

MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

UNIVERSITATEA DE STAT „ALECU RUSSO” DIN BĂLȚI

FACULTATEA DE ȘTIINȚE REALE, ECONOMICE ȘI ALE MEDIULUI

CATEDRA DE MATEMATICĂ ȘI INFORMATICĂ

# **ANGULAR ELEMENTS ȘI LAZY-LOADING ÎN PARADIGMĂ SSR**

## **TEZĂ DE MASTER**

**Autor:**

Studentul grupei AW21M

**Iurii ROSCA**

---

**Conducător științific:**

**Dumitru STOIAN**

lect. univ.

---

**Corina NEGARA**

dr., conf. univ.

---

**Bălți 2021**

МИНИСТЕРСТВО ОБРАЗОВАНИЯ, КУЛЬТУРЫ И ИССЛЕДОВАНИЯ РЕСПУБЛИКИ

МОЛДОВА

БЕЛЫЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ “А. РУССО”

ФАКУЛЬТЕТ ТОЧНЫХ НАУК, ЭКОНОМИКИ И ОКРУЖАЮЩЕЙ СРЕДЫ

КАФЕДРА МАТЕМАТИКИ И ИНФОРМАТИКИ

# **ANGULAR ELEMENTS И LAZY-LOADING В ПАРАДИГМЕ SSR**

## **МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**Автор:**

Студент группы AW21M

**Юрий РОШКА**

---

**Научные руководители:**

**Дмитрий СТОЯН**

лект. унив.

---

**Карина НЕГАРА**

др., конф. унив.

---

MINISTRY OF EDUCATION CULTURES AND RESEARCH OF THE REPUBLIC OF  
MOLDOVA

"ALECURUSSO" STATE UNIVERSITY OF BĂLȚI

FACULTY OF REAL, ECONOMIC AND ENVIRONMENTAL SCIENCES

DEPARTMENT OF MATHEMATICS AND COMPUTING

# **ANGULAR ELEMENTS AND LAZY-LOADING IN SSR PARADIGM**

## **MASTER THESIS**

**Author:**

The student of the group AW21M

**Iurii ROSCA**

---

**Scientific leader:**

**Dumitru STOIAN**

lect. univ.

---

**Corina NEGARA**

dr., conf. univ.

---

**BĂLȚI, 2020**

Controlată:

Data \_\_\_\_\_

Conducător științific:

Dumitru Stoian, lect. univ. \_\_\_\_\_

Corina Negara, dr., conf. univ. \_\_\_\_\_

Aprobată

și recomandată pentru susținere

la ședința Catedrei de matematică și informatică

Proces verbal nr. \_\_\_\_\_ din \_\_\_\_\_

Șeful catedrei de matematică și informatică

dr., conf. univ. C. Negara

\_\_\_\_\_

Aprobat  
Şeful catedrei de matematică şi informatică  
dr., conf. univ. C. Negara  
“ ” 202

**Graficul calendaristic de executare a tezei de licenţă**

Tema tezei de licenţă \_\_\_\_\_

confirmată prin ordinul rectorului USARB nr. \_\_\_\_ din „\_\_\_\_\_”

Termenul limită de prezentare a tezei de licenţă la Catedra de matematică şi informatică  
„\_\_\_\_\_”.

**Etapele executării tezei de licenţă:**

Etapele	Termenul de realizare	Viza de executare
1. Stabilirea temei; fixarea obiectivelor; selectarea surselor de informare		
2. Investigaţia cadrului teoretic al cercetării (teoria problemei); expunerea cadrului teoretic al cercetării;		
3. Întocmirea problemei cercetării; stabilirea tipului de cercetare; elaborarea ipotezelor		
4. Specificarea unităţilor (populaţiei) studiate; construcţia variabilelor (descrierea calitativă); cuantificarea (descrierea cantitativă)		
5. Alegerea metodelor de cercetare; stabilirea tehnicilor şi procedeelor de lucru - în conformitate cu decizia despre caracterul lucrării: experiment de constatare, experiment formativ etc.		
6. Culegerea datelor; selectarea modalităţilor de prelucrare a datelor; stocarea datelor; prelucrarea datelor; analiza datelor (verificarea ipotezelor)		
7. Rezolvarea aspectelor de grafică şi design la calculator; interpretarea rezultatelor		
8. Formularea propunerilor de soluţionare a problemei cercetării vizate în lucrare; elaborarea concluziilor şi a recomandărilor practice		
9. Susţinerea preventivă a tezei		

Student (a) \_\_\_\_\_

(semnătura)

Conducători ştiinţifici \_\_\_\_\_

(semnătura)

## ANOTARE

Rosca Iurii

### ANGULAR ELEMENTS AND LAZY-LOADING IN SSR PARADIGM

Teză de master. Bălți, 2020

*Structura tezei:* introducere, patru capitole, concluzii generale și recomandări, 19 titluri ale bibliografiei, 60 de pagini ale textului principal, 32 de figuri.

*Cuvinte cheie:* SEO, SPA, JS, Angular, Componente web, SSR, Lazy-Loading, Angular Elements, Angular Universal, Npm, NodeJS, RxJS, site web, redare, aplicație web, componentă, modul.

*Zona de studiu:* aplicație web. Redare pe server, Seo, Lazy-loading, Web components.

*Scopul cercetării:* optimizarea dezvoltării și a produselor finite ale aplicațiilor web pe scară largă pe o singură pagină.

*Obiective de cercetare:* (1) Analiza surselor speciale de informații. (2) Cercetarea cadrului unghiular. (3) Cercetarea componentelor web și a tehnologiilor conexe. (4) Cercetare Site Site Rendering și tehnologii conexe. (5) Cercetare de încărcare leneșă și tehnologii conexe. (6) Cercetarea Universului Angular și a tehnologiilor conexe. (7) Cercetarea elementelor unghiulare și a tehnologiilor conexe. (8) Cercetarea componentelor necesare unei aplicații cu drepturi depline în Angular: Material Design etc. (9) Dezvoltarea unei aplicații demo.

*Metodologia cercetării a constat în metode teoretice:* documentație științifică, analiză literară, sinteză, comparație, interpretare, generalizare, sistematizare și descriere.

## АННОТАЦИЯ

Рошка Юрий

### ANGULAR ELEMENTS И LAZY-LOADING В ПАРАДИГМЕ SSR

Магистерская диссертация. Balti, 2018

*Структура диссертации:* введение, четыре главы, общие выводы и рекомендации, 19 названий библиографии, 60 страниц основного текста, 32 рисунков.

*Ключевые слова:* SEO, SPA, JS, JavaScript, Angular, Web components, SSR, Lazy-Loading, Angular Elements, Angular Universal, Npm, NodeJS, RxJS, сайт, рендеринг, веб-приложение, компонент, модуль.

*Область изучения:* Веб-приложение. Серверный рендеринг, Seo, Lazy-loading, Web components,.

*Цель исследования:* оптимизация разработки и готовых продуктов масштабных одностраничных веб приложений.

*Цели исследования:* (1) Анализ специальных информационных источников. (2) Исследование фреймворка Angular. (3) Исследование Web components и сопутствующие технологии. (4) Исследование Server Site Rendering и сопутствующие технологии. (5) Исследование Lazy-Loading и сопутствующие технологии. (6) Исследование Angular Universal и сопутствующие технологии. (7) Исследование Angular Elements и сопутствующие технологии. (8) Исследование необходимых составляющих полноценного приложения на Angular: Material Design и тд. (9) Разработка демонстрационного приложение.

*Методология исследования состояла из теоретических методов:* научной документации, анализа литературы, синтеза, сравнения, интерпретации, обобщения, систематизации и описания.

## ANNOTATION

Rosca Iurii

### ANGULAR ELEMENTS AND LAZY-LOADING IN SSR PARADIGM

Bachelorthesis.Balti,2018

*The structure of the thesis: introduction, four chapters, general conclusions and recommendations, 19 titles of the bibliography, 60 pages of the main text, 32 figures.*

*Keywords: SEO, SPA, JS, Angular, Web components, SSR, Lazy-Loading, Angular Elements, Angular Universal, Npm, NodeJS, RxJS, website, rendering, web application, component, module.*

*Study area: Web application. Server-side rendering, Seo, Lazy-loading, Web components.*

*Purpose of the research: optimization of development and finished products of large-scale single-page web applications.*

*Research objectives: (1) Analysis of special information sources. (2) Researching the Angular framework. (3) Research Web components and related technologies. (4) Research Server Site Rendering and related technologies. (5) Lazy-Loading Research and Related Technologies. (6) Researching Angular Universal and related technologies. (7) Researching Angular Elements and related technologies. (8) Researching the necessary components of a full-fledged application in Angular: Material Design, etc. (9) Development of a demo application.*

*The research methodology consisted of theoretical methods: scientific documentation, literature analysis, synthesis, comparison, interpretation, generalization, systematization and description.*



## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	10
3 ПРОБЛЕМЫ БОЛЬШИХ ПРОЕКТОВ.....	12
Унификация кода между проектами.....	14
Высоконагруженный интерфейс .....	15
SEO для SPA приложений .....	15
РЕНДЕРИНГ НА СТОРОНЕ СЕРВЕРА (SSR).....	17
Серверный рендеринг .....	17
Angular и SEO .....	18
ЛЕНИВАЯ ЗАГРУЗКА (LAZY-LOADING) .....	22
Общее понимание ленивой загрузки .....	22
Ленивая загрузка модулей в Angular .....	26
ANGULAR ELEMENTS ИЛИ ВЕБ-КОМПОНЕНТЫ НА ANGULAR .....	35
Общее понимание Веб-компонентов.....	35
Что такое Angular Elements.....	44
Преобразование компонентов в пользовательские элементы.....	46
ОПИСАНИЕ ПРАКТИЧЕСКОЙ ЧАСТИ .....	50
Архитектура приложения и имплементированный функционал .....	50
Модульная реализация приложения .....	51
Механизм Lazy-Load в приложении .....	52
Использование сторонних сервисов .....	54
Использование Angular Universal.....	56
Использование Angular Elements .....	58
ЗАКЛЮЧЕНИЕ.....	62
БИБЛИОГРАФИЯ .....	63

## ВВЕДЕНИЕ

Уже несколько лет стандартом и хорошим тоном веб-программирования являются Single Page Application (далее SPA). Без SPA не существовало бы настолько быстрого обмена сообщениями, на таких ресурсах как facebook, ok, vk, instagram и т.д. Но данный метод имеет ряд серьезных технических упущений. Веб приложение всегда будет загружаться целиком. Даже если некий функционал по предусмотренной внутренней логике никогда не будет востребован пользователем так или иначе обязательно будет загружен. В данной курсовой работе представлен способ избежать загрузки этого нежелательного балласта с помощью Lazy- Loading.

Angular написан на языке TypeScript (являющийся продуктом Microsoft) разработчиками Google. Одни из главных преимуществ TypeScript это его компиляция в JavaScript код читаемый даже в старых версиях браузеров и возможность явного статического назначения типов с использованием полноценных классов, а также поддержкой подключения модулей, что призвано повысить скорость разработки. Angular собирает приложение в сплошной JavaScript код, что является большой проблемой для поисковых роботов SEO. Собранный для публикации проект Angular хоть и содержит множество технологических решений, тем не менее при загрузке в браузер представляет собой практически пустую HTML страницу. Для создания SEO ориентированного продукта, в Angular существует технология генерации HTML-шаблонов на сервере - Server-Side Rendering (SSR). В контексте Angular разработки данная технология носит название - Angular Universal.

Большие IT-компании часто занимаются брендированием своих продуктов. Под этим подразумевается: одинаковые кнопки, элементы страницы, логотипы, виджеты, чаты и т.п. Но при этом из-за большого количества команд и разработчиков внутри компании, и как следствие написание продуктов на разных языках, не представляется возможным использование универсальных библиотек с этими компонентами. Для создания таких универсальных решений были придуманы веб-компоненты. Веб-компоненты поддерживаются веб-браузерами напрямую и не требуют дополнительных библиотек для работы. Веб-компоненты включают три технологии, каждая из которых может использоваться отдельно от других: Custom Elements, HTML Templates, Shadow DOM. Веб-компоненты имеют свою реализацию в Angular и носит название Angular Elements. Данная технология позволяет использовать компоненты Angular в любых веб-приложениях.

Целью работы является исследование основных проблем разработки массивных веб-приложений. В качестве платформы для реализации был выбран фреймворк Angular так как содержит в себе множество готовых необходимых решений. Область исследования включает в себя технологии, используемые в процессе frontend разработки на языке TypeScript.

Достижение целей предполагает:

- Анализ специальных информационных источников;
- Исследование фреймворка Angular;
- Исследование Web components и сопутствующие технологии;
- Исследование Server Site Rendering и сопутствующие технологии;
- Исследование Lazy-Loading и сопутствующие технологии;
- Исследование Angular Universal и сопутствующие технологии;
- Исследование Angular Elements и сопутствующие технологии;
- Исследование необходимых составляющих полноценного приложения на Angular: Material Design и тд;
- Разработка демонстрационного приложение.

Работа состоит из: введение, заключение, 5-ти глав и библиографии.

Первая глава посвящена описанию 3-х проблем, которые существуют при построении больших веб приложений.

Во второй главе представлено общее понимание технологии SSR.

В третьей главе описываются теоретические основы фреймворка Angular и технологии Lazy-Loading.

В четвертой главе расположено общее понимание Веб-компонентов и Angular Elements

В пятой главе описывается практическая часть магистерской работы.

