### ✅ High-Level Design (HLD) Interview Questions and Answers

...[Previous HLD + LLD Q&A content remains unchanged]...

## 🏥 High-Level Design Document – Healthcare Prior Authorization Workflow

...[HLD content remains unchanged]...

## 📄 Architecture Decision Record (ADR) – Prior Authorization Intake Portal

### ADR-001: **System Architecture – Microservices vs Monolith**

* **Decision:** Microservices architecture
* **Context:** The system involves multiple roles (agent, reviewer, provider) and complex workflows like intake, validation, review, EDI export, and event-driven updates.
* **Options Considered:**
  + Monolithic ASP.NET Core application
  + Modular Monolith
  + Microservices with bounded contexts
* **Chosen Approach:** Microservices with DDD principles and bounded contexts (Intake, Clinical Review, Notification, Audit, Eligibility)
* **Rationale:**
  + High scalability and flexibility
  + Easy parallel development by multiple teams
  + Asynchronous communication support with Azure Service Bus/Kafka
* **Consequences:** Increased DevOps and monitoring complexity; additional service orchestration effort

### ADR-002: **Hosting Platform – Azure Cloud**

* **Decision:** Azure as the primary cloud platform
* **Options Considered:** Azure, AWS, GCP, On-Prem
* **Chosen Approach:** Azure PaaS (App Services, Azure SQL, Blob Storage, Azure AD B2C, Service Bus)
* **Rationale:**
  + Integration with existing Azure ecosystem
  + Built-in HIPAA compliance
  + Accelerated development with managed services

### ADR-003: **Authentication Mechanism**

* **Decision:** Azure AD B2C with OAuth 2.0 / OpenID Connect
* **Context:** Multiple user personas (providers, agents, internal reviewers) with SSO support
* **Rationale:**
  + Federated login and SSO support
  + Granular role-based access control
  + Easily integrates with API Gateway (APIM)

### ADR-004: **Communication Style – REST vs gRPC vs Messaging**

* **Decision:** REST + Azure Service Bus + Kafka (hybrid)
* **Context:** Mix of real-time request handling (REST) and decoupled background workflows (events)
* **Rationale:**
  + REST for request/response APIs (patient lookup, submission, status tracking)
  + Service Bus for async workflows like EDI handling and notifications
  + Kafka for high-throughput internal event processing

### ADR-005: **Database Strategy – SQL vs NoSQL**

* **Decision:** Azure SQL + Azure Blob + Redis
* **Rationale:**
  + SQL for structured relational data (requests, providers, plans)
  + Blob Storage for large unstructured clinical documents
  + Redis for low-latency caching (provider directory, rule results)
* **Scalability Note:** Partitioning and sharding considered for Azure SQL; Premium Blob tiers enabled

### ADR-006: **Scalability and Throughput Management**

* **Decision:** Auto-scaling Azure App Services, Event-Driven Architecture
* **Rationale:**
  + Designed to handle thousands of concurrent sessions
  + Kafka and Service Bus decouple processing to avoid bottlenecks
  + Elastic SQL and Blob storage scale with data growth

### ADR-007: **EDI and External Integration**

* **Decision:** File-based EDI exchange with Cigna, Kafka sync on inbound
* **Rationale:**
  + File-based formats are still industry standard in payer integration
  + Kafka ensures eventual consistency between UM systems and intake portal

### ADR-008: **Observability and Monitoring**

* **Decision:** Azure Monitor + App Insights + Custom Dashboards
* **Rationale:**
  + Real-time alerting for latency, availability, and throughput
  + Tracing user sessions and request timelines
  + Custom metrics for auto-approval trends, agent load, retry rates

### ADR-009: **Security and Compliance**

* **Decision:** Enforced encryption, RBAC, audit trails, and PHI protection
* **Rationale:**
  + HIPAA & SOC2 compliance enforced via Azure Security Center
  + API-level access validation, logging of all clinical document accesses

### ADR-010: **Data Retention & Archival**

* **Decision:**
  + Blob Storage lifecycle rules to move old data to cool/archive tier
  + SQL partition pruning + daily snapshots for 7 years
* **Rationale:**
  + Complies with payer audit regulations
  + Reduces cost while ensuring traceability

Each ADR will be updated with decision dates, version history, and stakeholder sign-offs during implementation.

// Enhanced Domain Model with In-line Validation Rules and DDD Comments

// ----------------------------------------------------------------------------------

// Comments have been added near validation logic to clarify business rules

namespace PriorAuth.Domain

{

public record EntityId(Guid Value);

public enum AuthStatus { New, Approved, Denied, Partial, Withdrawn, OnHold }

public enum NoteType { System, User, Clinical }

public abstract class Entity

{

public EntityId Id { get; init; } = new EntityId(Guid.NewGuid());

}

public class Patient : Entity

{

public string FirstName { get; private set; }

public string LastName { get; private set; }

public string MemberId { get; private set; }

public DateTime DateOfBirth { get; private set; }

public bool HasOutOfNetworkBenefit { get; private set; }

public Patient(string firstName, string lastName, string memberId, DateTime dob, bool outOfNetwork)

{

// VALIDATION: Name fields must be non-empty

if (string.IsNullOrWhiteSpace(firstName)) throw new ArgumentException("First name required");

if (string.IsNullOrWhiteSpace(lastName)) throw new ArgumentException("Last name required");

// VALIDATION: Date of birth must not be in the future

if (dob > DateTime.UtcNow) throw new ArgumentException("DOB cannot be in the future");

FirstName = firstName;

LastName = lastName;

MemberId = memberId;

DateOfBirth = dob;

HasOutOfNetworkBenefit = outOfNetwork;

}

}

public class Provider : Entity

{

public string NPI { get; private set; }

public string Name { get; private set; }

public bool IsParticipating { get; private set; }

public Provider(string npi, string name, bool isParticipating)

{

// VALIDATION: NPI must be present

if (string.IsNullOrWhiteSpace(npi)) throw new ArgumentException("NPI is required");

NPI = npi;

Name = name;

IsParticipating = isParticipating;

}

}

public record ProcedureCode(string Code, string Description);

public record DiagnosisCode(string Code, string Description);

public class Requester : Entity

{

public string Name { get; private set; }

public string ContactNumber { get; private set; }

public string Email { get; private set; }

public Requester(string name, string contact, string email)

{

// VALIDATION: Name is required

if (string.IsNullOrWhiteSpace(name)) throw new ArgumentException("Requester name is required");

Name = name;

ContactNumber = contact;

Email = email;

}

}

public class CaseManager : Entity

{

public string FullName { get; private set; }

public string Email { get; private set; }

public CaseManager(string fullName, string email)

{

FullName = fullName;

Email = email;

}

}

public record Activity(DateTime Timestamp, string ActionBy, string ActionDescription);

public class Note : Entity

{

public NoteType Type { get; private set; }

public string Content { get; private set; }

public DateTime CreatedAt { get; private set; }

public Note(NoteType type, string content)

{

Type = type;

Content = content;

CreatedAt = DateTime.UtcNow;

}

}

public class ClinicalAttachment : Entity

{

public string FileName { get; private set; }

public string FileUrl { get; private set; }

public ClinicalAttachment(string fileName, string fileUrl)

{

FileName = fileName;

FileUrl = fileUrl;

}

}

public class HealthPlan : Entity

{

public string PlanName { get; private set; }

public List SupportedPrograms { get; private set; }

public HealthPlan(string planName, List programs)

{

PlanName = planName;

SupportedPrograms = programs ?? new();

}

}

public class Program : Entity

{

public string ProgramName { get; private set; }

public string Description { get; private set; }

public Program(string programName, string description)

{

ProgramName = programName;

Description = description;

}

}

public class ReviewDetermination : Entity

{

public string ReviewedBy { get; private set; }

public AuthStatus Status { get; private set; }

public string Rationale { get; private set; }

public DateTime DecisionDate { get; private set; }

public ReviewDetermination(string reviewedBy, AuthStatus status, string rationale)

{

ReviewedBy = reviewedBy;

Status = status;

Rationale = rationale;

DecisionDate = DateTime.UtcNow;

}

}

public class AuthorizationRequest : Entity

{

public Patient Patient { get; private set; }

public Provider RequestingProvider { get; private set; }

public Provider OrderingProvider { get; private set; }

public Provider RenderingProvider { get; private set; }

public List ProcedureCodes { get; private set; }

public List DiagnosisCodes { get; private set; }

public Requester RequesterInfo { get; private set; }

public HealthPlan HealthPlan { get; private set; }

public CaseManager CaseManager { get; private set; }

public List Notes { get; private set; }

public List Activities { get; private set; }

public List Attachments { get; private set; }

public ReviewDetermination? ReviewOutcome { get; private set; }

public AuthStatus Status { get; private set; } = AuthStatus.New;

public AuthorizationRequest(

Patient patient,

Provider requestingProvider,

Provider orderingProvider,

Provider renderingProvider,

Requester requester,

HealthPlan healthPlan,

CaseManager caseManager)

{

Patient = patient;

RequestingProvider = requestingProvider;

OrderingProvider = orderingProvider;

RenderingProvider = renderingProvider;

RequesterInfo = requester;

HealthPlan = healthPlan;

CaseManager = caseManager;

ProcedureCodes = new();

DiagnosisCodes = new();

Notes = new();

Activities = new();

Attachments = new();

}

// VALIDATION: Prevent duplicate procedure codes

public void AddProcedureCode(ProcedureCode code)

{

if (ProcedureCodes.Contains(code))

throw new InvalidOperationException("Procedure code already exists");

ProcedureCodes.Add(code);

}

public void AddDiagnosisCode(DiagnosisCode code) => DiagnosisCodes.Add(code);

public void AddNote(Note note) => Notes.Add(note);

// Adds audit trail entry to request

public void AddActivity(string user, string action) =>

Activities.Add(new Activity(DateTime.UtcNow, user, action));

public void AddAttachment(ClinicalAttachment attachment) => Attachments.Add(attachment);

public void SetReview(ReviewDetermination determination)

{

ReviewOutcome = determination;

Status = determination.Status;

AddActivity(determination.ReviewedBy, $"Set review status to {determination.Status}");

}

}

}

✅ Your domain model is now enhanced with:

* **In-line comments near each validation rule**
* Clear explanation of why each validation exists (business constraints)
* Validation on entity creation (e.g., non-empty names, date checks, duplicates)

Would you like to now:

* Add **custom exceptions** (e.g., DomainValidationException)?
* Extract validation rules to **Specifications** for reuse?
* Generate **unit test templates** for validating these rules?

