University of Novi Sad
Technical Faculty “Mihajlo Pupin”
Zrenjanin

Using MongoDB, Express and Angular in Web Application Development

Original Serbian title: Primena MongoDB, Express i Angular Javaskript radnog okvira u izradi veb aplikacije

- Bachelor Thesis -

Student: Jovan Đukić
Programme: Information Technology
Student ID: IT 11/15

Zrenjanin, 2019

English revised translation, 2026

University of Novi Sad
Technical Faculty “Mihajlo Pupin”
Zrenjanin

Using MongoDB, Express and Angular in Web Application Development

Original Serbian title: Primena MongoDB, Express i Angular Javaskript radnog okvira u izradi veb aplikacije

- Bachelor Thesis -

Mentor: Doc. dr Ljubica Kazi

Student: Jovan Đukić
Programme: Information Technology
Student ID: IT 11/15

Zrenjanin, 2019

English revised translation, 2026

Editorial Note

This English edition was prepared in 2026 from the final Serbian version of the bachelor thesis accepted by the mentor and defended in 2019 before the faculty committee. It preserves the original structure, implementation, figures, historical examples, and source list. The translation corrects grammar, terminology, typographical errors, and a limited number of obvious technical inaccuracies where the intended meaning is clear. It does not modernize the 2019 application or rewrite the thesis as current technical guidance. Screenshots and diagrams remain in their original form where appropriate.

Contents

1. INTRODUCTION

The popularity of the Internet has grown enormously in the modern era. Computers were once a luxury reserved for government institutions and large companies. Today, they are found in almost every home, in people's pockets, and even in cars and household appliances. These devices are connected through the Internet.

User expectations have grown as well. Internet applications must be fast, continuously available, capable of processing large amounts of data, and able to meet a wide range of modern needs, from entertainment to everyday business operations.

At the same time, traditional technologies make these goals increasingly difficult to achieve. Development takes a long time, costs a great deal, and adapts poorly to frequent changes. JavaScript offers one way to address this problem because the language can be used throughout a web application.

This thesis examines the use of the JavaScript programming language to create modern web applications, from the user interface and business logic to the database—all with one language on one platform.

The chapter "Theoretical Foundations" discusses non-relational databases as flexible alternatives to traditional relational databases, the role of libraries and frameworks in application development, and types of programming languages.

The chapter "Existing Solutions" describes several internationally known organizations whose products and operations used the JavaScript platform at the time of writing.

The fourth chapter, "Description of the Technologies Used", provides a concise introduction to the technologies used in Chapters 5 and 6 to create the example application.

2. THEORETICAL FOUNDATIONS

2.1 Non-relational Databases

Non-relational databases are a broad category of database systems created to support more flexible and dynamic work with large amounts of data. They are also called NoSQL databases. Depending on the database, NoSQL systems may use document, key–value, graph, wide-column, or other non-relational data models rather than SQL tables and queries. [1]

In a document database such as MongoDB, data is stored as documents rather than rows and columns in traditional relational tables. These documents use a JSON-like structure.

The data model does not need to follow a rigid relational schema defined in advance. A document database is flexible enough for document structures to change as application requirements evolve.

Documents are grouped into collections, which play a role similar to tables in a relational database management system. A collection might contain registered users, vehicles in a taxi company, students at a faculty, or similar entities. One member of a collection is called a document, analogous to a row in a table.

Instead of issuing SQL queries, an application manipulates data through operations provided by the database. These commonly take the form of functions supplied with the arguments needed to insert, update, or delete data.

An important feature of many NoSQL databases is their ability to handle large amounts of data through replication and partitioning. The specific benefits and trade-offs depend on the database and its data model.

2.2 Software Frameworks

Software frameworks are tools used to develop applications more quickly and consistently. Many features and implementation concerns recur in every application. Frameworks abstract commonly repeated code, reducing duplicated effort, errors, development time, and cost.

By providing a large amount of prewritten code and implemented functionality, frameworks can make application development faster, easier, and less expensive. When several people work on an application with the same framework, shared conventions can also improve collaboration.

A framework resembles a code library, but it usually defines more of the application's structure and execution flow. Developers do not modify the framework itself; they configure and extend it with application-specific functionality.

Frameworks exist for many programming languages. Popular examples include Spring

for Java, Ruby on Rails for Ruby, and Django for Python. In the JavaScript ecosystem, well-known libraries and frameworks include jQuery, Angular, React, and Vue.

2.3 Scripting Languages

Programming languages provide a human-readable way to give instructions to a computer. Computers do not understand human language, while machine instructions represented as binary values are difficult for most people to use directly. Higher-level programming languages therefore express operations through words and symbols that are easier for people to understand.

As the need for more complex programs grew, compiled languages were developed. Their statements can represent multiple machine instructions. A compiler translates source code into a form that a computer can execute.

Scripting languages are commonly associated with execution through an interpreter or runtime. An interpreter reads and executes instructions within an existing system. In practice, modern runtimes may combine interpretation, compilation, and just-in-time compilation, so the boundary is not always strict.

Compared with lower-level compiled languages, interpreted languages often have simpler syntax, are easier to learn, and allow programs to be written more quickly with less code. The runtime handles concerns such as memory management, allowing the developer to focus on program behavior.

Their trade-offs vary by implementation, but interpreted or dynamic runtimes may execute more slowly, use more memory, and detect some classes of errors later.

Well-known scripting languages include Perl, PHP, Python, and JavaScript.

JavaScript was originally used mainly to add dynamic and interactive elements to web pages. It ran inside web browsers, so it was not initially used to create the same kinds of general-purpose applications as Java, C#, or Python.

JavaScript's object model also differed from the class-based object-oriented model found in Java, C#, and C++. In addition, early browsers had different JavaScript implementations, which caused compatibility problems. These limitations initially confined JavaScript largely to simple manipulation of elements on a web page.

The language reached an important turning point when ECMA, an association that standardizes information and communication systems, standardized it. [2] The first edition of the standard appeared in 1997 and the second in 1998.

ECMAScript 3, published in 1999, was an important early release that helped JavaScript gain broad adoption. Browser support continued to improve. Soon afterward, AJAX (Asynchronous JavaScript and XML) made it possible to update parts of a web page without reloading the entire page.

As AJAX became popular, JSON (JavaScript Object Notation) emerged as a format for

exchanging data between clients and servers, often replacing XML in web applications that preferred its lighter representation.

In 2009, Node.js introduced a cross-platform, open-source JavaScript runtime that allows JavaScript to run outside a web browser. [3] This made it practical to write both client and server applications with the same programming language.

MongoDB is a free, non-relational NoSQL document database. It stores documents internally as BSON, a binary format related to JSON, while client and server applications commonly exchange JSON-like data. This approach avoids mapping every document into relational tables and relationships.

Together with a large ecosystem of frameworks, libraries, and supporting tools, these technologies allow developers to use JavaScript throughout a web application. A popular group consists of MongoDB as the database, Express.js as the server-side JavaScript framework, Angular as the client framework, and Node.js as the runtime environment. These technologies are commonly called the “MEAN stack”.

JavaScript should not be confused with Java. Java is an object-oriented programming language associated with Oracle; despite the similarity in their names, the two languages are distinct.

2.4 REST Architecture

Applications built with the MEAN stack often use the REST (Representational State Transfer) architectural style. REST supports loosely coupled applications that communicate over HTTP, a common foundation for web services. It is not a protocol like SOAP (Simple Object Access Protocol), but it does define architectural constraints that guide an implementation. [4]

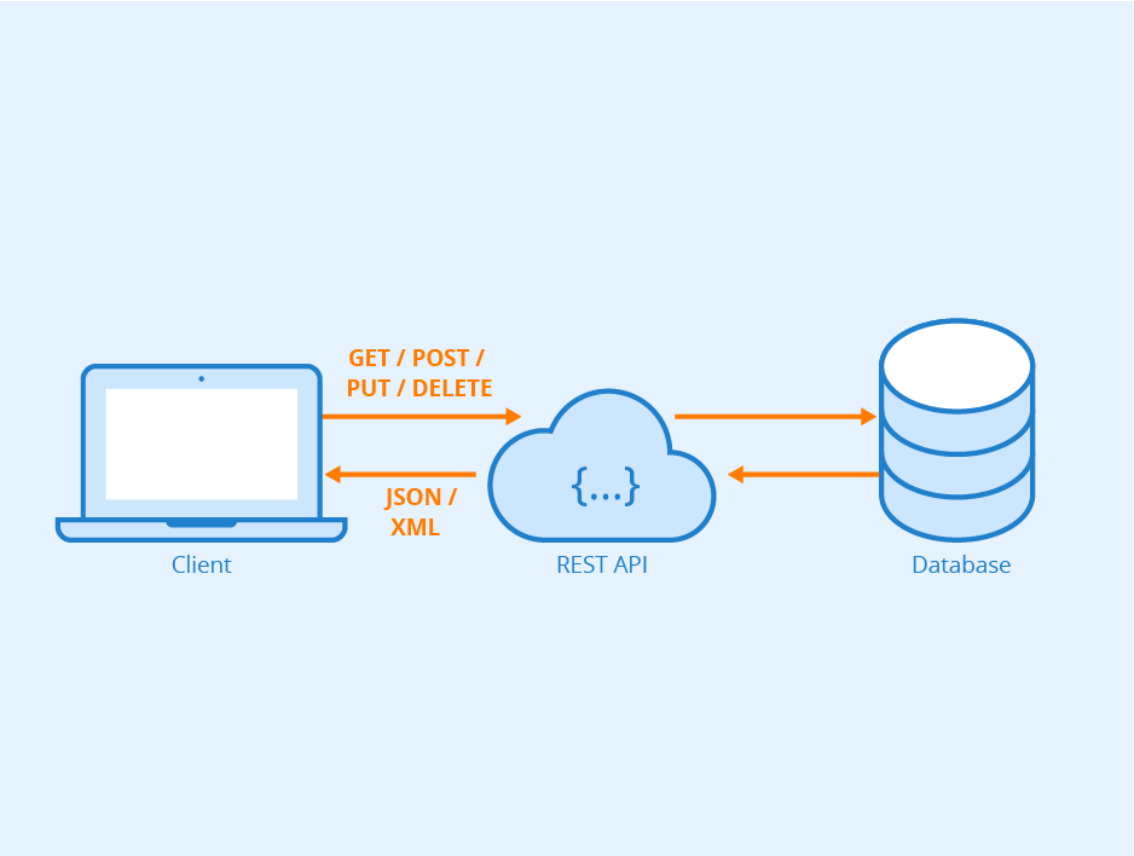


Figure 1. REST API architecture [5]

The constraints of REST architecture are: [6]

1. **Client–server**: separating the user-interface layer from the data layer improves the portability of the user interface and supports scalability by simplifying server components.
2. **Statelessness**: every client request sent to the server must contain all the information needed to understand it and cannot depend on session information stored by the server. The client therefore retains the session state.
3. **Cacheability**: a server response must state, implicitly or explicitly, whether its data may be cached. When a response is cacheable, the client cache may reuse it for later equivalent requests.
4. **Uniform interface**: applying a consistent interface simplifies the system architecture and makes interactions more visible. Four interface constraints contribute to a uniform interface: resource identification, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.
5. **Layered system**: a layered system organizes the architecture as a hierarchy. A component interacts only with its immediate layer and cannot “see” beyond it.
6. **Code on demand (optional)**: REST permits clients to extend their functionality by downloading and executing code such as scripts or applets. This can simplify clients by reducing the functionality they must implement in advance.

The central information abstraction in REST is a **resource**. A resource may be a document, image, service, person, or collection of other resources. REST uses a **resource identifier** to identify the resource involved in an interaction between components. [6]

The state of a resource at a particular moment is called a **resource representation**. [6] A representation contains data, metadata, and hypermedia links that can guide the client toward other application states.

In the simplified application architecture used in this thesis, the three layers are:

- the client, which forms the presentation layer;
- the server API (Application Programming Interface), which forms the business-logic layer; and
- the database, which forms the persistent-data layer.

Unlike a server-rendered MVC application, the presentation layer here is a separate application. It communicates with the API server, requests data for presentation to end users, and sends user-entered data that the server processes or passes to the database for storage.

A REST-based system need not have only one client application. A web application accessed through a browser, a mobile application on Android or iOS devices, and a desktop application running on Windows, macOS, or Linux can all communicate with the same server.

The number of servers is also not inherently limited. One client may communicate with several different servers, request their data, and pass data to other servers.

An Application Programming Interface is an interface through which software exposes operations. In this thesis, it is an application that runs on the server and mediates between clients and the database. It defines resources and access methods that clients request over HTTP. For example, users may be exposed as a resource with an identifier such as http://www.example.com/users. The API then defines operations on that resource, commonly create, read, update, and delete (CRUD).

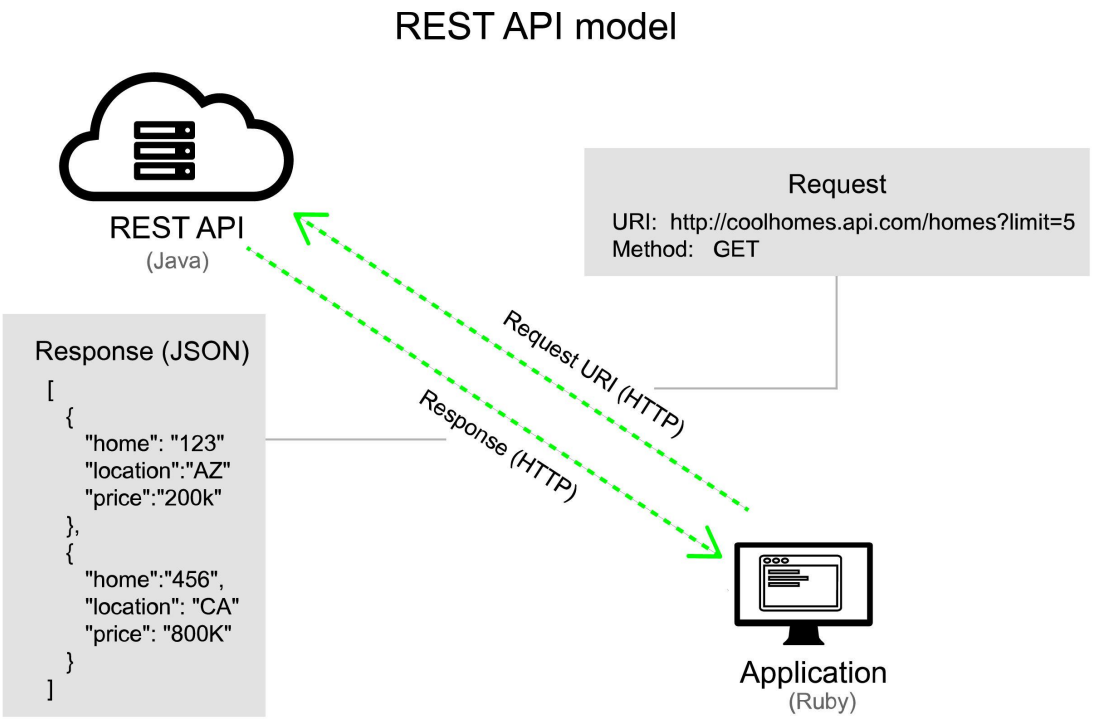


Figure 2. Example of client–server communication in a REST architecture [7]

HTTP methods such as GET and POST indicate the intended operation on a resource. For example, a client may send a GET request for the users resource. From the method and resource path, the API server determines that the client is requesting data, retrieves that data from the database or another API, and returns it to the client.

Common HTTP methods include GET, POST, PUT, PATCH, and DELETE. POST commonly creates a resource, PUT replaces a resource representation, PATCH applies a partial update, and DELETE removes a resource.

Method names express request semantics rather than implementing the operations themselves. A server could map methods to different operations, although doing so would violate established conventions. The client and server implementations may use different technologies; for example, the server may be written in Java and the client in Python, or vice versa.

