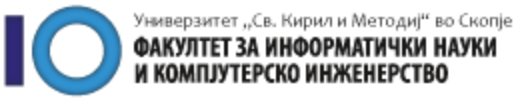
Technical Documentation

Digital Signature of PDF documents

Made by:
Filip Ilieski - 226022
Ilija Buncheski - 221094

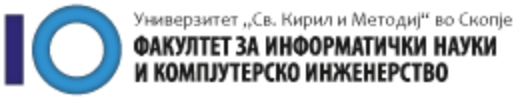
Mentor:
Prof. Dr. Ivan Chorbev

Skopje
2025



1. Introduction **4**
2. Project Objectives and Scopes **4**
2.1 Objectives 4
2.2 Scope 4
3. Project Overview **5**
4. System Architecture **5**
4.1 Architectural Style 5
4.2 Layered Structure 5
4.3 Benefits of the Chosen Architecture 6
5. Domain Layer **6**
6. Application Layer **6**
6.1 Responsibilities 6
6.2 Digital Signature Service 7
7. Infrastructure Layer **7**
7.1 Technical Responsibilities 7
7.2 Isolation of Technical Concerns 7
8. Web Layer **7**
8.1 Responsibilities 7
9. Digital Signature Workflow **8**
10. Digital Signature Concepts **8**
10.1 Integrity 8
10.2 Authenticity 8
10.3 Non-repudiation (Conceptual) 8
11. QR Code Integration **9**
11.1 Purpose 9
11.2 Advantages 9
12. Security Considerations **9**
13. Error Handling and Validation **9**
14. Non-Functional Requirements **10**
14.1 Maintainability 10
14.2 Testability 10
14.3 Extensibility 10
15. Execution Instructions **10**
15.1 Prerequisites 10
15.2 Cloning and Building 11

15.3 Running the application 11
15.4 Testing the API 12
16. Configuration Parameters 13
16.1 Application Settings 13
16.2 Launch Profiles and Environment Configuration 14
16.3 Required Assets 15
16.4 Dependency Injection Configuration 16
17. Deployment Instructions 16
18. User Scenarios 17
Scenario 1: Academic Document Verification 17
Scenario 2: Contract Signing Workflow 17
Scenario 3: Automated Document Processing 17
19. Limitations 18
20. Conclusion 18



1. Introduction

In the modern digital era, the exchange of electronic documents has become a fundamental aspect of academic, governmental, and business processes. As the reliance on digital documents increases, so does the need for mechanisms that ensure authenticity, integrity, and trust. Traditional handwritten signatures are insufficient in digital environments, making digital signing systems an essential component of secure information systems.

This project presents a backend-oriented Digital Signature System for PDF documents, developed using the C# programming language and the .NET framework. The system is designed to apply digital signing mechanisms to PDF files, enabling verification of document origin and detection of unauthorized modifications. The solution demonstrates how modern software architecture principles can be applied to security-oriented systems.

In addition to functional correctness, the project emphasizes clean architectural design, separation of concerns, and long-term maintainability.

2. Project Objectives and Scopes

2.1 Objectives

The main objective of this project is to design and implement a backend system capable of digitally signing PDF documents. The system aims to ensure document integrity after the signing process by embedding signature-related information directly into the document. In addition, the project provides a verification mechanism through the use of embedded metadata and QR codes, allowing signed documents to be validated externally. Another important goal of the project is to demonstrate the practical application of Clean Architecture principles, ensuring a clear separation of concerns and long-term maintainability. Finally, the system is designed to be modular and extensible, enabling future enhancements and adaptations with minimal impact on the existing codebase.

2.2 Scope

The scope of the project is limited to PDF document signing at the service level. The system does not include user identity management, certificate authorities, or advanced cryptographic trust chains. Instead, it focuses on demonstrating a clean and well-structured approach to document signing workflows.