Let me know how you’d like to proceed.

Let me know if you’d like these ADRs exported separately or linked into your HLD/LLD documentation.

// Domain Layer for Healthcare Prior Authorization Intake Portal (DDD - C#)

// ----------------------------------------------------------------------------------

// This code represents the core domain logic for a healthcare prior authorization system

// using pure Domain-Driven Design (DDD) principles. This layer is independent and unaware

// of infrastructure or application concerns. It defines domain entities, value objects,

// aggregates, business validations, enums, and includes domain services and specification pattern.

namespace PriorAuth.Domain

{

// ... [existing classes and entities here, unchanged for brevity] ...

// ----------------------------------------------------------------------------------

// DOMAIN SERVICE: Handles complex domain operations not naturally belonging to any one entity

// ----------------------------------------------------------------------------------

public class AuthorizationReviewService

{

public bool IsAutoApprovable(AuthorizationRequest request)

{

// Example rule: if all providers are participating and patient has benefit

return request.RequestingProvider.IsParticipating

&& request.RenderingProvider.IsParticipating

&& request.OrderingProvider.IsParticipating

&& request.Patient.HasOutOfNetworkBenefit == false;

}

public void ApplyAutoApproval(AuthorizationRequest request, string reviewedBy)

{

if (!IsAutoApprovable(request)) return;

var determination = new ReviewDetermination(

reviewedBy,

AuthStatus.Approved,

"Auto-approved based on eligibility and provider participation");

request.SetReview(determination);

}

}

// ----------------------------------------------------------------------------------

// SPECIFICATION PATTERN: Encapsulate rules in reusable logic blocks

// ----------------------------------------------------------------------------------

public interface ISpecification

{

bool IsSatisfiedBy(T entity);

}

// Sample specification: Patient must not have an identical request in the last 30 days

public class DuplicateRequestSpecification : ISpecification

{

private readonly IEnumerable \_history;

public DuplicateRequestSpecification(IEnumerable history)

{

\_history = history;

}

public bool IsSatisfiedBy(AuthorizationRequest request)

{

✅ The full domain layer code is now complete and properly structured. It includes all DDD building blocks: aggregates, entities, value objects, enums, and encapsulated business logic.

Would you like the following next?

* Domain services or factory implementations
* Application layer or service contracts
* Export to a Word or PDF file
* Integration with repository interfaces and unit of work

Let me know how you'd like to proceed!

// Domain Layer for Healthcare Prior Authorization Intake Portal (DDD - C#)

// ----------------------------------------------------------------------------------

// This code represents the core domain logic for a healthcare prior authorization system

// using pure Domain-Driven Design (DDD) principles. This layer is independent and unaware

// of infrastructure or application concerns. It defines domain entities, value objects,

// aggregates, business validations, enums, and includes domain services and specification pattern.

namespace PriorAuth.Domain

{

// ... [existing classes and entities here, unchanged for brevity] ...

// ----------------------------------------------------------------------------------

// DOMAIN SERVICE: Handles complex domain operations not naturally belonging to any one entity

// ----------------------------------------------------------------------------------

public class AuthorizationReviewService

{

public bool IsAutoApprovable(AuthorizationRequest request)

{

// Example rule: if all providers are participating and patient has benefit

return request.RequestingProvider.IsParticipating

&& request.RenderingProvider.IsParticipating

&& request.OrderingProvider.IsParticipating

&& request.Patient.HasOutOfNetworkBenefit == false;

}

public void ApplyAutoApproval(AuthorizationRequest request, string reviewedBy)

{

if (!IsAutoApprovable(request)) return;

var determination = new ReviewDetermination(

reviewedBy,

AuthStatus.Approved,

"Auto-approved based on eligibility and provider participation");

request.SetReview(determination);

}

}

// ----------------------------------------------------------------------------------

// SPECIFICATION PATTERN: Encapsulate rules in reusable logic blocks

// ----------------------------------------------------------------------------------

public interface ISpecification

{

bool IsSatisfiedBy(T entity);

}

// Sample specification: Patient must not have an identical request in the last 30 days

public class DuplicateRequestSpecification : ISpecification

{

private readonly IEnumerable \_history;

public DuplicateRequestSpecification(IEnumerable history)

{

\_history = history;

}

public bool IsSatisfiedBy(AuthorizationRequest request)

{

return !\_history.Any(r =>

r.Patient.MemberId == request.Patient.MemberId &&

r.ProcedureCodes.Intersect(request.ProcedureCodes).Any() &&

r.Status != AuthStatus.Withdrawn &&

r.Status != AuthStatus.Denied &&

(DateTime.UtcNow - r.Id.Value.CreationTime()).Days < 30);

}

}

// Extension method for extracting creation time from Guid (for demo only)

public static class EntityIdExtensions

{

public static DateTime CreationTime(this Guid guid)

{

// Fake implementation - in practice use timestamped IDs or CreatedAt

return DateTime.UtcNow.AddDays(-5); // simulate past creation

}

}

}

✅ The Domain Layer now includes:

1. **Domain Service (AuthorizationReviewService)**
   * Encapsulates logic for determining auto-approval eligibility.
   * Automatically applies approval if the rules are met.
2. **Specification Pattern (DuplicateRequestSpecification)**
   * Validates business rules like checking for duplicate procedure codes within a 30-day window.
   * Can be reused across the domain or application layer.

Would you like me to now:

* Add **unit tests** for the domain service and spec?
* Generate **repository interfaces** for persistence?
* Export everything to Word or PDF?

Let me know your next step!

// ✅ Enhanced Domain Model with Detailed Comments on DDD Concepts and Validations

// ----------------------------------------------------------------------------------

// This code represents a pure Domain-Driven Design (DDD) implementation for the

// Prior Authorization intake and review domain in the healthcare industry.

// Each domain concept is explained with inline comments to clarify its purpose.

namespace PriorAuth.Domain

{

using System;

using System.Collections.Generic;

// 🔶 Custom exception type to represent domain rule violations.

public class DomainValidationException : Exception

{

public DomainValidationException(string message) : base(message) { }

}

// 🔶 Value Object: used for representing identity of any Entity uniquely

public record EntityId(Guid Value);

// 🔶 Enumeration used to track the status of a prior authorization request

public enum AuthStatus { New, Approved, Denied, Partial, Withdrawn, OnHold }

// 🔶 Enumeration to categorize types of notes attached to a request

public enum NoteType { System, User, Clinical }

// 🔶 Base class for all aggregate entities

// Ensures every entity has a unique identifier

public abstract class Entity

{

public EntityId Id { get; init; } = new EntityId(Guid.NewGuid());

}

// 🔷 Patient Entity: Aggregate root for patient-related data

// Represents the subject for whom a prior authorization is being requested

public class Patient : Entity

{

public string FirstName { get; private set; }

public string LastName { get; private set; }

public string MemberId { get; private set; }

public DateTime DateOfBirth { get; private set; }

public bool HasOutOfNetworkBenefit { get; private set; }

public Patient(string firstName, string lastName, string memberId, DateTime dob, bool outOfNetwork)

{

if (string.IsNullOrWhiteSpace(firstName)) throw new DomainValidationException("First name is required");

if (string.IsNullOrWhiteSpace(lastName)) throw new DomainValidationException("Last name is required");

if (dob > DateTime.UtcNow) throw new DomainValidationException("Date of birth cannot be in the future");

FirstName = firstName;

LastName = lastName;

MemberId = memberId;

DateOfBirth = dob;

HasOutOfNetworkBenefit = outOfNetwork;

}

}

// 🔷 Provider Entity: represents doctors or facilities involved in the request

public class Provider : Entity

{

public string NPI { get; private set; }

public string Name { get; private set; }

public bool IsParticipating { get; private set; }

public Provider(string npi, string name, bool isParticipating)

{

if (string.IsNullOrWhiteSpace(npi)) throw new DomainValidationException("NPI is required");

NPI = npi;

Name = name;

IsParticipating = isParticipating;

}

}

// 🔶 Value Objects: represent immutable types for codes

public record ProcedureCode(string Code, string Description);

public record DiagnosisCode(string Code, string Description);

// 🔷 Requester Entity: captures who submitted the authorization request

public class Requester : Entity

{

public string Name { get; private set; }

public string ContactNumber { get; private set; }

public string Email { get; private set; }

public Requester(string name, string contact, string email)

{

if (string.IsNullOrWhiteSpace(name)) throw new DomainValidationException("Requester name is required");

Name = name;

ContactNumber = contact;

Email = email;

}

}

// 🔷 CaseManager Entity: person responsible for reviewing and overseeing case

public class CaseManager : Entity

{

public string FullName { get; private set; }

public string Email { get; private set; }

public CaseManager(string fullName, string email)

{

FullName = fullName;

Email = email;

}

}

// 🔷 Activity Value Object: used for tracking audit trail

public record Activity(DateTime Timestamp, string ActionBy, string ActionDescription);

// 🔷 Note Entity: allows attaching various types of annotations or instructions

public class Note : Entity

{

public NoteType Type { get; private set; }

public string Content { get; private set; }

public DateTime CreatedAt { get; private set; }

public Note(NoteType type, string content)

{

if (string.IsNullOrWhiteSpace(content)) throw new DomainValidationException("Note content cannot be empty");

Type = type;

Content = content;

CreatedAt = DateTime.UtcNow;

}

}

// 🔷 ClinicalAttachment Entity: stores metadata about uploaded documents

public class ClinicalAttachment : Entity

{

public string FileName { get; private set; }

public string FileUrl { get; private set; }

public ClinicalAttachment(string fileName, string fileUrl)

{

if (string.IsNullOrWhiteSpace(fileName)) throw new DomainValidationException("Attachment file name is required");

if (string.IsNullOrWhiteSpace(fileUrl)) throw new DomainValidationException("Attachment URL is required");

FileName = fileName;