RESTful API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://api.example.com/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[10]	**Delete** the entire collection.
Element URI, such as `http://api.example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.[10]	**Delete** the addressed member of the collection.

Figure 3. Examples of HTTP methods and resource operations

3. EXISTING SOLUTIONS

3.1 PayPal

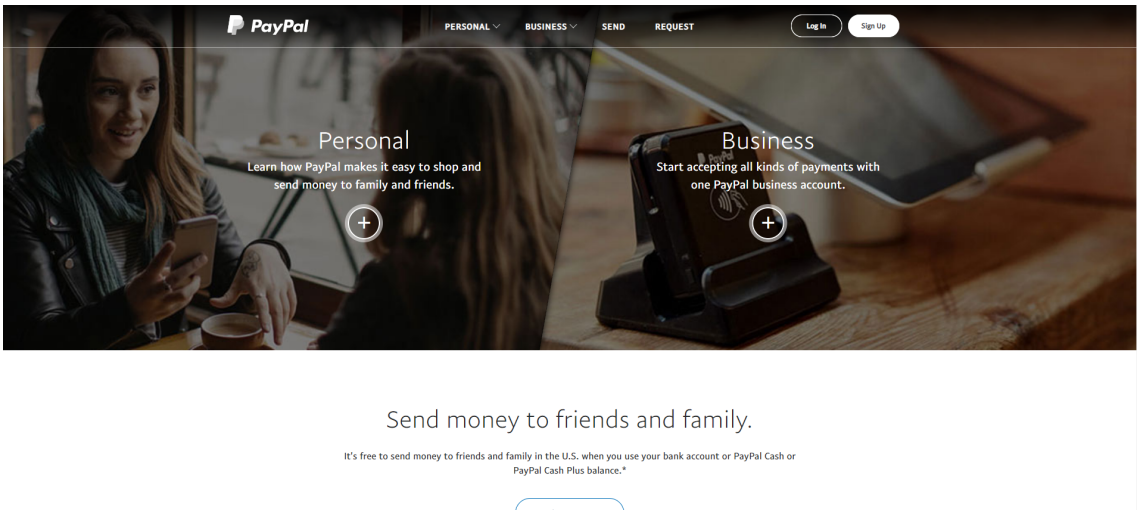


Figure 4. PayPal [8]

PayPal is an online payment platform whose early history includes X.com, founded by Elon Musk. At the time of writing, it was one of the world's most widely used methods for online payments and commerce, with 220 million users and 7.6 billion payment transactions. As membership grew, PayPal needed a way to provide better service. According to the cited case study, the move to Node.js allowed the application to be built twice as quickly with fewer people, 33% fewer lines of code, and 40% fewer files than the previous Java application. Page loading improved by 200 ms, and average response time fell by 35%. [9]

3.2 LinkedIn

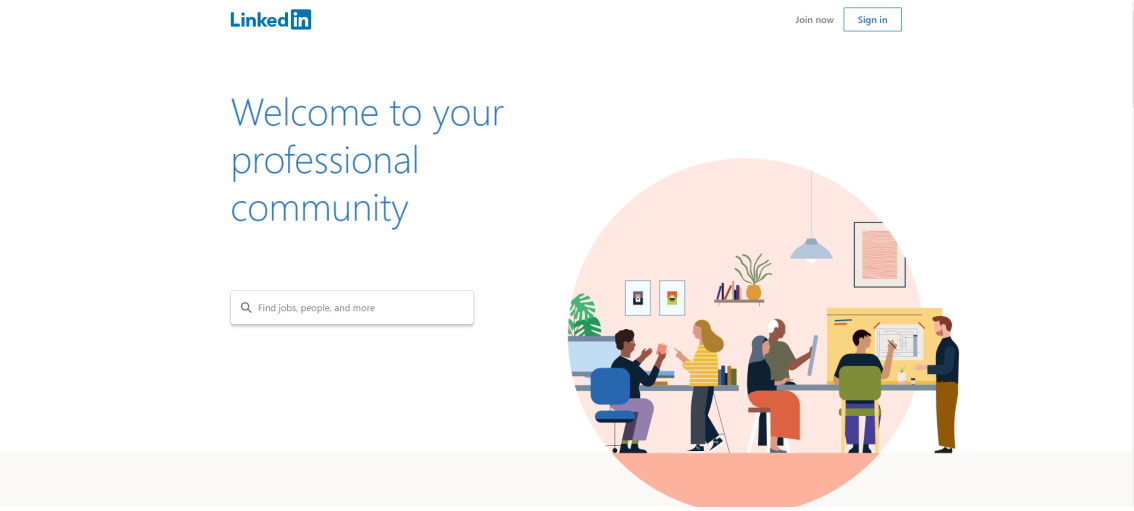


Figure 5. LinkedIn [10]

LinkedIn is a business-oriented social network founded in California in 2002. It allows users to establish professional connections with one another. At the time of writing, it was available in 24 languages and used by 400 million users in 200 countries. LinkedIn used Node.js for the server behind its mobile application.

Compared with the previous server, written in Ruby with the Ruby on Rails framework, the mobile application was reported to be 20 times faster, while the required number of servers fell from 30 to 3. Development time was also substantially reduced. [11]

3.3 Netflix

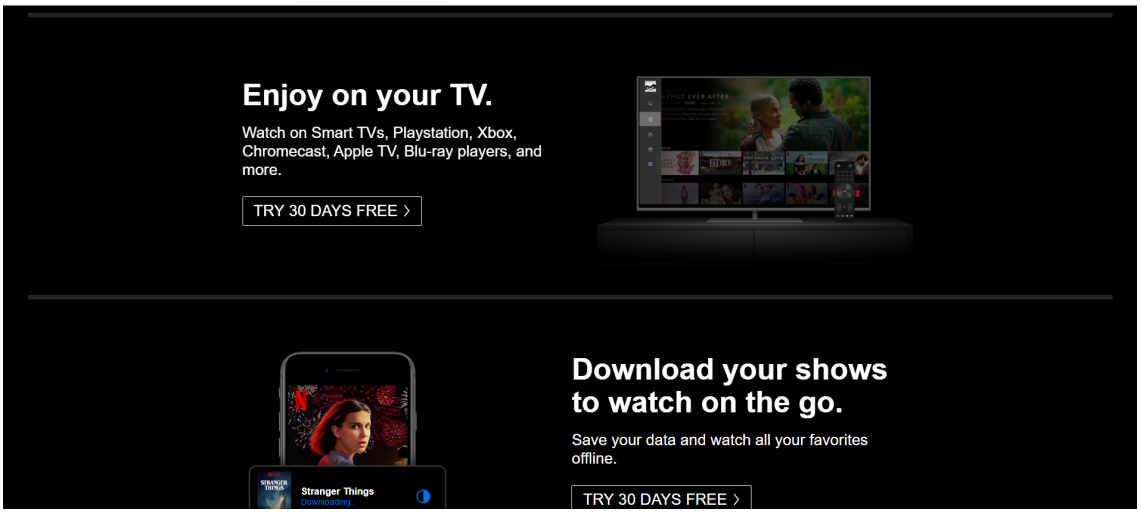


Figure 6. Netflix [12]

Netflix is a major global provider of video-streaming services. At the time of writing, it offered more than 5,500 titles, while 120 million users in 190 countries watched 100 million hours of video each day. After adopting Node.js, the company reported gains in

productivity and application performance. Application startup time fell to one minute, compared with 40 minutes for the previous Java application. [9]

3.4 Uber

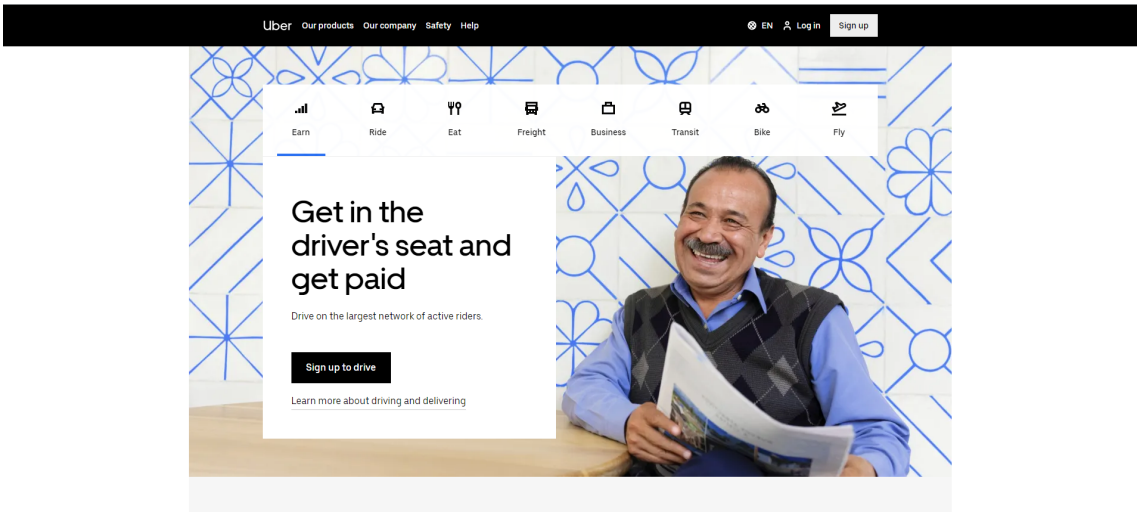


Figure 7. Uber [13]

Uber is an American company that operates a transportation network. At the time of writing, it operated in more than 40 countries and 404 cities, connecting passengers with drivers who used their own vehicles.

At the end of each trip, the fare was charged automatically to the passenger's card. Uber was among the early companies to use Node.js at large scale, building its passenger–driver matching system on the runtime.

According to the company, the benefits of Node.js included fast processing of large amounts of data, quick correction of errors, the ability to extend and deploy code frequently, and rapid development. [11]

3.5 NASA

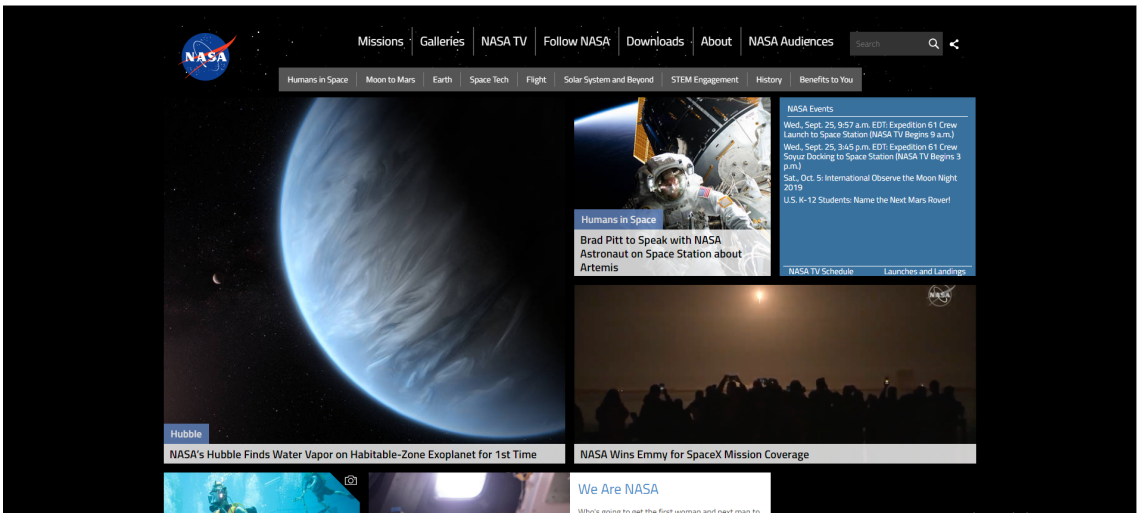


Figure 8. NASA—National Aeronautics and Space Administration [14]

NASA is an independent agency of the United States federal government responsible for the civilian space program and for aeronautics and aerospace research.

During a spacewalk in 2013, water leaked into an astronaut's helmet. In microgravity, the water quickly reached his eyes and ears, impairing his sight, hearing, and breathing. A nearby astronaut helped him return to safety.

To investigate the causes of the incident, NASA had to collect and process a large amount of data. It needed a new system that could process the information quickly and efficiently. Engineers based the system on Node.js and moved three older databases to the cloud. The resulting system allowed users to request all the relevant data from one place and reportedly improved retrieval time by 300%. [15]

4. DESCRIPTION OF THE TECHNOLOGIES USED

4.1 Angular

Angular is a JavaScript framework for creating large and complex single-page applications. This type of application uses one HTML (HyperText Markup Language) document whose contents change dynamically through JavaScript. Actions such as page-like navigation, form validation, animation, sending data to a server, and displaying newly received data occur within the application without a full page reload. Angular obtains the data needed by the user interface from external web services, commonly through asynchronous HTTP requests based on the AJAX technique.

Angular applications can be fast, scalable, and dynamic. The framework supports large and complex user interfaces on which several people or teams can work at the same time. Testability formed part of its design, which makes automated tests easier to write. It includes an extensive set of tools for developing demanding applications.

Angular is free and open-source software. Google develops and supports it together with a large community of contributors. It has extensive documentation and an active user community.

Angular is the successor to the older framework originally known by the same name. Because the earlier framework's design could not readily accommodate the planned changes, its successor became substantially different. Google therefore uses "Angular" for versions 2 and later and "AngularJS" for the older framework. This thesis uses the module-based Angular architecture current in 2019.

The framework itself is written in TypeScript. TypeScript is a superset of JavaScript: it extends the language with capabilities beyond the standard language. Its central addition is static type checking, which gives the language its name. It also provides syntax familiar from class-based object-oriented languages such as Java and C#, including classes, interfaces, inheritance, and abstraction. Developers with experience in Java or C# can therefore adapt quickly to TypeScript.

TypeScript is also an open-source language, created and developed by Microsoft.

Angular applications are written in TypeScript. TypeScript source is not compiled directly to native binary code; it is transpiled to a selected version of JavaScript that web browsers can execute.

The main building blocks of the Angular architecture described in this thesis are:

- modules;
- components;
- services;
- directives; and
- pipes.

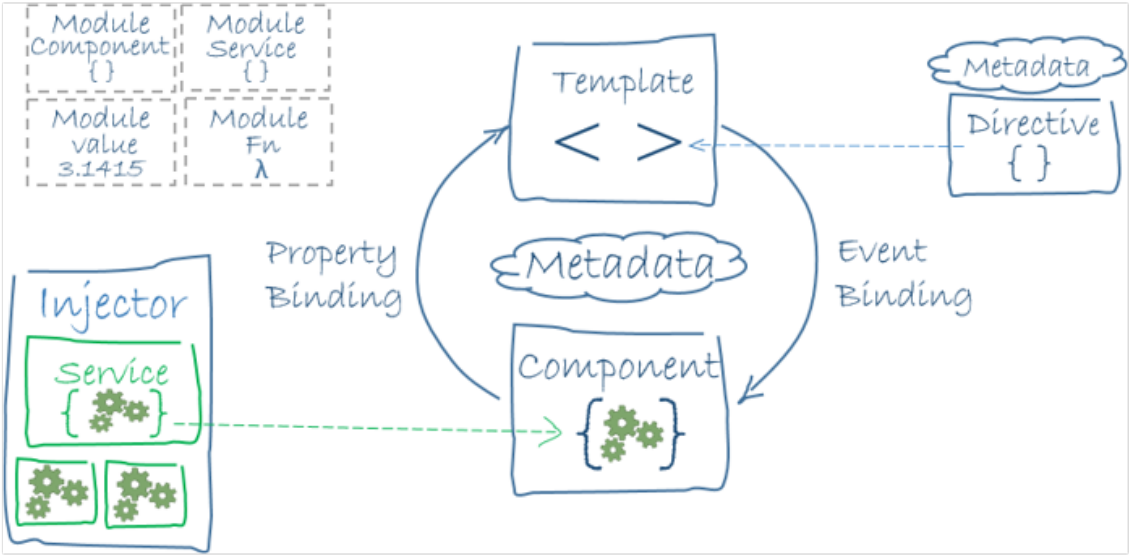


Figure 9. Angular application architecture [16]

4.1.1 Modules

Angular applications are modular, and Angular has its own module system. A module is a container for related code that belongs to one area or feature of an application. It may contain components, services, directives, and other files that together form a logical unit. [17] For example, one module might group the code for a landing page, while another might implement a dashboard. Modules can import other modules and export selected parts for use elsewhere, a pattern often used by code libraries.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Figure 10. A simple Angular module [17]

A module is a class described by Angular's `@NgModule()` decorator. The decorator supplies Angular with information about the module for later compilation.

