Migrating Java apps to the cloud

After you complete this unit, you should understand:

- What are the different levels of application migration?
- What types of apps should be migrated?
- How do you make Java apps cloud-ready?
- What cloud services should you choose while renovating?

Migration types

- Three entirely different levels of migration of applications to cloud: all three applicable under different circumstances.
- Application re-platforming
  - Do you want user experience to remain same?
  - Are you comfortable with quality of existing code base?
  - Are the equivalents for your dependencies in Bluemix?
  - If so: No code changes; some repackaging might be required; create new deployment automation (for example, manifest.yml) for Bluemix.
- Application modernization
  - Do you have dependencies (middleware, libraries) that are out of support?
  - Do you need to move to a more modern level of your platform?
  - Do you have connections into existing systems of record?
  - If so: Slight code changes required; at least retesting of new subsystems and end-to-end testing to ensure new code introduces no new bugs.
- Application re-envisioning
  - Do you need a new user experience? Are you trying to embrace multiplatform development?
  - Do you want to repackage your application as microservices?
  - If so: effectively a brand new development activity. You might be able to harvest some old code.

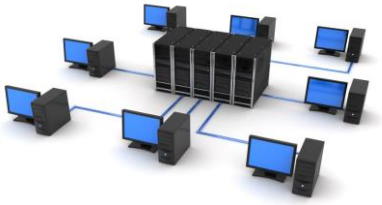
Getting ready to migrate

Common characteristics of enterprise applications

- Not built with cloud compatibility in mind
  - Trying to catch up to latest versions of libraries, etc.
- Might be built on WebSphere Stack products
  - Portals with WebSphere Portal Server
  - Departmental ESBs built with IBM DataPower and IBM Integration Bus
- Deep ties into internal IT
  - Update systems of record
  - Update multiple systems not specified as APIs
  - Deep ties into corporate security systems (LDAP, ISAM, Siteminder) and standards
- High and well-specified quality of service (QoS) attributes
  - Expectations for how often they are available and how they perform

You must consider all of the business, architectural, and design decisions of an application that you move to the cloud.

Differing assumptions



- Enterprise app assumptions
  - Infrastructure is stable.
  - Components of my application are co-located.
  - Operations team controls production servers.
  - If a disaster happens, it's someone's responsibility.

- Cloud-centric app assumptions
  - Infrastructure is constantly changing (elastic).
  - Components of my application can be globally distributed.
  - As a DevOps team member, I control production servers.
  - If a disaster happens, it's my responsibility to fix it.

Planning for migration

- Determine business drivers for your applications.
  - Are you already rewriting them for other reasons? If so, now's a great time to jump into the cloud.
  - Does migrating to cloud make business sense?
- Classify your applications.
  - How cloud-compatible are they?
  - How valuable are they to you?
  - How much work does the existing version create for your team?
- Build a migration approach and plan.
  - Develop a template architecture for your applications.
  - Use opportunity to pay technical debt.
  - Promote decoupling principles as you migrate.

Make applications cloud-ready

Moving to the cloud

- **Cloud-ready:** Apps can be deployed into public or private clouds.
  - Some architectural decisions can make apps unprepared for the cloud.

- **Cloud-centric:** Apps written to run on the cloud.
  - Built using tools and runtimes that are different from traditional apps.
    - For example, instead of relational database, use a NoSQL database.
  - Different infrastructure dependencies.