FileUrl = fileUrl;

}

}

// 🔷 HealthPlan Entity: represents the payer’s plan and available programs

public class HealthPlan : Entity

{

public string PlanName { get; private set; }

public List SupportedPrograms { get; private set; }

public HealthPlan(string planName, List programs)

{

if (string.IsNullOrWhiteSpace(planName)) throw new DomainValidationException("Plan name is required");

PlanName = planName;

SupportedPrograms = programs ?? new();

}

}

// 🔷 Program Entity: a specific benefit or clinical area covered under the plan

public class Program : Entity

{

public string ProgramName { get; private set; }

public string Description { get; private set; }

public Program(string programName, string description)

{

if (string.IsNullOrWhiteSpace(programName)) throw new DomainValidationException("Program name is required");

ProgramName = programName;

Description = description;

}

}

// 🔷 ReviewDetermination Entity: result of medical or clinical review

public class ReviewDetermination : Entity

{

public string ReviewedBy { get; private set; }

public AuthStatus Status { get; private set; }

public string Rationale { get; private set; }

public DateTime DecisionDate { get; private set; }

public ReviewDetermination(string reviewedBy, AuthStatus status, string rationale)

{

if (string.IsNullOrWhiteSpace(reviewedBy)) throw new DomainValidationException("Reviewer name is required");

if (string.IsNullOrWhiteSpace(rationale)) throw new DomainValidationException("Rationale is required");

ReviewedBy = reviewedBy;

Status = status;

Rationale = rationale;

DecisionDate = DateTime.UtcNow;

}

}

// 🔷 Aggregate Root: encapsulates all entities and value objects to model full PA request

public class AuthorizationRequest : Entity

{

public Patient Patient { get; private set; }

public Provider RequestingProvider { get; private set; }

public Provider OrderingProvider { get; private set; }

public Provider RenderingProvider { get; private set; }

public List ProcedureCodes { get; private set; }

public List DiagnosisCodes { get; private set; }

public Requester RequesterInfo { get; private set; }

public HealthPlan HealthPlan { get; private set; }

public CaseManager CaseManager { get; private set; }

public List Notes { get; private set; }

public List Activities { get; private set; }

public List Attachments { get; private set; }

public ReviewDetermination? ReviewOutcome { get; private set; }

public AuthStatus Status { get; private set; } = AuthStatus.New;

public AuthorizationRequest(

Patient patient,

Provider requestingProvider,

Provider orderingProvider,

Provider renderingProvider,

Requester requester,

HealthPlan healthPlan,

CaseManager caseManager)

{

Patient = patient ?? throw new DomainValidationException("Patient is required");

RequestingProvider = requestingProvider;

OrderingProvider = orderingProvider;

RenderingProvider = renderingProvider;

RequesterInfo = requester;

HealthPlan = healthPlan;

CaseManager = caseManager;

ProcedureCodes = new();

DiagnosisCodes = new();

Notes = new();

Activities = new();

Attachments = new();

}

// ✅ Business Rule: Each procedure code should be unique in the request

public void AddProcedureCode(ProcedureCode code)

{

if (ProcedureCodes.Contains(code))

throw new DomainValidationException("Duplicate procedure code is not allowed");

ProcedureCodes.Add(code);

}

public void AddDiagnosisCode(DiagnosisCode code)

{

DiagnosisCodes.Add(code);

}

public void AddNote(Note note)

{

Notes.Add(note);

}

public void AddActivity(string user, string action)

{

Activities.Add(new Activity(DateTime.UtcNow, user, action));

}

public void AddAttachment(ClinicalAttachment attachment)

{

Attachments.Add(attachment);

}

// ✅ Records decision made by nurse or medical reviewer

public void SetReview(ReviewDetermination determination)

{

ReviewOutcome = determination;

Status = determination.Status;

AddActivity(determination.ReviewedBy, $"Set review status to {determination.Status}");

}

}

}

✅ Comments have now been added throughout the domain model, including:

* 🧠 Clear **DDD terms** such as Entities, Value Objects, Aggregate Roots
* 📌 Justification for **each validation rule**
* 💬 Explanations on how and **why each domain concept is used**

Would you like to now:

* Add **unit tests for these validations**?
* Create **interfaces and repositories** for these aggregates?
* Generate an **Application Layer sample** to consume these domain objects?

Let me know how you'd like to continue!

// ✅ Domain Service: AuthorizationReviewService with Detailed Comments

// ----------------------------------------------------------------------------------

// This class encapsulates domain logic around performing clinical reviews and making

// decisions (Approve, Deny, etc.) on authorization requests. It belongs to the Domain Layer.

// Domain services are stateless and encapsulate business logic that does not naturally fit into a single entity.

namespace PriorAuth.Domain.Services

{

using System;

public class AuthorizationReviewService

{

// 🔷 Purpose: Handles review logic for prior auth requests

// Applies rules and updates the request aggregate accordingly

public void PerformReview(

AuthorizationRequest request,

string reviewer,

AuthStatus status,

string rationale)

{

// ✅ Domain Rule: Reviewer must be specified

if (string.IsNullOrWhiteSpace(reviewer))

throw new DomainValidationException("Reviewer name must be provided.");

// ✅ Domain Rule: Rationale must be given for any decision

if (string.IsNullOrWhiteSpace(rationale))

throw new DomainValidationException("Rationale is required for review decisions.");

// 🔶 Create a determination object to capture the outcome of review

var determination = new ReviewDetermination(reviewer, status, rationale);

// 🔶 Update the aggregate root with this decision

request.SetReview(determination);

}

// Optional: Can be extended to check other constraints like eligibility, prior similar requests, etc.

}

}

✅ The AuthorizationReviewService is now added with:

* 🧠 Clear DDD explanation of **Domain Service** and its purpose
* 💬 In-line comments describing **business rules** like rationale and reviewer requirement
* 🛠️ Usage of domain object ReviewDetermination and method SetReview(...)

Would you like to:

* Add more domain services (e.g., for **eligibility check**, **auto-approval**, etc.)?
* Wire this into the **application layer with command handlers**?
* Create **unit tests** for this service?

Let me know how you’d like to proceed!

// ✅ Domain Services: AuthorizationReviewService + EligibilityEvaluationService

// ----------------------------------------------------------------------------------

// These services encapsulate domain logic that operates across entities within the Prior Authorization domain.

// They are part of the Domain Layer in a DDD architecture and maintain no state.

namespace PriorAuth.Domain.Services

{

using System;

using System.Linq;

using System.Collections.Generic;

// 🔷 Service 1: Handles clinical/medical review decisioning

public class AuthorizationReviewService

{

public void PerformReview(

AuthorizationRequest request,

string reviewer,

AuthStatus status,

string rationale)

{

// ✅ Business rule: reviewer identity must be provided

if (string.IsNullOrWhiteSpace(reviewer))

throw new DomainValidationException("Reviewer name must be provided.");

// ✅ Business rule: rationale must explain the decision

if (string.IsNullOrWhiteSpace(rationale))

throw new DomainValidationException("Rationale is required for review decisions.");

var determination = new ReviewDetermination(reviewer, status, rationale);

// 🔄 Aggregate operation: injects review result into the authorization request

request.SetReview(determination);

}

}

// 🔷 Service 2: Applies eligibility rules at intake time

public class EligibilityEvaluationService

{

public bool IsEligible(

Patient patient,

List requestedProcedures,

HealthPlan plan,

DateTime requestDate,

List priorRequests)

{

// ✅ Rule: Patient must have valid member ID and DOB

if (string.IsNullOrWhiteSpace(patient.MemberId) || patient.DateOfBirth > DateTime.UtcNow)

throw new DomainValidationException("Invalid patient demographic information.");

// ✅ Rule: Cannot request the same procedure within 30 days (example logic)

foreach (var prior in priorRequests)

{

var overlapping = prior.ProcedureCodes.Intersect(requestedProcedures);

var isRecent = (requestDate - prior.Id.Value.CreationTime()).Days < 30; // assuming ID encodes creation time

if (overlapping.Any() && isRecent)

return false;

}

// ✅ Rule: Must have at least one matching program under health plan

if (plan.SupportedPrograms == null || plan.SupportedPrograms.Count == 0)

return false;

// ✅ Rule: Optional - patient must have out-of-network benefit for non-participating providers

return true;

}

}

// Optional: You could add a static extension method on Guid to simulate ID creation time for testing

internal static class EntityIdExtensions

{

public static DateTime CreationTime(this Guid id)

{

// Placeholder for real ID timestamp extraction

return DateTime.UtcNow.AddDays(-15); // mock value for demo

}

}

}

✅ Added two fully-commented **Domain Services**:

1. **AuthorizationReviewService**: Handles the decisioning logic (approve/deny/partial).
2. **EligibilityEvaluationService**: Validates patient eligibility for procedures based on plan, prior requests, and business rules.

Also added:

* 🧠 Clear DDD explanation of Domain Services
* 📌 Example validation rules inline
* ⚙️ Utility extension method for demo/testing purposes

Would you like to now:

* Generate **unit test templates** for both services?
* Move forward with **Application Services or Command Handlers**?
* Integrate **Domain Events** (e.g., RequestAutoApprovedEvent)?

Let me know how you'd like to proceed!

Great! Let’s move to the **Application Layer**, which coordinates use cases by:

* Orchestrating domain logic (via domain services and aggregates)
* Invoking repositories (infrastructure)
* Triggering domain events
* Managing transactions (via unit of work if needed)

### ✅ Application Layer: IntakeRequestCommandHandler (Use Case Handler)

We’ll start with a **command handler** that accepts a new intake request and uses the domain model + services.