Every Angular application in this version of the framework has at least one module, conventionally named "AppModule". It may also contain any number of feature modules,

which the main “AppModule” imports. When compiling and starting the application, Angular begins with “AppModule”.

AppModule must also have at least one component, conventionally named AppComponent. These are the two elements required for the simplest Angular application of this period.

4.1.2 Components

Components are the building blocks of the user interface in an Angular application. A component represents a part of the screen that the user sees and interacts with. Examples include a navigation menu, a list of users, a contact form, and a footer. Each part combines its component class with HTML and CSS (Cascading Style Sheets) to form a view.

Each component forms an encapsulated unit. Its HTML, CSS, and TypeScript code belong together, and Angular normally scopes component styles to that component. Components can be nested, and an application may contain many of them. Small, focused components are generally easier to understand and maintain.

Every Angular application must have at least one component, conventionally called “AppComponent”. An entire application could be implemented as one large component, but it becomes increasingly difficult to maintain as it grows. Other components are therefore nested beneath the root component.

Like a module, a component is a class described by an Angular decorator, `@Component()`. This decorator supplies metadata about the component, including the paths to its HTML and CSS files and its selector. Angular uses this information to associate a component class with its template and styles.

The component selector is the name used to place that component in a user interface. When its selector appears as an HTML element in a template, Angular renders the component’s template and applies its associated class and styles at that position.

For example, a component named “NavigationComponent” might use the selector “app-navigation” and the HTML element `<app-navigation></app-navigation>`. The custom element can then be composed with standard HTML elements. The “app” prefix helps avoid selector collisions and can be changed to suit the project.

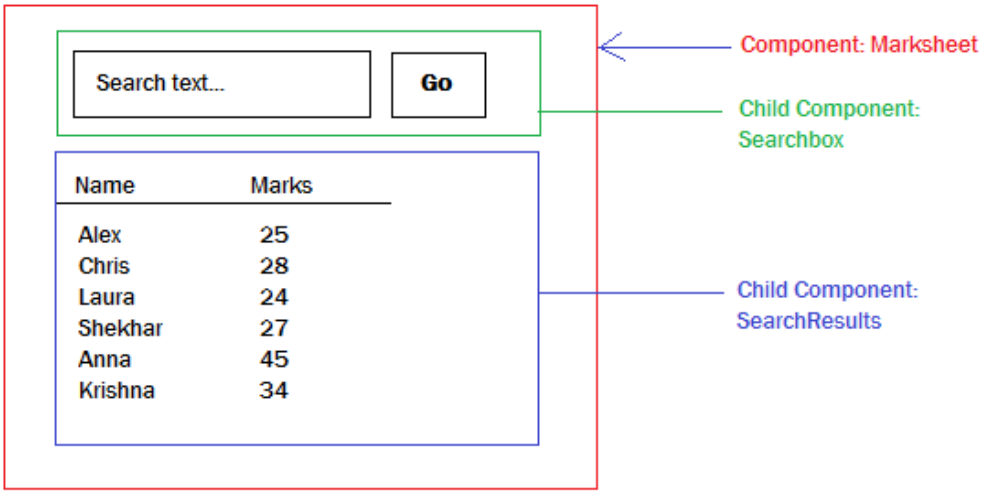


Figure 11. Example of nested components [18]

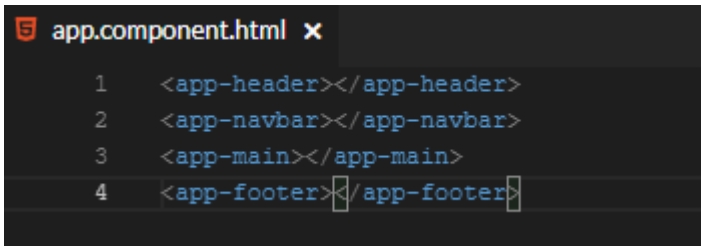


Figure 12. Example of dividing a user interface into components

4.1.3 Directives

Angular applications are highly dynamic. Directives instruct user-interface elements how to behave or change. A directive is a class described by the `@Directive()` decorator. [19]

A directive resembles a component, with one central difference: it has no template of its own. Components have a separate decorator because templates make them a distinct and central concept in Angular. [19]

The two directive categories discussed here are structural and attribute directives. [19] Angular provides built-in directives, and developers can also create custom directives when needed.

Structural directives change the structure of the user interface. They can repeat an element a specified number of times or insert and remove elements according to a condition. Two commonly used structural directives in the 2019 implementation are “*ngFor” and “*ngIf”. Their roles resemble iteration and conditional selection in programming languages.

With “*ngIf”, Angular includes the element in the DOM (Document Object Model) when the condition is true and removes it when the condition is false.

app.component.html

```
<section *ngIf="korisnkJePrijavljen === true">
  <h1>Welcome</h1>
</section>

<section *ngIf="korisnikJePrijavljen === false">
  <h1>Please log in.</h1>
</section>
```

*Figure 13. Example of the "*ngIf" directive*

The "*ngFor" directive repeats an element once for each member of a collection. The collection is an array of data that the interface needs to display. For example, if an application must display the names of five users, the directive creates five list elements and makes each user's data available to the corresponding element in the DOM.

app.component.html | app.component.ts

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

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

  korisnici = ['Petar', 'Nikola', 'Sanja', 'Danijela', 'Milos'];
}
```

Figure 14. Defining an array of users in a component

app.component.html | app.component.ts

```

<!--
  korisnici = ['Petar', 'Nikola', 'Sanja', 'Danijela', 'Milos'];
-->
<ol>
  <li *ngFor="let korisnik of korisnici">
    {{korisnik}}
  </li>
</ol>
```

*Figure 15. Applying the "*ngFor" directive to an array of users*

*Figure 16. User list produced by the "*ngFor" directive*

The asterisk is part of Angular's structural-directive shorthand. It indicates that the directive can create or remove the host element. The shorthand is required in these examples and is not used in the same way by attribute directives.

Attribute directives add or modify properties of the element to which they are applied, often changing its appearance. Angular provides several attribute directives; commonly used examples include "ngStyle" and "ngClass".

"ngStyle" and "ngClass" conditionally apply inline CSS styles and CSS classes, respectively. When the relevant logical condition is satisfied, the style or class is applied; otherwise it is omitted.

app.component.html

```
1  <p [ngClass]="{'green-bg': true}">Lorem ipsum dolor sit, amet
   consectetur adipisicing elit. Totam reiciendis velit ut ipsum cumque,
    rerum tempore praesentium voluptatem libero enim voluptas, neque
   dolores harum cupiditate perferendis iusto a fuga? Doloribus!</p>
2
3  <p [ngClass]="{'blue-bg': true}">
4    Lorem ipsum, dolor sit amet consectetur adipisicing elit.
     Perspiciatis praesentium distinctio placeat iure? Ullam, iure
     ratione! Fuga id est, ad molestias repellat maiores, iure ipsam,
     quam vitae voluptatum deserunt eaque.
5  </p>
6
7  <p [ngClass]="{'blue-bg': false}">
8    Lorem ipsum, dolor sit amet consectetur adipisicing elit.
     Perspiciatis praesentium distinctio placeat iure? Ullam, iure
     ratione! Fuga id est, ad molestias repellat maiores, iure ipsam,
     quam vitae voluptatum deserunt eaque.
9  </p>
```

Figure 17. Example of the "ngClass" directive on paragraphs

The first two paragraphs satisfy their conditions, so the first receives a green background and the second a blue one. The third paragraph does not satisfy its condition, so its background remains unchanged.

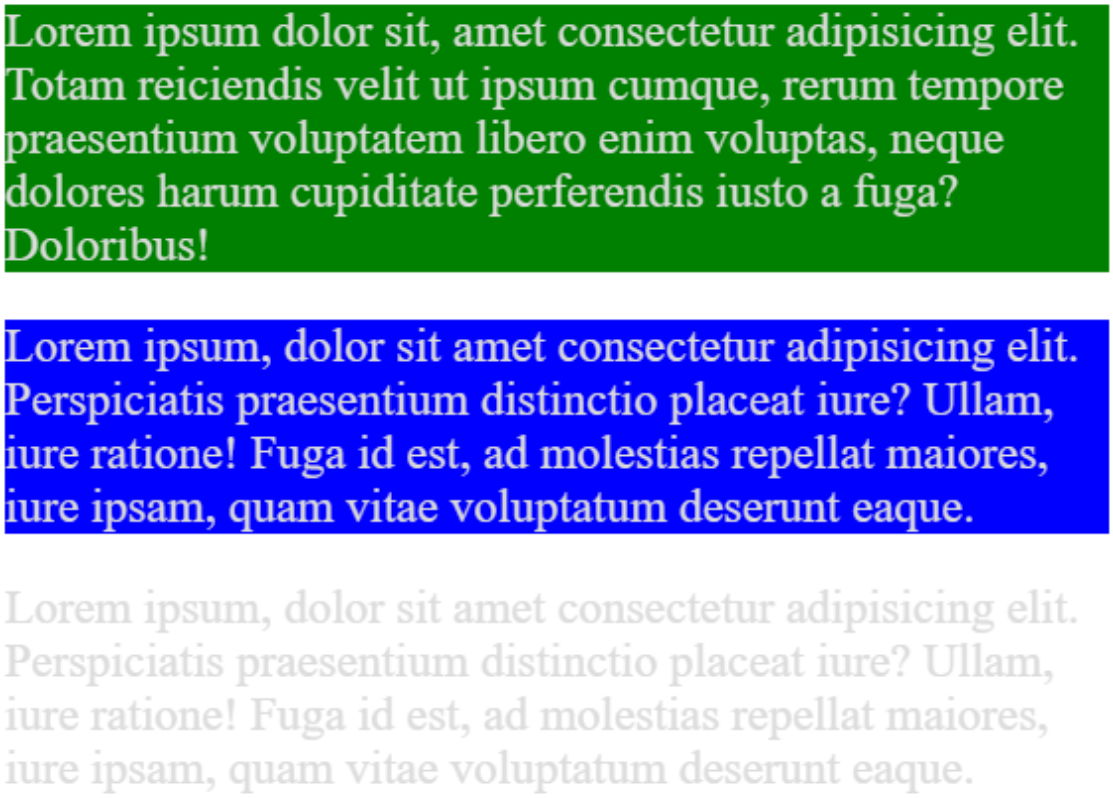


Figure 18. Dynamic assignment of CSS styles with "ngStyle"

4.1.4 Pipes

Pipes are classes described by the `@Pipe()` decorator. A pipe defines a transformation that receives a value and returns the display value used by the user interface immediately before rendering.

Frequently used pipes supplied by Angular include "PercentPipe", "DatePipe", "TitleCasePipe", "LowerCasePipe", and "CurrencyPipe". [20]

app.component.html

```
1
2   <p>Bez DatePipe: <br> {{datum}}</p>
3
4   <p>Sa DatePipe: <br> {{datum | date}}</p>
5   <p>PercentPipe: <br> {{0.24 | percent}}</p>
6   <p>TitleCasePipe: <br> {{'tiTlE caSE PiPe' | titlecase}}</p>
7   <p>LowerCasePipe: <br> {{ 'LOWERCASE PIPE' | lowercase}}</p>
8   <p>CurrencyPipe: <br> {{ 34 | currency}}</p>
```

Figure 19a. Using pipes in Angular

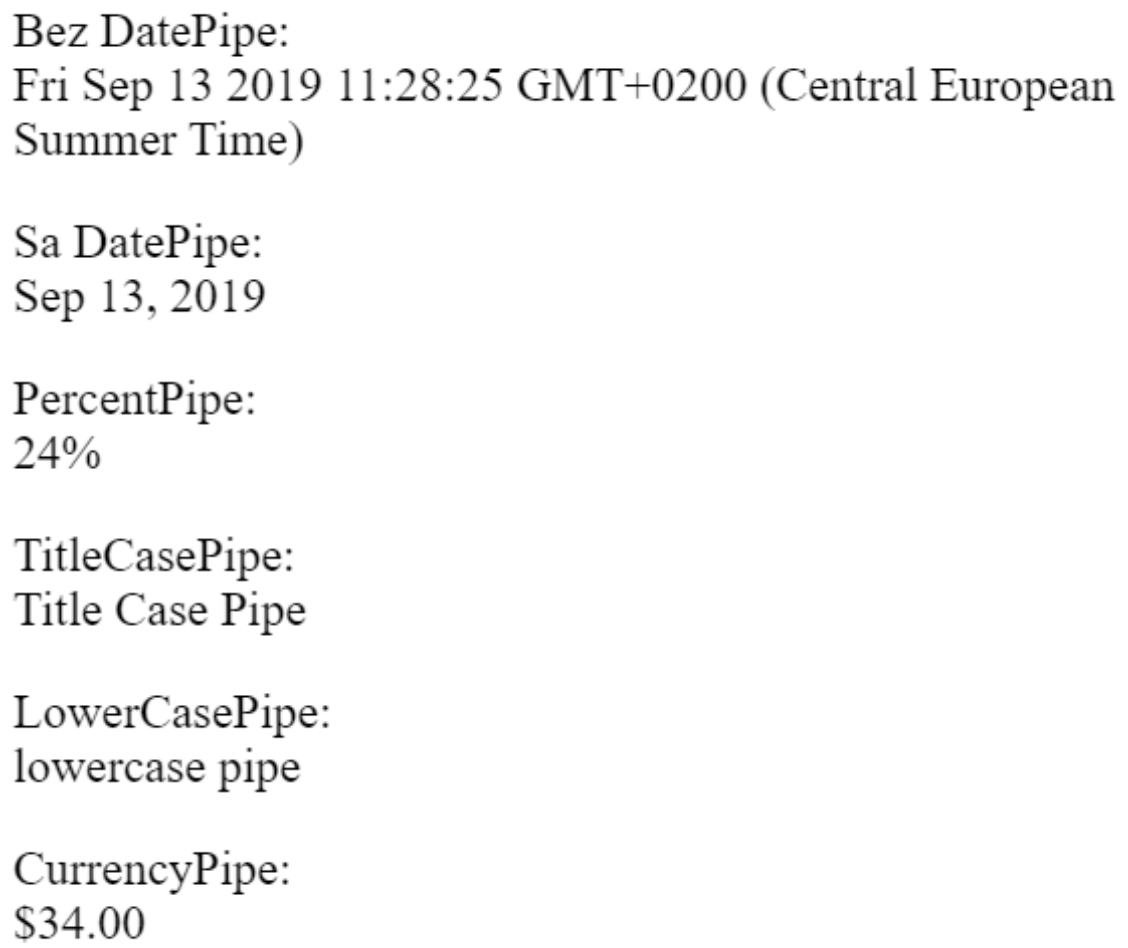


Figure 19b. Output produced by pipes

Pipes bridge the format in which a computer stores data and a format that people can read more easily. For example, a human-readable date is convenient for display even when the application stores the underlying value in another format. Pipes therefore improve the presentation of data to users.

Like the other building blocks of Angular, custom pipes can be created when the built-in transformations are insufficient.

4.1.5 Services

Services are among the simplest elements of Angular's architecture, but they serve an important role.

A service is an ordinary TypeScript class, comparable to a class in Java or C#. It may inherit from another class, implement interfaces, and declare public, private, or protected methods. A service should have a specific purpose and a small, clearly defined set of responsibilities. This improves modularity and code reuse. [22]

A service defines application logic for a particular domain. It keeps component code simpler and focused on presentation logic. Complex operations, calculations, and reusable logic can be placed in services. Services also commonly call external servers, receive data, and pass it to components, which use the data to construct the user interface.

Although business logic can be written directly in a component, doing so is considered an anti-pattern when the logic does not belong to presentation. Components should retain simple presentation logic, while services contain reusable application logic.

Angular provides a dependency-injection mechanism for making services available to components. Services use the `@Injectable()` decorator so that Angular can inject them into components or other services.

Figure 20a. Dependency injection [21]

The dependency-injection mechanism creates and manages service instances. These instances are often singletons within the scope of the configured provider. A component requests a service through a constructor parameter, and Angular performs the steps needed to supply the appropriate instance.

After a component receives a service, it can call the service's accessible methods and properties.

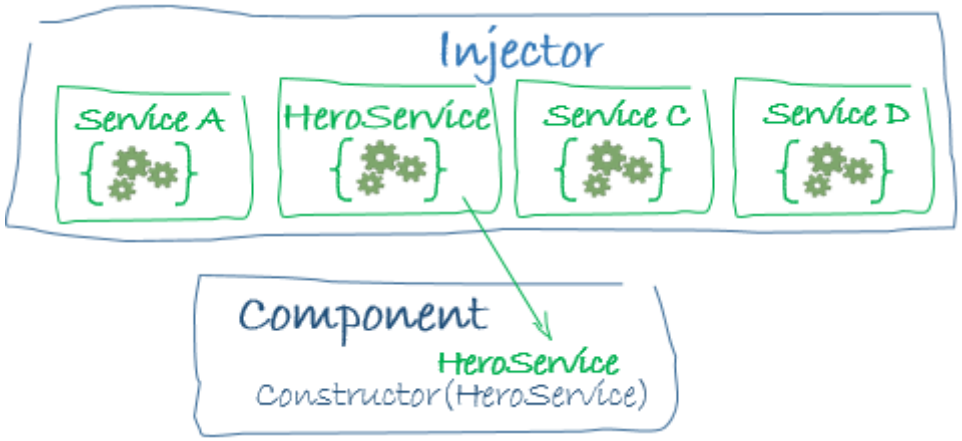


Figure 20b. Injecting a service from the container into a component [21]

Unlike the other elements described above, a service does not always need to be declared directly in a module. In the Angular version used for this thesis, it can be configured with a root-level provider so that it is available throughout the application, equivalent in effect to registering it with "AppModule".

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

@Injectable({
  providedIn: 'root',
})
export class UserService {
}
```

