CHAMA THE APP

Online courses assignment

REQUIREMENTS

Case description

- You start working at a company that offers online courses.
- For each of the courses, there is one teacher/lecturer, and for each of the courses there is a maximum number of students that can participate.
- To sign up, students need to supply their name and their age.

REQUIREMENTS

Part 1: API for signing up

- Create an API endpoint with which students can sign up for a course.
- If a course is full, it should not be possible to sign up any more.
- The endpoint's response should indicate whether signing up was successful.

REQUIREMENTS

Part 2: Scaling out

- After few months, the company's courses grow wildly successful, business is booming. There are many courses and millions of sign ups, and your synchronous in-process API which you have created in the Part 1 cannot handle the load any more.
- Create a new endpoint for your API that defers the actual processing to a worker process: signing up is processed asynchronously via a message bus.
- This works as follows. The API puts a command message on a queue, and the message is picked up by the worker process. The worker process tries to sign up the student; it then sends an e-mail to inform the student whether signing up succeeded.
- You can implement "sending an email" with a mock implementation that logs success or failure.

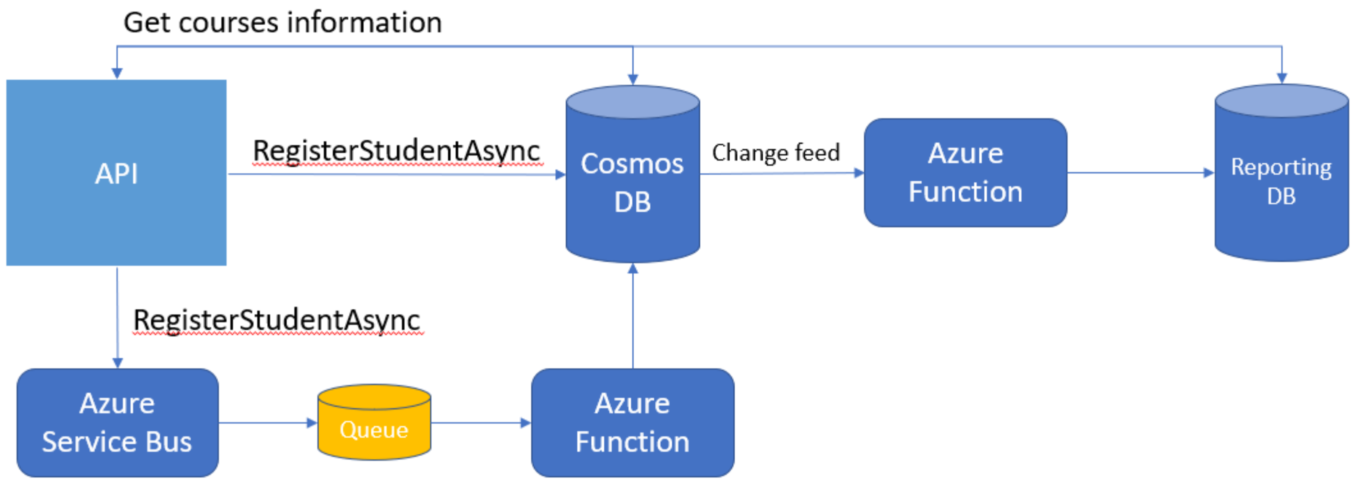
REQUIREMENTS

Part 3: Querying

- For analysis purposes, the company needs to know per course the minimum age, the maximum age and the average age of students that signed up for the courses. Consider that this needs to keep working efficiently when there are millions of sign-ups per day: calculating this information at every request is unfeasible.
- Create two API endpoints:
  - GET list: which returns a list with the above information for each course, plus the course total capacity and current number of students
  - GET details: which returns the above information for a single course, plus the teacher and the list of registered students