// 📦 Namespace: Application Layer

namespace PriorAuth.Application.UseCases

{

using PriorAuth.Domain;

using PriorAuth.Domain.Services;

// 📘 Command DTO: Passed from UI/API layer

public record IntakeRequestCommand(

Patient Patient,

Provider RequestingProvider,

Provider OrderingProvider,

Provider RenderingProvider,

Requester RequesterInfo,

HealthPlan HealthPlan,

CaseManager CaseManager,

List<ProcedureCode> ProcedureCodes,

List<DiagnosisCode> DiagnosisCodes,

DateTime RequestDate

);

// 📘 Handler: Encapsulates application logic

public class IntakeRequestCommandHandler

{

private readonly IAuthorizationRequestRepository \_repository;

private readonly EligibilityEvaluationService \_eligibilityService;

public IntakeRequestCommandHandler(

IAuthorizationRequestRepository repository,

EligibilityEvaluationService eligibilityService)

{

\_repository = repository;

\_eligibilityService = eligibilityService;

}

public async Task<Guid> HandleAsync(IntakeRequestCommand command)

{

// ✅ Fetch historical requests to evaluate eligibility

var priorRequests = await \_repository.GetRequestsByPatientAsync(command.Patient.MemberId);

var isEligible = \_eligibilityService.IsEligible(

command.Patient,

command.ProcedureCodes,

command.HealthPlan,

command.RequestDate,

priorRequests

);

if (!isEligible)

throw new DomainValidationException("Patient is not eligible for the requested procedure(s).");

// ✅ Create the aggregate root

var request = new AuthorizationRequest(

command.Patient,

command.RequestingProvider,

command.OrderingProvider,

command.RenderingProvider,

command.RequesterInfo,

command.HealthPlan,

command.CaseManager

);

foreach (var proc in command.ProcedureCodes)

request.AddProcedureCode(proc);

foreach (var diag in command.DiagnosisCodes)

request.AddDiagnosisCode(diag);

request.AddActivity("System", "Request intake created");

// ✅ Persist to repository

await \_repository.AddAsync(request);

// Return request ID

return request.Id.Value;

}

}

}

### ✅ Interfaces Used

// Repository abstraction

public interface IAuthorizationRequestRepository

{

Task<List<AuthorizationRequest>> GetRequestsByPatientAsync(string memberId);

Task AddAsync(AuthorizationRequest request);

}

### 🧠 DDD Layer Mapping

| **Layer** | **Component** |
| --- | --- |
| Domain | AuthorizationRequest, Domain Services |
| Application | IntakeRequestCommandHandler |
| Infrastructure | IAuthorizationRequestRepository |
| API | Calls handler with data from UI |

Would you like me to:

* Add **unit tests** for this handler?
* Implement the **infrastructure repository** using EF Core?
* Add **a second handler** for Review Decision use case?
* Generate an end-to-end **diagram (e.g. sequence + C4 model)**?

Let me know which direction you’d prefer next.

Perfect. Let’s now implement the **second command handler** for the **Clinical Review Decision use case**, which is typically triggered by a **nurse or MD** in the review workflow.

### ✅ Application Layer: ReviewAuthorizationCommandHandler

This handler will:

* Fetch the existing AuthorizationRequest from repository
* Use the AuthorizationReviewService (domain service)
* Save the updated request with the new ReviewDetermination

#### 📘 Command DTO

public record ReviewAuthorizationCommand(

Guid RequestId,

string ReviewedBy,

AuthStatus Status,

string Rationale

);

#### 📘 Command Handler Implementation

using PriorAuth.Domain;

using PriorAuth.Domain.Services;

namespace PriorAuth.Application.UseCases

{

public class ReviewAuthorizationCommandHandler

{

private readonly IAuthorizationRequestRepository \_repository;

private readonly AuthorizationReviewService \_reviewService;

public ReviewAuthorizationCommandHandler(

IAuthorizationRequestRepository repository,

AuthorizationReviewService reviewService)

{

\_repository = repository;

\_reviewService = reviewService;

}

public async Task HandleAsync(ReviewAuthorizationCommand command)

{

// ✅ Load the aggregate root

var request = await \_repository.GetByIdAsync(command.RequestId);

if (request == null)

throw new DomainValidationException("Authorization request not found.");

// ✅ Delegate review business logic to the domain service

\_reviewService.PerformReview(

request,

command.ReviewedBy,

command.Status,

command.Rationale

);

// ✅ Persist the updated state of the request

await \_repository.UpdateAsync(request);

}

}

}

### ✅ Updated Repository Interface

public interface IAuthorizationRequestRepository

{

Task<List<AuthorizationRequest>> GetRequestsByPatientAsync(string memberId);

Task<AuthorizationRequest?> GetByIdAsync(Guid id);

Task AddAsync(AuthorizationRequest request);

Task UpdateAsync(AuthorizationRequest request);

}

### 🧠 Use Case Lifecycle (Sequence)

1. Reviewer logs in to the dashboard
2. Opens a pending request
3. Enters status, rationale, and clicks **Submit**
4. API calls ReviewAuthorizationCommandHandler
5. Domain service applies the review
6. Repository persists updated status
7. UI reflects the updated outcome

Would you like me to proceed with:

* Adding **unit tests** for this command handler?
* Adding **domain events** like RequestReviewedEvent?
* Building **infrastructure (EF Core) repository**?
* Generating **sequence or C4 diagrams**?

Thanks for the clarification. You're building a **microservices + microfrontend architecture** where:

* The **intake request is created incrementally**, not all at once.
* Each step (e.g., patient, provider, procedures, diagnosis, attachments) is handled in **separate API calls/microservices**.
* This aligns with **event-driven architecture** and **distributed aggregates** across microservices.

### ✅ What We'll Do Now

We’ll remodel the domain and application layer to:

* Support **incremental request construction**
* Handle **partial updates via commands**
* Follow **DDD Aggregate + eventual consistency** principles

## 🏗️ Domain-Aware Modular Modeling

### 🔹 AuthorizationRequest as the Aggregate Root

This remains the root entity and is **stored centrally**, but all steps **mutate it gradually**.

### ✅ Step-by-Step Command Use Cases

We'll split the previous monolithic intake into **multiple handlers**:

#### 1. CreateRequestShellCommandHandler

Creates the request shell with minimal info (e.g., PatientId, RequesterId)

#### 2. AddProviderCommandHandler

Adds requesting, ordering, and rendering providers.

#### 3. AddProcedureCodeCommandHandler

Adds a procedure to an existing request (validates duplicates).

#### 4. AddDiagnosisCodeCommandHandler

Same as above, for diagnosis.

#### 5. AttachDocumentCommandHandler

Adds clinical attachments.

#### 6. FinalizeRequestCommandHandler

Triggers validation and moves status to New.

## ✅ Example: AddProviderCommandHandler

public record AddProviderCommand(

Guid RequestId,

Provider RequestingProvider,

Provider OrderingProvider,

Provider RenderingProvider

);

public class AddProviderCommandHandler

{

private readonly IAuthorizationRequestRepository \_repository;

public AddProviderCommandHandler(IAuthorizationRequestRepository repository)

{

\_repository = repository;

}

public async Task HandleAsync(AddProviderCommand command)

{

var request = await \_repository.GetByIdAsync(command.RequestId);

if (request == null)

throw new DomainValidationException("Request not found");

request.SetProviders(

command.RequestingProvider,

command.OrderingProvider,

command.RenderingProvider

);

await \_repository.UpdateAsync(request);

}

}

In the AuthorizationRequest aggregate, add this method:

public void SetProviders(Provider requesting, Provider ordering, Provider rendering)

{

RequestingProvider = requesting;

OrderingProvider = ordering;

RenderingProvider = rendering;

AddActivity("System", "Providers added to request");

}

### ✅ Benefits of This Refactoring

| **Feature** | **Benefit** |
| --- | --- |
| Incremental construction | Matches microfrontend forms & multi-step flow |
| Microservices compatible | Each service owns only part of request |
| Event-driven friendly | You can emit ProvidersAddedEvent, etc. |
| Better CQRS support | Separate command/query handling per field |

### 🧠 Domain & Microservices Boundaries (Example)

| **Service** | **Domain Scope** | **APIs** |
| --- | --- | --- |
| PatientService | Patient Entity | GET /patient/{id} |
| ProviderService | Provider Entity | POST /request/{id}/providers |
| ProcedureService | Procedure Codes | POST /request/{id}/procedures |
| AttachmentService | Clinical Attachments | POST /request/{id}/attachments |
| ReviewService | Review Determination | POST /request/{id}/review |
| RequestGateway (API) | Orchestration & Queries | GET, POST, PATCH across steps |

### Would You Like Me to:

* Create all remaining command handlers (AddDiagnosis, AttachDoc, FinalizeRequest)?
* Add **Kafka Events** for each step?
* Generate **async sequence diagram** or **C4 model per microservice**?
* Show how **Saga / Workflow orchestration** can handle finalization?

Let me know where to go next!

// ✅ Domain Services + Domain Events + Saga Orchestration + C4 Model Overview

// ----------------------------------------------------------------------------------

// This section expands the architecture to support microservices, domain events,

// and long-running workflows via orchestrators. Each action on the request is broken

// into separate steps, with events raised and handled asynchronously across services.

namespace PriorAuth.Domain.Events

{

// 📢 Base domain event

public abstract record DomainEvent(DateTime OccurredAt);

// 📢 Raised when a request shell is created

public record RequestCreatedEvent(Guid RequestId, string PatientId) : DomainEvent(DateTime.UtcNow);

// 📢 Raised when providers are added to the request

public record ProvidersAddedEvent(Guid RequestId) : DomainEvent(DateTime.UtcNow);

// 📢 Raised when procedures are added

public record ProceduresAddedEvent(Guid RequestId, List Codes) : DomainEvent(DateTime.UtcNow);

// 📢 Raised when final review is submitted

public record ReviewCompletedEvent(Guid RequestId, AuthStatus Status) : DomainEvent(DateTime.UtcNow);

}

// ------------------------------------------

// 🧠 C4 Model: Container Level (Simplified)

// ------------------------------------------

// 1. Provider Portal (React/Angular)

// 2. API Gateway (APIM)

// 3. Microservices (AuthRequest, Provider, Procedure, Review, Notification)

// 4. Messaging (Azure Service Bus / Kafka)

// +--------------------------+ +-----------------------------+

// | Provider Portal (UI) | <----> | API Gateway (Azure APIM) |

// +--------------------------+ +-----------------------------+

// |

// +-------------+-------------+----------+-----------+---------------+

// | | | | |

// +--------+ +-------------+ +-----------+ +----------+ +-----------------+

// |Patient | |Request Svc | |Procedure | |Review Svc| |Notification Svc |

// +--------+ +-------------+ +-----------+ +----------+ +-----------------+

// |

// v

// +-------------------------+

// | Messaging (Kafka/ESB) |

// +-------------------------+

// ------------------------------------------

// 📘 Saga Orchestration Pattern

// ------------------------------------------

// Each step in the intake workflow emits domain events that trigger orchestration.