- **Decision on whether you need an IaaS or a PaaS depends on the application.**

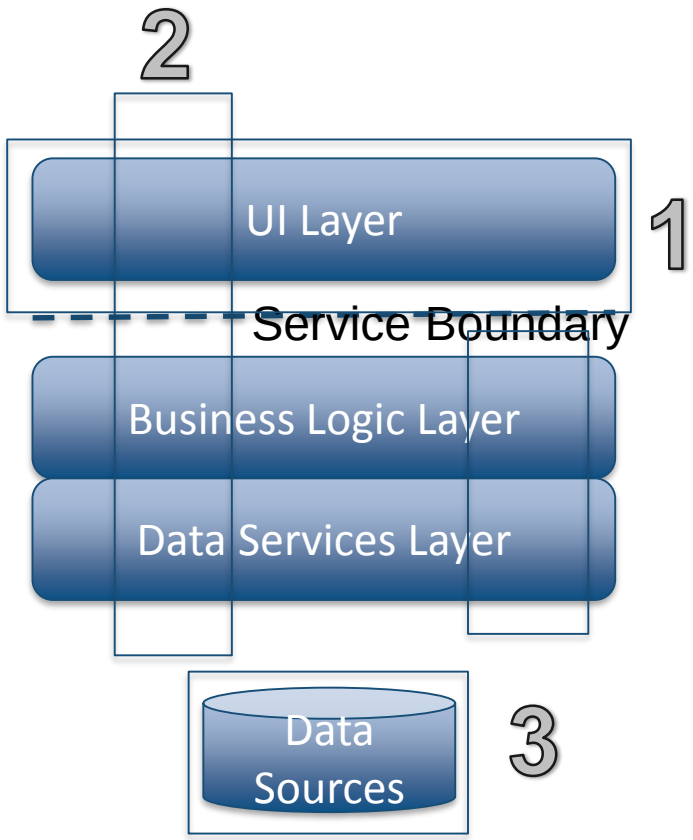
Criteria for cloud-ready applications

The application must:

- Have a design that is topology-agnostic.
  - No specific cluster size needed
- Be managed in system that is infrastructure-agnostic
  - Doesn’t depend on IP addresses, host names, or VLANs.
- Not use infrastructure-specific APIs.
- Not use operating system-specific features.
- Not use local file system.
- Send logs to persistent storage, not file system.
- Keep the session state only in the interaction layer.
- Use standard protocols to connect.
- Have installation and configuration that is scripted.
  - Deployment is easily repeatable.