Работа состоит из 60 страниц тематического содержания, 32 рисунка, 19 источников

## 1. ПРОБЛЕМЫ БОЛЬШИХ ПРОЕКТОВ

Работа веб-разработчиков заключается в том, чтобы выбирать правильные инструменты для любого конкретного проекта. Это может быть сложно, если важны не только насущные потребности проекта. И нужды всей команды, и если проект является частью более крупной экосистемы в компании, и как он будет поддерживаться, и как долго это нужно будет поддерживать, этот список можно ещё продолжать и продолжать. Веб-сообщество поставляет поразительное количество инструментов, библиотек и фреймворков. Порой бывает трудно успевать за всеми из них, ведь их настолько много, что этот феномен был темой разговоров в течение некоторого времени и даже сейчас. Внедрение новых инструментов происходит молниеносно. Отложив в сторону рамки на мгновение даже такая нишевая задача, как бегунки задач для создания JS - проектов, были кардинально изменены за последние несколько лет. Можно вспомнить переход с Grunt в 2012 году на Gulp всего пару лет спустя, и теперь есть тенденция к минимизации, используя Node.js. Менеджер пакетов NPM для запуска скриптов сборки. Говоря о менеджерах пакетов, разработчики колебались между NPM, Bower и Yarn для запуска интерфейсных зависимостей проектов. Инструменты сборки и менеджеры пакетов это небольшие, но важные части рабочего процесса веб-разработки. Однако такой же отток происходит и с тем, как на самом деле строятся приложения и пользовательский интерфейс, который, возможно, является самой центральной и важной частью веб-развития. Индивидуальному разработчику, это определенно может быть сложно уследить, хотя интересно изучить новый фреймворк или библиотеку. У некоторых «кривая» обучения более крутая, чем у других, и во многих случаях изучается «система» фреймворка, а не фундаментальные понятия HTML / JS / CSS. У разработчика в команде или в компании есть дополнительные проблемы. В начале проекта, необходимо договориться о том, какие инструменты будут использоваться для разработки в течение жизненного цикла проекта. В это входит: инструменты сборки, инструменты тестирования и, конечно же, любые фреймворки или библиотеки. Не каждый согласится с общим выбором. Если команда большая и работает над множеством проектов, может возникнуть соблазн позволить разработчикам в каждом проекте выбирать свои собственные инструменты. В конце концов, следует хорошо проанализировать потребности проекта и использовать соответствующие инструменты. В конце концов, используя разные инструменты и фреймворки могут обернуться проблемой для команды. Если все согласится, охотно или нет, с одними и теми же рамками, все может закончиться хорошо. Но даже в этом

случае, через два-три года, структура может устареть. Из-за этого устаревшие технологии начинают казаться немного удушающими, особенно для младших разработчиков в команде, которые хотят поддерживать свои навыки актуальными на уровне остальной части веб-сообщества. И в этом случае, организация стоит перед выбором: переделать весь стек технологий, используя новые решения или оставить старые, но сталкиваясь с восприятием не инновационного места работы. Это большая проблема, и ее решение обязательно!

## 1.1. Унификация кода между проектами

Одно из самых больших преимуществ отказа от фреймворка - возможность сосредоточиться на основных концепциях веб-разработки, а не на изучении конкретных фреймворков ведь это навыки, которые могут или не могут быть перенесены в следующую популярную структуру. Еще одно огромное преимущество - возможность попробовать небольшие библиотеки и микрофреймворки, которые решают очень специфические потребности проектов. Барьер доступа к ним и даже к новым интерфейсным инструментам сборки намного ниже, учитывая, что исключается борьба с конкретной средой разработки, предоставляемой последними популярными фреймворками [1]. Современные фреймворки чрезвычайно полезны и решают некоторые большие проблемы, но почему бы не задуматься об использовании так называемого чистого JS, учитывая желание попробовать другие вещи. Взглянув на опрос «State of JS», проведенный в 2017 году [1]. Можно заметить, что разработка без фреймворка является второй по популярности после React. Однако, не совсем понятны конкретные причины, по которым разработчики отдают предпочтение нефреймворкам, а чистому JS. Невозможно не восторгаться новыми технологиями и техниками, и как следствие будет нецелесообразно обойти вниманием одну из них. Веб-компоненты стали чем-то иным для веб-разработки. Особенно в такие моменты, когда создается некая часть приложения, как например, с использованием 3Dweb, или просто какой-то необычный «веб-эксперимент», в котором, для простоты и универсальности удобнее использовать простой JS, CSS и HTML конкретного пользовательского интерфейса. Но раньше не было отличного способа организовать код, создавая повторно используемые компоненты, при этом не возвращаясь к фреймворку. Это именно то, как веб-компоненты и новые функции JS меняют взгляд на веб-разработку [1]. Для начала следует рассмотреть, как новый синтаксис класса JS можно использовать при создании собственного настраиваемого элемента с использованием API веб-компонентов. А после продолжить изучать способы улучшения повторного использования, инкапсуляции и рабочего процесса: использование ShadowDOM, создание HTML шаблона, создание приложения только для компонентов и т.д. Однако, в настоящее время, веб-компоненты все еще являются чем-то новым, и веб-сообщество будет придумывать новые способы улучшения рабочего процесса. Без структуры исходный код превращается в так называемый «спагетти код». Написание новых функций могут быть огромной проблемой без предсказуемой организации проекта. Тем не менее, в попытке освободиться от больших, всеобъемлющих фреймворков веб-компоненты предоставляют

огромную возможность развития в этом направлении [1]. Прежде чем продолжать углубляться в данную тему имеет смысл выбрать в качестве наглядного примера, псевдо задачу, для реализации технологии. Например, средство выбора даты в браузере. Наверняка это то, с чем сталкивались, если не все, то по крайней мере большинство веб-разработчиков. Хотя сам по себе это не веб-компонент, он очень похожа концепция, если заглянуть внутрь.

## **1.2. Высоконагруженный интерфейс**

Последнее время множество людей и компаний ищут способы интересного представления своих данных для удобства их чтения и анализа. При разработке дизайна сложных систем приходится неизбежно сталкиваться с множеством пользователей или личностей, для которых это делается. Руководство, менеджеры, аналитики все эти категории пользователей имеют свои собственные потребности в данных и особенности работы. Важно идентифицировать типы пользователей заранее и организовать вокруг них задачи по информационной архитектуре и макетам. Создать приложение, которое будет удовлетворять одинаково всех пользователей - довольно сложная задача. Ещё сложнее продумать оптимизацию такого проекта, ведь всем пользователям нужны свои данные и соответственно свои интерфейсы. Если не разделять это всё на отдельные модули, на вряд ли можно достичь хороших результатов в поставленной задаче. Разделение кода процесс, который очень явно делит код на несколько частей. Это позволяет взять полный пакет и разбить его на несколько частей. В этом вся суть этого процесса, Webpack позволяет сделать это очень легко с загрузчиком для Angular. Приложение по сути становится множеством небольших приложений, которые обычно называются «модулями». Эти модули могут быть загружены по требованию [2]. В некоторых случаях какое-то содержание страницы нужно видеть не всем или не всегда, но на загрузку и генерацию этого содержания тратится много времени и ресурсов. В этом случае есть смысл сделать загрузку «ленивой», не загружая информацию, пока она не понадобится [3]. Так же исследования выявили, что 47.5% пользователи откажутся от ожидания сайта, если сайт загружается более двух-трех секунд. Пользователи интернет-магазинов, так же не станут ожидать это же время. 57% посетителей покинут сайт, в том случае если на мобильном устройстве сайт будет загружаться три секунды и более [4].

## **1.3. SEO для SPA приложений**

Уже пару лет большинство веб приложений, создаются с использованием AJAX-технологий, или же пишутся на JavaScript фреймворках. Такие приложения чаще всего имеют

одностраничную архитектуру и носят аббревиатуру SPA (Single Page Application) [5]. И хоть такой подход является более удобным и современным для разработки больших проектов, всё же имеет свои недостатки. Без дополнительного вмешательства их нельзя назвать годными для успешного продвижения. Отличием от «классических» сайтов, является то, что SPA архитектура устроена так: отрисовка страниц делается строго на клиенте. Браузер запускает JS приложение, при помощи технологии AJAX загружает содержимое этих страниц, которое необходимо пользователю [6]. Навигация так же проходит не перезагружаясь. Данная архитектура, позволяет веб приложениям работать быстрее и тратить меньше трафика.



Рис. 1.1. Сравнение между традиционной архитектурой сайта и SPA.

Загрузку и отрисовка содержимого в приложении SPA можно описать так (рис. 1.1). Браузер делает запрос на загрузку структуры сайта в HTML. Ответом получает JS приложение. Оно определяет, текущего пользователя и основываясь на его внутренних данных показывает содержимое. При помощи AJAX приложение получает нужные данные [6]. Веб приложение обрабатывает эти данные и демонстрирует их в браузере пользователю. Таким образом при использовании навигации загружается только содержимое, которое необходимо вместо целого сайта. Однако этот подход обладает рядом важных недостатков, касающихся продвижения [5]. JS код, к сожалению, пока ещё не обрабатывается поисковыми системами. К тому же их невозможно проверить на предмет ошибок специальными программами и инструментами. На текущий момент лишь сервисы Google умеют обрабатывать приложения SPA, по причине того, что они используют для данной задачи инструментарий из Chrome.



## 2. РЕНДЕРИНГ НА СТОРОНЕ СЕРВЕРА (SSR)

### 2.1. Серверный рендеринг

Когда заходит речь о серверном рендеринге веб-сайтов, обычно подразумевается, что приложение будет исполнено с помощью языка программирования, работающего на сервере. На сервере веб-страницы создаются, и выводятся уже готовыми в HTML, после чего отправляются в браузер, где отображаются напрямую [7]. Популярными примерами языков программирования, работающих по такой архитектуре, являются: PHP, Java, Python, .NET, Ruby. Преимущества рендеринга на стороне сервера заключаются в том, что основная работа происходит на сервере, это делает страницу готовой при загрузке в браузер и ему остается только показать её пользователю [7]. Такая страница на много лучше работает с индексацией поисковых систем и публикации в социальных сетях. Содержание готов к использованию, а клиенту или поисковой системе не нужно запускать какой-либо дополнительный код для анализа страницы. Недостатком серверного рендеринга является то, что страницы часто имеют только базовые возможности взаимодействия с пользователем, и когда содержимое изменяется или пользователь переходит на другую страницу, браузер должен повторно загрузить новую её. Это дает пользователю ощущение, что страница загружается медленнее, чем есть на самом деле.

Когда заходит речь о рендеринге на стороне клиента, обычно подразумевается приложение или веб-сайт, использующий JavaScript, работающий в браузер для отображения (рендеринга) страниц. Часто бывает загружается одна страница с файлом JavaScript, который создает фактическую страницу (отсюда и термин одностраничное приложение). Преимущество рендеринга на стороне клиента заключается в том, что страницы очень интерактивны. Части страницы можно перезагружать или обновляться без необходимости обновления вкладки браузера целиком [7]. Это использует меньшую пропускную способность и обычно дает пользователю ощущение что веб-сайт или приложение работают очень быстро. Сервер может быть в таком случае без шаблонизаторов, поскольку страница там не отображается, а просто обслуживает HTML, JavaScript и таблицы стилей. Это снижает нагрузку на сервер, и это приводит к наилучшей масштабируемости. Веб-сайты, созданные на стороне клиента, трудны для поисковой системы, для индексации, так как им нужно выполнить JavaScript, чтобы понять, как страница отображается. То же самое и при обмене ссылками на сайты в социальных сетях, поскольку они, как правило, статичны. Еще одна слабость в том, что размер начальной загрузки больше, и это может быть проблемой,

например, на мобильных устройствах с медленным соединением [7]. Пользователи будут видеть пустую страницу, пока все приложение загружено, и, возможно, они просто воспользуются небольшой его частью.

## 2.2. Angular и SEO

Веб-сайты состоят из кода. Код написан на языках. Большинство веб-сайтов составляют три языка. HTML создает контент. CSS делает макет, дизайн и визуальные эффекты. На этих двух языках можно создавать эстетически привлекательные, функциональные, плоские страницы, но в большинстве случаев этого недостаточно для современных продуктов [8]. С помощью JavaScript веб-сайты могут персонализировать интерактивный пользовательский опыт. Люди переходят на интересные сайты. JS делает привлекательные сайты, функциональными.

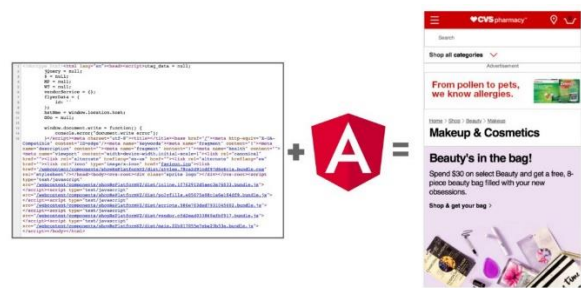


Рис. 2.1. Соотношение между изначальным кодом и конечным результатом.

Angular - это способ масштабирования JS для создания сайтов. С помощью Angular минимальное количество строк HTML-кода, отправленных с сервера, разворачивается и выполняет персонализированный интерактивный пользовательский интерфейс (рис. 2.1) [8].

	5	React	A	A <sub>2</sub>	5	V	jQuery	jQuery	plain	JS
Google	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
Bing	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
YAHOO!	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ask	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
Aol.	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
DuckDuckGo	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Yandex	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Baidu	Not indexed									

Рис. 2.2. Таблица взаимодействия поисков сервисов и фреймворков

Чтобы поисковые системы могли просматривать контент Angular, им необходимо выполнить JavaScript. Многие поисковые системы не могут отображать JavaScript (рис. 2.2).

В связи с этим 95% сайтов используют серверный рендеринг. Индексирование контента, созданного на основе JS, является хорошим бизнесом, когда модель полагается на то, чтобы являться самым надежным индексом веб-контента. Специалисты по SEO находятся в поисках решения по поводу возможностей работы Googlebot и его навыкам сканированию JS. Однако отсутствие ясности привело к предупреждению о том, что Angular может убить возможности реализации SEO для проекта. На I/O 2018 (Google I / O - фестиваль разработчиков, который проходил 8-10 мая в Shoreline Amphitheatre в Маунтин-Вью, Калифорния.) команда веб-мастеров открыто рассказала о проблемах, с которыми сталкивается Google при индексировании Angular и другого JS-контента [7]. До конференции разработчиков Google 2018 года специалисты по SEO исходили из того, что процесс работы робота Googlebot состоит из трех этапов: сканирование, индексирование и ранжирование.

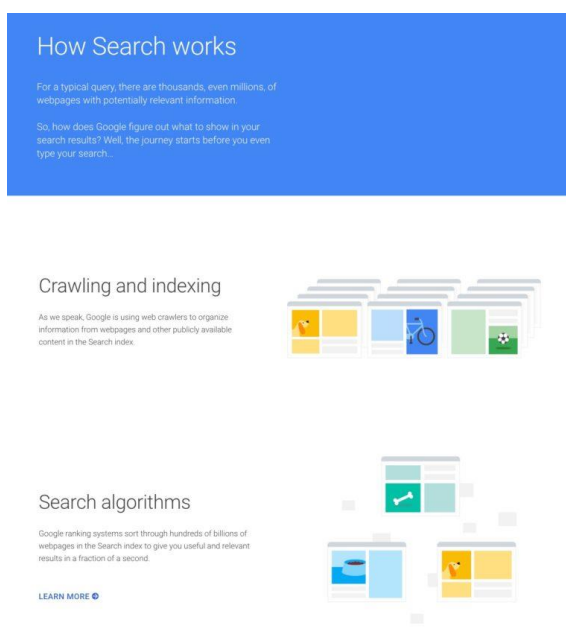


Рис. 2.3. Этапы процесса обработки страницы от Google

Даже до апреля 2019 года собственные ресурсы Google отражали простой трехэтапный процесс (рис. 2.3). В этом упрощенном процессе скрывается набор скрытых предположений:

- Робот Googlebot отображает JS при сканировании;
- Индексирование основано на отображаемом содержимом;
- Эти действия происходят одновременно в единой последовательности;
- Робот Google - все делает мгновенно.