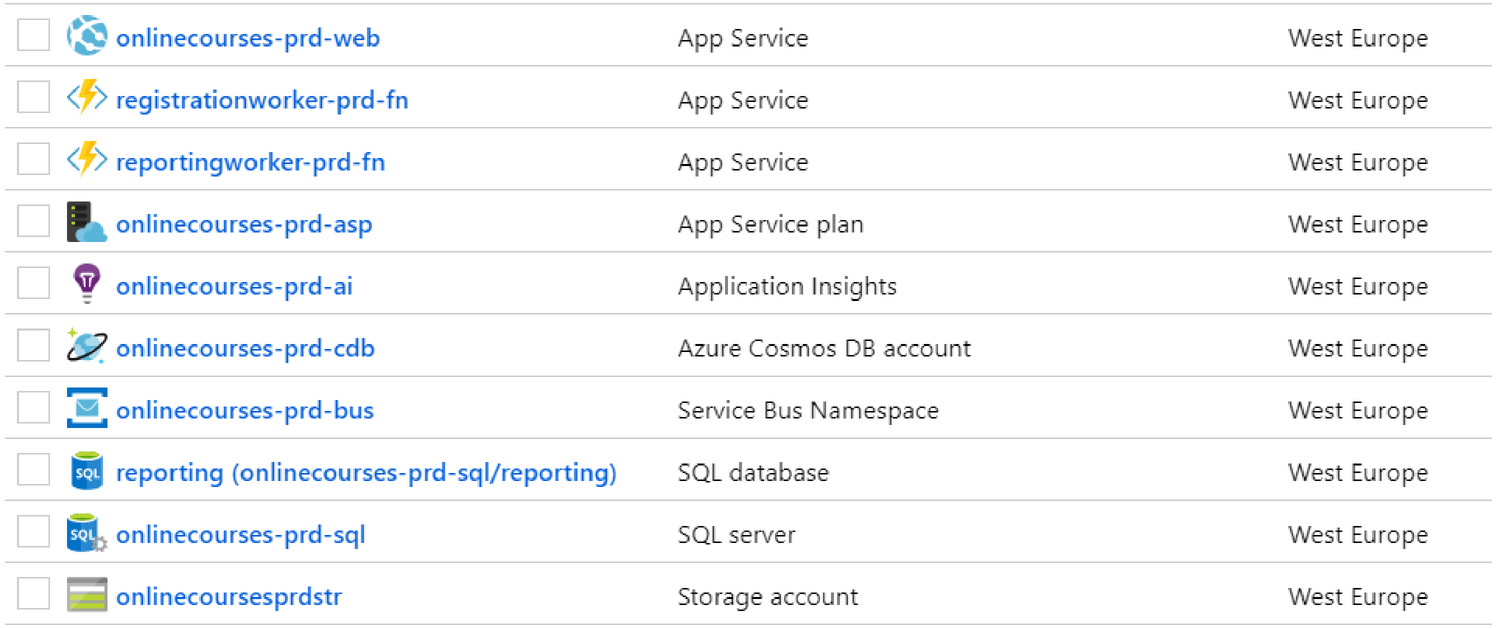
ASSUMPTIONS

- A list of courses is already available in the database
- People of all ages can be registered as students
- The name of a student is split into first and last name
- The system will determine the existence of a student within the course searching by name
- As no email is provided to the system, only the name of the user will be logged when the registration process is completed successfully



ARCHITECTURE COMPONENTS

- Web API
- Cosmos DB
- Azure Service Bus Queue
- Azure Function Apps
- SQL Server
- Application Insights



onlinecourses-prd-web	App Service	West Europe
registrationworker-prd-fn	App Service	West Europe
reportingworker-prd-fn	App Service	West Europe
onlinecourses-prd-asp	App Service plan	West Europe
onlinecourses-prd-ai	Application Insights	West Europe
onlinecourses-prd-cdb	Azure Cosmos DB account	West Europe
onlinecourses-prd-bus	Service Bus Namespace	West Europe
reporting (onlinecourses-prd-sql/reporting)	SQL database	West Europe
onlinecourses-prd-sql	SQL server	West Europe
onlinecoursesprdstr	Storage account	West Europe

RESOURCES

- Resources created as part of the project
- Web API is exposed through the following URL: http://onlinecourses-prd-web.azurewebsites.net/swagger

THE DESIGN DECISIONS

- Although an HTTP Azure Function could have been used instead of a Web API, the scenario looked "too complex" for a single function.
- Cosmos DB was used due to its scaling capabilities and the convenient Change Feed functionality that helped me keeping the analytics detached from the main process
- An Azure Function was used to listen to the Azure Service Bus Queue, again due to their simplicity and scaling capabilities
- An Azure Function was used to listen to the Cosmos Db Change Feed, as the requirements were too simple for a worker or service and the time limitations would not allow me to implement a change feed processor
- Sql server was the database engine of choice for the Reporting materialized view. Due to the simplicity of the data structure an Azure Storage Table could have been an option too, but I don't have many insights about its performance on high load scenarios