Figure 20c. Example of a globally provided service [22]

app.component.ts

```
1  import { Component } from '@angular/core';
2  import { UsersService } from './users.service';
3
4  @Component({
5    selector: 'my-app',
6    templateUrl: './app.component.html',
7    styleUrls: [ './app.component.css' ]
8  })
9  export class AppComponent  {
10
11   constructor(private usersService: UsersService) {
12     this.usersService.getUsers();
13   }
14 }
```

Figure 20d. Example of a component with an injected service

4.1.6 Angular CLI

Angular CLI (Command Line Interface) is a tool for quickly creating, developing, generating files for, and maintaining Angular applications through command-line commands. [23] When it initializes an Angular project, the CLI also creates the configuration needed for an efficient development workflow.

Some tasks performed by the CLI are:

- initializing the project;
- initializing a Git repository;
- configuring a development HTTP server with automatic reloading;
- compiling SASS or SCSS files;

– transpiling TypeScript files; and
– generating components, services, modules, and other building blocks.

At the end of development, the CLI prepares the application for production. This process applies a range of optimizations and compresses the code. The resulting bundle contains an HTML file, JavaScript and CSS files, and supporting static assets such as images and fonts. An HTTP server can then serve these files to users as the Angular application.

```
ng build my-app -c production
```

Figure 21. Command for creating a production build [23]

4.1.7 TypeScript

TypeScript arose from the need to make large and complex JavaScript applications easier to develop and maintain. As such applications grew, extending them became harder, and defects appeared more frequently and became more difficult to find and correct.

TypeScript is a superset of JavaScript, so every valid JavaScript program is also valid TypeScript. Developers can add language features that are absent from the chosen JavaScript target, and the TypeScript compiler—more precisely, a transpiler in this use—generates the corresponding JavaScript implementation. This can make application source shorter and clearer.

TypeScript adds static typing and class-based object-oriented syntax influenced by languages such as Java and C#. Object-oriented patterns are also possible in JavaScript, but TypeScript gives them explicit language constructs and compile-time checks.

The principal benefit discussed here is static checking of data and variable types. Explicit types allow the compiler to detect many mismatches during development, before those errors reach end users.

TypeScript also supports familiar object-oriented concepts, including abstraction, encapsulation, inheritance, polymorphism, and access modifiers such as `public`, `protected`, and `private`.

TypeScript source files use the `.ts` extension, while compilation produces ordinary JavaScript files with the `.js` extension.

```typescript
class Animal {
  constructor(public name: string) {}
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 3) {
    super.move(distanceInMeters);
  }
}

let zuca = new Dog("Zuca");

zuca.move();

```

Figure 22a. Example of inheritance and instantiation in TypeScript

```
"use strict";
var __extends = (this && this.__extends) || (function () {
    var extendStatics = function (d, b) {
        extendStatics = Object.setPrototypeOf ||
            ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
            function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
        return extendStatics(d, b);
    };
    return function (d, b) {
        extendStatics(d, b);                (parameter) d: any
        function __() { this.constructor = d; }
        d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
    };
})();
var Animal = /** @class */ (function () {
    function Animal(name) {
        this.name = name;
    }
    Animal.prototype.move = function (distanceInMeters) {
        if (distanceInMeters === void 0) { distanceInMeters = 0; }
        console.log(this.name + " moved " + distanceInMeters + "m.");
    };
    return Animal;
}());
var Dog = /** @class */ (function (_super) {
    __extends(Dog, _super);
    function Dog(name) {
        return _super.call(this, name) || this;
    }
    Dog.prototype.move = function (distanceInMeters) {
        if (distanceInMeters === void 0) { distanceInMeters = 3; }
        _super.prototype.move.call(this, distanceInMeters);
    };
    return Dog;
}(Animal));
var zuca = new Dog("Zuca");
zuca.move();
```

Figure 22b. JavaScript generated by compiling the code in Figure 22a

TypeScript lets the developer write the more concise source shown in Figure 22a. The compiler produces the equivalent JavaScript shown in Figure 22b.

4.2 Node.js and Express.js

Node.js is a platform for running JavaScript on a server. Created in 2009, it made JavaScript a practical tool for writing fast and efficient server applications.

Node.js uses the V8 JavaScript engine, which also powers the Google Chrome browser and is maintained by Google. [24] Node.js adds a rich standard library for working with the file system, operating system, cryptography, and network communication, making it straightforward to create web servers and similar applications.

Node.js is free and open-source software. After it is downloaded from https://nodejs.org/en/ and installed, the `node` command becomes available from the command line, including an interactive interface for executing JavaScript.

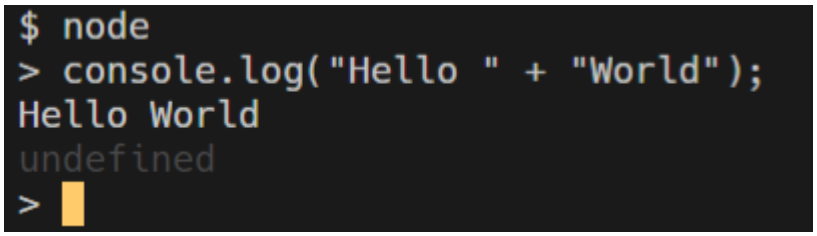


Figure 23. Executing JavaScript on a server

The Node.js installation also includes npm (Node Package Manager). In addition to the standard library, many third-party libraries and frameworks can be added to a project through npm. One of the most popular is Express.js.

Express.js is a small, fast, and minimalist JavaScript framework for building flexible web servers. It provides APIs related to HTTP and forms a thin layer over Node.js's standard HTTP facilities, making it well suited to creating an API quickly. [25]

Unlike Angular, Express imposes little application structure. Developers can organize an application according to their needs or the project's requirements. Express focuses on a small set of web-server concerns, so other aspects must be implemented separately or supplied by other libraries. For example, Express does not itself provide a complete authorization or database layer, but other libraries can be integrated with it.

Applications written with Express are often APIs that listen on a particular port. When such an application receives an HTTP request on a defined route, it handles the request and can return a response in JSON format.

```
1   const express = require('express')
2   const app = express()
3
4   app.get('/', (req, res) => res.send('Hello World!'))
5
6   app.listen(3000)
```

Figure 24. A minimal Express API

To create a simple API server, the application first imports Express and creates an application instance. The "get" method registers a handler for an HTTP GET request at the root path. When a request arrives, Express runs a function with the arguments "req" and "res", objects that represent the incoming request and outgoing response. The "send" method on "res" returns a simple "Hello World" string. Finally, "listen" starts the server on port 3000.

4.3 MongoDB

4.3.1 Basic Concepts

MongoDB, whose name is derived from the word “humongous”, is a document-oriented NoSQL database developed by the company of the same name. Instead of relational tables and rows, it organizes BSON documents into collections and offers its own query model rather than SQL joins and foreign keys. [26][1]

MongoDB aims to make document storage fast, scalable, and easy to use. Related data can be stored together in a document with a JSON-like structure. Documents in the same collection need not all have exactly the same fields, so their shape can evolve without a relational schema migration. Embedding related data can also avoid some joins, although the appropriate model depends on the application’s access patterns.

More precisely, MongoDB stores data as BSON (Binary JSON). BSON is a binary serialization format developed for MongoDB. It supports additional data types, including types for binary data, and is designed for efficient machine processing. [26]

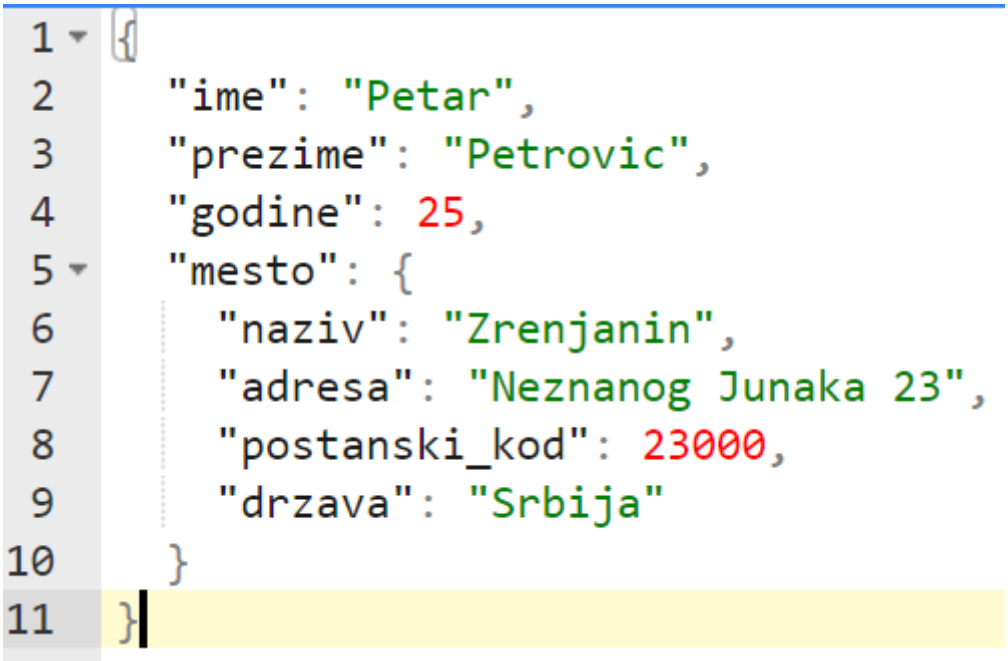


Figure 25. A MongoDB document

A document nested inside another document is called an embedded document. For example, a “Person” document may contain a name and surname together with an embedded place document that has its own name, address, and other fields. Documents and arrays can contain further embedded documents where the data model requires them.

A document does not have to follow a fixed structure shared by every document in its collection. It may contain a combination of field names and values, such as “place”: “Zren-

[1]Later MongoDB versions added support for multi-document ACID transactions.

janin". Fields can be added or removed as the model evolves, and an optional field may simply be absent from a document.

Like relational databases, non-relational databases must identify stored records uniquely. A relational table typically defines a primary-key column. MongoDB automates part of this process by adding an "_id" field to each document unless the application has supplied one. A developer may therefore provide identifiers or let MongoDB generate them.

Collections are analogous to tables in a relational database management system, while documents are analogous to rows. A database contains one or more collections, and each collection contains documents. Although MongoDB permits documents with different shapes in one collection, related data is normally kept together for coherent indexing and queries. A movies collection, for example, may contain a film title, director, and year rather than unrelated data such as a pet's name.

Collections can be created when first needed. MongoDB creates a collection implicitly when the first document is inserted if the collection has not already been declared. For example, a movies collection can appear when the first movie is added. A MongoDB database is therefore a set of collections.

4.3.2 Data Manipulation

Like Node.js, MongoDB can be accessed through a command-line interface. The MongoDB shell provides methods and mechanisms for working with data. Common operations such as reading, inserting, updating, and deleting data are built in.

```
> use people
switched to db people
> db.people.insert({name: "Petar Petrovic", age: 25})
WriteResult({ "nInserted" : 1 })
> db.people.insert({name: "Nikola Nikolic", age: 34})
WriteResult({ "nInserted" : 1 })
> db.people.insert({name: "Marko Markovic", age: 26})
WriteResult({ "nInserted" : 1 })
>
```

Figure 26. Inserting data into the selected database

The "use" command selects the database to work with, after which the "db" object represents that database and its collections. The shell syntax used in the thesis resembles JavaScript. Because the collection need not be created explicitly, the example immediately inserts a document with the built-in "insert" method. The method accepts a JavaScript object representing the document and stores it in the collection.

```
> db.people.find()
{
        "_id" : ObjectId("5d7d3d83f467b5b2089bf7f0"),
        "name" : "Petar Petrovic",
        "age" : 25
}
{
        "_id" : ObjectId("5d7d3d94f467b5b2089bf7f1"),
        "name" : "Nikola Nikolic",
        "age" : 34
}
{
        "_id" : ObjectId("5d7d3da5f467b5b2089bf7f2"),
        "name" : "Marko Markovic",
        "age" : 26
}
```

Figure 27. Selecting an entire collection

The “find” method retrieves documents from the selected collection. Without arguments, it resembles “SELECT * FROM” in a relational database. A query document can specify conditions, and “find” returns matching documents. For example, `db.people.find({age: 25})` returns people whose age is exactly 25.

```
> db.people.find({age: {$lt: 30}})
{
        "_id" : ObjectId("5d7d3d83f467b5b2089bf7f0"),
        "name" : "Petar Petrovic",
        "age" : 25
}
{
        "_id" : ObjectId("5d7d3da5f467b5b2089bf7f2"),
        "name" : "Marko Markovic",
        "age" : 26
}
>
```

Figure 28. Selecting people younger than 30

MongoDB provides many query operators. One is `$lt` (less than), which matches values below the specified value.

4.3.3 Database Scaling

One straightforward way to scale a relational database is to use a larger and faster server. This vertical-scaling strategy has practical cost and hardware limits, after which a system

may need to distribute work across multiple servers.

Although the idea sounds simple, distributing a database across physical machines introduces substantial complexity. Common relational deployments have historically found horizontal write scaling more difficult than running one primary server, while specialized database systems can support it at additional operational cost.

When a query reaches a distributed database, the system must locate the relevant data, process the query, and return the result even if the data is spread across machines.

If a database stores a smaller data set but receives frequent changes, replicas can distribute some work and improve availability. This introduces another problem: a change on one server must become available on the others, and concurrent changes must be coordinated.

MongoDB addresses these problems through mechanisms designed for document data, including replication and sharding. Because documents group related fields and do not depend on relational joins in the same way as normalized tables, a shard can process queries for the data assigned to it and return matching results. Distribution still requires an explicit shard key and coordination by the database system.