Changing your applications

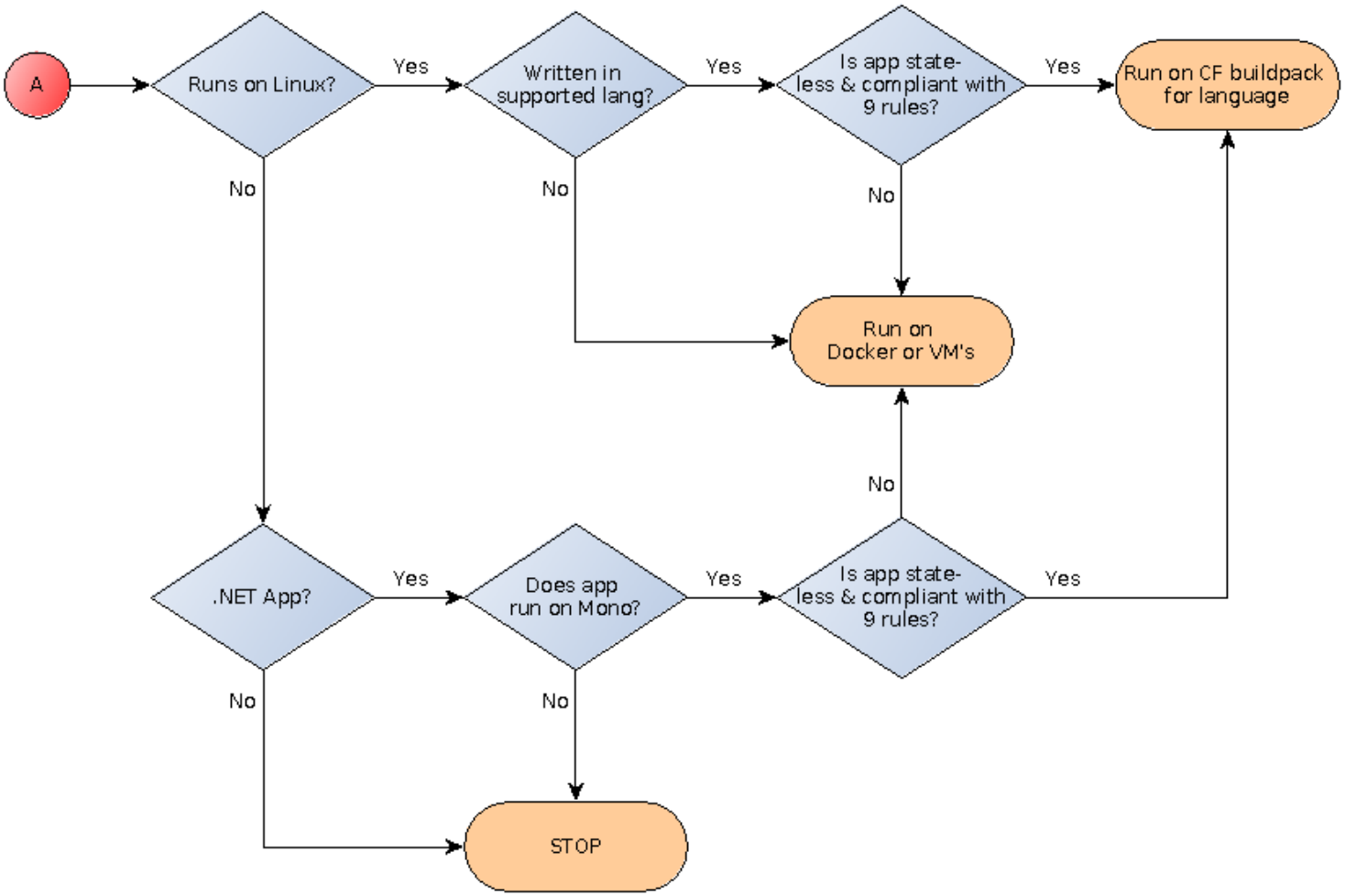
- The app must change in a cloud migration.
- You need a good set of:
  - App tests (preferably automated)
  - App performance baselines
  - Process to access test data
- Application refactoring
  - Break down applications by layer
  - Break down applications along functional boundaries
  - Break larger apps into smaller ones
  - Break out services
    - Develop a shared services architecture



