Azure Serverless Ticketly Project

Ticketly: Building a Serverless Event Platform on Azure

- Serverless Web Apps
- Azure Functions
- Queue Storage
- Service Bus
- Cosmos DB
- Observability with Application Insights

What is Serverless?

- Serverless is about **abstraction of infrastructure**:
- Azure provisions, scales, and maintains servers.
- **Execution model:** event-driven, short-lived processes.
- **Billing:** pay per execution + resource consumption (CPU/Memory).
- **Automatic scaling:** thousands of concurrent requests supported.
- **Cold start vs. warm execution** — Functions may need warm-up strategies.

Azure Serverless Building Blocks

1. **Azure Static Web Apps**
2. Globally distributed frontend hosting.
3. Built-in auth providers (AAD, GitHub, Google, etc.).
4. Auto-provisioned Functions API integration.
5. **Azure Functions**
6. Triggers: HTTP, Timer, Queue, Service Bus, Event Grid, etc.
7. Bindings: input/output shortcuts for DBs, storage, messaging.
8. **Azure Storage Queues**
9. Lightweight async message broker.
10. Max 64KB message size.
11. Used for jobs not requiring complex workflows.

12. **Azure Service Bus**
13. Enterprise-grade messaging.
14. Topics, Subscriptions, Dead-letter queues.
15. FIFO with sessions, duplicate detection.
16. **Cosmos DB**
17. NoSQL DB with multi-region writes.
18. Millisecond latency, tunable consistency.
19. Serverless capacity or provisioned throughput.

Ticketly Architecture Overview

```
User (browser)
  ↓
Azure Static Web Apps (React frontend + auth)
  ↓
Azure Functions (serverless API)
  ├── Cosmos DB (Events, Orders)
  ├── Azure Queue Storage (image-jobs)
  │       └── thumbnailWorker (Function)
  └── Azure Service Bus Topic (orders)
          ├── orderProcessor (Function)
          └── paymentCapture (Function)
```

- **CQRS style:** Reads from Cosmos DB, writes via Commands (Functions + SB).
- **Decoupling:** Async queues prevent frontend blocking.

Azure Functions

- **Execution context:** ephemeral, stateless; use Cosmos/Storage for state.
- **Triggers used in Ticketly:**
- HTTP → Expose REST APIs.
- Queue Trigger → background image tasks.
- Service Bus Trigger → order workflow.
- **Bindings example (createOrder):**

```
{
  "type": "cosmosDB",
  "direction": "out",
  "name": "newOrder",
  "databaseName": "ticketly",
```

```
  "containerName": "orders",
  "connectionStringSetting": "COSMOS_CONN"
}
```

- **host.json** controls retries, concurrency, batching.

Queue Storage

- Designed for **simple message handling**:
- Max message size: 64KB.
- Throughput limited but cost-efficient.
- Operations: enqueue, dequeue, peek.
- Visibility timeout prevents multiple consumers from processing simultaneously.
- Poison queue: messages exceeding `maxDequeueCount` move to `<queue>-poison`.
- In Ticketly:
- `thumbnailWorker` processes image jobs asynchronously.
- Ensures UI not blocked during uploads.

Service Bus

- **Advanced messaging scenarios**:
- Supports **topics** with multiple subscriptions.
- Guarantees **at-least-once** delivery.
- Enables FIFO with **sessions**.
- DLQ (dead-letter queue) for unprocessable messages.
- Configurable **retry policies** (exponential backoff).
- In Ticketly:
- `createOrder` publishes to topic `orders`.
- `orderProcessor` validates inventory.
- `paymentCapture` handles payment workflow.
- Failed payments moved to DLQ for manual re-drive.

Cosmos DB

- **NoSQL containers** with partitioning.
- Partition key: `/id` — chosen here for simplicity so every document can be looked up directly by its unique id. This avoids dealing with partitioning logic in a workshop. **However, in production this is not ideal** because `/id` usually leads to poor partition distribution: all queries that filter by user, event, or status will scatter across partitions. Better keys (e.g. `/userId` for orders or `/eventId` for tickets) provide higher cardinality, even data distribution, and scalable query performance.
- **Consistency levels:** Strong, Bounded Staleness, Session (default), Eventual.
- Functions integration:
- Input binding → auto-fetch doc by `{id}`.
- Output binding → upsert doc back into container.
- In Ticketly:

- `getEvents` queries `events` container.
- `createOrder` inserts into `orders`.
- `orderProcessor` and `paymentCapture` update order status.

Observability with Application Insights

- **Telemetry captured:**
- Request rates, latency, error rates.
- Dependency calls (Cosmos, SB, Queues).
- Custom events: e.g., `OrderValidated`, `OrderCompleted`.
- **KQL Queries (Kusto Query Language):** KQL is the query language used in Azure Monitor, Application Insights, and Log Analytics. It is designed for fast, read-only queries that explore telemetry data. With KQL you can filter, transform, and aggregate logs to answer questions like 'how many failed Function executions happened yesterday?' or 'what is the distribution of order statuses?'. Here is an example:

```
traces
| where customDimensions.OrderId != ''
| summarize count() by customDimensions.Status
```

- **Dashboards:** visualize queue depth, SB DLQ size.
- Alerts: queue > 100 messages, DLQ > 0, error rate > 5%.

Slide 10 — Benefits of Serverless on Azure

- **Dev speed:** minimal boilerplate; bindings reduce glue code.
- **Scaling:** from 0 to thousands of concurrent executions.
- **Reliability:** retries, poison queues, DLQs.
- **Security:** identity via Static Web Apps + Functions auth.
- **Cost efficiency:** pay for consumption; ideal for spiky workloads.

Demo Flow (Technical)

1. User browses events (GET `/api/events` → Cosmos DB).
2. Creates an order (POST `/api/orders` → Cosmos insert + SB publish).
3. `orderProcessor` updates order status to Validated.
4. `paymentCapture` completes payment → status Completed.
5. Failed payments → DLQ.
6. Image upload triggers queue → `thumbnailWorker` generates thumbnails.

Wrap Up

Concepts Learned:

- **Serverless Apps**: Static Web Apps + Functions.
- **Async Messaging**: Queue Storage for lightweight jobs.
- **Workflows**: Service Bus for robust order/payment pipelines.
- **Persistence**: Cosmos DB bindings simplify CRUD.
- **Monitoring**: App Insights for tracing & dashboards.

Next Steps:

- Add **Durable Functions** for stateful orchestrations.
- Use **Event Grid** for event-driven fan-out.
- Secure config in **Azure Key Vault**.
- Automate infra with **Bicep/Terraform**.
- Implement CI/CD via **GitHub Actions + Static Web Apps**.