Рендеринг - это процесс, в котором скрипты, вызываемые при первоначальном синтаксическом анализе HTML, выбираются и выполняются. Если сайт использует JavaScript,

HTML будет отличаться от DOM. Два представления одной страницы могут сильно отличаться. Первоначальный HTML состоял всего из 16 строк. После выполнения JavaScript DOM заполнен богатым контентом. Команда веб-мастеров Google пояснила, что, учитывая предположения о трехэтапном процессе и их влияние на органическую производительность, существует два этапа индексации. Первая волна индексирует страницу только на основе исходного HTML (также известного как исходный код страницы).

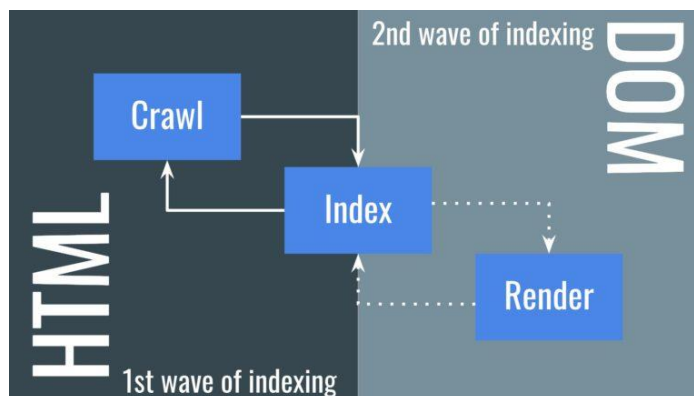


Рис. 2.4. Процессы индексирования.

Второй индекс основан на модели DOM (рис. 2.4). JavaScript - самый дорогой ресурс на сайте [7]. 1 МБ скрипта может занять 5 секунд при подключении к сети 3G. Загрузка страницы размером 1,5 МБ может стоить 0,19 доллара США. Для робота Googlebot эта стоимость зависит от ЦП для выполнения сценария.

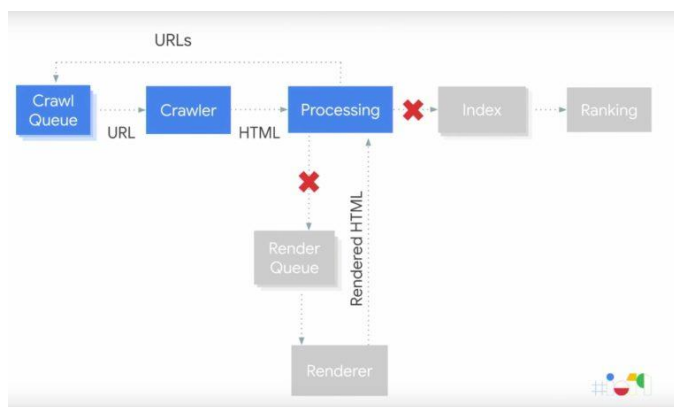


Рис. 2.5. Доступность ресурсов для индексации.

При таком большом количестве JavaScript в Интернете для механизма рендеринга робота Googlebot сформировалась буквальная очередь. Это означает, что контент, созданный с помощью JavaScript, обнаруживается роботом Googlebot только после того, как ресурсы становятся доступными. Часто технологический долг необходимо погасить, прежде чем

можно будет внести большие изменения. Одним из главных препятствий, мешающих Google понять большую часть разнообразного веб-контента, была его служба веб-рендеринга (WRS). Одним из основных компонентов поискового робота была версия Chrome, выпущенная в 2015 году. Для профессионалов SEO и разработчиков, это означает, большие базы кода полных polyfill и модифицирование ES6, а так же функциональные возможности ES5.

Пока эти идеи не удалось до конца воплотить в жизнь. Тем не менее необходимо продвигать сайты на Angular уже сейчас. Для этого существует библиотека для рендеринга приложений Angular на стороне сервера. Angular Universal создает готовую страницу на сервере и посылает её браузеру [8]. Это позволяет приложению отображаться быстрее, что дает пользователям возможность просмотреть макет приложения, прежде чем он станет полностью интерактивным. Angular Universal может сгенерировать статическую версию вашего приложения, которая легко доступна для поиска, ссылок и навигации без JavaScript. Universal также делает доступным предварительный просмотр сайта, поскольку каждый URL-адрес возвращает полностью обработанную страницу.

### 3. ЛЕНИВАЯ ЗАГРУЗКА (LAZY-LOADING)

#### 3.1. Общее понимание ленивой загрузки

Lazy loading - так же является шаблоном проектирования в компьютерном программировании для отсрочки инициализации объекта до его вызова. Применение данной концепции очень велико на сегодняшний день и наибольшая его концентрация собралась вокруг веб-программирования. Загрузка контента по частям делает возможным реализацию, очень большого функционала веб-приложения, который может довольно быстро и стабильно работать на “клиенте” [9]. Но есть и другие примеры. Один из таких примеров это, ОРМ фреймворки, которые для удобства работы с ними разработчику, загружают данные в готовые, но пустые объекты. В последствии к ним легко обращаться средствами синтаксиса языка, а также, к их сопряжениям с другими объектами [10]. Это в разы ускоряет разработку, однако нагружает “машину” и тратит ресурсы на задачи не требуемые в данный момент. Что бы не загружать лишние данные и экономить ресурсы используется ленивая загрузка. Это может повысить эффективность работы программы, при правильном использовании [9]. Существует четыре основных способа реализации данного шаблона проектирования. Каждый из них имеет свои плюсы и минусы:

- Lazy initialization (Отложенная инициализация);
- Virtual proxy (Виртуальный прокси);
- Ghost (Призрак);
- Value holder (Контейнер значения).

Lazy initialization (Отложенная инициализация) - при данной реализации загружаемый объект изначально имеет значение null.

Пример кода на языке C#:

```
private int myWidgetID;
private Widget myWidget = null;
public Widget MyWidget
{
    get {
        if (myWidget == null)
            myWidget = Widget.Load(myWidgetID);
        return myWidget;
    }
}
```

При каждом запросе объекта проверяется его значение на наличие null и создает его «на лету», прежде чем возвращать его первым. Этот метод наиболее прост в реализации.

Virtual proxy (Виртуальный прокси) - эта реализация построена как объект с тем же интерфейсом, что и у реального объекта. При первом вызове одного из его методов он загружает реальный объект, а затем делегирует.

Ghost (Призрак) - данная реализация, это объект без каких-либо данных, который будет загружен в частичном состоянии [9]. Он может содержать только идентификатор объекта, но загружает свои собственные данные при первом обращении к одному из его свойств. Например, предположим, что пользователь собирается запросить контент через онлайн-форму. Все, что известно знаем во время создания, это то, что к контенту будет доступ, но какое действие или контент неизвестен.

Value holder (Контейнер значения) - это общий объект, который обрабатывает ленивое поведение загрузки, и появляется на месте полей данных объекта. Обычно объект с методом getValue, который вызывает загрузку.

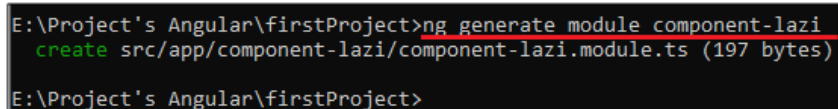
Пример кода на языке C#. Клиент вызывает метод getValue:

```
private ValueHolder < Widget > valueHolder ;
public Widget MyWidget
{
    get {
        return valueHolder.GetValue ();
    }
}
```

В Angular уже есть программный API для ленивой загрузки - NgModule. В Angular CLI он имеет прямую зависимость от базовой инструментальной привязки webpack для разделения блоков и ленивой загрузки. Это значит, чтобы реализовать технологию Lazy-Loading в приложении Angular нужно выполнить 3 пункта, а именно [8]:

- Создать функциональный модуль;
- Создать модуль маршрутизации модуля функций;
- Настроить маршруты.

Для компонентов, которые требуется загружать с помощью ленивой загрузки создаются функциональные модули. Для этого зайдём в папку с проектом через, командную строку, и после этого введём команду для генерации функционального модуля: ng generate module [имя модуля] (рис. 3.1).



```
E:\Project's Angular\firstProject>ng generate module component-lazi
create src/app/component-lazi/component-lazi.module.ts (197 bytes)
E:\Project's Angular\firstProject>
```

Рис. 3.1. Создаем модуль с помощью Angular CLI в командной строке.

После этого в проекте появляется новая папка и модуль в ней (рис. 3.2).

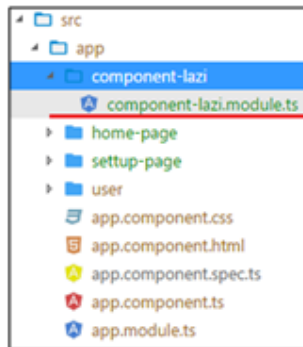


Рис. 3.2. Созданный модуль.

Теперь в этот модуль (`component-lazy.module.ts`) нужно добавить новые маршруты, и декларировать компоненты.

```
import{ HomePageComponent }from'./homepage.component';
import{ RouterModule, Routes} from
'@angular/router';
const routes: Routes=[
{
  path: '',
  component: HomePageComponent
}
];
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ],
  declarations: [ HomePageComponent ],
  exports: [ RouterModule ]
})
export class ComponentLazyModule { }
```

Выше представлено импортирование, а после декларирование `HomePageComponent`, который будет загружаться с помощью ленивой загрузки. Если до этого он был добавлен в корневой модуль, то теперь его следует оттуда удалить. После так же импортируется модуль маршрутизации [8]. Следующим шагом будет настройка корневого модуля и корневого маршрута.

```
const routes: Routes=[
{
  path: 'lazy-loading',
  loadChildren : 'app/component-lazy/component-' +
  'lazy.module#ComponentLazyModule'
}
];
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ],
  declarations: [ HomePageComponent ],
```



```

exports: [ RouterModule ]
})
export class ComponentLaziModule { }

```

loadChildren сопровождается строкой, являющейся путем к модулю, хэш-меткой или #именем класса модуля. Можем проверить, загружается ли модуль, с помощью инструментов разработчика Chrome. В Chrome откройте инструменты разработчика, с помощью команды Cmd+Option+i на Mac или Ctrl+Alt+i на ПК и перейдя на вкладку «Network» (рис.3.3) [8].

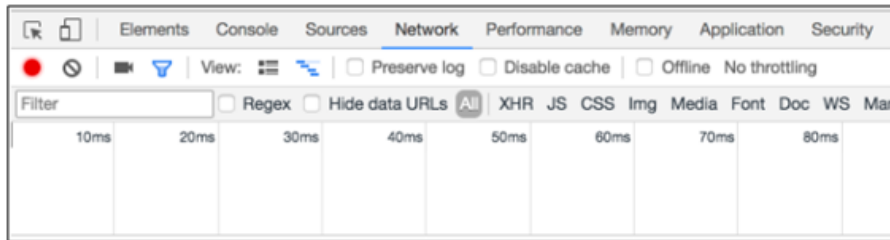


Рис. 3.3. DevTools в браузере Google Chrome.

Предварительно очистим то, что записал браузер (рис. 3.4) во время загрузки приложения.

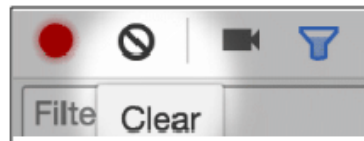


Рис. 3.4. Кнопка Clear в DevTools.

Теперь нужно спровоцировать загрузку компонента путем маршрутизации. После этого можно будет увидеть новую запись (рис. 3.5).

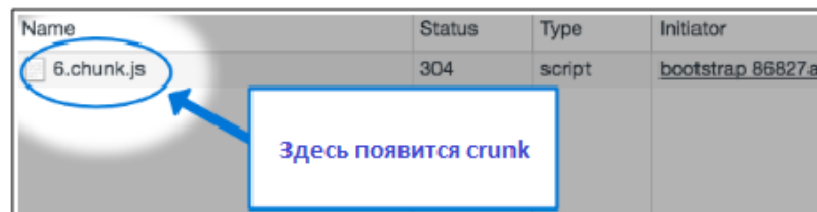


Рис. 3.5. Отслеживается загруженный файл.

Можно заметить, что RouterModule.forRoot(routes) был добавлен к app-routing.module.ts imports массиву. Это говорит Angular, что данный модуль AppRoutingModule является модулем маршрутизации и forRoot() указывает, что это корневой модуль маршрутизации[11]. Он настраивает все маршруты, которые передаются ему, дает доступ к директивам маршрутизатора и регистрирует RouterService. Использовать forRoot() можно лишь один раз в приложении в AppRoutingModule на корневом

уровне. Также добавляется `RouterModule.forChild(routes)` модули маршрутизации функций. Таким образом, Angular знает, что этот список маршрутов отвечает только за предоставление дополнительных маршрутов и предназначен для функциональных модулей. Можно использовать `forChild()` в нескольких модулях. `forRoot()` содержит конфигурацию инжектора, которая является глобальной; таких как настройка маршрутизатора [12]. `forChild()` не имеет конфигурации инжектора, только директивы, такие как `RouterOutlet` и `RouterLink`.

### 3.2. Ленивая загрузка модулей в Angular

Angular это фреймворк для создания клиентских веб приложений на HTML и TypeScript. Он реализует основные и дополнительные функции для приложения представляя их библиотеками на языке TypeScript, которые могут быть импортированы в модули [8].

```
import { NgModule } from '@angular/core';
```

Основными строительными блоками Angular приложения являются модули, которые предоставляют контекст компиляции для компонентов. Директива `NgModules` собирает связанный код в функциональные множества. Angular приложение определяется набором таких директив [11].

```
@NgModule({imports : [ BrowserModule ],  
providers : [ Logger ],  
declarations : [ AppComponent ],  
exports : [ AppComponent ],  
bootstrap : [ AppComponent ]})  
export class AppModule {}
```

У приложения всегда есть корневой модуль, который определяет загрузку приложения и обычно содержит много функциональных модулей. Компоненты содержат представления, которые являются набором HTML элементов, которые Angular может изменять в соответствии с программной логикой и данными. У каждого приложения есть хотя бы один корневой компонент (в только что созданном проекте им по умолчанию будет `AppComponent`).

```
@Component({  
selector: 'app-root',  
templateUrl : './app.component.html',  
styleUrls : ['./app.component.css']  
})  
export class AppComponent
```

Компоненты используют сервисы, которые обеспечивают определенную функциональность, не связанную напрямую с представлениями [12]. Сервисы (они же

провайдеры) могут быть внедрены в компоненты в качестве зависимостей, что делает код модульным, многоразовым и эффективным. Компоненты и сервисы - это простые классы, с декораторами, которые определяют их тип и предоставляют метаданные, которые сообщают Angular, как их использовать [8]. Метаданные класса компонента связывают его с шаблоном (свойство `templateUrl`) определяющим представление. Шаблон объединяет обычный HTML с Angular директивами, которые позволяют Angular изменять HTML-код перед его отображением. Метаданные класса сервиса предоставляют информацию, необходимую Angular, чтобы сделать ее доступной для компонентов через Injection Dependency (DI) [13]. Компоненты приложения обычно определяют много представлений, упорядоченных иерархически. В Angular есть Router сервис, который призван помочь назначить пути навигации между ними [14]. Маршрутизатор предоставляет сложные навигационные возможности в браузере. Шаблон сочетает HTML с привязкой Angular к разметке с помощью директив, которые могут изменять элементы HTML до их отображения.

```
<p myDirective> using my directive! </p>
```

Директивы шаблонов предоставляют программную логику, а привязка к разметке связывает данные приложения и объектную модель документа (DOM). Связывание событий позволяет приложению реагировать на ввод пользователя в целевой среде путем обновления данных приложения [13]. Связывание свойств позволяет интерполировать значения, которые вычисляются из 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 программными данными и логикой. Angular поддерживает двустороннюю привязку данных. Это означает, что изменения в DOM, такие как выбор пользователя, также могут быть отражены обратно в данные программы [11]. Шаблоны также могут использовать `pipe`(каналы), чтобы улучшить работу пользователя, изменив значения для отображения. Используются `pipe` для отображения, например, дат меняя формат с '02 февраль 2018 09:25:18:123' на '02.02.18 09:25' или наоборот, или для других похожих операций с любыми данными [11].

```
<p> using pipe for date {{ myDate | date }}</p>
```

Angular уже имеет `pipe`'ы для общих преобразований, но также можно определить свои собственные. Для данных или логики, которые не связаны с определенным представлением и должны быть использованы в разных компонентах, обычно создается сервис.

```
constructor (private service: MyService){}
```

Для определения сервиса есть специальный декоратор `@Injectable`. Данный декоратор содержит метаданные, которые позволяют сервису внедряться в клиентские компоненты в качестве зависимости.