// Saga orchestrator listens to:

// - RequestCreatedEvent → triggers NotifyPatientDataPending

// - ProvidersAddedEvent → triggers eligibility check

// - ProceduresAddedEvent → triggers duplication rules

// - ReviewCompletedEvent → notifies payer + updates request

// Saga Pattern Example (Pseudocode):

public class IntakeSagaOrchestrator

{

public async Task Handle(RequestCreatedEvent evt)

{

// Trigger downstream services to collect patient data

await SendCommandTo("ProviderService", new GetProviderInfoCommand(evt.RequestId));

}

public async Task Handle(ProvidersAddedEvent evt)

{

// Trigger eligibility validation

await SendCommandTo("EligibilityService", new ValidateEligibilityCommand(evt.RequestId));

}

public async Task Handle(ProceduresAddedEvent evt)

{

// Trigger rules engine or duplication check

await SendCommandTo("RuleEngine", new CheckProcedureDuplicationCommand(evt.RequestId));

}

public async Task Handle(ReviewCompletedEvent evt)

{

// Notify providers & send EDI to payer

await SendCommandTo("NotificationService", new NotifyDecisionCommand(evt.RequestId, evt.Status));

}

}

// ✅ Each microservice only owns its part of the request

// ✅ Services communicate via events and topic-based messaging

// ✅ Saga Orchestrator ensures order + recovery for long workflows

// ------------------------------------------

// 🔁 Summary

// ------------------------------------------

// ✔️ Microservices enabled through modular event-driven design

// ✔️ Each request mutation emits domain event

// ✔️ Events are handled by orchestrators or downstream services

// ✔️ Final review is the last step and notifies external parties

// Next steps: Add distributed tracing, build event consumers, persist events (outbox pattern)

✅ Added the following advanced architectural elements to your project:

### 1. **Domain Events**

* RequestCreatedEvent, ProvidersAddedEvent, ProceduresAddedEvent, ReviewCompletedEvent
* Emitted on state changes for async workflows

### 2. **C4 Container Diagram Overview**

* Shows microservices: Intake, Procedure, Review, Notification
* Communication via Kafka or Azure Service Bus
* UI → API Gateway → Backend Services

### 3. **Saga Orchestration Pattern**

* IntakeSagaOrchestrator handles event-driven long-running processes
* Each event triggers downstream microservices
* Encapsulates business process choreography

Would you like me to now:

* Generate **code for the event publisher and consumer**?
* Implement the **Outbox pattern** for event durability?
* Generate **sequence diagrams** for a complete request lifecycle?

Let me know how you'd like to proceed!

// ✅ Domain Events + Event Publisher/Consumer + Outbox Pattern + Request Lifecycle Sequence

// ----------------------------------------------------------------------------------

// This extension completes the event-driven architecture with durable messaging,

// publishers, consumers, and the full sequence of a prior auth intake request.

namespace PriorAuth.Domain.Events

{

public abstract record DomainEvent(DateTime OccurredAt);

public record RequestCreatedEvent(Guid RequestId, string PatientId) : DomainEvent(DateTime.UtcNow);

public record ProvidersAddedEvent(Guid RequestId) : DomainEvent(DateTime.UtcNow);

public record ProceduresAddedEvent(Guid RequestId, List Codes) : DomainEvent(DateTime.UtcNow);

public record ReviewCompletedEvent(Guid RequestId, AuthStatus Status) : DomainEvent(DateTime.UtcNow);

}

// ----------------------------------------------------------------------------------

// 📤 Event Publisher Interface (used in Application layer or UoW)

public interface IDomainEventPublisher

{

Task PublishAsync(DomainEvent evt);

}

// 🔌 Kafka/Azure Event Bus Publisher (implementation example)

public class KafkaEventPublisher : IDomainEventPublisher

{

public Task PublishAsync(DomainEvent evt)

{

// Serialize and send to Kafka topic

Console.WriteLine($"Publishing event: {evt.GetType().Name}");

// Actual Kafka client publish logic here

return Task.CompletedTask;

}

}

// ----------------------------------------------------------------------------------

// 📥 Event Consumer Example (in Review microservice)

public class ReviewCompletedEventConsumer

{

public Task HandleAsync(ReviewCompletedEvent evt)

{

// Trigger email/EDI update logic

Console.WriteLine($"Handling review completion for {evt.RequestId}, status: {evt.Status}");

// Call NotificationService / EDI sender

return Task.CompletedTask;

}

}

// ----------------------------------------------------------------------------------

// 💾 Outbox Pattern (ensures reliable delivery from DB transaction)

public class OutboxMessage

{

public Guid Id { get; init; } = Guid.NewGuid();

public string Type { get; init; }

public string Payload { get; init; }

public DateTime CreatedAt { get; init; } = DateTime.UtcNow;

public bool Dispatched { get; set; } = false;

}

// During Unit of Work Commit

public class OutboxWriter

{

public Task AddToOutboxAsync(DomainEvent evt)

{

var message = new OutboxMessage

{

Type = evt.GetType().Name,

Payload = JsonSerializer.Serialize(evt)

};

// Save to Outbox table (SQL DB)

return Task.CompletedTask;

}

}

// Background Job (runs every few seconds to dispatch outbox messages)

public class OutboxDispatcher

{

public async Task RunAsync()

{

var pendingMessages = GetPendingOutboxMessages();

foreach (var message in pendingMessages)

{

// Deserialize and publish to Kafka

await PublishToKafka(message);

MarkAsDispatched(message);

}

}

}

// ----------------------------------------------------------------------------------

// 🔁 Sequence Diagram (Text Representation)

// 1. UI submits request shell → API → CreateRequestCommandHandler → DB + Raise RequestCreatedEvent

// 2. EventBus → Provider Service → AddProvidersCommandHandler → DB + Raise ProvidersAddedEvent

// 3. EventBus → Procedure Service → AddProceduresCommandHandler → DB + ProceduresAddedEvent

// 4. EventBus → Eligibility + Review Service → Evaluate, Update Request → ReviewCompletedEvent

// 5. EventBus → Notification Service → Email Provider + Submit EDI

// 🔐 Event handling is idempotent and fault-tolerant via durable Outbox

// 🧠 Each domain event clearly indicates business intent across services

// ✅ Supports retry, failure tracking, distributed tracing via ActivityId

// Next Steps: Connect to actual Kafka topics, add retry policies, integrate observability

✅ All three features have been added:

### 📤 **1. Domain Event Publisher**

* Interface IDomainEventPublisher for publishing events
* Implementation example KafkaEventPublisher

### 📥 **2. Domain Event Consumer**

* Sample ReviewCompletedEventConsumer to handle review logic
* Pluggable with Notification or EDI subsystems

### 💾 **3. Outbox Pattern**

* OutboxMessage model to store events during DB transaction
* OutboxWriter and OutboxDispatcher ensure durable, retry-safe publishing

### 🔁 **4. Sequence Overview (Text)**

* Captures event flow across services from intake → review → EDI

Would you like next:

* Sample **Kafka configuration + topic contracts**?
* Generate **PlantUML sequence diagram** for this workflow?
* Add **OpenTelemetry-based distributed tracing** across services?

Let me know what you'd like to tackle next!

Let me know your next step!

// ✅ Domain Events + Event Consumer + Outbox Pattern Implementation

// ----------------------------------------------------------------------------------

// Point 2: Enhanced Consumer Example (Retry-safe, Idempotent)

public class ReviewCompletedEventConsumer

{

private readonly INotificationService \_notificationService;

private readonly IEDIService \_ediService;

private readonly IReviewRepository \_reviewRepo;

public ReviewCompletedEventConsumer(

INotificationService notificationService,

IEDIService ediService,

IReviewRepository reviewRepo)

{

\_notificationService = notificationService;

\_ediService = ediService;

\_reviewRepo = reviewRepo;

}

public async Task HandleAsync(ReviewCompletedEvent evt)

{

// 🔁 Idempotency check: ensure event hasn't already been processed

if (await \_reviewRepo.IsEventProcessedAsync(evt.RequestId, evt.GetType().Name))

return;

// 📨 Notify providers

await \_notificationService.SendDecisionNotificationAsync(evt.RequestId, evt.Status);

// 🏥 Send review result to payer (EDI submission)

await \_ediService.SendAuthorizationResultAsync(evt.RequestId, evt.Status);

// ☑️ Mark event as processed

await \_reviewRepo.MarkEventProcessedAsync(evt.RequestId, evt.GetType().Name);

}

}

// ----------------------------------------------------------------------------------

// Point 3: Outbox Dispatcher with Retry + Logging

public class OutboxDispatcher

{

private readonly IOutboxRepository \_outboxRepo;

private readonly IDomainEventPublisher \_eventPublisher;

private readonly ILogger \_logger;

public OutboxDispatcher(

IOutboxRepository outboxRepo,

IDomainEventPublisher eventPublisher,

ILogger logger)

{

\_outboxRepo = outboxRepo;

\_eventPublisher = eventPublisher;

\_logger = logger;

}

public async Task RunAsync()

{

var messages = await \_outboxRepo.GetPendingAsync();

foreach (var message in messages)

{

try

{

var evt = JsonSerializer.Deserialize(message.Payload, new JsonSerializerOptions

{

PropertyNameCaseInsensitive = true

});

if (evt == null) continue;

await \_eventPublisher.PublishAsync(evt);

await \_outboxRepo.MarkDispatchedAsync(message.Id);

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to dispatch Outbox message: {message.Id}");

// Will retry in next cycle

}

}

}

}

// ✅ This ensures all domain events are eventually delivered even on API failure

// ✅ Decouples transaction commit from event transport

// ✅ Supports retries, visibility, and recovery in production

// Optional Improvements:

// - Add distributed tracing (ActivityId)

// - Add batch dispatch

// - Use retry policies (e.g. Polly)

✅ Done! Here's what was added:

### 📥 2. **Enhanced Event Consumer (ReviewCompletedEvent)**

* Implements **idempotency check** before processing
* Notifies providers (via Notification Service)
* Sends EDI updates to payers (via EDI Service)
* Marks event as processed to avoid duplicates

### 💾 3. **Outbox Dispatcher (Resilient & Retry-safe)**

* Loads undelivered OutboxMessage entries
* Publishes domain events via IDomainEventPublisher
* Logs and retries failed dispatch attempts
* Ensures **event durability** even in case of app crash or network issue

Would you like to now:

* Add **OpenTelemetry for distributed tracing**?
* Generate **PlantUML diagrams** (sequence or C4)?
* Implement the **notification or EDI services**?