Separate MongoDB servers can divide the data for which they are responsible, a process called sharding. Data is distributed across servers, and each shard stores a particular subset. For example, the split might be based on ranges of user identifiers or another suitable shard key. This allows a MongoDB deployment to scale horizontally from a small cluster to a larger number of servers.

5. DESCRIPTION OF THE DEVELOPMENT PROCESS

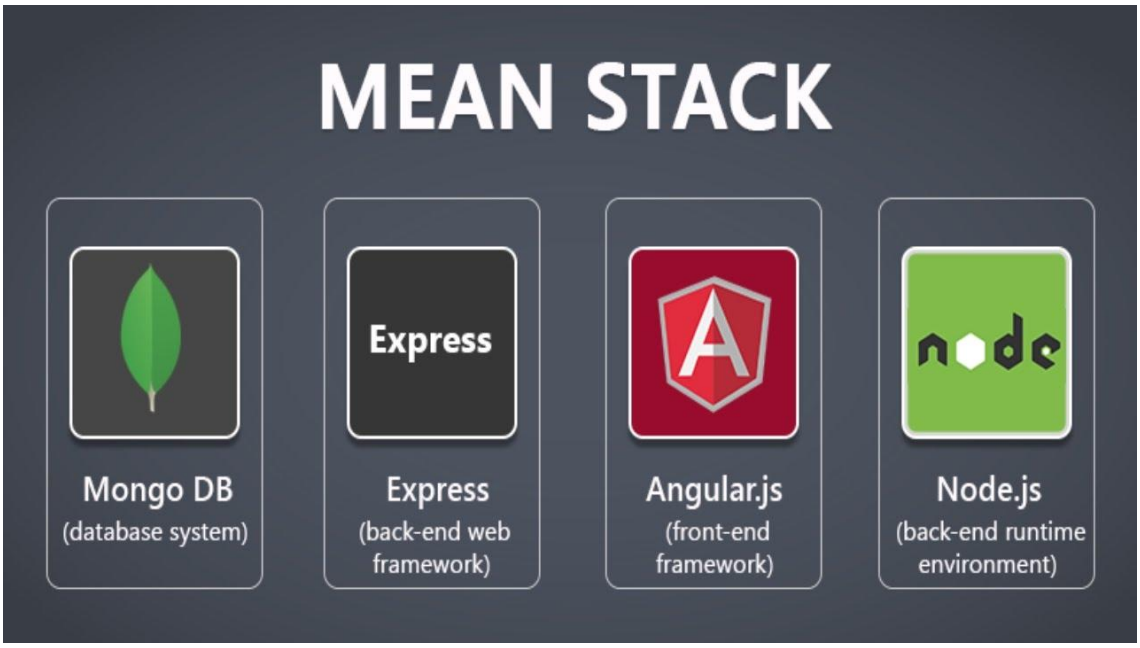


Figure 29. Technology stack for creating JavaScript applications [27]

Before application development begins, Node.js and MongoDB must be installed. They can be downloaded free of charge from https://nodejs.org/en/download/ and https://www.mongodb.com/download-center/community.

The front-end and back-end applications—the user interface and business-logic layer—are developed separately. Several people or teams can work on them in parallel and largely independently. Each team can develop its own application without detailed knowledge of the other's implementation. During the early stages, the applications may have no direct contact until they are ready to exchange data through an agreed interface.

One person can also work on both applications. This kind of work is called full-stack development.

Node.js is a shared tool in the development of both applications. Both use the npm package manager to install libraries and frameworks.

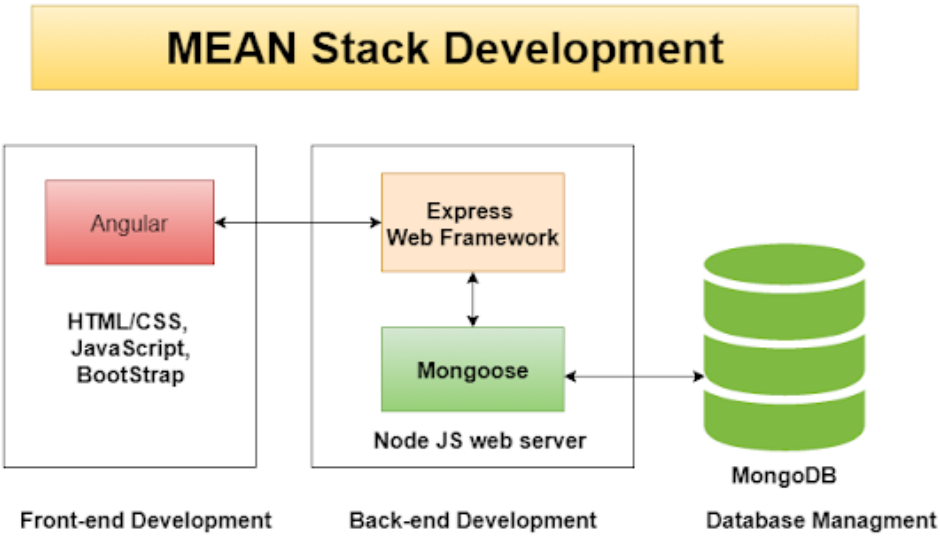


Figure 30. Application architecture with the MEAN technology stack [28]

5.1 Setting Up the Structure

To create the Angular application, the developer first installs Angular CLI, which manages the project. The command “ng new filmovi” creates a new Angular project.

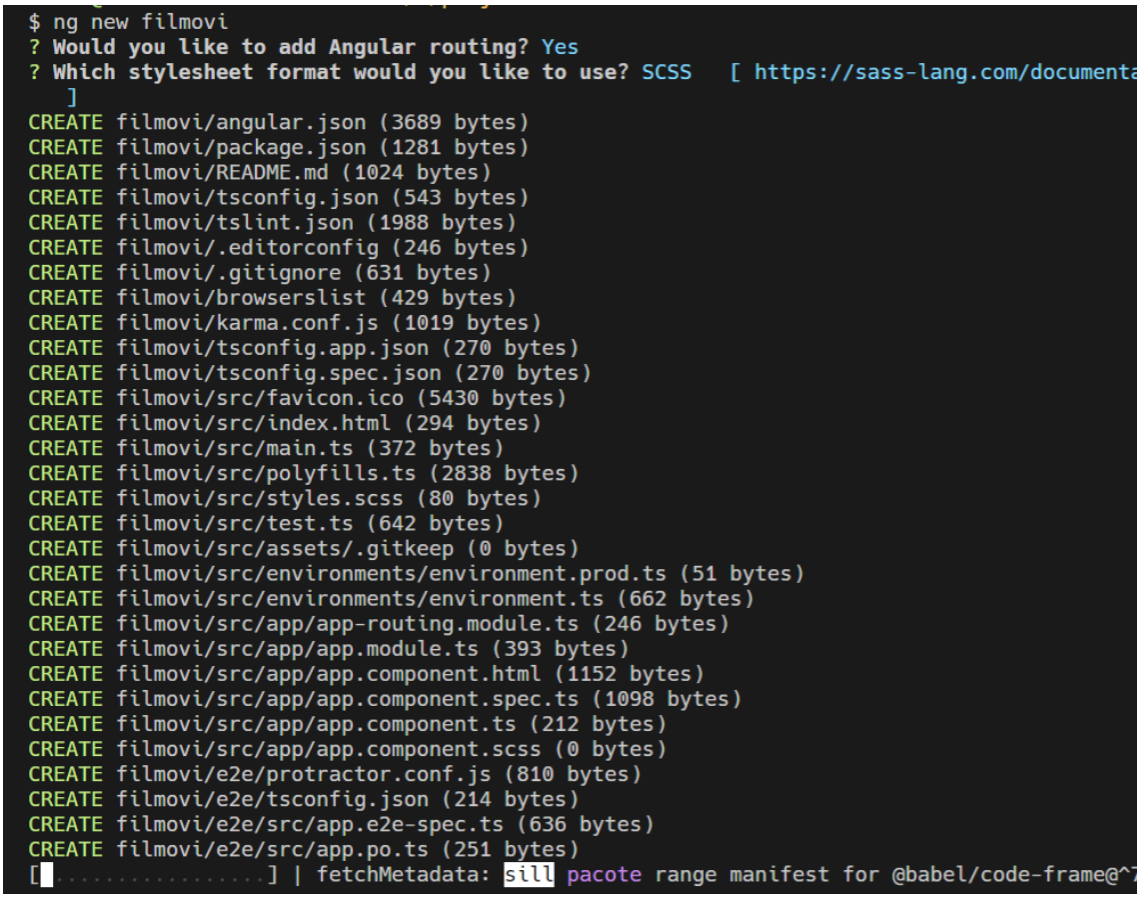


Figure 31. Creating a new Angular project

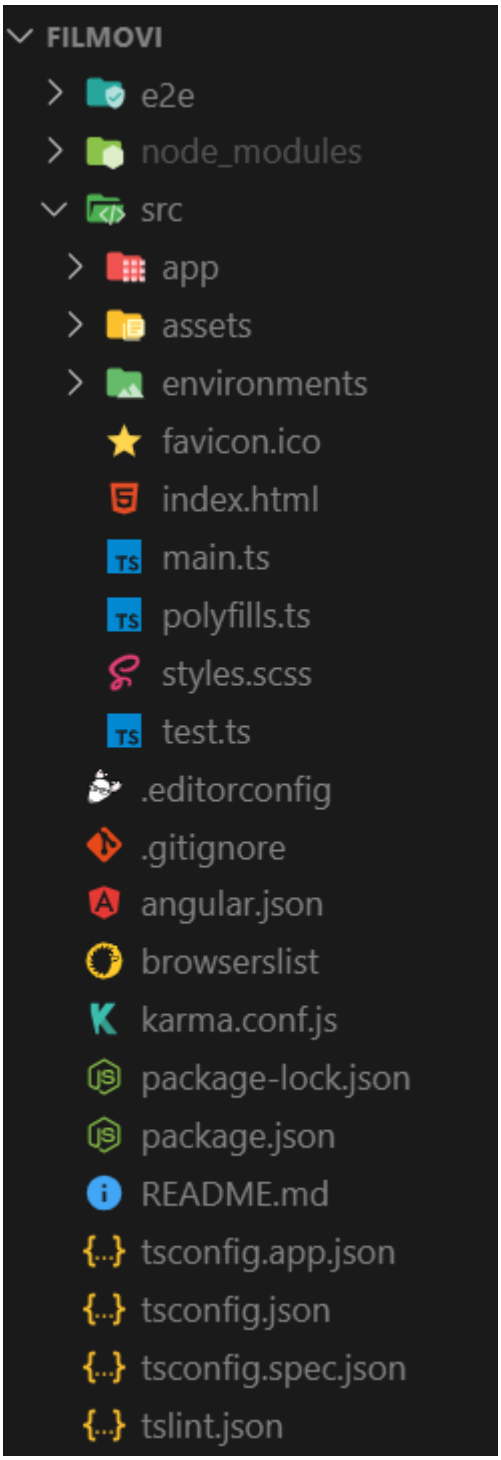


Figure 32. File structure of the Angular application

The command “ng s” starts the application, which can then be opened in a browser at http://localhost:4200.

```
$ ng s
 10% building 3/3 modules 0 activei [wds]: Project is running at http://localhost:4200/webpack-dev-server/
i [wds]: webpack output is served from /
i [wds]: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 11.5 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 251 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.09 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.6 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.1 MB [initial] [rendered]
Date: 2019-09-15T09:59:43.320Z - Hash: 06cc2c85db7a563be03b - Time: 5722ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/
 **
i [wdm]: Compiled successfully.
```

Figure 33. Starting the Angular application

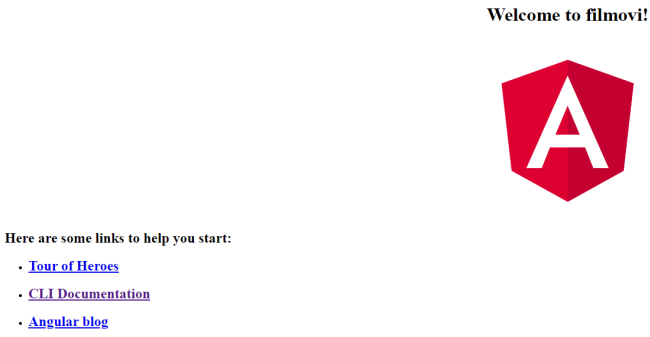


Figure 34. Successfully created and running Angular application

The application needs two views: one that lists films and another that shows the details of one film. Because Angular implements a single-page application, these views are not created as separately loaded HTML pages.

Angular provides “RouterModule” for working with routes and views. This module contains the tools needed for client-side routing.

The module must first be imported, and the `<router-outlet></router-outlet>` directive is then placed in “app.component”. The directive comes from the router module and marks the place where routed components are dynamically inserted and removed, simulating page changes in a traditional application. [29]

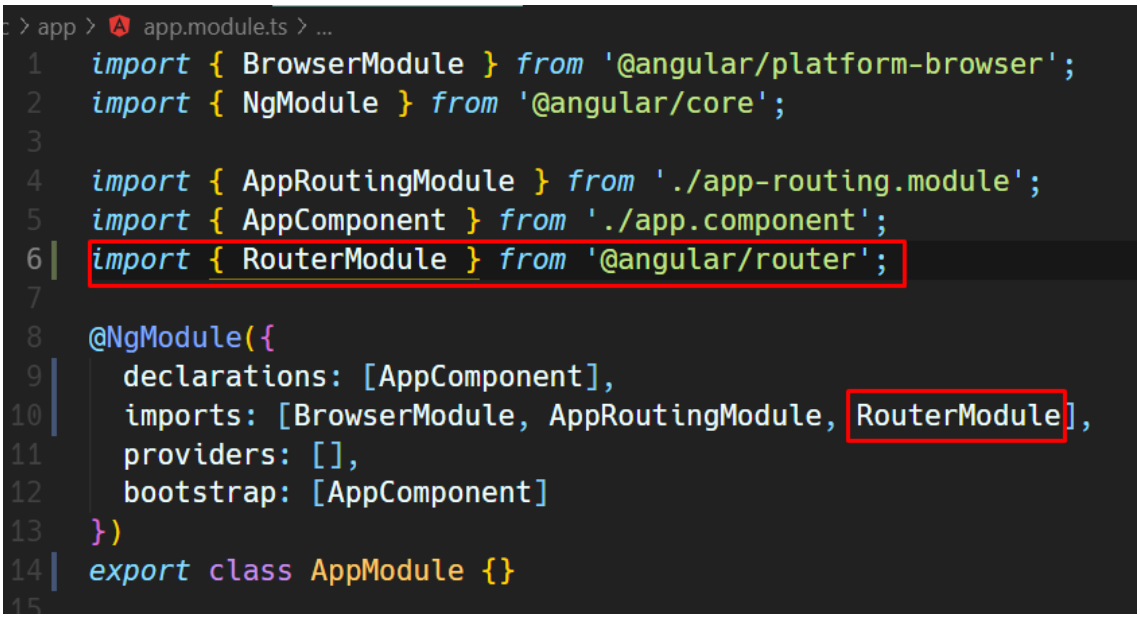


Figure 35. Importing the router module

Figure 36. Marking the outlet for routed components

The film page and film-list page are generated as components, which the router displays as needed. The command for creating the film-list component is “ng g c lista-filmova”. The film component is created in the same way.

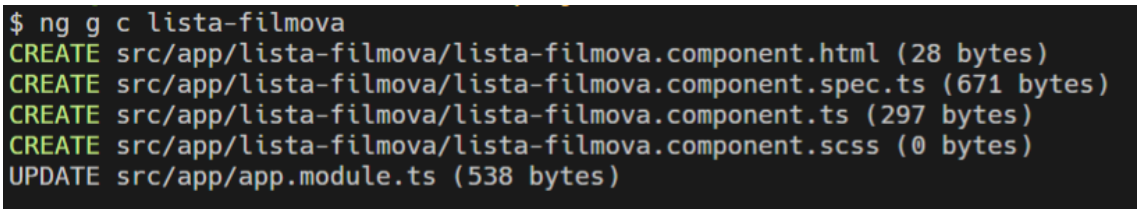


Figure 37. Generating a component

Routes are configured in “app-routing.module”, which defines the paths used by the module. The routes are named “/lista-filmova” and “/film”. The film route also has a parameter that identifies the film by name.