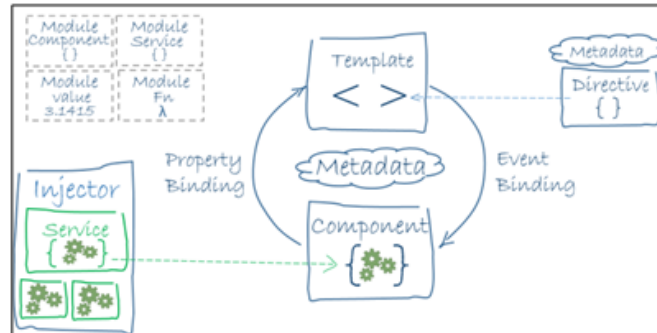


Рис. 3.6. Диаграмма показывает, как эти основные части связаны.

Dependency injection (или DI) позволяет сохранить класс компонентов и повысить эффективность (рис. 3.6). Компоненты не извлекают данные с сервера и не проверяют вход пользователя, они перекладывают эти задачи на сервисы. Благодаря языку TypeScript в Angular есть возможность полноценного использования ООП (Объектно-Ориентированное Программирование). И конечно же можно создавать классы наделяя их определенным функционалом. Но на самом деле классы в Angular играют куда большую роль. Angular, кроме объектно-ориентированной парадигмы использует так же декларативное программирование. Декларативное программирование - это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат [13]. Можно интерпретировать это примерно так: классы будут задекларированы для того, чтобы Angular раздал им свои роли. Это делается с помощью декораторов и вот некоторые из них [8]:

- `@NgModule` - этот декоратор говорит Angular, что данный класс является модулем, который нужен для определения сервисов, компонентов, маршрутов, пайпов и тд. С помощью этого декоратора происходит разделение приложения на модули;
- `@Component` - этот декоратор говорит Angular, что класс является компонентом. Компоненты нужны для отображения и логических операций над данными;
- `@Injectable` - этот декоратор говорит Angular, что класс является сервисом. Сервисы нужны для хранения и использования общим данных которые должны быть доступны нескольким компонентом (Например, данные о пользователе);

- @Pipe - этот декоратор говорит Angular, что класс является фильтром. Фильтры используются для динамического изменения данных или формата данных внутри HTML шаблона;
- @Directive - этот декоратор говорит Angular, что класс является директивой. Директивы нужны для того что производить манипуляции над HTML шаблоном.

После этой процедуры классы обретут гораздо большие возможности. Декораторы могут быть использованы для аннотации класса, свойства, метода или параметра. Давайте подробно рассмотрим каждый из этих типов. Декомпозиция - разделение целого на части. Часто в веб приложениях существует необходимость обратиться к компоненту по ссылке [16]. И SPA не являются исключением. В Angular есть Router, который отвечает за навигацию приложения. Он позволяет через Url переходить с одного компонента на другой [8].

У браузера имеется модель навигации приложения:

- Если ввести URL-адрес в адресной строке, браузер перейдет на соответствующую страницу;
- Нажимая ссылку на странице, браузер перейдет на новую страницу;
- Используя кнопки браузера назад и вперед, браузер переместится назад и вперед по истории страниц, которые открывались.

Router в Angular заимствует эту модель навигации. Он может интерпретировать URL-адрес браузера в качестве инструкции для перехода к представлению, создаваемому клиентом. Он может передавать необязательные параметры вместе с компонентом. Можно привязать маршрутизатор к ссылкам на странице, и по нажатию на них он перейдет к соответствующему компоненту приложения. Маршрутизатор регистрирует активность в журнале истории браузера, так что кнопки назад и вперед будут работать [17]. Для добавления маршрутизации в веб приложение нужно, чтобы в файле index.html в качестве первого дочернего элемента тега <head> был добавлен элемент <base>.

```
<base href="/">
```

Это нужно для того, чтобы сообщить маршрутизатору, как составлять URL-адреса навигации. Angular Router - не является обязательным условием для проектирования приложения, но он позволяет закрепить за конкретным компонентом собственный URL-адрес. Это не часть ядра Angular. Он находится в пакете библиотеки @angular/router. Импортируйте то, что нужно из данного пакета, как из любого другого пакета Angular.

```
import { RouterModule, Routes }
from '@angular/router';
```

Маршрутизированное Angular приложение содержит один экземпляр Router сервиса. Когда URL-адрес браузера изменяется, маршрутизатор ищет соответствующий, Route из которого он может определить отображаемый компонент. Маршрутизатор не имеет маршрутов, пока его не настроить. Настраивается маршрутизатор с помощью RouterModule.forRoot метода и добавляется в AppModule, в декоратор NgModule, в свойство imports [14].

```
const appRoutes :Routes=[{
  path: 'crisis-center',
  component: CrisisListComponent
},
{
  path: 'heroes',
  component: HeroListComponent
  data: {title:'HeroesList'}
},
{
  path: '',
  redirectTo :'/heroes',
  pathMatch : 'full'
},
{
  path:'**',
  component: PageNotFoundComponent
}];
```

Созданный appRoutes - это массив маршрутов, который описывает перемещения по Url - адресам. :id во втором маршруте является маркером для параметра маршрута. В URL-адресе, таком как /hero/42, "42" будет значением id параметра. Каждый из маршрутов сопоставляется с URL-адресом (в свойстве path) компонента [8]. Маршрутизатор анализирует и создает окончательный URL-адрес, позволяя использовать как относительные, так и абсолютные пути при навигации между видами приложений. Теперь его нужно передать в RouterModule.forRoot метод в модуле, в свойство imports для настройки маршрутизатора.

```
@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing :true}
    )
  ],
  ...}) export class AppModule { }
```

Если нужно посмотреть, какие события происходят во время жизненного цикла навигации, есть опция enableTracing. При значении true выводится каждое событие маршрутизатора, которое будет происходить в течение каждого жизненного цикла навигации

в консоль браузера. Эту опцию следует использовать только для отладки. Устанавливаете `enableTracing: true` опцию в объекте, переданном в качестве второго аргумента `RouterModule.forRoot()` методу (так же как это показано на картинке выше) [15]. Когда URL-адрес браузера для приложения становится `/heroes`, маршрутизатор сопоставляет этот URL-адрес с маршрутом `/heroes` и отображает `HeroListComponent` внутри тега `<router-outlet>` который размещён в HTML-представлении хоста.

```
<router-outlet></router-outlet>
```

URL-адрес может поступать непосредственно из адресной строки браузера. Но большую часть времени перемещение происходит в результате какого-либо пользовательского действия, например, щелчка мыши по кнопке.

```
<h1>AngularRouter</h1>
<nav>
  <a routerLink="/crisiscenter" routerLinkActive="active"> CrisisCenter </a>
  <a routerLink="/heroes" routerLinkActive="active">
    Heroes </a>
</nav>
<router-outlet></router-outlet>
```

В директивах `routerLink` по тегам предоставляется контроль над маршрутизатором этих элементов. Директивы `routerLinkActive` меняют стиль отображения элемента если маршрут элемента в данный момент активен.

Так же у маршрутов имеется защита, называемая `Guard`. `Guard` может разрешить или запретить навигацию по определенному маршруту. К примеру, для доступа к определенному компоненту требуется наличие каких-то условий, отталкиваясь от которых можно предоставить или не предоставить пользователю доступ. `Guards` защищают доступ к маршруту. Один из типов защиты `CanActivate` позволит управлять доступом к компоненту при маршрутизации. Для того, чтобы его реализовать создадим файл `about.guard.ts` в папке `src/app`.

```
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
} from '@angular/router';
import { Observable } from 'rxjs'; export
class AboutGuard implements CanActivate {
  canActivate (route : ActivatedRouteSnapshot, state
    : RouterStateSnapshot) : Observable<boolean>
  | boolean {
    return true;
  } }
}
```

Класс AboutGuard реализует интерфейс CanActivate. Метод CanActivate() принимает параметры - ActivatedRouteSnapshot и RouterStateSnapshot. Они содержат информацию о запросе [13].

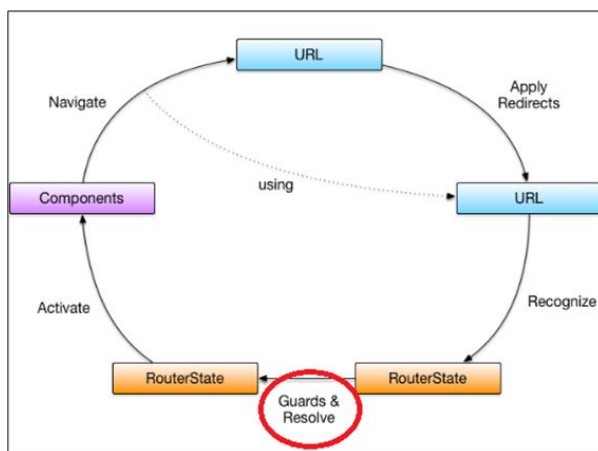


Рис. 3.7. Жизненный цикл маршрутизации.

Guard проверяется раньше, чем загружается компонент (рис. 3.7). ActivatedRouteSnapshot позволяет получить различную информацию из запроса, такую как, например, параметры маршрута и строки запроса. Если CanActivate() вернет true это будет означать что переход по маршруту одобрен и маршрутизация работает.

```

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent, canActivate: [AboutGuard] }
];

@NgModule({
  imports: [ BrowserModule, RouterModule.forRoot(appRoutes) ],
  declarations: [ AppComponent, HomeComponent, AboutComponent ],
  providers: [ AboutGuard ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
  
```

Рис. 3.8. Добавляем в свойство canActivate и providers Guard.

Чтобы использовать Guard нужно добавить созданный класс AboutGuard в корневой модуль приложения в свойство providers, а также добавить свойство canActivate и в него так же добавить класс AboutGuard (рис. 3.8) [11]. Безусловно Angular не единственное SPA решение для веб приложения. Тем не менее его рейтинг ежегодно растет во всевозможных топах и сравнениях с другими инструментами.

*Преимущества:*



- Поддержка вэб-компонентов. Весь код делится на компоненты, сервисы и модули. Это позволяет разработчику быстро найти нужный участок кода и начать работать в нём. Так как компоненты разделены изменение кода в одном ни коим образом не повлияет на другой;
- Использование TypeScript. Самое большое коммерческое преимущество TypeScript состоит в его инструментарии. Этот язык даёт возможности современного автозаполнения, навигации и рефакторинга. Такие инструменты становятся практически незаменимыми при работе с большими проектами;
- Отличная производительность. Angular не проводит глубокий сравнительный анализ объектов. Если какой-то элемент добавить в массив данных, изменение пути не будет обнаружено. Это касается и свойств объекта, пока они не связаны напрямую с View;
- Angular CLI. CLI фреймворка Angular даёт возможность с лёгкостью создавать приложение, которое работает по умолчанию. Это соответствует лучшим современным тенденциям.

#### *Недостатки:*

- Тяжело освоить. Это ни на что не похожий полноценный фреймворк с уже существующими в нём реактивными привязками к шаблонам. Это не игрушка для новичка, это профессиональный и очень мощный инструмент для разработки больших проектов.

Всё чаще для разработки своих веб приложений выбирают Angular. Главная причина тому усердный труд разработчиков Angular. У данного фреймворка за спиной поддержка огромной компании, которая очень редко дает усомниться в качестве своих разработок. За последние года Angular стал одним из самых популярных фреймворков в мире front-end разработки. По этой причине многие производители сторонних сервисов (библиотек, инструментов) разрабатывают так же совместимые с Angular версии своих продуктов. Некоторые из них Angular уже использует внутри своей архитектуры (например, RxJS библиотеку) и загружает по умолчанию при создании нового проекта (но не импортирует их), другие сервисы нужно самостоятельно искать и устанавливать. Как правило местом для хранения этих сервисов вполне обоснованно является облачный сервер npm. Npm это пакетный менеджер, использующийся для скачивания или загрузки на

сервер пакетов (пакетом в Node.js называется один или несколько JavaScript-файлов, представляющих собой какую-то библиотеку или инструмент), обладающий самой большой пакетной экосистемой в мире с открытым исходным кодом [13]. Именно с помощью npm мы, кстати говоря, устанавливаем на свой ПК Angular CLI и даже при создании нового проекта неявно его используем. Npm уже входит в состав программы Node.js поэтому не требует отдельной установки.

## 4. ANGULAR ELEMENTS ИЛИ ВЕБ-КОМПОНЕНТЫ НА ANGULAR

### 4.1. Общее понимание Веб-компонентов

Популярные современные фреймворки в основном предлагают возможность повторного использования кода в виде компонентов или модей. Вообще говоря, это отдельная часть кода (HTML / JS / CSS), которая предлагает визуальный стиль, интерактивность и, возможно, имеет API или параметры. Все эти свойства уже предоставляет браузер, ведь в нём уже есть многоцветные, модульные элементы, которые предлагают стиль, интерактивность и поставляются с API. Конечно, речь идет о тегах HTML или элементах DOM [1]. Они отображаются в DOM и имеют очень специфический тип функциональности. Тег <div> или тег <span> довольно общий и используется для хранения текста или смеси элементов. Элементы <button> и <input> являются более конкретными по функциональности и стилю. Когда кнопка помещается в HTML-код, она выглядит как стандартная кнопка, и когда на неё нажимают, она действует как кнопка. Подобные различия в стилях и возможностях у <input>, будь то создание средства выбора даты, слайдер или поле ввода текста.