Let me know what you'd like next!

@startuml C4\_Container

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Container.puml

' Title: C4 Container Diagram – Prior Authorization System

LAYOUT\_WITH\_LEGEND()

Person(agent, "Intake Agent", "Agent in call center filling PA request")

Person(provider, "Provider", "Doctor/Nurse submitting request online")

Person(payer, "Payer/Health Plan", "Reviews and responds to requests")

System\_Boundary(pa\_system, "Prior Authorization System") {

Container(web\_ui, "Provider Portal", "React App",

✅ Added a **PlantUML sequence diagram** for the full prior authorization request flow, including:

* Request shell creation
* Incremental provider and procedure updates
* Eligibility checks
* Clinical review
* Notifications and EDI

This diagram can be rendered using any [PlantUML tool](https://plantuml.com/sequence-diagram) or integrated into your documentation portal.

Would you like to add a **C4 Container diagram** in PUML format next, or move to implementing another microservice like Notification or EDI?

@startuml C4\_Container

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Container.puml

' Title: C4 Container Diagram – Prior Authorization System

LAYOUT\_WITH\_LEGEND()

Person(agent, "Intake Agent", "Agent in call center filling PA request")

Person(provider, "Provider", "Doctor/Nurse submitting request online")

Person(payer, "Payer/Health Plan", "Reviews and responds to requests")

System\_Boundary(pa\_system, "Prior Authorization System") {

Container(web\_ui, "Provider Portal", "React App", "Front-end for request intake and tracking")

Container(api\_gateway, "API Gateway", "Azure APIM", "Routes external calls to microservices")

Container(auth\_svc, "Auth Service", "ASP.NET Core", "Handles user login, role, permissions")

Container(request\_svc, "Request Service", ".NET Core API", "Manages creation and updates to request shell")

Container(provider\_svc, "Provider Service", ".NET Core API", "Manages provider selection")

Container(procedure\_svc, "Procedure Service", ".NET Core API", "Handles CPT/HCPCS code input")

Container(review\_svc, "Review Service", ".NET Core API", "Performs nurse/MD review & status update")

Container(notification\_svc, "Notification Service", "Worker Service", "Sends email/SMS & EDI updates")

Container(event\_bus, "Kafka Event Bus", "Kafka", "Publishes and routes domain events")

Container(blob\_storage, "Azure Blob Storage", "Azure", "Stores attachments/clinical documents")

Container(sql\_db, "SQL Database", "Azure SQL", "Stores transactional request data")

Container(redis\_cache, "Redis Cache", "Azure Redis", "Holds payer rules and lookup cache")

}

Rel(agent, web\_ui, "Uses")

Rel(provider, web\_ui, "Uses")

Rel(web\_ui, api\_gateway, "Sends REST calls")

Rel(api\_gateway, auth\_svc, "Validates token, role")

Rel(api\_gateway, request\_svc, "Route: /create-request")

Rel(api\_gateway, provider\_svc, "Route: /add-provider")

Rel(api\_gateway, procedure\_svc, "Route: /add-procedure")

Rel(api\_gateway, review\_svc, "Route: /submit-review")

Rel(request\_svc, sql\_db, "Stores request shell")

Rel(provider\_svc, sql\_db, "Stores provider info")

Rel(procedure\_svc, sql\_db, "Stores CPT/HCPCS info")

Rel(review\_svc, sql\_db, "Updates status")

Rel(review\_svc, event\_bus, "Publishes ReviewCompletedEvent")

Rel(request\_svc, event\_bus, "Publishes RequestCreatedEvent")

Rel(provider\_svc, event\_bus, "Publishes ProvidersAddedEvent")

Rel(procedure\_svc, event\_bus, "Publishes ProceduresAddedEvent")

Rel(event\_bus, notification\_svc, "Routes events")

Rel(notification\_svc, blob\_storage, "Reads attachments")

Rel(notification\_svc, payer, "Sends EDI message")

@enduml

✅ C4 Container Diagram has been added in **PlantUML C4 format**. It includes:

* External actors: Provider, Intake Agent, Payer
* Internal containers: UI, Gateway, Services (Request, Provider, Procedure, Review), Kafka Event Bus, SQL DB, Blob Storage, Redis
* All interactions and responsibilities modeled clearly

You can visualize this with any PlantUML tool that supports [C4-PlantUML](https://github.com/plantuml-stdlib/C4-PlantUML).

Would you like to proceed with:

* The **Component-level C4 model**?
* Implementation of **Notification or EDI microservice**?
* Add OpenTelemetry-based tracing across services?

Let me know!

@startuml C4\_Component

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Component.puml

LAYOUT\_WITH\_LEGEND()

Person(agent, "Intake Agent")

Person(provider, "Provider")

System\_Boundary(pa\_system, "Prior Authorization System") {

Container(web\_ui, "Provider Portal", "React App", "UI for PA request creation and tracking")

Container\_Boundary(request\_api, "Request Service") {

Component(req\_controller, "RequestController", "ASP.NET Controller", "Exposes API endpoints for request creation")

Component(req\_agg, "AuthorizationRequest Aggregate", "Domain Model", "DDD aggregate root with validation & events")

Component(req\_repo, "IAuthorizationRequestRepository", "Interface", "Persists request to database")

Component(req\_uow, "UnitOfWork", "Transaction Manager", "Manages transaction & domain event commit")

Component(outbox\_writer, "OutboxWriter", "Outbox Handler", "Stores domain events for reliable publish")

}

Container(sql\_db, "SQL DB", "Azure SQL")

Container(event\_bus, "Kafka", "Event Bus")

}

Rel(agent, web\_ui, "Creates prior auth request")

Rel(provider, web\_ui, "Submits clinical request")

Rel(web\_ui, req\_controller, "Calls API")

Rel(req\_controller, req\_agg, "Validates + mutates aggregate")

Rel(req\_agg, req\_repo, "Saves via repository")

Rel(req\_repo, sql\_db, "Persists to DB")

Rel(req\_agg, outbox\_writer, "Appends domain events")

Rel(outbox\_writer, sql\_db, "Stores to Outbox table")

Rel(outbox\_writer, event\_bus, "Dispatched asynchronously (by OutboxDispatcher)")

@enduml

✅ Added the **C4 Component Diagram** (PlantUML) for the Request Service, showing:

* Internal components like RequestController, AuthorizationRequest aggregate, Repository, UnitOfWork, and OutboxWriter
* Interaction with external SQL DB, Kafka, and Provider Portal

This gives clarity into how the microservice applies **Domain-Driven Design**, **clean architecture**, and **event publishing**.

Would you like to:

* Create component diagrams for other services (e.g., Provider, Procedure, Review)?
* Proceed with OpenTelemetry tracing integration?
* Add API specification (OpenAPI/Swagger YAML)?

Let me know!

### 🗄️ Database Design – Prior Authorization Intake System

This schema is designed to support a modular, event-driven PA intake system using normalized relational tables.

---

#### 🔹 `AuthorizationRequests`

Stores main request shell.

```sql

CREATE TABLE AuthorizationRequests (

RequestId UNIQUEIDENTIFIER PRIMARY KEY,

PatientId NVARCHAR(100) NOT NULL,

RequesterId NVARCHAR(100),

HealthPlanId NVARCHAR(100),

ProgramId NVARCHAR(100),

Status NVARCHAR(50) DEFAULT 'New',

CreatedAt DATETIME2 DEFAULT SYSDATETIME(),

UpdatedAt DATETIME2

);

```

---

#### 🔹 `Providers`

Stores individual provider records.

```sql

CREATE TABLE Providers (

ProviderId NVARCHAR(100) PRIMARY KEY,

Name NVARCHAR(200),

NPI NVARCHAR(20),

Type NVARCHAR(50), -- e.g., Requesting, Rendering, Ordering

IsParticipating BIT,

IsOutOfNetwork BIT

);

```

---

#### 🔹 `RequestProviders`

Associates providers to a request with their role.

```sql

CREATE TABLE RequestProviders (

Id INT IDENTITY PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

ProviderId NVARCHAR(100) FOREIGN KEY REFERENCES Providers(ProviderId),

Role NVARCHAR(50) -- Requesting, Rendering, Ordering

);

```

---

#### 🔹 `ProcedureCodes`

Static CPT/HCPCS codes.

```sql

CREATE TABLE ProcedureCodes (

Code NVARCHAR(50) PRIMARY KEY,

Description NVARCHAR(200)

);

```

---

#### 🔹 `RequestProcedures`

Join table for requests and procedures.

```sql

CREATE TABLE RequestProcedures (

Id INT IDENTITY PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

ProcedureCode NVARCHAR(50) FOREIGN KEY REFERENCES ProcedureCodes(Code),

IsPrimary BIT DEFAULT 0

);

```

---

#### 🔹 `DiagnosisCodes`

Static ICD-10 code master.

```sql

CREATE TABLE DiagnosisCodes (

Code NVARCHAR(20) PRIMARY KEY,

Description NVARCHAR(200)

);

```

---

#### 🔹 `RequestDiagnoses`

Join table for diagnosis codes and request.

```sql

CREATE TABLE RequestDiagnoses (

Id INT IDENTITY PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

DiagnosisCode NVARCHAR(20) FOREIGN KEY REFERENCES DiagnosisCodes(Code)

);

```

---

#### 🔹 `ClinicalAttachments`

Stores uploaded files.

```sql

CREATE TABLE ClinicalAttachments (

AttachmentId UNIQUEIDENTIFIER PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

FileName NVARCHAR(255),

FileUrl NVARCHAR(500),

UploadedAt DATETIME2 DEFAULT SYSDATETIME()

);

```

---

#### 🔹 `RequestNotes`

Free-form user notes.

```sql

CREATE TABLE RequestNotes (

NoteId UNIQUEIDENTIFIER PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

Note NVARCHAR(MAX),

Author NVARCHAR(100),

CreatedAt DATETIME2 DEFAULT SYSDATETIME()

);

```

---

#### 🔹 `RequestReviews`

Captures nurse/MD decisions.

```sql

CREATE TABLE RequestReviews (

ReviewId UNIQUEIDENTIFIER PRIMARY KEY,

RequestId UNIQUEIDENTIFIER FOREIGN KEY REFERENCES AuthorizationRequests(RequestId),

Reviewer NVARCHAR(100),

Status NVARCHAR(50), -- Approved, Denied, Partial, Withdrawn, Hold

Rationale NVARCHAR(MAX),

ReviewedAt DATETIME2 DEFAULT SYSDATETIME()

);

```

---

#### 🔹 `OutboxMessages`

Durable event delivery table.

```sql

CREATE TABLE OutboxMessages (

Id UNIQUEIDENTIFIER PRIMARY KEY,

Type NVARCHAR(100),

Payload NVARCHAR(MAX),

CreatedAt DATETIME2 DEFAULT SYSDATETIME(),

Dispatched BIT DEFAULT 0

);

```

Would you like me to generate:

- ER diagram for this schema?