3. Project Overview

The Digital Signature System is implemented as a backend service that processes PDF documents submitted by clients. Upon receiving a document, the system generates a digitally signed version by embedding signature-related information directly into the PDF. This information serves as proof that the document has passed through the system and has not been modified afterward.

A QR code is added to the document to support verification workflows. The QR code can encode metadata or references that allow users to validate the signed document externally. This design enhances transparency and traceability while remaining lightweight.

Beyond its functional role, the project prioritizes architectural clarity, making it easy to understand, test, and extend.

4. System Architecture

4.1 Architectural Style

The system follows Clean Architecture (Onion Architecture) principles. This architectural approach enforces a strict separation between business logic and technical implementation details. Dependencies always point inward, ensuring that core business rules are isolated from frameworks and external tools.

4.2 Layered Structure

The architecture consists of four main layers:

1. Domain Layer
2. Application Layer
3. Infrastructure Layer
4. Web Layer

Each layer has a clearly defined responsibility and communicates with other layers only through abstractions.

4.3 Benefits of the Chosen Architecture

The chosen architectural approach reduces coupling between system components by enforcing clear dependency boundaries and communication through abstractions. This design significantly improves testability, as dependency inversion allows individual components to be tested in isolation. Another important benefit is the ease with which external libraries can be replaced, since infrastructure-specific details are isolated from core business logic. As a result, the system achieves long-term maintainability and remains adaptable to future changes. Additionally, the architecture establishes clear responsibility boundaries between layers, making the overall system easier to understand, develop, and extend.

5. Domain Layer

The Domain layer represents the conceptual foundation of the system. It defines what the system is capable of doing without specifying how those capabilities are implemented.

Within this layer, interfaces are defined for PDF processing, digital signature operations, and QR code generation, describing the core responsibilities of the system in an abstract manner.

The Domain layer does not depend on any external frameworks, libraries, or infrastructure. As a result, it remains stable even when technical decisions change in other layers.

6. Application Layer

The Application layer implements the system’s use cases and business workflows. It coordinates interactions between domain interfaces and infrastructure implementations.

6.1 Responsibilities

Its primary responsibility is to orchestrate the digital signing process, ensuring that all steps are executed in the correct order. The Application layer enforces the overall workflow logic and handles application-level decisions, while delegating all technical and low-level tasks to the infrastructure services. This approach ensures that business logic remains clearly separated from implementation details and external dependencies.

6.2 Digital Signature Service

The central component of this layer is the digital signature service. It receives input documents, triggers signature generation, and ensures that all required steps are executed correctly.

This design ensures that business logic remains independent of low-level technical details.

7. Infrastructure Layer

The Infrastructure layer contains concrete technical implementations. It fulfills the contracts defined by the Domain layer and performs all low-level operations.

7.1 Technical Responsibilities

These operations include manipulating and modifying PDF documents, embedding signature-related data into the documents, generating QR codes used for verification purposes, interacting with the file system for document storage, and managing external libraries and dependencies. By encapsulating all technical responsibilities within this layer, the system maintains a clear separation between business logic and implementation details.

7.2 Isolation of Technical Concerns

By isolating all technical logic in this layer, the system ensures that changes to libraries or tools do not affect higher-level logic.

8. Web Layer

The Web layer exposes the system’s functionality through RESTful endpoints. It acts as the interface between external clients and the application logic.

8.1 Responsibilities

Its responsibilities include receiving HTTP requests from clients, performing initial validation of input data, forwarding validated requests to the appropriate application services, and returning responses back to the clients.

The Web layer does not contain any business logic, which ensures a clean separation between presentation concerns and application processing and helps maintain the overall architectural integrity of the system.

9. Digital Signature Workflow

The digital signing workflow is designed to be deterministic and repeatable:

1. A PDF document is submitted to the system
2. The document is validated
3. Signature data is generated
4. The PDF is modified to include signature information
5. A QR code is generated
6. The final signed document is produced

The Application layer ensures that each step is executed in the correct sequence.