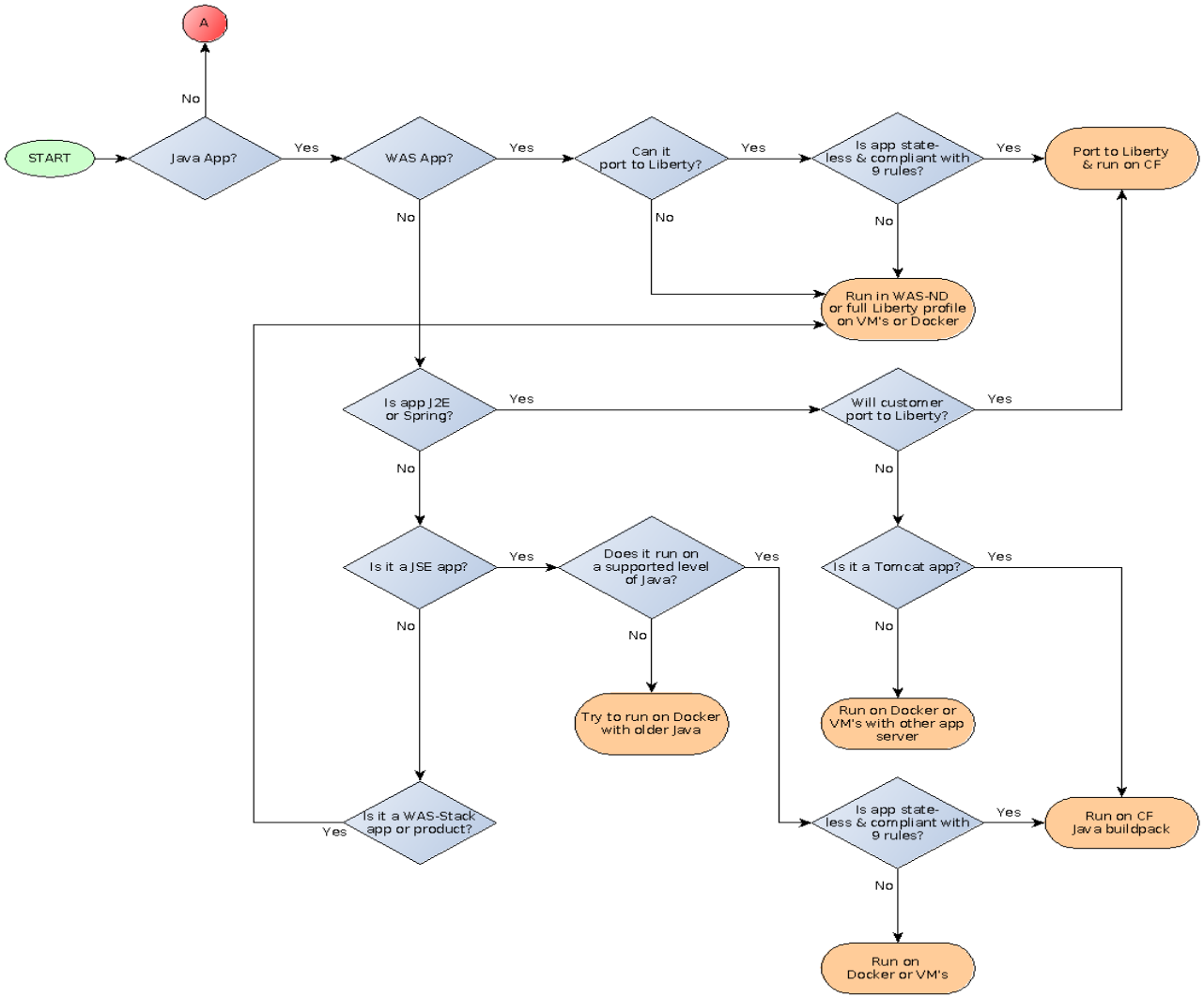
Paying technical debt

- Fully automated functional and performance test suite
- Fully automated code and configuration deployment scheme
- Clean up unneeded internal dependencies
- Refactor to classical SOA or **microservices**
- Modernize your services infrastructure
  - Add caching, monitoring, and so on
- Modernize your user interface
  - Update framework versions
  - Move to Web 2.0/3.0 hybrid model or native mobile apps

Simplified migration flowchart



More complete migration options



Rebuilding apps for cloud-centricity

Design principles for Bluemix applications

- Bluemix applications are *born on the cloud*
  - Built by using **microservices** architectures
  - Follow **DevOps** principles
  - Built in accordance with **12-factor** rules

- Born-on-the-cloud applications
  - Have shorter lifetime: new versions roll out very quickly
  - Built by using principles of **polyglot programming** and **polyglot persistence**

An analogy for constructing Bluemix applications

- Anyone who is familiar with cooking knows that an experienced cook doesn't follow a recipe exactly—they have in mind the outline of the recipe, and they vary what goes in to the final dish based on what's available and what ingredients work well together
- For instance noodle soup will always have noodles and broth. But you can you can add many other ingredients based on what meat or vegetables you have and what ingredients work together with each other.
- Pizza is another good example. In Europe, they often put an egg on the pizza before it goes into the oven. In Japan, squid and sweet corn are favorite toppings. Every pizza has a crust and sauce and usually cheese, but the toppings vary by region and preference.

How do you get started?

- Most Bluemix applications fit roughly into three different **recipes**. The recipes are, not surprisingly, drawn from the different types of applications that we see together:
  - Systems of Record
  - Systems of Interaction
  - Systems of Insight
- Within each recipe, you see a repeating set of compositions that show how to combine runtimes with “Core Services.”
- Note that within this formulation, [brackets] indicates 0-N of these might be needed:
  - System of Interaction Recipe
    - [Gateway] + Runtime + Data Store + [Data Movement]
  - System of Insight Recipe
    - [Visualization] + Runtime + Data Store + [Analysis Tool]
  - System of Record Recipe
    - [Gateway] + Runtime + Data Store
- There are also two different layers of “Ecosystem Services” that might be added to an application to help it fulfill its functional and nonfunctional requirements.