- SQL scripts for seeding master data?

- Indexes and partitioning recommendations?

✅ Database schema with full SQL table definitions and detailed explanations has been added. It includes:

* Core domain entities (request, provider, procedure, diagnosis, attachments)
* Association tables and review logic
* Durable event handling (Outbox pattern)

Would you like to generate:

* An ER Diagram from this schema?
* SQL seed scripts for static masters like ProcedureCodes and DiagnosisCodes?
* Advanced topics like partitioning or indexing strategy for high volume scale?

Let me know your next step!

### 🧠 Advanced Database Design Strategies for High-Volume PA Intake System

---

#### 📌 1. \*\*Indexing Strategy\*\*

Add appropriate indexes to improve query performance for filters and joins:

```sql

-- For fast patient-based lookups

CREATE INDEX IX\_Requests\_PatientId ON AuthorizationRequests(PatientId);

-- For filtering by request status

CREATE INDEX IX\_Requests\_Status ON AuthorizationRequests(Status);

-- Join performance on foreign keys

CREATE INDEX IX\_Providers\_ProviderId ON Providers(ProviderId);

CREATE INDEX IX\_RequestProviders\_RequestId ON RequestProviders(RequestId);

CREATE INDEX IX\_RequestProcedures\_RequestId ON RequestProcedures(RequestId);

CREATE INDEX IX\_RequestDiagnoses\_RequestId ON RequestDiagnoses(RequestId);

-- Date filtering for history

CREATE INDEX IX\_Requests\_CreatedAt ON AuthorizationRequests(CreatedAt);

```

---

#### 📌 2. \*\*Partitioning Strategy\*\*

For scalability and better performance over large data volume:

```sql

-- Partition table by month or year based on CreatedAt

-- Useful for AuthorizationRequests and RequestReviews

-- Step 1: Create partition function

CREATE PARTITION FUNCTION pfCreatedDate (DATETIME2)

AS RANGE LEFT FOR VALUES (

'2024-01-01', '2024-02-01', '2024-03-01', '2024-04-01', ...

);

-- Step 2: Create partition scheme

CREATE PARTITION SCHEME psCreatedDate

AS PARTITION pfCreatedDate ALL TO ([PRIMARY]);

-- Step 3: Create table on partition scheme

-- (this must be done when designing the table schema)

```

---

#### 📌 3. \*\*Read Replica and Offloading\*\*

- Use \*\*Azure SQL Read Replicas\*\* to offload reporting queries

- Dashboard and history queries can point to read-only DB

---

#### 📌 4. \*\*Data Archival\*\*

- Move closed/aged-out requests to a historical table

```sql

CREATE TABLE AuthorizationRequests\_Archive (... same schema ...);

-- Schedule daily/weekly archival job for requests older than 1 year

```

---

#### 📌 5. \*\*Table Compression\*\*

- Enable \*\*row/page compression\*\* for static tables like ProcedureCodes, DiagnosisCodes

```sql

ALTER TABLE ProcedureCodes REBUILD PARTITION = ALL WITH (DATA\_COMPRESSION = PAGE);

```

---

#### 📌 6. \*\*Query Optimization Tips\*\*

- Always filter on \*\*indexed columns\*\* (avoid wrapping columns in functions)

- Avoid SELECT \* in high-volume services

- Use \*\*TOP + ORDER BY\*\* + pagination for UI-driven fetches

Would you like help implementing these in a migration script or a tuning checklist?

✅ Advanced database strategies have been added, covering:

* 📌 Indexing for performance
* 📌 Partitioning for large-volume request data
* 📌 Read replicas for offloading analytics
* 📌 Archival strategies for aging data
* 📌 Compression tips for static tables
* 📌 Query optimization practices

Would you like a SQL Server migration script with all these baked in, or should we now generate a tuning/monitoring checklist for Azure SQL?

Great! Below are **React application recommendations** tailored to your **Prior Authorization Intake Portal**, followed by **React + API security practices** with examples.

## ⚛️ React Recommendations for Prior Authorization Intake Portal

### ✅ 1. **Micro-Frontend Architecture**

* Split modules like Patient Search, Provider Lookup, Procedure Entry, Review Dashboard into individual micro frontends.
* Use tools like **Module Federation (Webpack 5)** or **Single SPA** for integration.

### ✅ 2. **Componentization**

* Atomic Design (Atom, Molecule, Organism) for reusability.
  + PatientInfoForm, ProviderCard, DiagnosisCodeSelector, etc.

### ✅ 3. **State Management**

* Use **React Query (TanStack)** for server state and caching.
* Use **Zustand** or **Redux Toolkit** (RTK) for local UI state.

### ✅ 4. **Form Handling & Validation**

* Use **React Hook Form** + **Zod** or **Yup** for schema validation.
  + Async validation for NPI, diagnosis, and procedure codes.

### ✅ 5. **Accessibility**

* Follow WCAG 2.1:
  + Use aria-\* attributes, keyboard navigations, and screen reader support.

### ✅ 6. **Routing**

* Use **React Router v6+** with code-splitting using React.lazy.
  + Protect review routes using route guards.

### ✅ 7. **Performance**

* Lazy load components
* Enable React DevTools profiler
* Debounce patient search inputs

## 🔐 React + API Security Practices

### 🔒 1. **Authentication via Azure AD B2C**

* Use **MSAL.js** (Microsoft Authentication Library) in frontend
* Configure B2C App Registration with redirect URI and scopes

**Example: MSAL Setup**

import { PublicClientApplication } from '@azure/msal-browser';

const msalInstance = new PublicClientApplication({

auth: {

clientId: 'your-client-id',

authority: 'https://yourdomain.b2clogin.com/yourdomain.onmicrosoft.com/B2C\_1\_signin',

redirectUri: 'http://localhost:3000',

}

});

### 🔒 2. **Token Handling**

* Store **access tokens** in **memory** or **HttpOnly cookie** (avoid localStorage).
* Refresh tokens securely using MSAL refresh flow.

### 🔒 3. **API Protection**

* Backend API (ASP.NET Core) should:
  + Validate JWT via Azure AD B2C middleware.
  + Define required scopes in [Authorize] attributes.

**Backend Example (ASP.NET Core)**:

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddMicrosoftIdentityWebApi(Configuration, "AzureAdB2C");

[Authorize(Policy = "ProviderOnly")]

[HttpPost("api/requests")]

public IActionResult CreateRequest([FromBody] RequestDto dto) { ... }

### 🔒 4. **Route Guards in React**

import { useIsAuthenticated } from "@azure/msal-react";

function ProtectedRoute({ children }) {

const isAuthenticated = useIsAuthenticated();

return isAuthenticated ? children : <Navigate to="/login" />;

}

### 🔒 5. **CORS Policy**

* Set CORS policy to allow only your frontend origin in backend.

app.UseCors(builder =>

builder.WithOrigins("https://yourportal.com")

.AllowAnyHeader()

.AllowAnyMethod()

.AllowCredentials());

## 📦 Bonus: Secure File Upload (Clinical Attachments)

* Upload directly to **Azure Blob SAS URL** from frontend.
* Keep blob write access outside your API (prevent bypass risk).

**React Upload to Blob Example**

const uploadToAzure = async (file: File, sasUrl: string) => {

await fetch(sasUrl, {

method: "PUT",

headers: {

"x-ms-blob-type": "BlockBlob",

"Content-Type": file.type,

},

body: file,

});

};

## Would You Like Help With:

* 📄 OpenID Connect flow explanation with React and .NET?
* 📁 Secure file download pattern?
* 🛡️ RBAC role-based UI control for React?

Let me know how deep you'd like to go!

Excellent — let's now extend your **security practices** to cover deeper threats, including **XSS**, **SQL Injection (Time-Based)**, **CSP**, and other advanced topics relevant to a **high-security healthcare React + API platform** like yours.

## 🛡️ Comprehensive Security Practices for React + API (Healthcare App)

### 🔐 **1. Authentication & Authorization Recap**

* ✅ Use **OAuth2.0 / OpenID Connect via Azure AD B2C**
* ✅ Always use **Access Token Validation** in API
* ✅ Enforce **Role-Based Access Control (RBAC)** both in frontend and backend
  + E.g., IntakeAgent, Provider, NurseReviewer

## 🔏 2. **Cross-Site Scripting (XSS)** Protection

### What is it?

XSS allows attackers to inject malicious JS into your app (especially dangerous in healthcare apps).

### React Defense:

* **React is XSS-safe by default** due to its DOM sanitization.
* But be careful when using:
  + dangerouslySetInnerHTML
  + Unescaped values inside attributes

### ✅ Best Practices:

// ❌ BAD (dangerous)

<div dangerouslySetInnerHTML={{ \_\_html: userComment }} />

// ✅ GOOD (escaped by React)

<p>{userComment}</p>

## 💣 3. **SQL Injection (Time-Based and Others)**

Happens if user input is directly injected into DB queries. **Time-based injection** is used to extract data by delaying responses.