10. Digital Signature Concepts

10.1 Integrity

The system ensures that any modification of the document after signing can be detected.

10.2 Authenticity

The embedded signature information confirms that the document was processed by the system.

10.3 Non-repudiation (Conceptual)

While full cryptographic non-repudiation is outside the scope, the system demonstrates the foundational concepts.

11. QR Code Integration

The QR code serves as a verification aid by encoding information related to the signed document.

11.1 Purpose

Its primary purpose is to support external verification by allowing third parties to access or confirm document-related data. In addition, the QR code can be used for metadata storage, providing contextual information about the signed document, and for improving traceability by linking the document to a verification source or reference.

11.2 Advantages

One of the key advantages of integrating QR codes is that they enable user-friendly verification without requiring specialized tools. Furthermore, QR codes are platform independent and can be scanned using a wide range of devices, while introducing minimal visual intrusion into the document layout.

12. Security Considerations

Although the system is not implemented as a full PKI-based solution, several important security aspects are taken into consideration. The system ensures controlled modification of documents by applying all changes in a structured and predictable manner, reducing the risk of unintended alterations. Cryptographic and signature-related operations are isolated within dedicated components, which limits their exposure to other parts of the system and improves overall security. In addition, the architecture helps prevent unauthorized workflow manipulation by clearly defining and enforcing the sequence of operations involved in the digital signing process. Future versions of the system could further enhance security by integrating certificate authorities and providing stronger cryptographic guarantees.

13. Error Handling and Validation

The system includes validation mechanisms that ensure only valid PDF documents are processed and that invalid inputs are rejected at an early stage of execution. Error handling is designed to be robust and graceful, allowing the system to respond appropriately to exceptional

situations without compromising stability. By detecting and handling errors in a controlled manner, the system improves overall robustness and provides a more reliable and user-friendly experience.

14. Non-Functional Requirements

14.1 Maintainability

Clear separation of concerns enables easier code maintenance.

14.2 Testability

Interfaces allow dependency mocking and unit testing.

14.3 Extensibility

New signing algorithms or verification methods can be added with minimal changes.

15. Execution Instructions

This section describes the practical steps required to execute the Digital Signature System in a local development environment. The application is implemented as an ASP.NET Core Web API and is executed as a backend service that exposes RESTful endpoints for PDF document signing.

15.1 Prerequisites

Before executing the system, the target machine must have the .NET 10 SDK installed. The installed version of the .NET SDK can be verified by executing the “dotnet --version” command in a terminal or command prompt. A compatible development environment such as Visual Studio or Rider may also be used, although execution through the .NET CLI is fully supported.

15.2 Cloning and Building

To begin the execution process, the source code of the project must first be obtained from the remote repository. The repository is cloned to the local machine using a Git client, after which the working directory is changed to the root folder of the project:

git clone https://github.com/ibunceski/DigitalSignature.git
cd DigitalSignature

Once the repository has been successfully cloned, all required NuGet dependencies must be restored. This step ensures that all external packages referenced by the solution are available before building or running the application:

dotnet restore

After restoring the dependencies, the solution should be built to verify that all projects compile correctly and that no compilation errors are present:

dotnet build

15.3 Running the application

After a successful build, the application can be started. The system is designed to be executed as a Web API service, with the Web.Api project acting as the main entry point. To run the application using the .NET CLI, navigate to the Web.Api project directory and execute the following command:

cd src/Web.Api
dotnet run

When the application starts, it launches a local web server and begins listening for incoming HTTP requests. By default, the service is accessible through the following endpoints:

- HTTP: http://localhost:5140
- HTTPS: https://localhost:7152

These endpoints can be used to access and test the exposed REST API, including the PDF signing functionality.

As an alternative to command-line execution, the application can also be run using an integrated development environment. To do so, the solution file (DigitalSignature.slnx) should be opened in an IDE such as Visual Studio or Rider. The Web.Api project must be selected as the startup project, as it represents the entry point of the system. Once configured, the application can be launched using the IDE’s built-in debugger, which starts the Web API service and displays the active listening endpoints in the output console.