Core services categorization

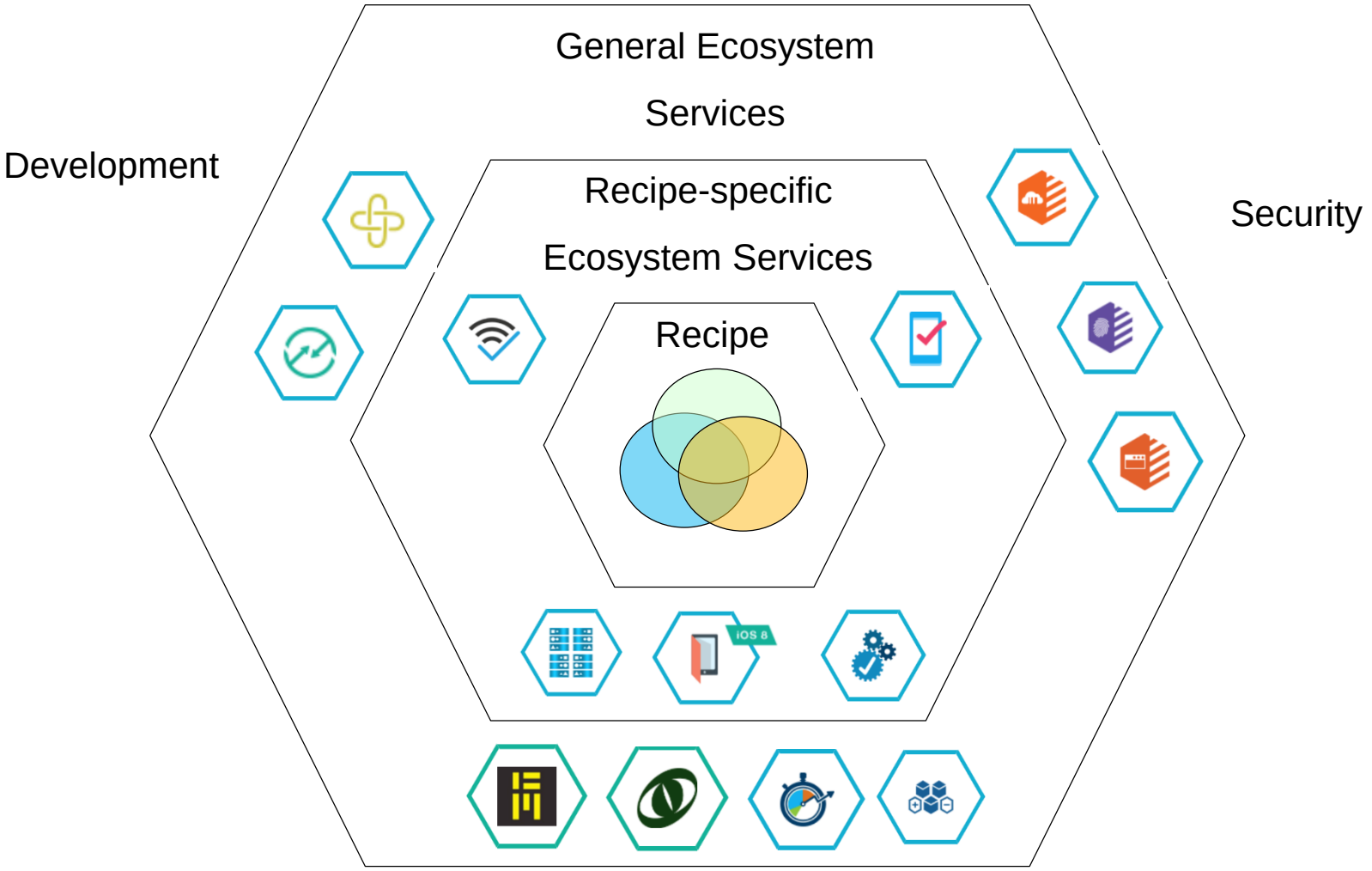
Data Storage	Gateway	Visualization	Analysis	Data Movement, Augmentation, and Validation
SQL Options	IOT	Embeddable Reporting	Analytics for Hadoop	DataWorks
NoSQL Options	Twilio	Cognitive Graph	Apache Spark	Pitney Bowes Geocoding
	Secure Gateway	IOT Realtime Insights	BigInsights for Hadoop	Watson Natural Language Translation
	API Management	Document Generation	DashDB	
	SendGrid		Geospacial Analytics	
			Streaming Analytics	
			Predictive Modeling	

