеры и т.д. есть несколько разных событий, которые вы можете использовать. По общему шаблону, большая часть задач выполняется внутри функции Update, но есть также ещё другие функции, которые вы можете использовать.

FixedUpdate: Зачастую случается, что FixedUpdate вызывается чаще чем Update. FU может быть вызван несколько раз за кадр, если FPS низок и функция может быть и вовсе не вызвана между кадрами, если FPS высок. Все физические вычисления и обновления происходят сразу после FixedUpdate. При применении расчётов передвижения внутри FixedUpdate, вам не нужно умножать ваши значения на Time.deltaTime. Потому что FixedUpdate вызывается в соответствии с надёжным таймером, независящим от частоты кадров.

Update: Update вызывается раз за кадр. Это главная функция для обновлений кадров.

LateUpdate: LateUpdate вызывается раз в кадр, после завершения Update. Любые вычисления произведённые в Update будут уже выполнены на момент начала LateUpdate. Часто LateUpdate используют для преследующей камеры от третьего лица. Если вы перемещаете и поворачиваете персонажа в Update, вы можете выполнить все вычисления перемещения и вращения камеры в LateUpdate. Это обеспечит то, что персонаж будет двигаться до того, как камера отследит его позицию.

1.3.6.1 Рендеринг

OnPreCull: Вызывается до того, как камера отсечёт сцену. Отсечение определяет, какие объекты будут видны в камере.

OnPreCull вызывается прямо перед тем, как начинается отсечение.

OnBecameVisible/OnBecameInvisible: Вызывается тогда, когда объект становится видимым/невидимым любой камере.

OnWillRenderObject: Вызывается один раз для каждой камеры, если объект в поле зрения.

OnPreRender: Вызывается перед тем, как камера начнёт рендерить сцену.

OnRenderObject: Вызывается, после того, как все обычные рендеры сцены завершатся. Вы можете использовать класс GL или Graphics.DrawMeshNow, чтобы рисовать пользовательскую геометрию в данной точке.

OnPostRender: Вызывается после того, как камера завершит рендер сцены.

OnRenderImage (только в Pro версии): Вызывается после завершения рендера сцены, для возможности пост-обработки изображения экрана.

OnGUI: Вызывается несколько раз за кадр и отвечает за элементы интерфейса (GUI). Сначала обрабатываются события макета и раскраски, после чего идут события клавиатуры/мышки для каждого события.

OnDrawGizmos Используется для отрисовки гизмо в окне Scene View в целях визуализации.

Сопрограммы.

Нормальные обновления сопрограмм запускаются после завершения из функции Update. Сопрограмма это функция, которая приостанавливает своё исполнение (yield), пока данные YieldInstruction не завершатся. Разные способы использования сопрограмм:

yield Сопрограмма продолжит выполнение, после того, как все Update функции были вызваны в следующем кадре.

yield WaitForSeconds Продолжает выполнение после заданной временной задержки, и после все Update функций, вызванных в итоговом кадре.

yield WaitForFixedUpdate Продолжает выполнение после того, как все функции FixedUpdate были вызваны во всех скриптах

yield WWW продолжает выполнение после завершения WWW-загрузки.

yield StartCoroutine сцепляет сопрограмму, и будет ждать, пока не завершится сопрограмма MyFunc.

Когда объект разрушается.

OnDestroy: Эта функция вызывается после всех обновлений кадра в последнем кадре объекта, пока он ещё существует (объект может быть уничтожен при помощи Object.Destroy или при закрытии сцены).

При выходе.

Эти функции вызываются во всех активных объектах в вашей сцене:

OnApplicationQuit: Эта функция вызывается для всех игровых объектов перед тем, как приложение закрывается. В редакторе вызывается тогда, когда игрок останавливает игровой режим. В веб-плеере вызывается по закрытия веб окна.

OnDisable: Эта функция вызывается, когда объект отключается или становится неактивным.