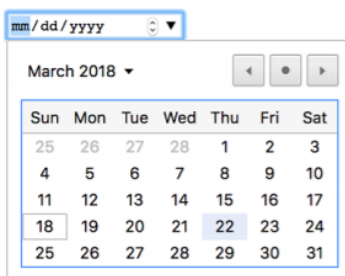


Рис. 4.1 Расширенный пользовательский интерфейс выбора даты

Чтобы создать средство выбора даты, необходимо просто поместит следующий тег в HTML документе: <input type= "date">. То, что в действительности получается от этого простого тега, довольно сложно, но все это делает браузер сам. Этот тег (при использовании типа «дата») предлагает текстовое поле ввода, но можно нажать на месяц, день или год и перейти вверх или вниз по любому из них [1]. Кроме того, если щелкнуть стрелку вниз сбоку, откроется представление календаря (рис. 4.1). Пользователь может взаимодействовать с этим интерфейсом, чтобы выбрать дату. Кроме того, в мобильном телефоне он работает немного иначе. Он не откроется таким образом, а вместо этого покажет модальное окно для выбора даты. Более того, у средства выбора даты есть свойства, которые можно запросить, включая

выбранное значение. Это можно увидеть используя свойства в консоли `JavaScript.console.log(document.querySelector('ввод').значение)`. Средство выбора даты - отличный пример многоразового «компонента» или «модуля» с довольно сложными визуальными стилями и шаблонами взаимодействия, которые были запрограммированы в браузер разработчиками. Они работают в самых разных ситуациях. Это также отличный пример популярной веб-компонентной концепции, называемой Shadow DOM.

Shadow DOM (теневая модель документа) - часть документа, реализующая инкапсуляцию в DOM дереве. Она является частью документа и встраивается непосредственно внутрь страницы [18]. При открытии инструментов разработчика, чтобы посмотреть на DOM, будет виден просто тег `<input type="date">`. Однако, если использовать Chrome и включить функцию отображения «User Agent Shadow DOM» в настройках инструментов разработчика, тот же тег расширится, и выйдет так (рис. 4.2).



Рис. 4.2. Включение настроек Shadow Root в инструментах разработчика Chrome

В этом скрытом «теневом корне» гораздо больше разметки. Было бы здорово увидеть это в HTML и CSS. Его нет, потому что этот элемент пользовательского интерфейса является частью браузера и привязан к ОС. Тем не менее, есть изрядное количество элементов, скрытых в Shadow DOM, которые появляются в элементе поля ввода. Присмотревшись, можно заметить, что Shadow DOM содержит смесь `<div>` и `<span>` [1]. Может прийти в голову, что это опасно. Что ж, в CSS приложения можно очень просто определить все теги `<div>`, чтобы они имели синий фон с очень большим размером шрифта и все теги `<span>` для отображения с непрозрачностью 10%. Это бы прекрасно могло работать, за исключением одного «но»! Shadow DOM защищает внутреннюю работу веб-компонента извне. Поэтому эти синие и большие стили в тегах `<div>` и `<span>` не проникнут в Shadow DOM.

```
let myElement = document.getElementById ('clear');
```

Более того, можно было бы написать некоторый Javascript, чтобы попытаться получить и управлять кнопкой очистки средства выбора даты (пример кода выше) [18]. При попытке получить этот элемент, поскольку он находится в пределах Shadow DOM, объект элемента не будет найден, а переменная myElement имеет значение null.

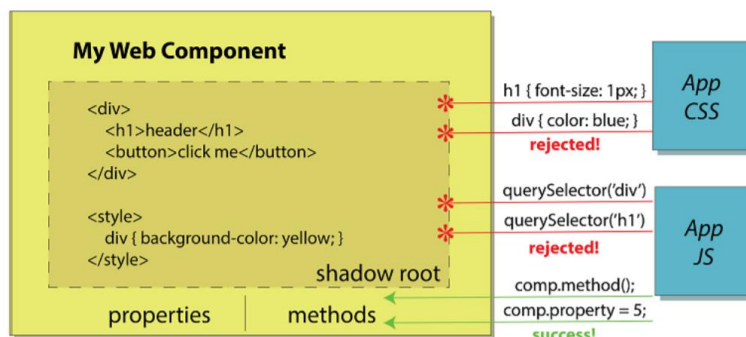


Рис. 4.3 Shadow DOM защищает компонент от непредвиденных последствий.

Это и делает Shadow DOM, защищает теневую область видимости. Можно использовать этот «теневого корень» где угодно (рис. 4.3). Это имеет смысл в настраиваемом элементе, который создается во избежание непреднамеренного сбоя, когда разработчик устанавливает правило CSS, которое имеет то же самое имя, как и то, что используется в созданном компоненте. Или же когда тот же разработчик запрашивает элемент по классу, и что-то в настраиваемом элементе случайно выбирается [18]. Становится понятным что, выбор даты - очень полезный элемент, как и много других полезных элементов, которые используются ежедневно. Многие элементы используются для семантических целей, как например, тег <footer>. Но другие элементы имеют особый API и стиль, например, теги <button>, <option> или <video>

Каким бы красивым и удобным ни был выбор даты и любой другой элемент, было бы замечательно, если бы была возможность создавать собственные элементы с собственным визуальным стилем, внутренней логикой, возможностью повторного использования и инкапсуляция. Именно это имеется в виду, когда речь идет о веб-компонентах [1]. В добавок инкапсуляция, обеспечиваемая Shadow DOM, может использовать API Custom Element для создания собственных компонентов, которые делают вещи, для специфичных нужд. Это возможность взять то, что интересно и создать объект многоразового использования, которым можно поделиться со всем миром, своей командой или пользоваться самому, используя его в нескольких проектах там, где это необходимо. Или даже лучше, может некий

разработчик создал веб-компонент для чего-то, что нужно другому, у которого нет времени или опыта, чтобы создать это самому [18]. Разработчики могут делиться своими компонентами, и просто использовать их его как обычный элемент DOM не зависимо от фреймворка.

Говоря о конкретных технических функциях, которые предлагают веб-компоненты, видение начало разваливаться после первоначальной шумихи вокруг веб-компонентов. Это прижилась несколько лет назад [1]. Примерно в 2015 году было широко распространено мнение, что стандартный веб-компонент будет создан с помощью имплементирования трех новых функций:

- Пользовательские элементы;
- Теневой DOM;
- Импорт HTML.

Эта концепция никогда не принималась в качестве стандарта. Фактически, вначале Google в значительной степени отвечали за создание рабочих проектов веб-компонентов [18]. Они взяли на себя создание API-интерфейсов и отправили их в Chrome как оснащающий эксперимент, чтобы увидеть, получат ли распространение веб-компоненты. HTML-импорт так и не появился.

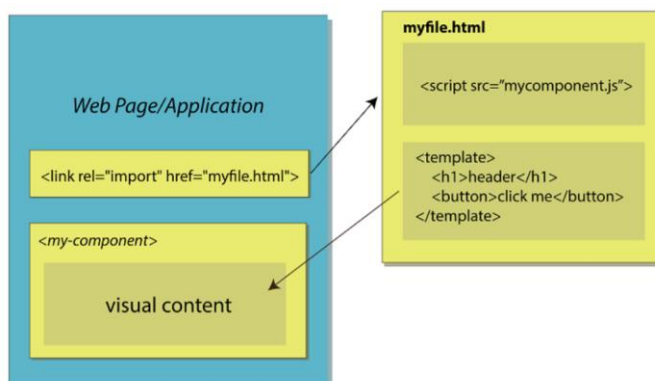


Рис. 4.4. Файл, содержащий определение компонента и разметку компонента

С помощью импорта HTML файл, содержащий определение компонента и разметку компонента, может быть импортированным прямо в документ Shadow DOM (рис. 4.4). Другие поставщики браузеров в то время не планировали выпускать эту функцию. Firefox, в частности, хотели подождать, чтобы увидеть, насколько большой будет всплеск модулей ES6/ES2015 и, возможно, когда-нибудь импортировать не только JS, но и HTML. HTML-импорт был довольно большой потерей. С самого начала в планах Google от него зависели

веб-компоненты. HTML Import был фрагментом HTML для объявления разметки или структуры компонента, а также включал JS, определяющий логику составной части. Импорт HTML был основной точкой входа для веб-компонентов, и без них не было понятно, как вообще использовать веб-компоненты с разметкой и стилем. Единственным браузером, который принял его, был Chrome. Даже сейчас это просто Chrome и Safari. Хотя Firefox поставляет его за флагом конфигурации, а в Edge он в настоящее время находится в разработке [18]. И Shadow DOM, и API Custom Element также перешли с версии 0 на 1. Для Custom Elements это было немного неприятно, учитывая, что разработчики, знакомые с веб-компонентам в то шаткое время надеялись перейти на новый API. Учитывая все это, разработчиков, которые называли веб-компоненты «нарушенным обещанием» и продолжили пользоваться фреймворками вряд ли можно критиковать. В 2015 году было немного сложно правильно работать с ними, особенно при ориентации на браузеры, отличные от Chrome. Полимер и X-теги еще один аспект того, что имелось в виду, когда говорилось о веб-компонентах. Были фреймворки, появившиеся в то время, которые использовали веб-компоненты в качестве своих основных структур. Из-за нестабильности, окружающей простые компоненты без каркаса в то время, Polymer Google и X-теги Mozilla были даже более тесно связаны с веб-компонентами, чем лежащие в основе технологии. Как библиотека, Polymer довольно хороша [1]. Она основана на главных концепциях веб-компонентов и предлагает отличные помощники и инструменты. Некоторые из них, например, lit-html и PolymerCLI могут работать даже с веб-компонентами, не основанными на полимерах. Несмотря на твердую версию 3.0 сейчас, первые дни Polymer до версии 1.0 были немного шаткими. Как и ожидалось с любой библиотекой до версии 1.0, API-интерфейсы немного изменились, особенно когда они пытались сохранить спецификации с отсутствием Shadow DOM во всех браузерах, кроме Chrome. Особенно сложно было иметь дело с Shadow DOM. Полнофункциональные полифиллы, в которые включенная инкапсуляция CSS была слишком сложной и влияла на производительность [1]. Чтобы компенсировать это, «Shady DOM» был изобретен как облегченная реализация, которую можно было заполнить полифиллами. Это было непростое время для веб-компонентов в целом, и Polymer казался еще одним фреймворком / библиотекой, который должен был конкурировать с более устоявшимися, не имевшими дело с этими веб-стандартами.

Можно успешно использовать современные веб-компоненты для проектов, но не быть полностью удовлетворенными ими, или начать использовать некоторые совершенно новые

языковые функции JS [1]. Функция жирной стрелки оказалась отличным способом управления прицелом при работе с событиями мыши или таймером. Что еще более важно, концепция «модулей» и ключевое слово `import` стали удобными. Благодаря импорту можно избавиться от хрупкого беспорядка, связанного с тем, чтобы каждый используемый файл JS, был связан в теге сценария на главной HTML-странице. Каждый веб-компонент полностью отвечает за импорт собственного кода. Это означало, что на главной HTML-странице можно иметь один тег сценария на основе модуля, импортирующего веб-компонент, содержащий всё приложение. Каждый дочерний компонент просто импортирует все, что ему нужно. Это открыло двери для многоуровневых модулей кода, написанных на чистом JS, и дало возможность для создания нескольких уровней наследования, когда требуется, чтобы компоненты разделяли API и были немного умнее, чем базовый `HTMLElement` API. Наконец, можно хранить свой HTML/CSS в отдельный файл `template.js`, который можно импортировать, сохраняя мои визуальные проблемы отдельно от логики контроллера компонента. Последней «огромной» функцией JS, с которой приятно работать с веб-компонентами - «Шаблонный литерал» [1]. Можно не только хранить свой HTML / CSS в отдельном файле шаблона, но и заменить выражения-заполнители в разметке на переменные, а также вложить несколько шаблонов вместе с использованием функций JS. Эти функции ES6 / ES2015 внезапно сделали работу с веб-компонентами «счастьем». Даже ранее с устаревшим HTML-импортом, можно было подумать, что комбинация модули и литералы шаблонов - лучший способ. Правда, это дает дочерним элементам компонента некоторую хорошую защиту от стиля и JS, вкрадывающихся и оказывающих неблагоприятное воздействие, но это новое решение проблемы, которая всегда присутствовала. Как бы там ни было в будущем с веб-компонентов, на данный момент не слышно ни о каких альтернативных подобных решениях [18]. Как только Shadow DOM будет работать повсюду, это станет важным дополнением к общему набору инструментов. Никогда не бывает легко предсказать будущее, особенно в Интернете, где все меняется в безумном темпе. Тем не менее, есть несколько веских предпосылок, указывающих, куда могут двигаться веб-компоненты. Уже были видны эксперименты с React, Angular, и Vue при упаковке компонентов в каждой из этих платформ как отдельный веб-компонент. Дополнительно инструменты такие как StencilJS и Svelte позволяют создавать и компилировать в автономные веб-компоненты. Это значит, что вскоре можно будет создавать компоненты без фреймворка или с рамками по индивидуальному выбору.



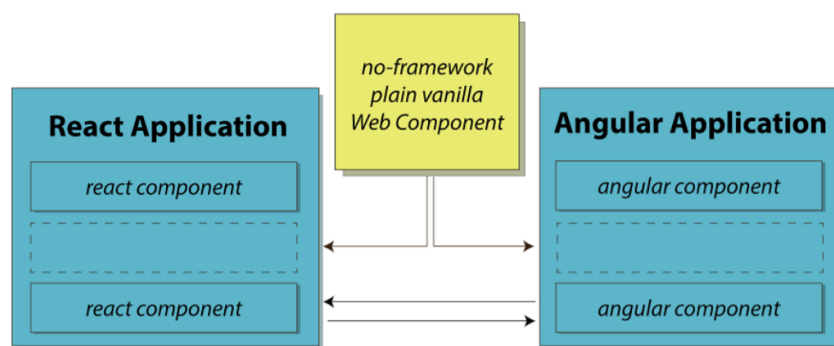


Рис. 4.5. Веб-компонент и фреймворки.