15.4 Testing the API

After the application is running, the PDF signing functionality can be tested by sending a request to the exposed REST endpoint. This can be performed using tools such as Postman, curl, or the .http request file included in the Web.Api project. Testing the endpoint confirms that the system is able to receive PDF documents, process them, and generate signed output correctly.

The following example demonstrates how to test the PDF signing endpoint using the curl command-line tool. The request is sent as an HTTP POST call, with the PDF file included as multipart form-data:

```
curl -X POST http://localhost:5140/api/PdfSigning/upload \
  -F "file=@yourfile.pdf" \
  -H "Content-Type: multipart/form-data"
```

If the request is processed successfully, the system responds with a JSON object containing a URL that references the signed PDF document. An example response is shown below:

```
{

  "signedPdfUrl": "http://localhost:5140/signed/signed_{guid}.pdf"

}
```

The returned URL can be opened in a web browser to download and inspect the signed PDF file. The presence of signature-related information and the optional QR code in the document confirms that the signing process was executed correctly.

16. Configuration Parameters

This section describes the configuration parameters that control the runtime behavior of the Digital Signature System. The application follows the standard ASP.NET Core configuration model and uses configuration files to define environment settings, network ports, logging behavior, and required runtime assets. Centralizing configuration in this manner allows the system to be adjusted without modifying the source code.

16.1 Application Settings

The application uses standard ASP.NET Core configuration files located in the src/Web.Api directory. These files define global and environment-specific configuration values that are loaded automatically at runtime.

The appsettings.json file provides the base configuration that applies to all environments. It primarily defines logging behavior and host-related settings, as shown in the following configuration:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

This configuration specifies the default logging level for the application and adjusts logging verbosity for framework components. Additional configuration values can be introduced in this file as the system evolves.

The appsettings.Development.json file is used to override or extend configuration values when the application is running in a development environment. In the current implementation, this file is minimal, but it provides a structured mechanism for introducing development-specific settings without affecting other environments.

16.2 Launch Profiles and Environment Configuration

The application defines multiple launch profiles in the launchSettings.json file, which is located in the src/Web.Api/Properties directory. These launch profiles control how the application is started during execution and specify both the network configuration and environment-specific variables required at runtime.

By default, two launch profiles are provided. The HTTP profile runs the application using an unsecured HTTP endpoint at http://localhost:5140. The HTTPS profile runs the application using a secured HTTPS endpoint at https://localhost:7152, while also exposing the HTTP endpoint at http://localhost:5140. These profiles allow the application to be executed in different network configurations without modifying the application code.

In addition to defining network ports, the launch profiles are also used to configure environment variables required for the application to function correctly. In particular, certificate-related configuration values are provided through environment variables defined within the selected launch profile. The application requires the AppSettings__PfxPath and AppSettings__PfxPassword environment variables to be set in order to load the digital certificate used during the signing process.

```
{
  "profiles": {
    "http": {
      "commandName": "Project",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "AppSettings__PfxPath": "Certificates/certificate.pfx",
        "AppSettings__PfxPassword": "password123"
      },
      "dotnetRunMessages": true,
      "applicationUrl": "http://localhost:5140"
    },
    "https": {
      "commandName": "Project",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "AppSettings__PfxPath": "Certificates/certificate.pfx",
        "AppSettings__PfxPassword": "password123"
      },
      "dotnetRunMessages": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5140"
    }
  },
  "$schema": "https://json.schemastore.org/launchsettings.json"
}
```

16.3 Required Assets

In addition to configuration files, the application depends on specific runtime assets that must be present in the wwwroot directory of the Web API project. These assets are required for successful PDF processing and visual rendering of the signed documents.