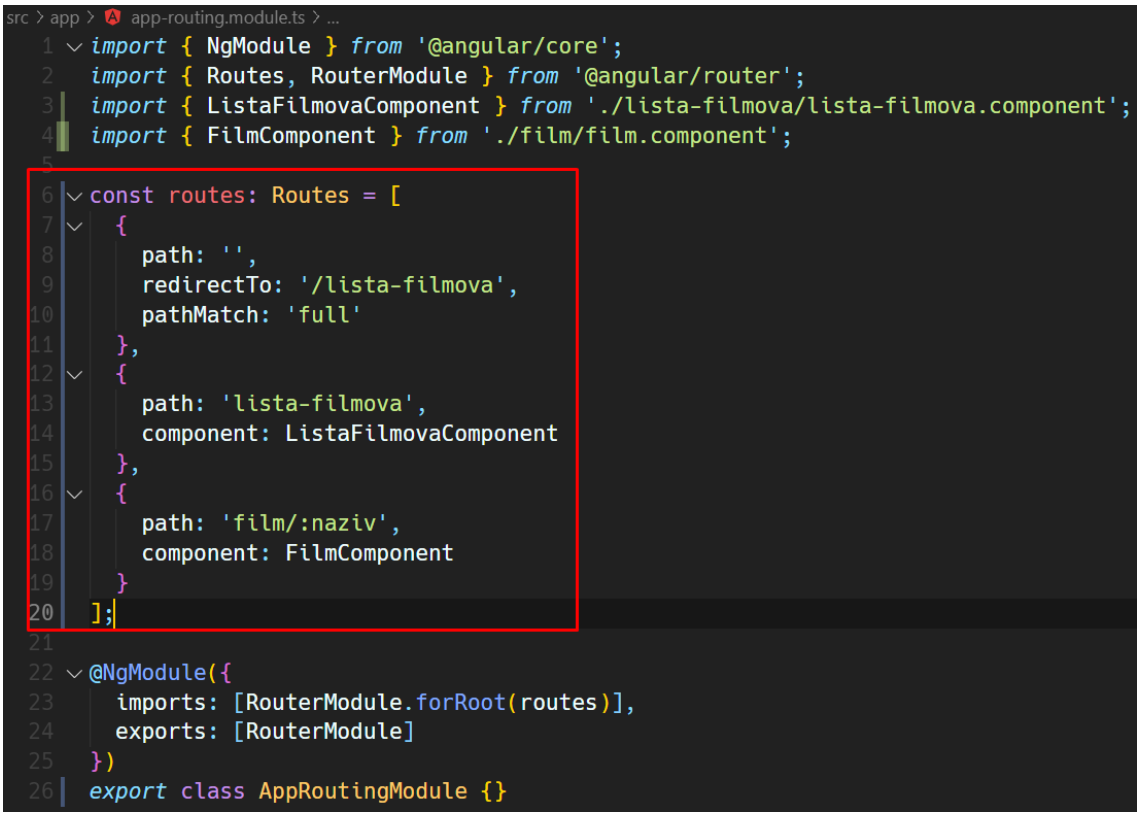


Figure 38. Defining routes

The film-list component initially receives a list of films with static data. Later, the server supplies the contents of this list.

Each film in the list has a link to its own page. Angular's "routerLink" directive is used instead of an ordinary "href" attribute so that navigation stays within the running single-page application. Clicking a link opens the view for the selected film.

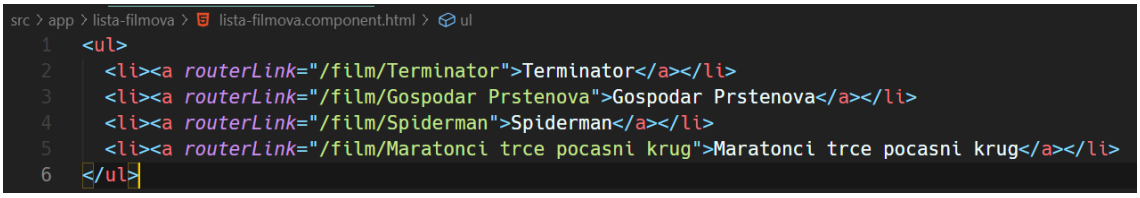


Figure 39a. Film list with the routerLink directive

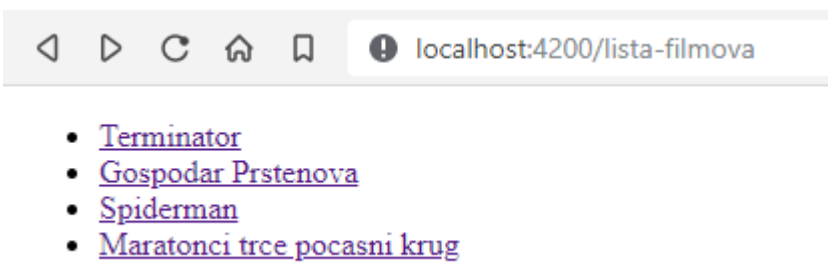


Figure 39b. Film list in the browser

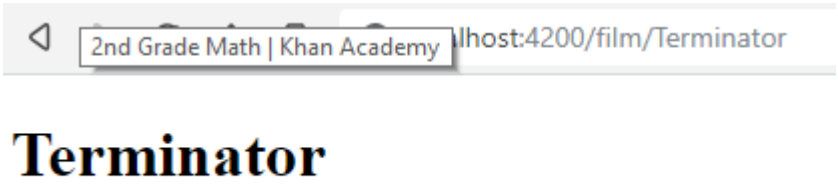


Figure 39c. Opening a film page

5.2 API

The API is a separate application developed in its own directory. The command “npm init -y” initializes a Node.js project, and “npm install express” downloads and installs Express.

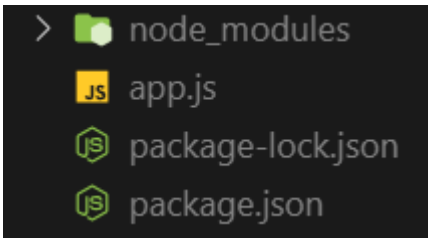


Figure 40. Initial API structure

The “app.js” file contains the configuration for starting the server and connecting to the database. The application uses the Mongoose ODM (Object Document Mapper) to simplify the connection, define data models, and provide higher-level features over the official MongoDB driver. The database does not need to be created in advance because MongoDB creates it when data is first written.

app.js > ...

```
const express = require('express');
const app = express();
const mongoose = require('mongoose');

mongoose
  .connect('mongodb://localhost:27017/filmovi', {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => {
    console.log('Uspesno uspostavljena konekcija sa bazom podataka.');
  })
  .catch((err) => {
    console.log('Greska u konekciji sa bazom: ', err);
  });

const port = 3000;

app.listen(port, () => {
  console.log(`Server slusa na portu ${port}.`);
});
```

Figure 41. API server and database

Although a document database does not require every document to follow one rigid structure, defining the data expected by the application makes communication between the server and database clearer. Mongoose maps a schema definition to a model with properties and methods that the business logic can use.

The example does not need separate top-level collections for both entities. An actor is modeled as an embedded document inside a film document, keeping the data needed by the example’s film queries together.

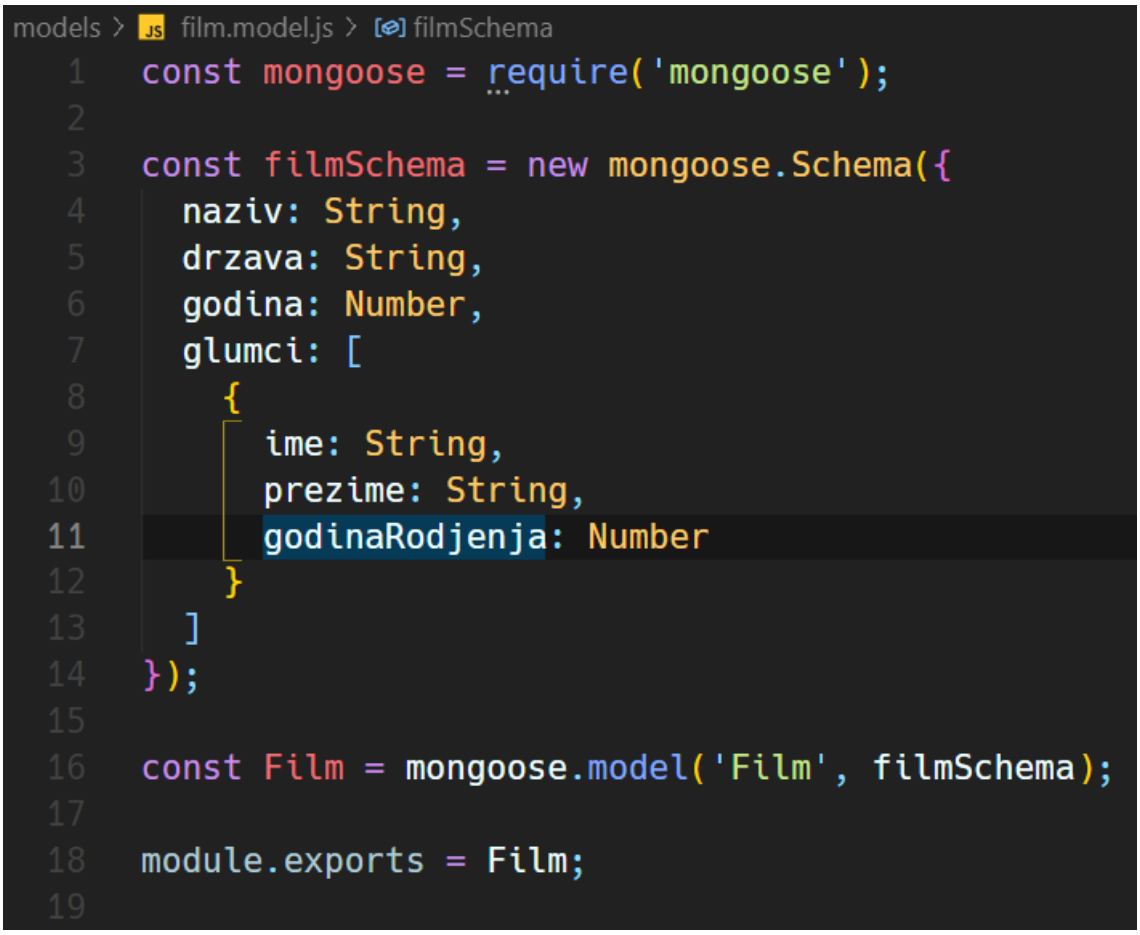


Figure 42. Data schema and model with two entities

The server finds data in the database and sends it to the client. At the client's request, it must also insert, delete, or update data. The application therefore defines routes to which clients send requests and from which they receive responses.

```
router.get('/', async (req, res) => {
  const godina = req.query.godina;

  try {
    let filmovi = [];

    if (godina) {
      filmovi = await Film.find({ godina });
    } else {
      filmovi = await Film.find();
    }

    res.status(200).json(filmovi);
  } catch (error) {
    res.status(500).json({ greska: 'Greska pri pronalazenju filmova' });
  }
});
```

Figure 43. Route for reading films

```
33  router.post('/', async (req, res) => {
34    try {
35      const noviFilm = new Film(req.body);
36      await noviFilm.save();
37
38      res.status(201).json(noviFilm);
39    } catch (error) {
40      res.status(500).json({ greska: error });
41    }
42  });
```

Figure 44. Route for creating a new film and storing it in the database

5.3 Service and Data Models

To use the static typing provided by TypeScript, the client defines models for the film data received from the server and displayed in the user interface.

These models are written as interfaces, similar to interface-based models in other object-oriented languages.

```
1   import { Glumac } from './glumac.model';
2
    1 implementation
3   export interface Film {
4     _id: string;
5     naziv: string;
6     godina: number;
7     drzava: string;
8     glumci: Glumac[];
9   }
10
```

Figure 45. Film model

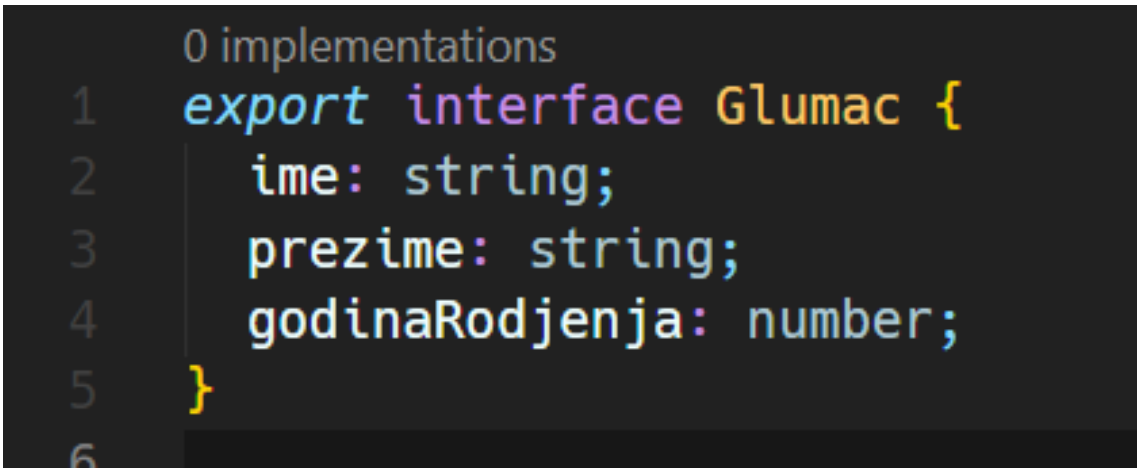
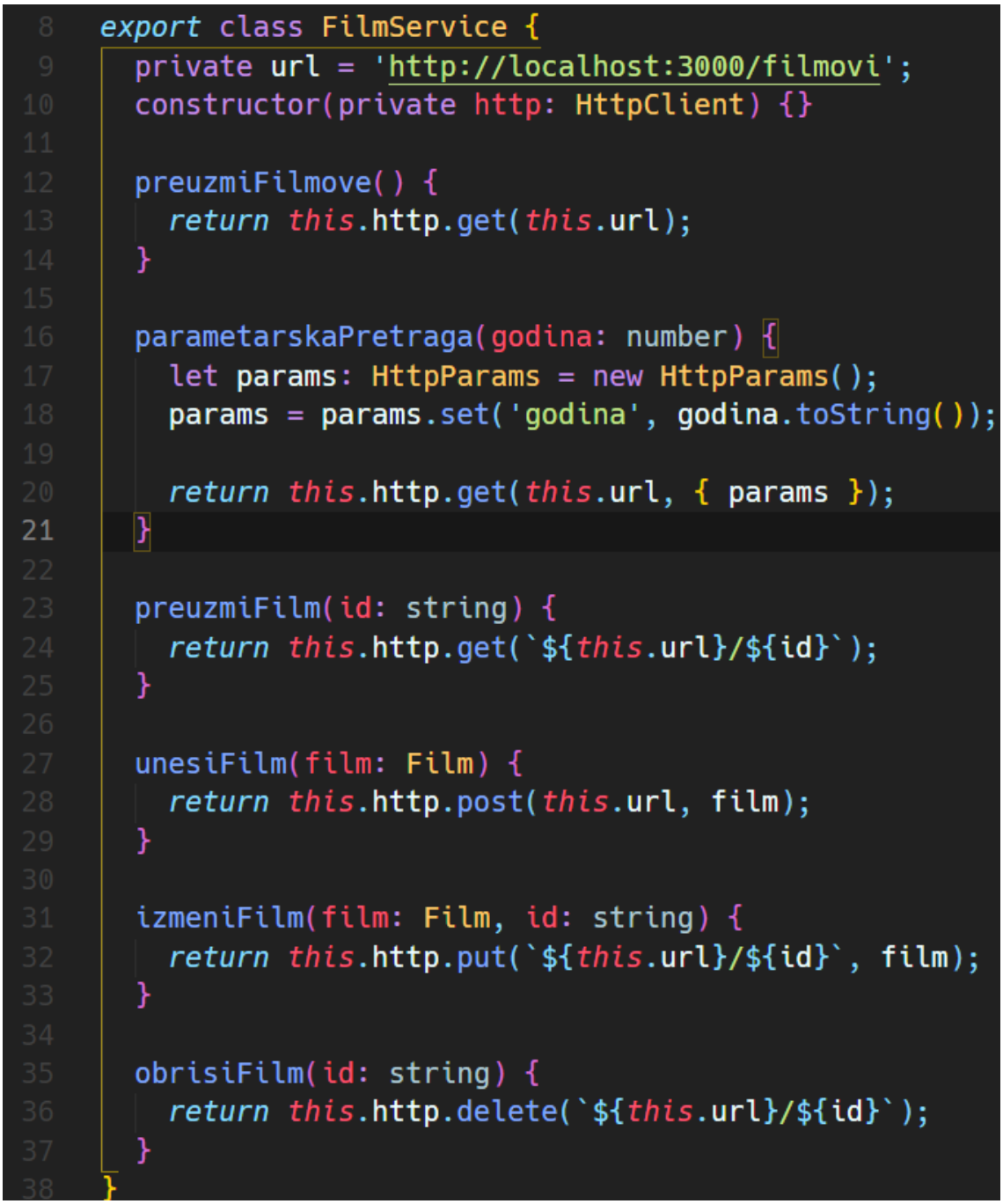