Веб-компоненты могут ликвидировать разрыв между популярными фреймворками (рис. 4.5). Веб-компоненты фреймворка можно использовать в этих фреймворках, но уже есть проекты, компилирующие компонент React, Angular и Vue для независимого запуска компонентов, которые можно использовать где угодно [18]. Эта концепция может даже расширяться, позволив совершенно разным языкам работать вместе. В одном приложении могут быть разные компоненты, разработанные на Javascript, Typescript, и Coffeescript, и поскольку каждый из них представляет собой модульный компонент, предоставляющий API, это не имеет значения [8]. Еще более безумно то, что с появлением WebAssembly стало возможным видеть такие языки, как C++, Lua, Go и др., скомпилированными в байт-код и обернутый веб-компонентом, выглядящий как полностью нормальный элемент снаружи, одновременно обеспечивающий высокую производительность графики, которая может работать быстрее, чем обычно работает JS. Скорее всего использование модулей ES6 / ES2015 и импорта изменит образ мышления индустрии о библиотеках и фреймворках. Уже видно два похожих инструмента: lit-html и hyperHTML для расширенного управления разметкой. У обоих есть модули, которые разработчики могут импортировать вместо того, чтобы загружать всю библиотеку и решать очень конкретную проблему. В связи с этим, в будущем появятся еще много удивительных библиотек. Импортируется только то, что необходимо, и только когда это нужно. Время покажет, откажутся ли основные фреймворки от функций из-за html в отдельные импорты, которые можно использовать вне фреймворка, как это сделал Polymer, но кажется это неизбежно, особенно глядя на другие языки, в которых была функция импорта навсегда. Задолго до появления веб-компонентов были отличные способы взаимодействия с обычными элементами DOM. Можно использовать эти же методы, чтобы придать структуру всему, что строится с помощью веб-компонентов, как и

обычные теги `<div>`, `<video>` или `<input>` [1]. Веб-компоненты такие же, как и любой другой элемент DOM.

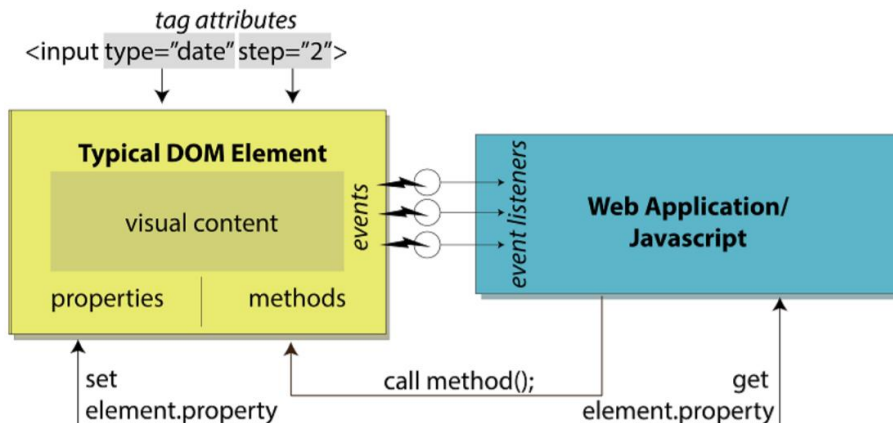


Рис. 4.6. Публичный API элемента.

Во-первых, у каждого элемента есть своего рода публичный API (рис. 4.6). Под этим подразумевается, возможность получить и установить свойства элемента и вызвать функции. Например, с помощью элемента видео можно вызывать функции «`pause()`» и «`play()`» для управления воспроизведением видео [1]. Также можно проверить, как долго длится видео, проверив свойство «`duration`». Наконец, чтобы перейти к текущему моменту видео, можно установить свойство «`currentTime`». Очевидно, что методы и функции для объектов используются в программировании повсюду. Элементы DOM ничем не отличаются - и, кроме того, пользовательские веб-компоненты тоже [18]. Что-то столь же простое, как тег `<img>`, имеет атрибут «`src`», который указывает элементу на расположение изображения. Веб-компоненты имеют API, так что можно внутренне отслеживать изменения атрибутов. Наконец, можно прослушать изменения из пользовательского веб-компонента. Обычно делается следующие, чтобы слушать `click`: `mybutton.addEventListener('click', functionToCall)`. Также можно создавать и отправлять собственные «настраиваемые события». Говорить об отдельных компонентах - это одно, но как насчет, создания всего веб-приложение. Веб-компоненты могут быть настолько большими или маленькими, насколько это необходимо. Можно создать несколько очень детализированных компонентов, например кнопку, а затем вложить их внутри более крупного веб-компонента, такого как собственная панель инструментов [1]. Компонент панели инструментов может обрабатывать более тонкие детали работы с кнопками, возможно, включать и выключать их или отключать определенные при определенных обстоятельствах. определенные при определенных обстоятельствах.

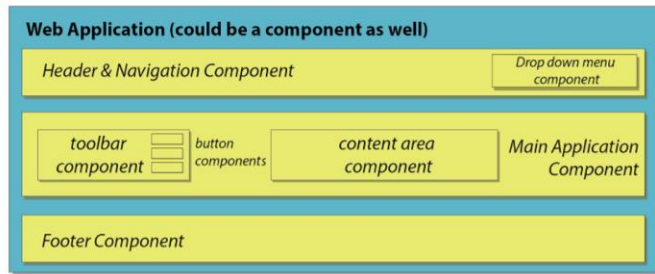


Рис. 4.7. Вложенная панель инструментов в родительский элемент.

Панель инструментов, наряду с другими компонентами, может быть дополнительно вложена в другой родительский элемент, компонент и так далее (рис. 4.7). Веб-компоненты и JS без фреймворка могут многое предложить для веб-приложений. Но по мере роста приложения оно будет усложняться. Будет труднее координировать взаимодействие компонентов друг с другом [18]. Иногда обнаруживается, что даже с внутренней структурой, которую предоставляют веб-компоненты, недостаточно для создания сложного приложения. И может возникнуть соблазн обратиться к популярным фреймворкам и библиотекам, помогающие структурировать. Популярные фреймворки, такие как Angular предлагают привязку данных, шаблоны MVC и многое другое. Безусловно, они могут быть полезны при создании традиционного веб-приложение [1]. С другой стороны, можно писать и импортировать простой JS-код на основе проверенных и надежных шаблонов проектирования, которые существуют уже много лет, избегая этих больших рамок. Например, собственные события DOM могут не подойти [1]. Часто есть необходимость в одной части веб-приложения для сообщения совершенно в другую часть приложения, и не хочется беспокоиться о том, как событие проходит через DOM. Можно обратиться к библиотекам на подобе RXjs или Redux, но это может оказаться излишним.

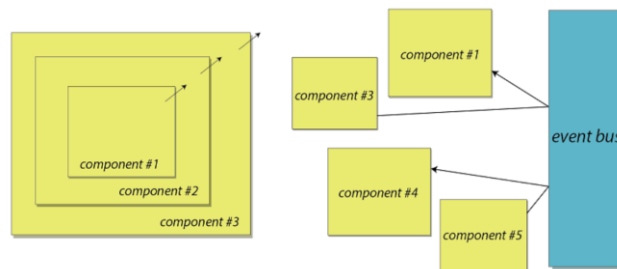


Рис. 4.8. Простая шина событий.

Вместо этого можете написать простую шину событий с небольшим объемом кода (рис. 4.8). Также есть золотая середина с микрофреймворками. Микрофреймворк может быть

отличным, минималистичным способом организовать приложение и добавить определенную функциональность более широкой структуре [1]. В конце концов, несмотря на то, что есть отличные аргументы в пользу веб-компонентов без фреймворка, проект и команда в конечном итоге повлияют на то, что используется для создания продукта. В любом развивающемся стандарте веб-компоненты пока не предлагают всех ответов. Бывают случаи, когда веб-приложение очень простое и современный фреймворк может быть идеальным ответом, потому что он обрабатывает все, что нужно. Иногда приходится работать над типом проекта, в котором фреймворки просто мешают. В решениях, из которых можно выбрать, широкий спектр вариантов, некоторые из которых пересечение [18]. Даже если веб-компоненты без фреймворка не подходят, фреймворк, вероятно, однажды будет построен с их помощью, даже если это не очевидно. Ознакомление с основами веб-стандартов любого фреймворка - это всегда отличная идея, даже если они не используются напрямую.

## 4.2. Angular Elements

Angular Elements - это пакет в Angular, который помогает публиковать компоненты Angular как пользовательские элементы. Для этого он берет компонент Angular и компилирует его в веб-компонент [19]. Пользовательские элементы - это функция веб-платформы, которая в настоящее время поддерживается Chrome, Edge (на основе Chromium), Firefox, Opera и Safari, а также, доступна в других браузерах через полифилы. Пользовательский элемент расширяет HTML, позволяя определить тег, содержимое которого создается и контролируется кодом JavaScript. Браузер поддерживает CustomElementRegistry определенные настраиваемые элементы, которые сопоставляют создаваемый класс JavaScript с тегом HTML. @angular/elements пакет экспортирует createCustomElement() API [19]. Преобразование компонента в настраиваемый элемент делает всю необходимую инфраструктуру Angular доступной для браузера. Создание настраиваемого элемента является простым и понятным, и оно автоматически связывает определяемое компонентом представление с обнаружением изменений и привязкой данных, сопоставляя функциональность Angular с соответствующими эквивалентами в собственном HTML.

Простой динамический контент в приложении Angular - преобразование компонента в настраиваемый элемент обеспечивает простой путь к созданию динамического HTML-содержимого в приложении Angular. HTML-контент, который добавляется непосредственно в DOM в приложении Angular, обычно отображается без обработки Angular, если не

определяется динамический компонент, добавляя собственный код для подключения тега HTML к данным приложения и участвует в обнаружении изменений. С настраиваемым элементом вся эта проводка выполняется автоматически.

Приложения с богатым содержанием - если в приложении с богатым контентом, настраиваемые элементы позволяют предоставить поставщикам контента сложные функции Angular, не требуя знания Angular. Например, руководство по Angular, добавляется непосредственно в DOM с помощью инструментов навигации Angular, но может включать в себя специальные элементы, подобные тем, `<code-snippet>` которые выполняют сложные операции [18]. Все, что нужно сообщить поставщику контента - это синтаксис настраиваемого элемента. Поставщику контента не нужно ничего знать об Angular или о структурах данных или реализации компонента. Используется `createCustomElement()` функция для преобразования компонента в класс, который можно зарегистрировать в браузере как настраиваемый элемент. После того, как регистрируется, сконфигурированный класс в реестре настраиваемых элементов браузера, можно использовать новый элемент так же, как встроенный элемент HTML в контенте, который добавляется непосредственно в DOM:

```
<my-popup message="Use Angular!"></my-popup>
```

Пользовательские элементы загружаются сами по себе - они запускаются автоматически при добавлении в DOM и автоматически уничтожаются при удалении из DOM [19]. После добавления настраиваемого элемента в DOM для любой страницы он выглядит и ведет себя как любой другой элемент HTML и не требует каких-либо специальных знаний терминов Angular или соглашений об использовании.

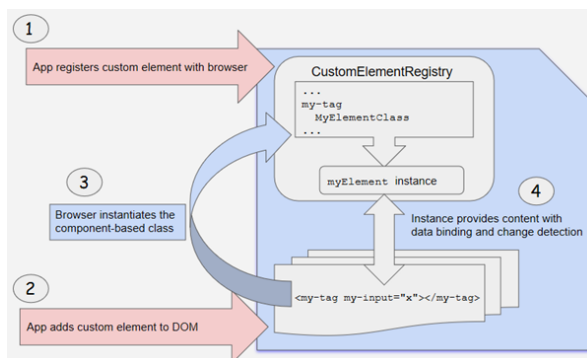


Рис. 4.9. Компонет, использует синтаксис шаблона Angular и отображается DOM.

Когда настраиваемый элемент размещается на странице, браузер создает экземпляр зарегистрированного класса и добавляет его в DOM. Содержимое предоставляется шаблоном

компонента, который использует синтаксис шаблона Angular и отображается с использованием данных компонента и DOM (рис. 4.9). Входные свойства в компоненте соответствуют входным атрибутам для элемента.

### 4.3. Преобразование компонентов в пользовательские элементы

Angular предоставляет `createCustomElement()` функцию преобразования компонента Angular вместе с его зависимостями в пользовательский элемент [19]. Функция собирает наблюдаемые свойства компонента, а также функции Angular, необходимые браузеру для создания и уничтожения экземпляров, а также для обнаружения изменений и реагирования на них. Процесс преобразования реализует `NgElementConstructor` интерфейс и создает класс конструктора, который настроен на создание экземпляра самозагрузки компонента. Используется функция JavaScript, `customElements.define()`, чтобы зарегистрировать сконфигурированный конструктор и связанный с ним тег настраиваемого элемента в браузере `CustomElementRegistry` [18]. Когда браузер встречает тег для зарегистрированного элемента, он использует конструктор для создания экземпляра настраиваемого элемента.

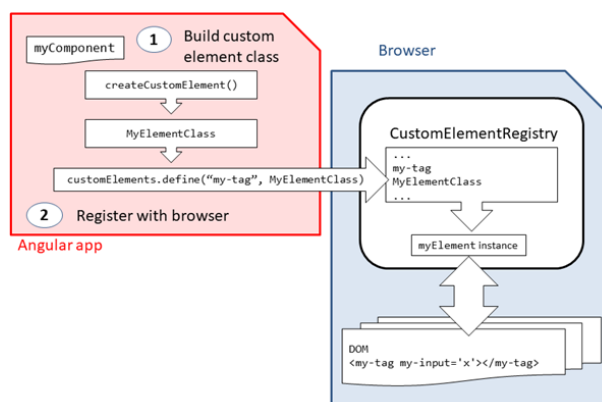


Рис. 4.10. Мост между данными и логикой, определенной в компоненте, и стандартными API-интерфейсами DOM.

Пользовательский элемент содержит компонент Angular, обеспечивая мост между данными и логикой, определенной в компоненте, и стандартными API-интерфейсами DOM (рис. 4.10). Свойства и логика компонента отображаются непосредственно в атрибуты HTML и систему событий браузера. API создания анализирует компонент в поисках входных свойств и определяет соответствующие атрибуты для настраиваемого элемента. Он преобразует имена свойств, чтобы сделать их совместимыми с настраиваемыми элементами, которые не распознают регистр символов [19]. В именах полученных атрибутов

используются строчные буквы, разделенные тире. Например, для компонента с соответствующим настраиваемым элементом определяется атрибут `@Input('myInputProp')` `inputPropmy-input-prop`. Выходные данные компонентов отправляются как настраиваемые события HTML, при этом имя настраиваемого события совпадает с именем вывода. Например, для компонента с соответствующим настраиваемым элементом будут отправляться события с именем «valueChanged», а отправленные данные будут сохранены в свойстве события. Если указывать псевдоним, используется это значение [18]. Например, приводит к отправке событий с именем «myClick».

```
@Output() valueChanged=new EventEmitter();
@Output('myClick') clicks =
new EventEmitter<string>();
```

Недавно разработанная функция веб-платформы пользовательских элементов в настоящее время изначально поддерживается в ряде браузеров (рис. 4.11).