THE API

- Built using .NET Core 2.2
- Based on a CQRS pattern, the application has the shape of an RPC API, as the requirements listed in the section “Part 2: Scaling out” made me think REST may not be ideal for the scenario
- The API consists in several projects:
  - **Api**: Contains the commands/queries references, handlers, application level exceptions and controllers.
  - **Api.Models**: Basic implementations for commands, queries and POCOs
  - **Domain**: Contains the business models and exceptions
  - **Infrastructure**: Implementations for the persistence and transport layer (CosmosDb, Sql Server and Azure Service Bus) and repositories
  - **IntegrationEvents**: Used for communication between the API and the Registration function

THE API

- Basic input validation for Commands has been implemented using FluentValidation
- The controllers were shaped in a way they would support versioning, although only V1 of the CourseController is implemented.
- Communication between the controllers and the handlers is made using the Mediator pattern (MediatR)
- Application logging is handled by Application Insights
- Some of the exceptions are handled by the application layer, intercepted and mapped to HttpStatusCodes using GlobalExceptionHandler
- The API exposes a Swagger UI to easily interact with the different end-points

THE REGISTRATION FUNCTION

- An Azure Service Bus queue was created as part of the changes implemented for the requirement described in the section "Part 2: Scaling out"
- In order to listen and react to the events sent out to this new queue, I created a ServiceBusTriggered Azure Function
- This function is responsible for picking up the integration commands sent by the API
- Changes are saved to Cosmos DB using the same domain and infrastructure classes as the API
- Application Insights is used for logging purposes

THE REPORTING FUNCTION

- In order to keep the analytics part of the application detached from the main process, I created a new Azure Function triggered by the Change Feed of the Cosmos DB
- The function is triggered every time a document is created or updated in the courses collection
- The minimum, maximum and average age of the updated course is then calculated
- The results is saved to a materialized view into a SQL database
- Logging is also handled by Application Insights

RETRIEVING COURSE INFORMATION

- As the basic course information and analytics are separated into different sources, the API needs to consolidate the data and return it as a single object
- The handlers behind the GetCourses and GetCourseDetails end-points will be responsible of querying both the Cosmos DB and the reporting materialized view and return the information required as part of the section "Part 3: Querying"

PROJECT CHALLENGES

- One of the main challenges was of course implementing an architecture with so many pieces in just 5 hours
- I used most of the time implementing the foundations in the API and ran out of time towards the end of the assignment, where I had to create the functions and the materialized view
- Cosmos Db is not supporting collections without a partition key anymore. When I realized it, I had to come up with a work-around (hard-coding a partition key in the repository) to avoid having to refactor the repositories and context

IMPROVEMENT POINTS

- As I mentioned before, I had to sacrifice some quality towards the end of the project in order to meet the proposed timeframe
- Implementing security into the API. Azure AD could be used for this purpose
- I only unit tested a few classes created during the first part of the project.
  With some additional time I would have spent more time testing the repositories and context wrappers (using In-Memory DB could have been an option), query handlers, functions and end-points
- When an API has integration to external parties, I usually create a flag in the application configuration file, so testing in isolation is possible.
  As an example, I can create a Postman script that is expecting a 200 OK response from the API, but I'm not interested about the message that is sent to the service bus
- Move the responsibility of logging (or mailing) a Student after the registration process to a mocked class that can easily be replaced using DI instead of logging directly from the Registration Azure Function

IMPROVEMENT POINTS

- More generic and better-defined repositories.
  As I mentioned before, I had to work-around a small issue I had with the Cosmos DB that made me rethink the repositories I created.
  With some more time, I would have had the possibility to change the generic repositories so it can accept an entity's property to be used as partition key
- Better defined partition keys in Cosmos Db.
  Related to my previous point, choosing the right partition key in Cosmos DB is not only important for performance and integrity reasons, but it also defines the order the Change Feed events are triggered.
  This point is not critical in this scenario, but it could affect the data accuracy in some cases
- Dependency Injection in Azure Functions is a bit more time-expensive and I couldn't implement it into the Registration function

IMPROVEMENT POINTS

- Creating NuGet packages from the most commonly used pieces of functionality like abstractions, generic repositories and integration events
- Creating a CI/CD pipeline using Azure DevOps
- Test the API responses using an application like Postman that can also be integrated as part of a CI/CD pipeline
- Creating an ARM script to deploy the application's infrastructure to Azure
- Using Azure KeyVault to store credentials instead of having them hard-coded into the configuration file
- Using Entity Framework migrations to create the SQL database structure used for reporting
- General improvements to error handling and explicit logging