A font file located at wwwroot/fonts/NotoSans-Regular.ttf is used for rendering text within the signed PDF documents. An image file located at wwwroot/images/finki_logo.png is embedded into the generated QR codes to provide institutional branding. The directory wwwroot/signed/ is used as the output location for signed PDF documents and is created automatically by the application if it does not already exist.

If any of these assets are missing or incorrectly placed, the PDF signing process may fail at runtime. Therefore, it is important to verify their presence before executing the application.

16.4 Dependency Injection Configuration

The application uses dependency injection to manage service lifetimes and enforce separation of concerns between system components. All core services are registered during application startup in the Program.cs file.

The following service registrations are used to compose the system at runtime:

```
builder.Services.AddScoped<IQrCodeService, QrCodeService>();
builder.Services.AddScoped<IPdfProcessingService, PdfProcessingService>();
builder.Services.AddScoped<IDigitalSignatureService, DigitalSignatureService>();
```

These services are responsible for QR code generation, PDF processing, and orchestration of the digital signing workflow. By configuring dependencies through interfaces, the system remains modular, testable, and easily extensible.

17. Deployment Instructions

The Digital Signature System is deployed as an ASP.NET Core Web API application. Deployment consists of preparing the application for execution in a production environment and starting it as a backend service that listens for HTTP requests.

To deploy the application, it must first be published in Release mode. Publishing generates a set of compiled binaries and runtime files that can be executed independently of the development environment:

dotnet publish -c Release -o ./publish

The published output can then be transferred to a target machine that has the appropriate .NET runtime installed. The application is started by executing the generated Web API assembly, which launches the built-in Kestrel web server and begins listening on the configured network ports.

Static files, including generated signed PDF documents, are served from the wwwroot directory. The application must have write permissions for the output directory used to store signed documents. Configuration values such as ports and environment settings are controlled using standard ASP.NET Core configuration files and environment variables.

This deployment approach allows the system to run as a standalone backend service in both development and production environments.

18. User Scenarios

This section presents representative usage scenarios that illustrate how the Digital Signature System can be applied in practical contexts. The selected scenarios demonstrate typical interactions with the system and highlight its role in document verification and integrity assurance.

Scenario 1: Academic Document Verification

A university distributes course-related documents, such as lecture notes or official materials, to students in digital form. Before distribution, the documents are uploaded to the system in PDF format using the exposed API. The system processes each document and embeds signature-related information together with a QR code. The QR code encodes a verification reference that allows students to verify that the document they received corresponds to the version processed by the system and has not been altered after signing.

Scenario 2: Contract Signing Workflow

A legal or administrative department integrates the Digital Signature System into an internal document management workflow. When a contract reaches its finalized form, the PDF document is submitted to the signing API. The system returns a signed version of the document that includes a QR code linking to verification information stored on the server. Parties involved in the agreement can later verify the integrity of their copy by accessing the verification reference embedded in the QR code and confirming that the document matches the signed version.

Scenario 3: Automated Document Processing

An organization processes a large number of official documents as part of an automated workflow. The Digital Signature System is integrated into a batch processing pipeline, where documents are uploaded programmatically to the API. The system returns signed PDF documents, which are then archived or distributed automatically. The embedded QR codes provide a convenient mechanism for recipients to verify that the documents were processed by the system, even when no manual interaction is involved.

19. Limitations

The system has several intentional limitations that define the scope of the project. It does not include a graphical user interface and is designed to function exclusively as a backend service. Additionally, the system does not integrate with certificate authorities and does not provide user identity management or authentication mechanisms. Full document lifecycle management, such as versioning or long-term archival, is also outside the scope of the project. These limitations are deliberate and help keep the project focused on demonstrating core digital signing concepts and clean architectural design in an educational context.

20. Conclusion

The Digital Signature System demonstrates a structured and well-architected approach to PDF document signing. By combining Clean Architecture principles with practical document processing, the project achieves both technical clarity and academic value. The modular design makes it suitable for further development and real-world adaptation while serving as a strong example of clean backend system design.