Блок-схема жизненного цикла скрипта.

Следующая диаграмма совмещает порядок и повтор функций событий в течение жизни скрипта.

Блок схема жизненного цикла скрипта представлена на рисунке 1.2.

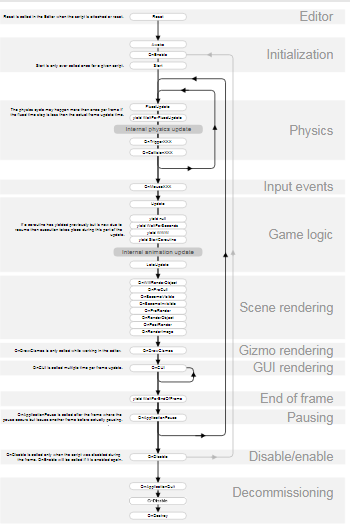


Рисунок 1.2– Блок-схема жизненного цикла скрипта

1.3.7 Общие функции

Некоторые функции в справке по скриптам (например, различные функции GetComponent) перечислены в варианте, который содержит букву T или имя типа в угловых скобках после имени функции: -

//C#

void FuncName<T>();

//JS

function FuncName.<T>(): T;

Они известны как общие функции. Их значимость для программирования заключается в том, чтобы указать типы параметров и/или возвращаемого типа при вызове функции. В JavaScript, это может быть использовано, чтобы обойти ограничения динамической типизации: -

// The type is correctly inferred since it is defined in the function call.

//In C#

var obj = GetComponent<Rigidbody>();

//In JS

var obj = GetComponent.<Rigidbody>();

В C# это может сократить количество кода: -

Rigidbody rb = go.GetComponent<Rigidbody> ();

// ...as compared with:

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));

Любая функция, имеющая общий вариант, который указан на своей странице справки, позволяет использовать специальный синтаксис вызова.

1.3.8 Важные классы

Это некоторые из наиболее важных классов, которые вы будете использовать при написании сценариев в Unity. Они охватывают некоторые основные области скриптовых систем Unity и являются хорошей отправной точкой для поиска доступных функций и событий.

Описание классов представлено на рисунке 1.3.

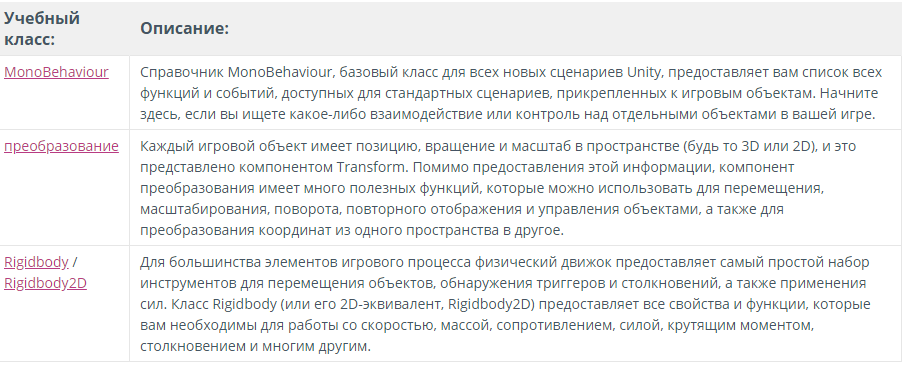


Рисунок 1.3– Описания классов

Unity события.

UnityEvents это способ позволяющий поддерживать управляемую пользователем функцию обратного вызова от момента редактирования до момента запуска без необходимости дополнительного программирования и конфигурации скриптов.

UnityEvents полезны для целого ряда вещей:

Содержит управляемую пользователем функцию обратного вызова

Системы связей

Постоянная функция обратного вызова

Предварительно настроенные события вызова

UnityEvents могут быть добавлены к любому MonoBehavior и вызываться в коде как стандартные .net делегаты. После добавления UnityEvent в MonoBehaviour, оно отобразится в инспекторе и можно будет добавлять постоянные функции обратного вызова.