Recipe subtypes

Recipes often come in specific subtypes that help you decide both what core services and recipe-specific ecosystem services you need:

- Systems of Interaction are divided into *Web Systems of Interaction* and *Mobile Systems of Interaction*
  - A Web System of Interaction might use recipe-specific ecosystem services such as **Session Caching** and **Single-Sign On.**
  - A Mobile System of Interaction might use recipe-specific services such as **Push Notification**, **Mobile Quality Assurance,** and **Mobile Security.**
- We also often see *Internet of Things Systems of Record* that send and receive messages through the **IOT Gateway**.
- Systems of Insight are often divided into *Historical Systems of Insight* and *Realtime Systems of Insight.*
  - Historical Systems of Insight can use techniques such as table pivoting available in **DashDB***.*
  - Realtime Systems of Insight can use techniques such as **Streaming Analytics** or **Apache Spark.**

Layers of a Bluemix application



What do the pieces mean?

- A recipe provides some hints about how to construct a business application out of a set of runtimes and services.
  - They're not *boilerplates* because that implies a specific combination of specific services. A boilerplate is more like a frozen pizza or a packaged cup of Ramen noodles.
  - For instance, a System of Record usually has a database—but which database? That depends on your requirements.
- Recipe-specific ecosystem services provide features that are helpful for one recipe or another.
  - If you have a Web System of Interaction Recipe, then you might need **Session Caching.**
  - If you have a Mobile System of Interaction Recipe, then **Mobile Quality Assurance** might be useful.
- Ecosystem services provide features that are generally helpful for a software development lifecycle stage, either development, testing, or production.
  - If you're building a web application, you'll want to performance test that application: use either **BlazeMeter** or **LoadImpact.**
  - But you can't see the how that impacts your application unless you have monitoring: use either **NewRelic** or **Monitoring and Analytics.**
  - If you suspect you need scaling, then **Auto-scaling** can provide that, but you'll need to prove that your thresholds are the right ones through performance testing….

Application architecture (or re-architecture) template

- What recipe does your application fit?
  - Start looking at services based on the formulas stated previously.
  - If it is a System of Interaction, is it of the Mobile Subtype?
  - The Mobile Subtype System of Interaction brings in Mobile-Specific Core services to implement functional requirements

- The next several steps happen in an iterative way: you can add only the outer services layers in later iterations as you need them.
- Step 1: Build out your recipe core.
  - Experiment and determine what the right runtimes and services are for your core.
- Step 2: Does your application need recipe ecosystem services to implement recipe-specific functionality or meet NFR's of your recipe core?
  - If so, add recipe-specific ecosystem services.
- Step 3: Does your application need ecosystem services to fulfill the NFR's of your solution in the areas of maintainability, performance or reliability, or security?
  - If so, add in one or more general ecosystem services such as one or more of the DevOps Services or the security services.