### ✅ Defenses in .NET API:

* **Use parameterized queries** in SQL access
* **Avoid string concatenation** in any database command
* If using **EF Core**, it's safe **unless using FromSqlRaw/Interpolated**

// ✅ Parameterized EF query

var result = await \_context.Requests

.Where(r => r.PatientId == patientId)

.ToListAsync();

// ❌ Vulnerable raw SQL

context.Requests.FromSqlRaw($"SELECT \* FROM Requests WHERE Id = '{userInput}'");

## 🎯 4. **Content Security Policy (CSP)**

### What is CSP?

A powerful browser mechanism that **restricts what scripts can run**, stopping many XSS attacks.

### ✅ Add CSP Headers:

In your backend API (ASP.NET Core):

app.Use(async (ctx, next) =>

{

ctx.Response.Headers.Add("Content-Security-Policy",

"default-src 'self'; script-src 'self'; object-src 'none'; base-uri 'self';");

await next();

});

## 🚫 5. **Clickjacking Defense**

Prevent the app from being embedded in another malicious iframe.

### ✅ Use this header:

app.Use(async (ctx, next) =>

{

ctx.Response.Headers.Add("X-Frame-Options", "DENY");

await next();

});

## 🔄 6. **CORS Configuration (Strict)**

Ensure only your frontend origin is allowed.

app.UseCors(builder =>

builder.WithOrigins("https://intake.evicore.com")

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

## 🧪 7. **Rate Limiting & DDoS Defense**

* Use **Azure API Management policies** or **middleware** to rate-limit endpoints.
* Protect login and file upload endpoints more aggressively.

## 🔐 8. **Secure File Upload (Azure Blob)**

* Use **SAS Tokens** to upload directly to blob.
* Validate:
  + MIME type (application/pdf, image/png)
  + File size
  + Filename normalization

## 📉 9. **Audit Logging & Tamper Protection**

* Log:
  + Who created/updated requests
  + Review decisions
  + Attachment access
* Use **immutable audit log tables** or **append-only blob containers**

## 🔎 10. **Security Headers (Full Set)**

Add these to all API responses:

| **Header** | **Purpose** |
| --- | --- |
| Content-Security-Policy | Prevents XSS |
| Strict-Transport-Security | Forces HTTPS |
| X-Frame-Options | Prevents clickjacking |
| X-Content-Type-Options | Prevents MIME sniffing |
| Referrer-Policy | Controls referrer leakage |

## 🧼 11. **Input Validation & Output Encoding**

* ✅ Validate all inputs on both client and server using:
  + zod / yup in React
  + FluentValidation / data annotations in .NET
* ❌ Never trust frontend-only validation.

## 🔐 12. **OWASP ASVS Coverage (Partial)**

| **Control** | **Covered?** |
| --- | --- |
| Input Validation | ✅ Yes |
| Auth (B2C) | ✅ Yes |
| Role Authorization | ✅ Yes |
| Rate Limiting | 🔄 Partial |
| Session Mgmt | ✅ MSAL Managed |
| XSS/CSRF | ✅/🔄 |
| File Upload Defense | ✅ SAS & Type Check |
| SQL Injection | ✅ Param Queries |

### 🔧 BONUS: Tools You Can Use

| **Tool** | **Purpose** |
| --- | --- |
| **SonarQube** | Static code security scanning |
| **OWASP ZAP** | Runtime API scanning |
| **Helmet.js** | Add security headers (for Node, if used in UI) |
| **MSAL.js** | Azure B2C integration |
| **Express-rate-limit** (if Node API) | Rate limiting |

## ✅ Next Steps (Let Me Know Which You Want):

* 🔐 CSP header for React frontend (index.html)
* 🛠️ Example MSAL + Role guard React wrapper
* 📜 Swagger Security Schemes
* 🧱 RBAC at UI component level

Would you like these in a bundle? Or should I generate a **Security Playbook document** for you?

Great! Here's how to implement the **first two** requested items for your healthcare intake portal frontend built with **React + Azure B2C**:

## ✅ 1. Content Security Policy (CSP) in React (index.html)

**React doesn't inject CSP by default**, so you must manually configure it in the public/index.html.

### 🔒 Add this in your <head>:

<meta

http-equiv="Content-Security-Policy"

content="

default-src 'self';

script-src 'self' https://\*.msidentity.com https://\*.b2clogin.com;

style-src 'self' 'unsafe-inline';

img-src 'self' data:;

connect-src 'self' https://your-api.azurewebsites.net https://login.microsoftonline.com;

object-src 'none';

frame-ancestors 'none';

base-uri 'self';

"

/>

### ✅ Explanation:

* default-src 'self': Only load resources from same origin
* script-src: Allow scripts only from self + Azure B2C
* connect-src: Allow API and login endpoints
* img-src: Images from self and inline data URIs
* frame-ancestors: Prevent embedding in iframes (anti-clickjacking)

⚠️ If you’re loading dynamic assets or using analytics (like Google), include their URLs explicitly.

## ✅ 2. Role Guard in React for Azure B2C (with MSAL)

Use **@azure/msal-react** to guard routes by role or authentication state.

### 🔧 Step 1: Wrap your app with MsalProvider

import { MsalProvider } from "@azure/msal-react";

import { PublicClientApplication } from "@azure/msal-browser";

import App from "./App";

const msalInstance = new PublicClientApplication({

auth: {

clientId: "<your-client-id>",

authority: "https://<tenant>.b2clogin.com/<tenant>.onmicrosoft.com/B2C\_1\_signin",

redirectUri: "/"

}

});

const root = createRoot(document.getElementById("root"));

root.render(

<MsalProvider instance={msalInstance}>

<App />

</MsalProvider>

);

### 🔐 Step 2: Create ProtectedRoute with Role Check

import { useIsAuthenticated, useMsal } from "@azure/msal-react";

import { Navigate } from "react-router-dom";

export const ProtectedRoute = ({ children, requiredRole }) => {

const isAuthenticated = useIsAuthenticated();

const { accounts } = useMsal();

const account = accounts[0];

const hasRequiredRole = () => {

const roles = account?.idTokenClaims?.roles || [];

return roles.includes(requiredRole);

};

if (!isAuthenticated) return <Navigate to="/login" />;

if (requiredRole && !hasRequiredRole()) return <Navigate to="/unauthorized" />;

return children;

};

### 🔐 Step 3: Use Route Guard in Routing

<Route

path="/review"

element={

<ProtectedRoute requiredRole="NurseReviewer">

<ReviewDashboard />

</ProtectedRoute>

}

/>

### 📌 Backend (ASP.NET) Reminder:

Make sure you're adding the roles claim to your B2C app registration (in manifest):

"appRoles": [

{

"allowedMemberTypes": [ "User" ],

"displayName": "Nurse Reviewer",

"id": "role-guid-here",

"isEnabled": true,

"value": "NurseReviewer"

}

]

Would you like:

* ✅ Ready-made RoleContext provider for managing roles globally?
* 📋 Swagger/OpenAPI with OAuth2 scopes?
* 📄 Security checklist for frontend deployment?

Let me know — or I can export this entire security config to a **Security Playbook** document.

Perfect! Here's how to implement the **third** item you asked for:

## ✅ 3. **Swagger / OpenAPI Security Scheme for OAuth2 (Azure AD B2C)**

Used to protect and document your **ASP.NET Core API** for frontend apps (like your React intake portal) that authenticate using Azure AD B2C.

### 🔧 Step 1: Define OAuth2 in Swagger Configuration (Program.cs or Startup.cs)

Add this inside services.AddSwaggerGen(...):

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "PA Intake API", Version = "v1" });

// OAuth2 configuration for Azure AD B2C

c.AddSecurityDefinition("oauth2", new OpenApiSecurityScheme

{

Type = SecuritySchemeType.OAuth2,

Flows = new OpenApiOAuthFlows

{

AuthorizationCode = new OpenApiOAuthFlow

{

AuthorizationUrl = new Uri("https://<tenant>.b2clogin.com/<tenant>.onmicrosoft.com/B2C\_1\_signin/oauth2/v2.0/authorize"),

TokenUrl = new Uri("https://<tenant>.b2clogin.com/<tenant>.onmicrosoft.com/B2C\_1\_signin/oauth2/v2.0/token"),

Scopes = new Dictionary<string, string>

{

{ "https://<tenant>.onmicrosoft.com/api/read", "Read Access" },

{ "https://<tenant>.onmicrosoft.com/api/write", "Write Access" }

}

}

}

});

// Apply OAuth2 globally

c.AddSecurityRequirement(new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference { Type = ReferenceType.SecurityScheme, Id = "oauth2" }

},

new[] { "https://<tenant>.onmicrosoft.com/api/read" }

}

});

});

🔁 Replace <tenant> with your actual Azure AD B2C tenant and user flow.

### ✅ Step 2: Enable OAuth2 UI in Swagger UI

In app.UseSwaggerUI(...):

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "PA Intake API v1");

c.OAuthClientId("your-client-id-here");

c.OAuthUsePkce(); // Required for B2C

c.OAuthScopeSeparator(" ");

});

### ✅ Step 3: Protect Your Controllers with Scopes or Roles

[Authorize(Roles = "NurseReviewer")]

[HttpGet("api/review")]

public IActionResult GetPendingReviews() { ... }

Or, using custom policies:

services.AddAuthorization(options =>

{

options.AddPolicy("CanCreateRequest", policy =>

policy.RequireClaim("scp", "https://<tenant>.onmicrosoft.com/api/write"));

});

### 🔐 Bonus: Swagger Login Flow

1. Run your API
2. Open /swagger
3. Click **Authorize** 🔒 button
4. Login via Azure B2C
5. Token is passed automatically with every Swagger request

✅ **Now your Swagger UI is fully secured and aligned with your frontend's Azure B2C login.**

Would you like me to:

* 🔁 Generate a reusable Swagger/OpenAPI YAML file for this?
* 📄 Export a “Security Playbook” combining React, Swagger, API & Auth practices?
* 🧪 Add Postman collection with OAuth2 configuration?

Let me know your next move!