Браузер	Поддержка настраиваемых элементов
Хром	Поддерживается изначально.
Edge (на основе хрома)	Поддерживается изначально.
Fire Fox	Поддерживается изначально.
Опера	Поддерживается изначально.
Сафари	Поддерживается изначально.

Рис. 4.11.Список браузеров поддерживающих пользовательские элементы.

В браузерах, которые изначально поддерживают пользовательские элементы, спецификация требует, чтобы разработчики использовали классы ES2015 для определения пользовательских элементов - разработчики могут отказаться от этого, установив target: "es2015" свойство в файле конфигурации TypeScript своего проекта .

```
ng add @angular/elements --project=*your_project_name*
```

Рис. 4.12. Angular CLI установка Angular Elements.

Рекомендуется использовать Angular CLI для автоматической настройки проекта с правильным полифилом (рис. 4.12) [8]. Поскольку поддержка Custom Element и ES2015

может быть доступна не во всех браузерах, разработчики могут вместо этого использовать полифил для поддержки старых браузеров и кода ES5 [1]. Раньше, когда требовалось добавить компонент в приложение во время выполнения, нужно было определить динамический компонент, а затем загрузить его, прикрепить к элементу в DOM и подключить все зависимости, изменив обнаружение и обработку событий. Использование настраиваемого элемента Angular делает процесс намного проще и прозрачнее за счет автоматического предоставления всей инфраструктуры и фреймворка - все, что нужно сделать, это определить тип обработки событий [19]. Все равно нужно исключить компонент из компиляции, если не предполагается использование его в своем приложении. Универсальные API - интерфейсы DOM, такие как `document.createElement()` или `document.querySelector()`, возвращают тип элемента, соответствующий указанным аргументам. Например, вызов `document.createElement('a')` вернет объект `HTMLAnchorElement`, который, как известно TypeScript, имеет свойство `href`. Точно так же `document.createElement('div')` вернет объект `HTMLDivElement`, который, как известно TypeScript, не имеет свойства `href`. При вызове с неизвестными элементами, такими как имя настраиваемого элемента (например `popup-element`), методы будут возвращать общий тип, например `HTMLElement`, поскольку TypeScript не может определить правильный тип возвращаемого элемента. Пользовательские элементы, созданные с помощью Angular, расширяются с `NgElement` (который, в свою очередь, расширяется `HTMLElement`) [18]. Кроме того, эти настраиваемые элементы будут иметь свойство для каждого входа соответствующего компонента. Например, у `popup-element` будет `message` свойство типа `string`. Есть несколько вариантов, если требуется получить правильные типы для пользовательских элементов. Предположим, создается `my-dialog` собственный элемент на основе следующего компонента:

```
@Component(...)
class MyDialog {
  @Input() content: string; }
```

Самый простой способ получить точную типизацию - привести возвращаемое значение соответствующих методов DOM к правильному типу. Для этого, можно использовать `NgElement` и `WithProperties` тип (как экспортируемый из `@angular/elements`):

```
const aDialog = document.createElement('my-dialog')
as NgElement & WithProperties<{content: string}>;
aDialog.content = 'Hello, world!'; aDialog.content =
123; // <-- ERROR: TypeScript knows this should be a
string. aDialog.body = 'News'; // <-- ERROR:
TypeScript knows there is no `body` property on
```



```
`aDialog`.
```

Это хороший способ быстро получить возможности TypeScript, такие как проверка типов и поддержка автозаполнения, для пользовательского элемента. Но это может стать громоздким, если это нужно в нескольких местах, потому что приходится приводить возвращаемый тип при каждом возникновении [19]. Альтернативный способ, который требует определения типа каждого настраиваемого элемента только один раз - это расширение `HTMLTagNameMap`, которое TypeScript использует для определения типа возвращаемого элемента на основе его имени тега (для методов DOM, таких как `document.createElement()`, `document.querySelector()` и т.д.). Теперь TypeScript может определить правильный тип так же, как и для встроенных элементов:

```
document.createElement('div')    //--> HTMLDivElement
(built-in element) document.querySelector('foo') //--
->      Element      (unknown      element)
document.createElement('my-dialog')  //--> NgElement
& WithProperties<{content: string}> (custom element)
document.querySelector('my-other-element')    //-->
NgElement & WithProperties<{foo: 'bar'}> (custom
element)
```

В итоге с помощью Angular Elements можно поместите компоненты Angular в другие библиотеки / фреймворки JavaScript, такие как React и Vue, осуществлять передачу данных из React и Vue в компонент Angular, использовать компонент Angular в приложении AngularJS. Разработчик Angular, знают, насколько разные Angular и AngularJS.

## 5. ОПИСАНИЕ ПРАКТИЧЕСКОЙ ЧАСТИ

### 5.1. Архитектура приложения и имплементированный функционал

В данном проекте использовались следующие технологии:

- Node.js;
- Angular;
- Web-components;
- Lazy-loading.

Языки программирования и гипертекстовой разметки:

- TypeScript;
- HTML.

Дополнительные пакеты:

- Angular Elements;
- Angular Universal.

Сторонние средства:

- npm http-server;
- npm random-words;
- Bootstrap.

Задача разработать два веб приложения для демонстрации совместной работы упомянутых технологий.

Первое приложение должно быть представлено в виде веб сервиса, предоставляющего готовые веб-компоненты. Данные компоненты многоразового использования, должны быть не зависимы от фреймворков или библиотек. Так же компоненты должны предоставляться по средствам скриптового запроса, то есть, так же как имплементируется любая JS библиотека.

Второе приложение должно быть представлено в виде пользовательского сайта, который будет демонстрировать следующие идеи:

- Ленивая загрузка модулей в SPA приложениях;
- Серверный рендеринг SPA приложения (SSR);
- Использование готовых веб-компонентов из другого источника.

Для удобства последующее упоминания о приложениях будут такими:

- widgets: для первого приложения, в котором будет создаваться сервис для веб-компонентов;

- website: для второго приложения, в котором будет создаваться пользовательское веб приложение.

Оба приложения должны быть созданы с помощью фреймворка Angular.

## 5.2. Модульная реализация приложения

Приложение Angular состоит из модулей. Обычно, приложение состоит из нескольких модулей, но как минимум оно имеет один корневой модуль, который по умолчанию называется AppModule. Модулю необходим ряд библиотек, поэтому в самом начале файла нужно их подключить. Имена библиотек Angular начинаются с префикса @angular. Библиотеки импортируются с помощью директивы import, а устанавливаются обычно через пакетный менеджер npm. Главный модуль в приложении Angular используется как точка входа. При создании нового проекта через команду в командной строке ng new project-name генерируется такой AppModule. Как аргумент в декораторе @NgModule должен использоваться JavaScript объект. @NgModule - это декоратор, который принимает объект, свойства которого описывают метаданные модуля. Наиболее важные свойства:

- declarations: классы представлений (view classes), которые принадлежат модулю. Angular имеет три типа классов представлений: компоненты (components), директивы (directives), каналы (pipes);
- exports: набор классов представлений, которые должны использоваться в шаблонах компонентов из других модулей;
- imports: другие модули, классы которых необходимы для шаблонов компонентов из текущего модуля;
- providers: классы, создающие сервисы, используемые модулем;
- bootstrap: корневой компонент, который вызывается по умолчанию при загрузке приложения.

Директива @NgModule определяет класс, как модуль. Такие классы помогают разбивать приложение на части (модули), которые взаимодействуют между собой представляя в конечном итоге целостное приложение. Другими словами, модуль – это упаковка части функционала приложения. Если нужно реализовать lazy-loading в приложении, то необходимо использовать концепцию Angular Modules при проектировании приложения [13]. В свойстве declarations объявляются компоненты, которые содержатся в этом модуле. В данном случае это компонент AppComponent. Компонентов может быть

несколько, они объявляются в этих же квадратных скобках через запятую. Допустим, у нашего компонента есть селектор `my-app`. Когда в шаблоне HTML пишется `<my-app></my-app>` приложение загружает компонент и тот HTML код, который есть в нём. AppModule говорит браузеру, о том, что нужно встроить в DOM компонент AppComponent. AppModule импортирует служебные модули Angular, такие как BrowserModule, который отвечает за работу приложения в браузере. Это сервис, который взаимодействует с нашим кодом и API браузера. Так же BrowserModule включает в себя директивы NgIf и NgFor, которые могут использоваться в шаблонах компонентов [11]. AppModule связывает данные, их представление и браузер. В приложении создаются компоненты, директивы и пайпы. Чтобы сообщить приложению какой функционал необходим, требуется добавить их через запятую в квадратные скобки в свойства `declarations` объекта, который является аргументом декоратора `@NgModule`. После того как компоненты задекларированы можно использовать их внутри других компонентов через их селектор, который указывается в описании компонента.

### 5.3. Механизм Lazy-Load в приложении

В приложении `website` механизм ленивой загрузки состоит из 7-ми дополнительных модулей. Первый модуль AppModule корневой он загружается первым и дает старт приложению.

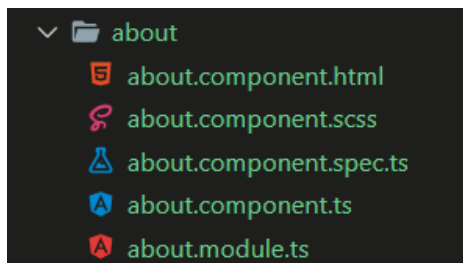


Рис. 5.1. Модуль about.

После создания модулей (рис. 5.1) в них были добавлены компоненты, к которым они должны быть привязаны (на примере модуля `about`):

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { AboutComponent } from './about.component';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', component: AboutComponent },
];

@NgModule({
```

```

    declarations: [AboutComponent],
    imports: [
        CommonModule,
        RouterModule.forChild(routes)
    ]
  })
  export class AboutModule { }

```

Затем были настроены маршруты в главном модуле:

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module')
    .then(m => m.HomeModule)
  },
  {
    path: 'galery',
    loadChildren: () => import('./galery/galery.module')
    .then(m => m.GaleryModule)
  },
  {
    path: 'contacts',
    loadChildren: () => import('./contacts/contacts.module')
    .then(m => m.ContactsModule)
  },
  {
    path: 'about',
    loadChildren: () => import('./about/about.module')
    .then(m => m.AboutModule)
  },
  {
    path: 'help',
    loadChildren: () => import('./help/help.module')
    .then(m => m.HelpModule)
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {
    initialNavigation: 'enabled'
  })],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

Таким образом был реализован механизм ленивой загрузки в приложении.

## 5.4. Использование сторонних сервисов

За последние года Angular стал одним из самых популярных фреймворков в мире front-end разработки. По этой причине многие производители сторонних сервисов (библиотек, инструментов) разрабатывают так же совместимые с Angular версии своих продуктов. Некоторые из них Angular уже использует внутри своей архитектуры (например, RxJS библиотеку) и загружает по умолчанию при создании нового проекта (но не импортирует их), другие сервисы нужно самостоятельно искать и устанавливать. Как правило местом для хранения этих сервисов вполне обоснованно является облачный сервер npm. Npm это пакетный менеджер, использующийся для скачивания или загрузки на сервер пакетов (пакетом в Node.js называется один или несколько JavaScript-файлов, представляющих собой какую-то библиотеку или инструмент), обладающий самой большой пакетной экосистемой в мире с открытым исходным кодом [13]. Именно с помощью npm мы, кстати говоря, устанавливаем на свой ПК Angular CLI и даже при создании нового проекта неявно его используем. Npm уже входит в состав программы Node.js поэтому не требует отдельной установки.

Однако Angular не обязательно нуждается в библиотеках написанных именно под него. Он вполне дружелюбен к обычным JavaScript библиотекам, таким как random-words. Эта библиотека умеет генерировать случайные слова или предложения. В ней есть возможность регулировать количество слов в предложениях. Это довольно удобная библиотека для предположения веб приложения клиенту. В данном случае, так как у приложения website нет backend'a (серверной части приложения и базы данных), появляется необходимость использовать подобные библиотеки.

Чтобы установить эту библиотеку нужно в консоли, в папке приложения ввести команду для установки её в проект Angular:

```
npm i random-words
```

Использоваться она будет следующим образом:

```
import { Component, OnInit } from '@angular/core';

declare var require;

@Component({
  selector: 'app-gallery',
  templateUrl: './gallery.component.html',
  styleUrls: ['./gallery.component.scss']
})
export class GalleryComponent implements OnInit {
  randomWords;
```

```

items = [];

constructor() { }

ngOnInit(): void {
  this.randomWords = require('random-words');
  this.generateContent(30);
}

generateContent(count = 10) {
  this.items = [];
  [...Array(count).keys()].forEach((a, i) => this.
items.push({
  description: this.sentenceOfWords(20),
  down: this.sentenceOfWords(1)
})))
}

sentenceOfWords = (count = 1) => !this.randomWords
? 'error' : this.randomWords(count).join(' ');
}

```

В приложении website, которое является клиентским и должно взаимодействовать с пользователем должны быть прописаны или имплементированы готовые стили. Для этой задачи отлично подойдет библиотека Bootstrap. Данная библиотека содержит в себе множество готовых и удобных компонентов для большого количества задач. Чтобы добавить её в проект нужно в index.html внутри тега <head></head> прописать следующий код:

```

<link href="https://cdn.jsdelivr.net/npm/bootstrap
@5.0.0-
beta2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
BmbxuPwQa2lc/FVzBcNJ7UAyJxM6wuwqIj61tLrc4wSX0szH/Ev+n
YRRuWlolflfl" crossorigin="anonymous">
<script src="https://cdn.jsdelivr.net/npm/bootstra
p@5.0.0-beta2/dist/js/bootstrap.bundle.min.js"
integrity="sha384-
b5kHyXgcpbZJO/tY9U17kGkflS0CWuKcCD38l8YkeH8z8QjE0GmW
1gYU5S9FOnJ0"
crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperj
s/core@2.6.0/dist/umd/popper.min.js"
integrity="sha384-
KsvDlyqQ1/1+IA7gi3P0tyJcT3vR+NdBTt13hSJ2lnve8agRGXTT
yNaBYmCR/Nwi"
crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstra
p@5.0.0-beta2/dist/js/bootstrap.min.js"
integrity="sha384-
nsg8ua9HAw1y0W1btsyWgBklPnCUAFLuTMS2G72MMONqmOymq585
AcH49TLBQObG"
crossorigin="anonymous"></script>
</head>

```

Это cdn для скачивания библиотек Bootstrap во время загрузки приложения браузером.

Далее уже внутри других компонентов можно прописывать стили:

```
<footer class="container py-5">
  <div class="row">
    <div class="col-12 col-md"></div>
  </div>
</footer>
```

Таким образом наше приложение приобретает презентабельный внешний вид.

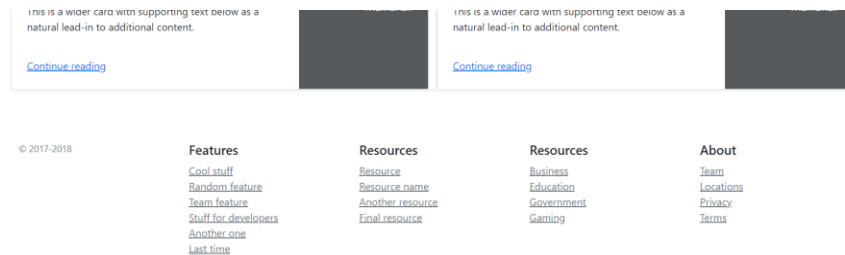


Рис. 5.2. Footer для сайта созданный с помощью Bootstrap

Таким образом (рис. 5.2) быстро и качественно преобразуется приложение за минимально затраченное время.

## 5.5. Использование Angular Universal

Обычное приложение Angular выполняется в браузере, отображая страницы в DOM в ответ на действия пользователя. Angular Universal выполняется на сервере, генерируя статические страницы приложений, которые позже загружаются на клиенте. Это означает, что приложение обычно отображается быстрее, что дает пользователям возможность просмотреть макет приложения, прежде чем он станет полностью интерактивным. Так же это дает более лучшую взаимодействию с SEO [15].

Установка происходит следующим образом. Чтобы создать серверный модуль приложения `app.server.module.ts`, выполните следующую команду CLI:

```
ng add @nguniversal/express-engine
```

Это создаст особую структуру папок (рис. 5.3).



src/	
index.html	app web page
main.ts	bootstrapper for client app
main.server.ts	* bootstrapper for server app
style.css	styles for the app
app/ ...	application code
app.server.module.ts	* server-side application module
server.ts	* express web server
tsconfig.json	TypeScript base configuration
tsconfig.app.json	TypeScript browser application configuration
tsconfig.server.json	TypeScript server application configuration
tsconfig.spec.json	TypeScript tests configuration

Рис. 5.3. Созданные модули для SSR (отмечены знаком «\*»)

В главном модуле приложения необходимо добавить строчку в imports:

```
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports: [
    BrowserModule.withServerTransition({ appId: 'serverApp' })
  ],
})
export class AppModule { }
```

Теперь можно запускать рендеринг с помощью Universal используя команду CLI:

```
npm run dev:ssr
```

В консоли должен появиться результат успешной сборки приложения (рис. 5.4).

Build at: 2021-04-18T20:40:25.264Z - Hash: 69f8198e5f2516833e37 - Time: 8653ms		
✓ Server application bundle generation complete.		
Initial Chunk Files	Names	Size
main.js	main	6.70 MB
	Initial Total	6.70 MB
Lazy Chunk Files	Names	Size
help-help-module.js	help-help-module	23.38 kB
vendors~galery-galery-module.js	vendors~galery-galery-module	19.68 kB
home-home-module.js	home-home-module	16.26 kB
about-about-module.js	about-about-module	12.87 kB
contacts-contacts-module.js	contacts-contacts-module	11.52 kB
galery-galery-module.js	galery-galery-module	8.74 kB
Build at: 2021-04-18T20:40:26.893Z - Hash: eb99ba8087f93b1f42c7 - Time: 13997ms		
Compiled successfully.		

Рис. 5.4. Результат успешного выполнения команды.

Навигация по routerLinks будет работать правильно, поскольку используются собственные <a> теги anchor() [15]. Переход от приложения, отрисованного на сервере, к клиентскому приложению происходит быстро на машине разработки, однако всегда нужно тестировать свои приложения в реальных сценариях.

Стоит помнить, что приложение больше не запускается в браузере. Некоторые его API-интерфейсы и возможности просто отсутствуют на сервере. К примеру, серверные приложения не умеют ссылаться на глобальные глобальные объекты API, такие как navigator, window, location, document. Angular предоставляет некоторые абстракции над этими объектами, как например Location, DOCUMENT. В большинстве случаев это заменяет API. Но если Angular не предоставляет нужных абстракций, можно написать новые, которые делегируют API-интерфейс браузера, находясь в браузере, и альтернативную реализацию, находящуюся на сервере (также называемую шиммингом). Точно так же без событий мыши или клавиатуры серверное приложение не может полагаться на то, что пользователь нажимает кнопку, чтобы отобразить компонент.

```
import { PLATFORM_ID, Inject } from '@angular/core'
import { isPlatformBrowser } from '@angular/common'

export class MyComponent {
  browserIsReady: boolean;
  constructor( @Inject(PLATFORM_ID) platformId:
    string) {
    this.browserIsReady=isPlatformBrowser(platformId
  );
    if (this.browserIsReady) { //... using }
  }}
}
```

Приложение должно определять, что отображать, исключительно на основе входящего клиентского запроса. Это хороший аргумент в пользу маршрутизации приложения.

## 5.6. Использование Angular Elements

Для того, чтобы использовать Angular Elements в создании веб сервера, для начала необходимо создать простое Angular приложение. Далее следует добавить компоненты, которые будут в последующем играть роль загружаемых веб-компонентов.

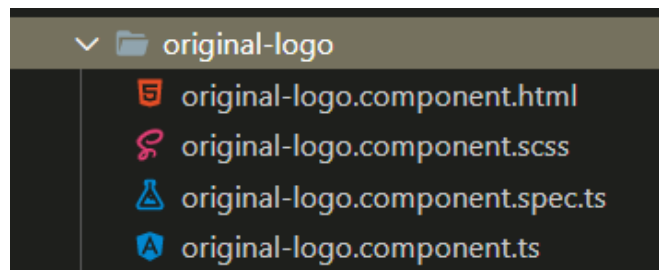


Рис. 5.5. Созданный компонент для хранения оригинального логотипа.

По структуре парок эти компоненты ничем не отличаются от обычных (рис. 5.5). Внутри компоненты так же ничем не отличаются от стандартных.

Теперь необходимо установить библиотеку Angular Elements с помощью CLI:

```
ng add @angular/elements
ng add ngx-build-plus
```

Первая команда обеспечивает поддержку Angular элементов. Вторая расширяет CLI для конечной сборки проекта. Так же необходимо установить модуль *http-server*:

```
npm i -g http-server -save
```

Это простой http-сервер командной строки с нулевой конфигурацией. С помощью него будет запускаться приложение. Следующим шагом необходимо создать веб компоненты на основе обычных компонентов. Делается это в главном модуле приложения:

```
import { BrowserModule } from '@angular/platform-browser';
import { Injector, NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { OriginalLogoComponent } from './original-logo/original-logo.component';
import { createCustomElement } from '@angular/elements';
import { environment } from 'src/environments/environment';
import { ContentOneComponent } from './content-one/content-one.component';
import { ContentTwoComponent } from './content-two/content-two.component';

@NgModule({
  declarations: [
    AppComponent,
    OriginalLogoComponent,
    ContentOneComponent,
    ContentTwoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [
    ...(!environment.production && [AppComponent]) ||
  ])
  ],
  entryComponents: [
    OriginalLogoComponent,
    ContentOneComponent,
    ContentTwoComponent
  ]
})
export class AppModule {
  constructor(private injector: Injector) {}
```

```

ngDoBootstrap() {
  customElements.define('original-
logo', createCustomElement(OriginalLogoComponent, {
injector: this.injector }));
  customElements.define('content-
one', createCustomElement(ContentOneComponent, { inj
ector: this.injector }));
  customElements.define('content-
two', createCustomElement(ContentTwoComponent, { inj
ector: this.injector }));}}

```

Теперь нужно слегка отредактировать файл `angular.json`:

```

"architect": { "build": { "builder": "ngx-build-
plus:build", .... "serve": { "builder": "ngx-build-
plus:dev-server", ... "test": { "builder": "ngx-build-
plus:karma",

```

В этом файле хранятся конфигурации для сборки приложения. Последним шагом перед запуском приложения, является его сборка:

```

ng build --prod --output-hashing none --single-
bundle true

```

После первой сборки в файлах проекта появляется новая папка «`dist`» (рис. 5.6).

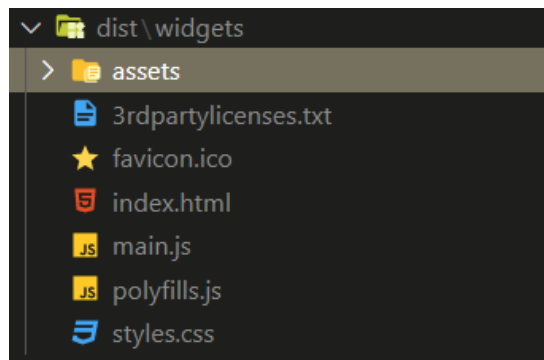


Рис. 5.6. Папка «`dist`»

Эта папка содержит в себе готовое для публикации на сервер приложение со всеми необходимыми файлами и стилями.

Запуск приложения осуществляется с помощью модуля `http-server`:

```

http-server ./dist/widgets -p 8081

```

Теперь данный сервис готов к использованию. Осталось лишь настроить веб сайт который будет к нему обращаться. Для этого в `index.html` сайта добавляется строки с полифилами для браузеров, которые не поддерживают веб компоненты, а так же строка обращения к веб сервису:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/zone.js/0.9.1/zone.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/webcomponentsjs/2.2.10/custom-elements-es5-adapter.js"></script>
<script type="text/javascript" src="http://localhost:8081/main.js"></script>
```

Так же необходимо настроить Angular так чтобы можно было добавлять custom components. Это делается в главном модуле приложения:

```
import { CUSTOM_ELEMENTS_SCHEMA }
from '@angular/core';
@NgModule({
  schemas: [ CUSTOM_ELEMENTS_SCHEMA ]
})
export class AppModule { }
```

Стоит помнить, что данную настройку модуля необходимо применить ко всем модулям где предполагается использование веб компонентов. Осталось только добавление сами веб компонентов на сайт. Это делается простым добавлением тега в шаблон:

```
<original-logo></original-logo>
```

Таким образом можно настроить и использовать Angular Elements.

## ЗАКЛЮЧЕНИЕ

Разработка веб приложений с помощью подобных технологий по началу кажется сложной, в итоге оказывается довольно интересной. Причиной этому служит тот факт, что это современные технологии, а значит они должны заинтересовывать и вдохновлять. Специальность «информатика» всегда подразумевает постоянный рост знаний в области и изучение новых технологических решений. Сейчас в лучшее из времен, где имеется доступ ко всем мировым знаниям и большинству современных технологий. Опубликован большой список необходимой и понятной литературы, которую можно совершенно бесплатно загрузить на своё мобильное устройство или же распечатать (кому как удобно). Поэтому на сегодняшний день практически не бывает нерешенных задач.

Lazy-Loading оптимизирует работу приложения и повышает защиту, позволяет одностраничным веб приложениям (SPA) загружаться частями. Одностраничные веб приложения (SPA) не требуют перезагрузки браузера. Это значит, что JavaScript код не будет прерываться и может хранить данные во время всего сеанса. Более того отсутствует потребность загружать страницу с данными целиком, а значит загружать можно только чистые данные.

Angular Universal - технология, которая позволяет разработчикам выполнять рендеринг приложений Angular на стороне сервера в различных сценариях. Обращение к API в Angular Universal осуществляется только по абсолютным URL. Но в случае, если приложение и API находятся на одном сервере, то клиентское приложение должно выполнять запросы по относительному URL, в то время как серверное приложение должно обращаться по абсолютному пути. Это проблема автоматически решается модулем Angular Express Engine.

Angular Elements - это компоненты Angular, упакованные в виде настраиваемых элементов, веб-стандарта для определения новых элементов HTML независимо от платформы. С помощью Angular Elements можно сделать компоненты действительно многоразовыми. Это означает, что можно использовать компоненты Angular в других фреймворках и библиотеках, таких как React, Vue и Ember. С помощью Angular Elements можно добавить Angular, известный как интерфейсный фреймворк, в бэкэнд проекта. Angular Elements также позволяет самостоятельно разрабатывать и публиковать части приложения. Используя Angular Element, все, что необходимо сделать, это динамически вставить тег, и Angular сам создаст для нас экземпляр компонента.

## БИБЛИОГРАФИЯ

1. FARRELL, БУТ. Web Components in Action. 2019. ISBN 9781617295775;
2. Lazy loading. [On-line]. [14.02.2021]. Disponibil:  
[https://en.wikipedia.org/wiki/Lazy\\_loading](https://en.wikipedia.org/wiki/Lazy_loading);
3. An object that doesn't contain all of the data you need but knows how to get it. [On-line]. [05.03.21]. Disponibil: <https://martinfowler.com/eaCatalog/lazyLoad.html>;
4. How to Design Your Site to Make it Super-fast. [On-line]. [05.01.21]. Disponibil:  
<https://www.awwwards.com/how-to-design-your-site-to-make-it-super-fast.html>;
5. LINCOLN, John. Angular Universal: всё, что нужно знать SEO-специалисту. [On-line]. [05.01.21]. Disponibil: <https://searchengines.guru/ru/articles/2020148>;
6. MILER, Jason, OSMANY, Addy. Rendering on the Web. [On-line]. [26.11.2020].  
Disponibil: <https://developers.google.com/web/updates/2019/02/rendering-on-the-web>;
7. BORGGREVE, Bram. Server-Side Enterprise Development with Angular. 2018. ISBN 1789806267;
8. Официальная документация фреймворка Angular. [On-line]. [04.11.2020].  
Disponibil: <https://angular.io/docs>;
9. FOWLER, Martin. Шаблоны корпоративной архитектуры приложений. Addison-Wesley. С. 200–214. 2003. ISBN 0-321-12742-0;
10. ORM Lazy Loading Pitfalls. [On-line]. [12.01.2020]. Disponibil:  
<http://gorodinski.com/blog/2012/06/16/orm-lazy-loading-pitfalls>;
11. FREEMAN, Adam. Pro Angular 2017. 2018. ISBN 978-5-4461-0451-2;
12. Routing & Navigation. The Basics. [On-line]. [07.02.2021]. Disponibil:  
<https://angular.io/guide/router>;
13. FAIN, Yakov, MOISEEV, Anton. Angular 2 Development With Typescript. 2018. ISBN 978-5-4461-0496-3;
14. Introduction to services and dependency injection. [On-line]. [06.02.2021]. Disponibil:  
<https://angular.io/guide/architecture-services>;
15. Server-side rendering (SSR) with Angular Universal. [On-line]. [01.02.2021].  
Disponibil: <https://angular.io/guide/universal>;
16. HOROSHEV, А. Введение в управление проектированием механических систем: Учебное пособие. 1999. ISBN 5-217-00016-3.;

17. Lazy Loading Feature Modules. [On-line]. [08.02.2021]. Disponibil:  
<https://angular.io/guide/lazy-loading-ngmodules>;
18. Официальная документация web-components. [On-line]. [12.02.2021]. Disponibil:  
<https://www.webcomponents.org/introduction>;
19. Официальная документация Angular Elements. [On-line]. [25.02.2021]. Disponibil:  
<https://angular.io/api/elements>.