Figure 46. Actor model

All server communication is placed in a service. The service defines methods that send HTTP requests with the required parameters and pass the returned data to components, which render it in their views.



```
export class FilmService {
  private url = 'http://localhost:3000/filmovi';
  constructor(private http: HttpClient) {}

  preuzmiFilmove() {
    return this.http.get(this.url);
  }

  parametarskaPretraga(godina: number) {
    let params: HttpParams = new HttpParams();
    params = params.set('godina', godina.toString());

    return this.http.get(this.url, { params });
  }

  preuzmiFilm(id: string) {
    return this.http.get(`${this.url}/${id}`);
  }

  unesiFilm(film: Film) {
    return this.http.post(this.url, film);
  }

  izmeniFilm(film: Film, id: string) {
    return this.http.put(`${this.url}/${id}`, film);
  }

  obrisiFilm(id: string) {
    return this.http.delete(`${this.url}/${id}`);
  }
}
```

Figure 47. Service for communicating with the server

5.4 Components and Presentation Logic

When the component initializes, it calls service methods to obtain films for display. When a user enters a year in the field, the component also sends a request to filter films by year.

```
export class ListaFilmovaComponent implements OnInit {
  filmovi: Film[] = [];
  pretraga: FormGroup;

  constructor(private filmService: FilmService, private fb: FormBuilder)
    this.pretraga = this.fb.group({
      godina: ['']
    });
  }

  parametarskaPretraga() {
    this.filmService
      .parametarskaPretraga(this.pretraga.get('godina').value)
      .subscribe((filmovi: Film[]) => {
        this.filmovi = filmovi;
      });
  }

  ngOnInit() {
    this.filmService.preuzmiFilmove().subscribe((filmovi: Film[]) => {
      console.log(filmovi);
      this.filmovi = filmovi;
    });
  }
}
```

Figure 48. Film-list component, initial data, and filtering

```
<table class="table">
  <thead>
    <tr>
      <th scope="col">#</th>
      <th scope="col">Naziv</th>
      <th scope="col">Godina</th>
      <th scope="col">Drzava</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let film of filmovi; let i = index">
      <th scope="row">{{ i + 1 }}</th>
      <td
        ><a [routerLink]="['/film', film._id]">{{ film.naziv }}</a></td
      >
      <td>{{ film.godina }}</td>
      <td>{{ film.drzava }}</td>
    </tr>
  </tbody>
</table>
```

Figure 49. Table that lists films dynamically

A user adds a film through a form. Angular represents the entered values as a JavaScript

object that the service serializes as JSON and sends to the server for storage.

Editing a film is almost identical. The edit form first requests the selected film and fills its fields with the existing data. The user can then change the values and send them to the server for processing.

```
export class UnosComponent {
  unosForma: FormGroup;

  constructor(private fb: FormBuilder, private filmService: FilmService) {
    this.unosForma = this.fb.group({
      naziv: [''],
      godina: [''],
      drzava: [''],
      glumci: this.fb.array([])
    });
  }

  unesiFilm() {
    this.filmService.unesiFilm(this.unosForma.value).subscribe(
      () => {
        alert('Film uspesno unet');
      },
      () => {
        alert('Greska pri unosenju filma');
      }
    );
  }
```

Figure 50. Film-entry component and form creation

Each film has its own page, which shows more detail than the table view. The page also contains a delete button that sends the identifier of the film to be removed to the server.

```
29    obrisiFilm() {
30      this.filmService.obrisiFilm(this.id).subscribe(
31        () => {
32          alert('Film uspesno obrisan.');
33          this.router.navigate(['/lista-filmova']);
34        },
35        () => {
36          alert('Greska pri brisanju');
37        }
38      );
39    }
```

Figure 51. Function for deleting a film

6. IMPLEMENTED EXAMPLE

6.1 Models

This chapter presents a use-case diagram, conceptual model, component diagram, deployment diagram, and sequence diagram for one use case for the example described in Chapter 5.

6.1.1 Use-case Diagram

The following figure presents the use-case diagram.

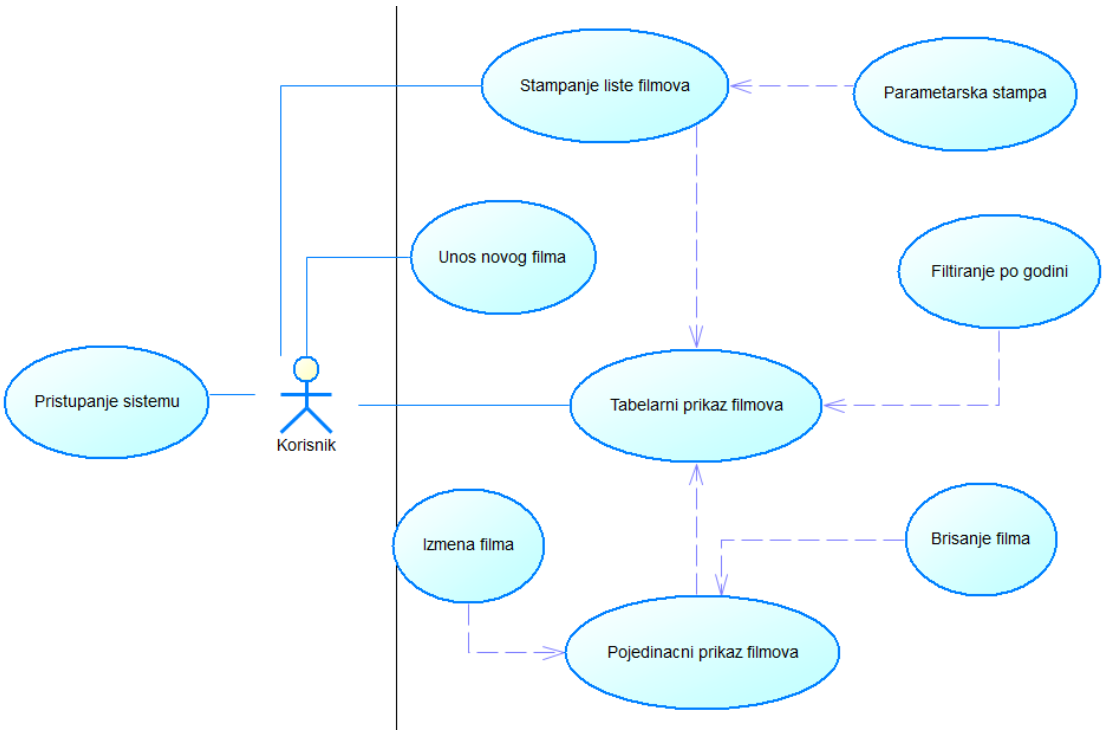


Figure 52. Use-case diagram

6.1.2 Component Diagram

The following figure presents the component diagram.

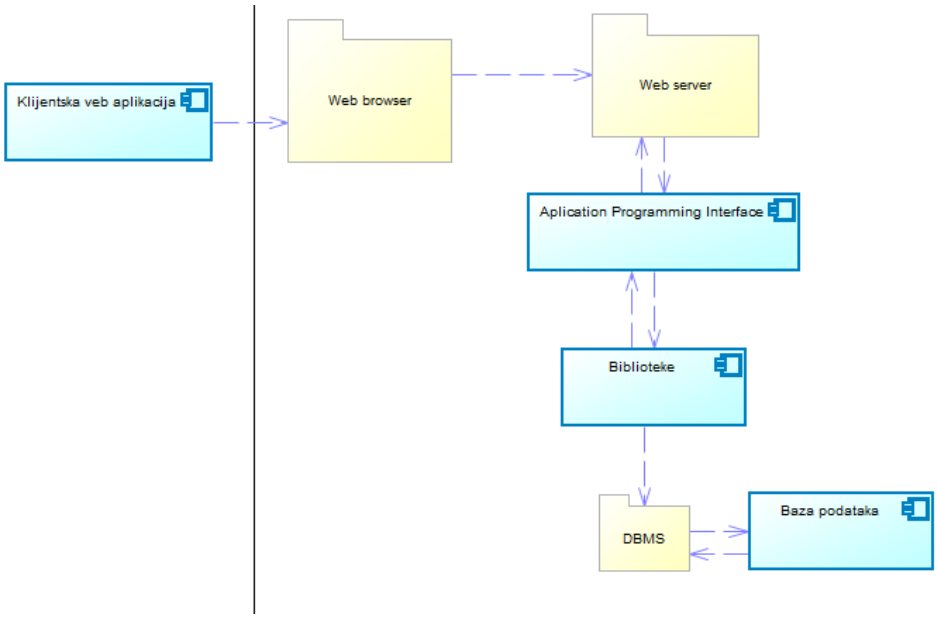


Figure 53. Component diagram

6.1.3 Deployment Diagram

The following figure presents the deployment diagram.

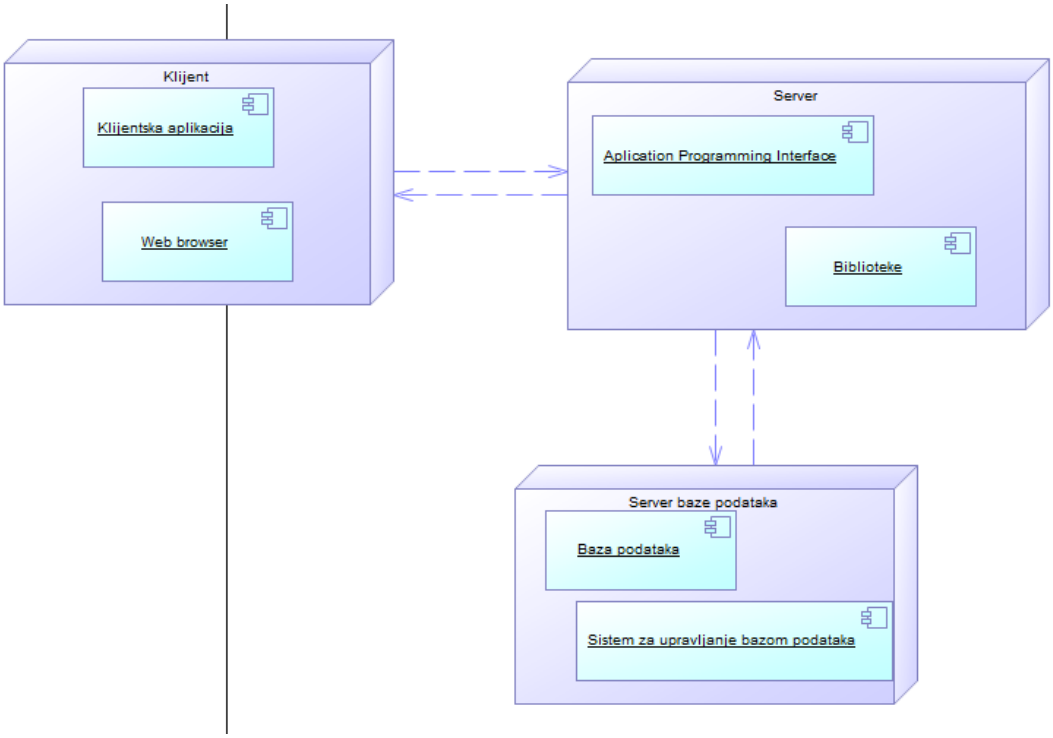


Figure 54. Deployment diagram

6.1.4 Sequence Diagram for One Use Case

The following figure presents the sequence diagram for one use case—adding a new film to the collection.

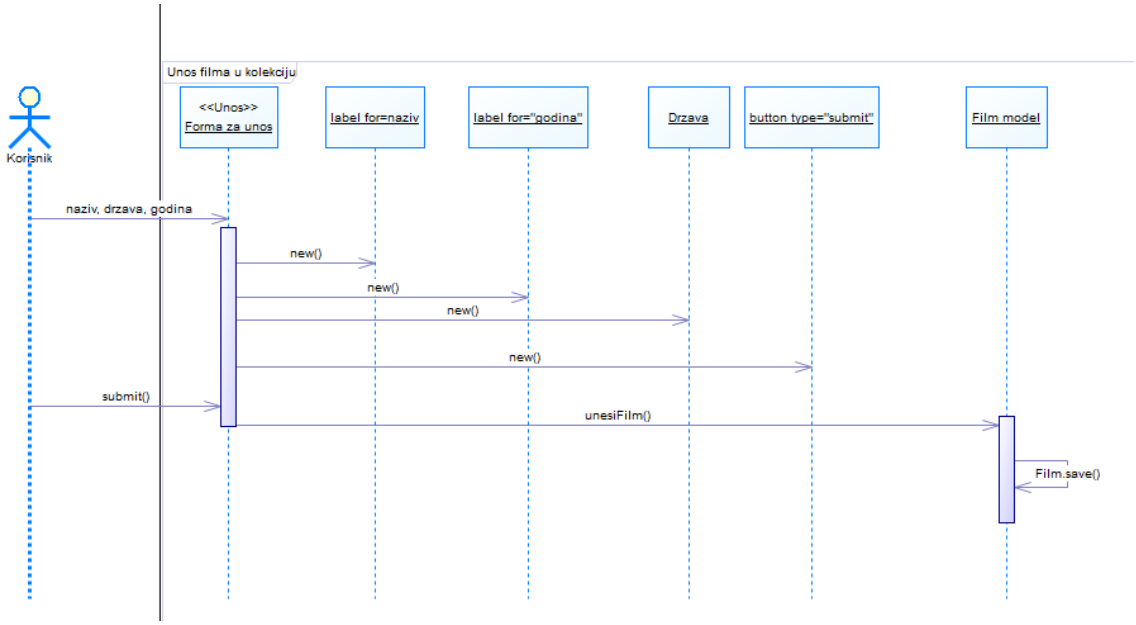


Figure 55. Sequence diagram

6.1.5 Conceptual Diagram

The following figure presents the conceptual diagram.