Choosing core services

Polyglot persistence

- Your application will probably need to store some data persistently, for example, for longer than the length of a single user session.
- At one time, it was a simple assumption that web programs should run on a relational database.
  - Multiple relational database options exist for Bluemix.
- However, the assumption of using a SQL database (DB2, MySQL, etc.) should be challenged by the set of open-source data options collectively called *NoSQL.*
  - These data stores are characterized into four different basic types that are each specifically qualified for different purposes

NoSQL options

- **Cloudant**: High performance JSON document database built on Apache CouchDB. Built in support for map/reduce. Useful in nearly any NoSQL situation.
- **Redis by Compose**: BSD-licensed key-value store. Especially useful for leaderboards and document ranking.
- **MongoDB by Compose**: Open-source document database (documents similar to JSON objects). Often used for log data, product data management, and content management.
- **Redis Cloud and MongoLab**: Provide third-party Redis and Mongo hosting services.
- **Graph Data Store**: Built on Apache TinkerPop and is especially useful for modeling social networks.

SQL database options

The SQL options for storing data vary by functionality and the amount of scaling and multitenancy that they support.

- The SQL Database service provides a DB2 Database (not a DB2 instance) that is unique to a space and suitable for low-volume applications.
- The DB2 on Cloud Service provides a DB2 instance that is unique to a space and can scale up to enterprise sizes.

Both DB2 options are fully compatible with existing DB2 drivers and SQL syntax.

- PostgreSQL on Compose and Elephant SQL (third party) provide implementations of Postgres.
- ClearDB and the (experimental) MySQL Service provide implementations of MySQL.

In the end, the decision is usually based on what database the team (and their DBAs) are most comfortable with.

Storing data temporarily

For performance reasons (or for scalability of session-type data), you might need to store data for the duration of a user session.

There's also the need to cache commonly accessed data rather than pull it from a database, especially a relational database.

- The Session Cache service is specific to the Liberty buildpack and implements the JEE Session API.
- The Data Cache service implements a highly available key-value cache with APIs for Java, Node.js, and REST.
- The Memcached service hosts a distributed key-value cache by using the widely available Memcached APIs.

These should not be used as systems of record—they are explicitly unreliable and might have any of the standard problems (for example, staleness) that distributed caches have. A cache is not a database!

Communicating between services asynchronously

Sometimes, you need to add asynchronicity between stages of a business application. For example, commonly a website will need immediate response whereas processing an order could be an operation that takes hours or days.

Asynchronous messaging systems allow different services to run at different speeds

- MQ Light is a fully-featured AMQP-based messaging engine with clients for Java, Node.js, Ruby, and Python.
- CloudAMQP hosts the open source RabbitMQ AMQP-based messaging engine.
- There is also an experimental RabbitMQ service available from Bluemix Labs.

None of these solutions are capable of being an enterprise messaging solution or tying to those. You still need to use MQ Series and the Secure Gateway for that.

Communicate with the world outside Bluemix: gateways

A gateway allows you to bridge between code running inside a Bluemix runtime and people and services that exist outside the Bluemix environment.

- Twilio allows you to communicate via voice and SMS
- SendGrid allows you to communicate via email.
- IOT allows you to send and receive MQTT messages through the Internet Of Things Foundation.
- API management allows you to create Bluemix services from existing APIs in your enterprise and to manage those APIs.
- Secure Gateway allows you to connect to resources behind your corporate firewall.

While Twilio and SendGrid normally act as *data sinks,* the Secure Gateway might support limited cases of bidirectional communication.

Validating, cleansing, and converting data

- The Pitney-Bowes Geocoding and Reverse Geocoding services map GPS coordinates to postal addresses and vice versa.
- With DataWorks, you can move data from one database to another and also validate postal addresses.
- Watson Language Translation converts text between several natural languages.
- Presence Insights helps you geo-locate a user's device in a venue to provide that user with location-specific information.
- Geospatial analytics gives you MQTT notification when a device moves in a bounded area.
- Think of these as pipes or filters in a data pathway. Either you're moving data from one place to another, or you're changing, converting, or augmenting that data.

Recap

- In this session, you learned:
  - The different types of application migration
  - How to make Java apps cloud-ready
  - What apps you should and should not migrate
  - The cloud services you should choose while renovating your apps