UnityEvents have similar limitations to standard delegates. That is, they hold references to the element that is the target and this stops the target being garbage collected. If you have a UnityEngine.Object as the target and the native representation disappears the callback will not be invoked.

Использование UnityEvents

Вот несколько шагов по настройке обратного вызова в редакторе:

Make sure your script imports/uses UnityEngine.Events.

Выберите иконку +, чтобы добавить слот для функции обратного вызова

Выберите объект UnityEngine.Object к которому вы хотите обратиться посредством функции обратного вызова (Для этого вы можете использовать селектор объектов)

Выберите функцию, которую вы хотите вызвать

Вы можете добавить больше одной функции обратного вызова к событию

Во время настройки UnityEvent в инспекторе, вам доступно два типа поддерживаемых функций вызова:

Статические вызовы, это - пред настроенные вызовы с предустановленными значениями, которые настраиваются в UI (пользовательском интерфейсе). Это означает, что при срабатывании функции обратного вызова, целевая функция вызывается с аргументами предварительно введенными в UI. Динамические вызовы при срабатывании используют аргументы получаемые из кода, и ограничены типом вызванного UnityEvent. UI фильтрует функции обратного вызова и показывает только те динамические вызовы, которые действительны для данного UnityEvent.

Общие UnityEvents

По умолчанию UnityEvent в Monobehaviour связывает динамически функции без аргументов. Это не обязательно должно быть так, потому что UnityEvents поддерживает связь функций содержащих до 4-х аргументов. Чтобы сделать это, вам необходимо переопределить стандартный класс UnityEvents как поддерживающий множественные аргументы. Сделать это довольно просто:

[Serializable]

public class StringEvent : UnityEvent <string> {}

Потом это может быть вызвано с помощью вызова функции Invoke со ‘строкой(string)’ в качестве аргумента.

UnityEvents могут быть переопределены с наличием до 4-х аргументов в их общем определении. Добавление экземпляра этого класса к вашему, вместо базового UnityEvent, даст возможность функции обратного вызова связывать динамические строковые функции.

1.3.9 Понимание векторной арифметики

Векторная арифметика - основа 3D графики, физики и анимации, и, для получения максимальной отдачи от Unity, весьма полезно досконально разбираться в этой теме. Ниже приведены описания основных операций и несколько советов о том, для чего они могут быть использованы.

Сложение.

При сложении 2 векторов результат эквивалентен тому, что получится если исходные векторы принять за следующие друг за другом “шаги”. Заметьте, что порядок двух слагаемых не важен, т.к. в любом случае результат будет одинаковый.

График сложение представлен на рисунке 1.4.

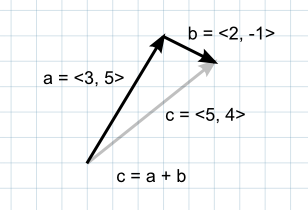


Рисунок 1.4– График сложения

Если первый вектор принять за точку в пространстве, то второй вектор можно интерпретировать как сдвиг или “прыжок” из этой точки. Например, чтобы для поиска точки 5-тью единицами выше точки на земле, вы могли бы использовать следующий расчёт: -

var pointInAir = pointOnGround + new Vector3(0, 5, 0);

Вычитание.

Вычитание векторов чаще всего используется, чтобы узнать расстояние и направление одного объекта относительно другого. Заметьте, что при вычитании порядок параметров имеет значение: -

// The vector d has the same magnitude as c but points in the opposite direction.

var c = b - a;

var d = a - b;

Как и с обычными числами, прибавление отрицательного вектора - это то же

самое, что и вычитание положительного.

// These both give the same result.

var c = a - b;

var c = a + -b;

Отрицательный вектор имеет ту же величину, что и исходный вектор, и лежит на той же прямой, только в обратном направлении.