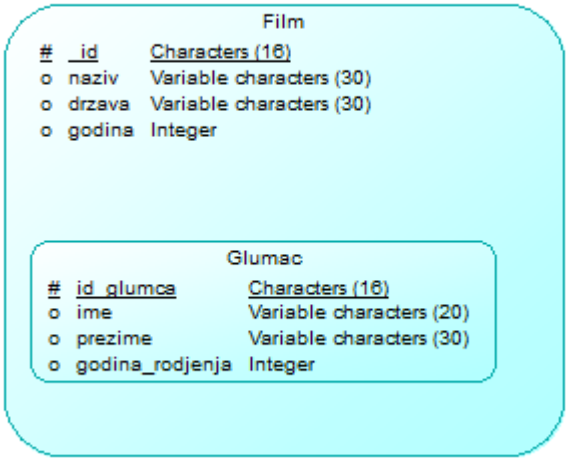


Figure 56. Conceptual diagram

6.2 User Guide

Start the Angular application with the command “ng s” and the Express application with “node app”. Then open the application in a web browser at http://localhost:4200.

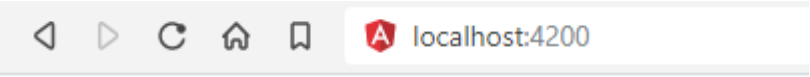


Figure 57. Web application URL

6.2.1 Table View and Filtering

When the application opens, it displays the existing films in a table. Enter a film’s release year in the “search by year” field to list all films released in that year.

Figure 58. Table-view page

Figure 59. Filtering films by release year

6.2.2 Printing and Parameterized Printing

The table-view page contains a “Print” button below the year field. Clicking it opens a print-friendly page listing all films. If the year field was filled in first, the print view

contains only the filtered data. The navigation menu and filter field are omitted from the print view.

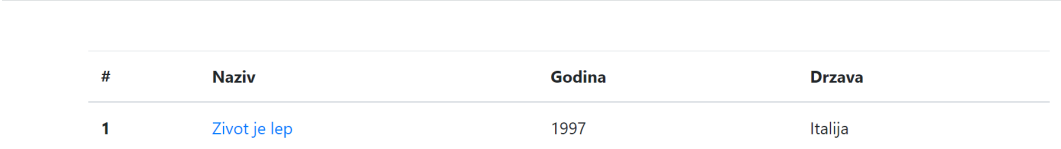


Figure 60. Parameterized print view

6.2.3 Adding a New Film

Clicking the “Add” button in the navigation menu opens a page with a form for adding a new film. Clicking “Add actor” displays additional fields for the actor’s data. The form allows any number of actors, and a particular actor can be removed by clicking the “x” button to the right. After “Submit” is clicked, the application displays a message indicating whether the film was added successfully.

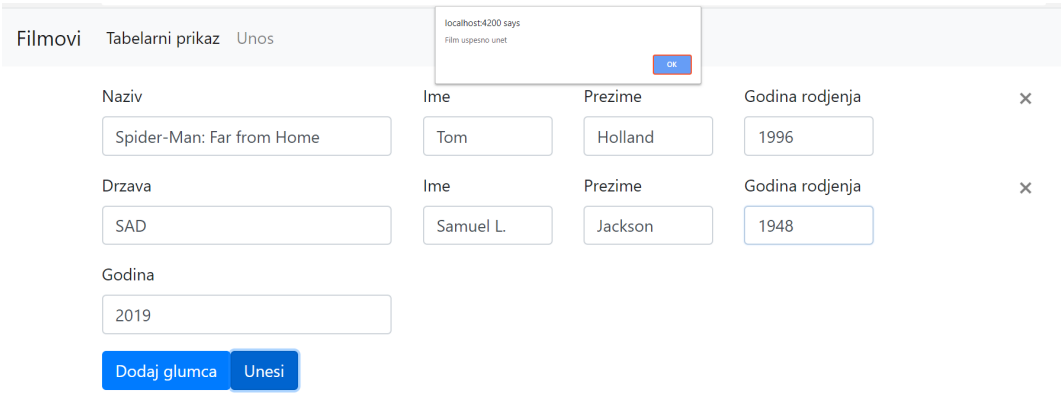


Figure 61. Adding a new film with actors

6.2.4 Editing and Deleting a Film

Clicking a film in the table opens a page with more details about that film. The page also contains buttons for editing or deleting it.

Figure 62. Individual film view with edit and delete buttons

Clicking “Edit” opens the same form used for adding a film, prefilled with the existing data. The user can change the values and confirm them with the “Edit” button.

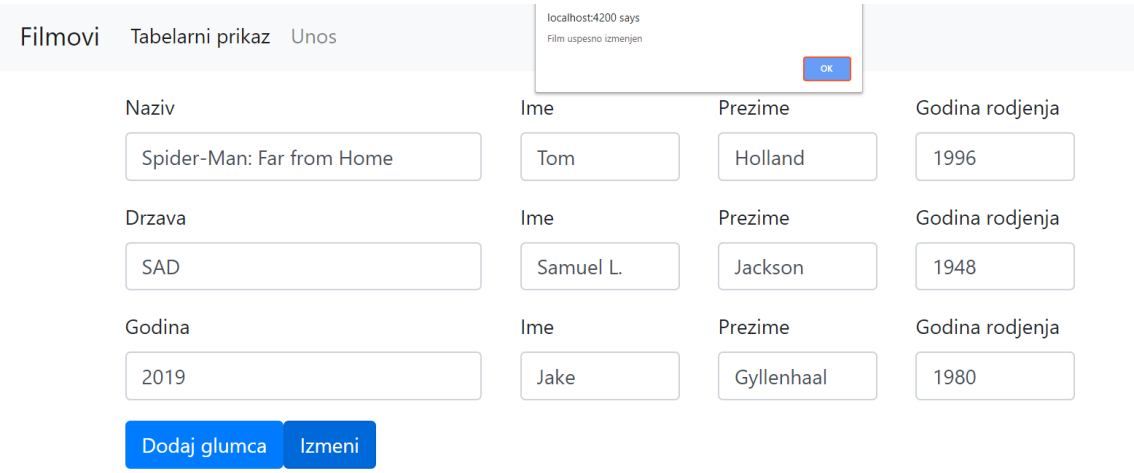


Figure 63. Adding an actor to an existing film

Clicking “Delete” removes the film from the database, after which it no longer appears in the table.

6.3 Application Layers and Sublayers

Main layer	**REST architecture**	**Sublayer**	**Technology**	
Presentation layer	Client	User interface	Angular	Components

Service layer	Client	Presentation logic	Angular	Angular services
Business-logic layer	Server API		Express REST server	
Data-access layer	Database server	Model layer	Mongoose ODM	
Database		Non-relational	MongoDB database	

Table 1. Application layers and sublayers

6.4 Source-code Excerpts with Explanations

6.4.1 Presentation-logic Layer and User Interface

The Angular application has one “index.html” document whose contents change dynamically to produce the effect of moving between pages.

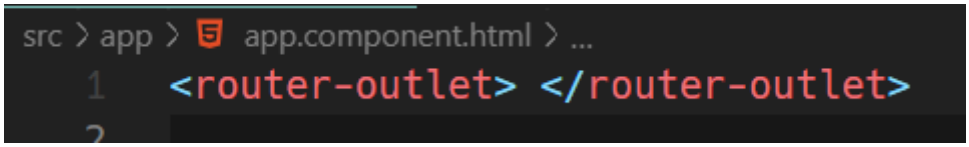


Figure 64. Router outlet

The router outlet is the directive responsible for inserting and removing components during navigation. The navigable components and their routes are defined in the routing module.

src > app > app-routing.module.ts > routes

```typescript
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ListaFilmovaComponent } from './lista-filmova/lista-filmova.component';
import { FilmComponent } from './film/film.component';
import { UnosComponent } from './unos/unos.component';
import { IzmenaComponent } from './izmena/izmena.component';
import { StampaComponent } from './stampa/stampa.component';
import { MainComponent } from './main/main.component';

const routes: Routes = [
  {
    path: '',
    component: MainComponent,
    children: [
      {
        path: '',
        redirectTo: '/lista-filmova',
        pathMatch: 'full'
      },
      {
        path: 'lista-filmova',
        component: ListaFilmovaComponent
      },
      {
        path: 'film/:id',
        component: FilmComponent
      },
      {
        path: 'unos',
        component: UnosComponent
      },
      {
        path: 'izmena/:id',
        component: IzmenaComponent
```

Figure 65. Components and their routes

6.4.2 Service Layer and Presentation Logic

Services keep complex logic out of components so that components can focus on presentation. The service layer calls external servers and acts as a bridge between the user interface and business logic. Service logic is written once and reused by every component that needs it.

```
export class FilmService {
  private url = 'http://localhost:3000/filmovi';
  constructor(private http: HttpClient) {}

  preuzmiFilmove() {
    return this.http.get(this.url);
  }

  parametarskaPretraga(godina: number) {
    let params: HttpParams = new HttpParams();
    params = params.set('godina', godina.toString());

    return this.http.get(this.url, { params });
  }

  preuzmiFilm(id: string) {
    return this.http.get(`${this.url}/${id}`);
  }

  unesiFilm(film: Film) {
    return this.http.post(this.url, film);
  }

  izmeniFilm(film: Film, id: string) {
    return this.http.put(`${this.url}/${id}`, film);
  }

  obrisiFilm(id: string) {
    return this.http.delete(`${this.url}/${id}`);
  }
```

Figure 66. A service as the bridge between the interface and server

6.4.3 Business-logic Layer

This layer contains the API, which listens for client requests on defined routes and handles them as they arrive. Requests may involve database queries and changes, disk input and output, or calculations. Each route is responsible for one specific operation.

```
router.put('/:id', async (req, res) => {
  const id = req.params.id;
  const izmene = req.body;

  try {
    let film = await Film.findByIdAndUpdate(id, izmene, { new: true });

    res.status(200).json(film);
  } catch (error) {
    res.status(500).json({ greska: error });
  }
});

router.delete('/:id', async (req, res) => {
  const id = req.params.id;

  try {
    await Film.findByIdAndRemove(id);

    res.status(200).json({ message: 'Film obrisan' });
  } catch (error) {
    res.status(500).json({ greska: error });
  }
});
```

Figure 67. Routes for updating and deleting database records

6.4.4 Model and Database Layer

The model layer sits between the database and the business logic. It defines data models with properties and methods that the business-logic layer uses and maps those models to documents in the database.

```
const mongoose = require('mongoose');

const filmSchema = new mongoose.Schema({
  naziv: String,
  drzava: String,
  godina: Number,
  glumci: [
    {
      ime: String,
      prezime: String,
      godinaRodjenja: Number
    }
  ]
});

const Film = mongoose.model('Film', filmSchema);

module.exports = Film;
```

Figure 68. Data schema mapped to a model

The database layer is an instance of a MongoDB server. It performs database reads and writes and supports administration and scaling. An administrator can issue instructions through the command line or use a graphical interface.

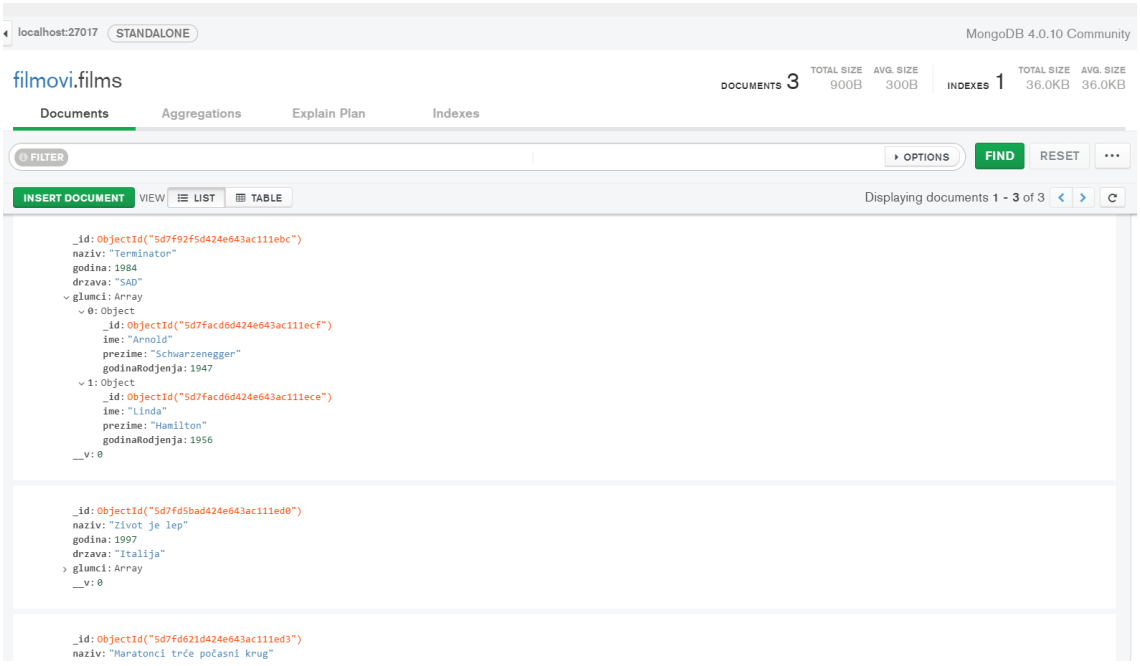


Figure 69. Graphical view of the database

7. CONCLUSION

This thesis described a process for developing fast and scalable applications with JavaScript technologies across the application's layers, illustrated by an example collection of films and actors.

The development process for these technologies was drawn from their official presentations and documentation. The thesis also presented five internationally known organizations that used these technologies in parts of their operations at the time of writing.

The development process was explained for every part of the system. Two separate applications were created to cooperate and communicate closely. Five types of diagram describe the resulting application: a use-case diagram, component diagram, deployment diagram, conceptual data model, and sequence diagram for one use case.

To make the application's structure easier to understand, a table presents every layer and sublayer together with its implementation. The thesis also provides instructions for using the completed application.

The application is simple and minimal, but it can be adapted and extended in many ways.

Possible extensions include:

- improving the visual design;
- adding animations;
- improving performance with reactive-programming techniques;
- storing more film data, such as genre and director, and actor data, such as biographies and achievements;
- creating a mobile application with the Ionic framework; and
- creating a desktop application with the Electron framework.

8. BIBLIOGRAPHY

1. https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data
2. https://auth0.com/blog/a-brief-history-of-javascript/
3. Introduction to Node.js, https://nodejs.dev/introduction-to-nodejs
4. REST API architecture, https://restfulapi.net/rest-architectural-constraints/
5. https://www.seobility.net/en/wiki/REST_API
6. https://restfulapi.net/
7. https://idratherbewriting.com/learnapidoc/docapis_what_is_a_rest_api.html
8. https://www.paypal.com/us/home
9. https://naturaily.com/blog/node-js-applications
10. https://www.linkedin.com/
11. https://brainhub.eu/blog/9-famous-apps-using-node-js/
12. https://www.netflix.com
13. https://www.uber.com/
14. https://www.nasa.gov/
15. https://foundation.nodejs.org/wp-content/uploads/sites/50/2017/09/Node_CaseStudy_Nasa_FNL.pdf
16. Angular architecture, https://angular.io/guide/architecture
17. Angular module architecture, https://angular.io/guide/architecture-modules
18. https://vitalflux.com/wp-content/uploads/2016/02/components.png
19. Angular directives, https://angular.io/guide/architecture-components#directives
20. Built-in Angular pipes, https://angular.io/guide/pipes#built-in-pipes
21. Angular service architecture, https://angular.io/guide/architecture-services
22. Angular providers, https://angular.io/guide/providers
23. Angular CLI, https://angular.io/cli
24. Technologies used by Node.js, https://nodejs.org/en/docs/meta/topics/dependencies/#v8
25. Official Express website, https://expressjs.com/
26. Eelco Plugge, Peter Membrey, Tim Hawkins – The Definitive Guide to MongoDB (2010)
27. https://i.ytimg.com/vi/Lzi2xYQdwWc/maxresdefault.jpg
28. https://4.bp.blogspot.com/-zQRHoujtNTQ/WiK1d8AqKOI/AAAAAAAAF_U/FPhQIEebkmM9gp6TYxrADhSFugqmxPNVwCLcBGAs/s1600/mean%2Bstack%2Bdevelopment.png
29. Angular router outlet, https://angular.io/guide/router#router-outlet