Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

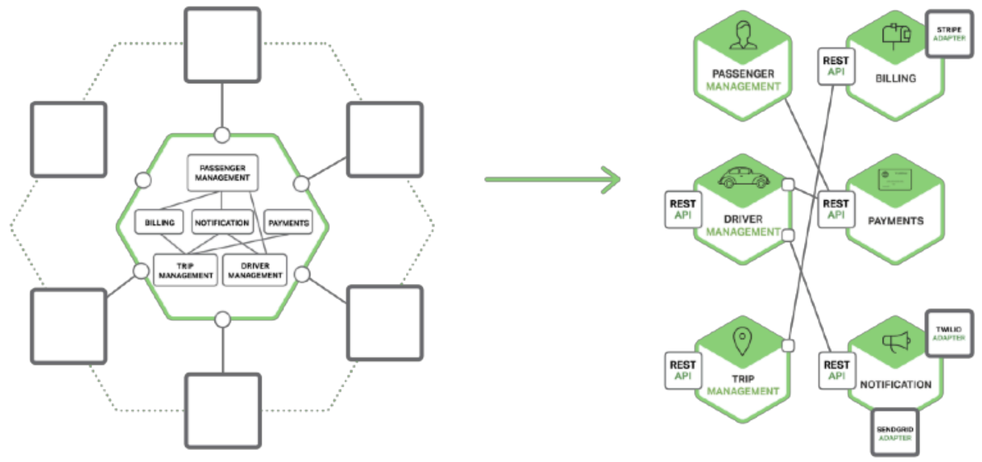
Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/IF1007/if1007

Licença do material

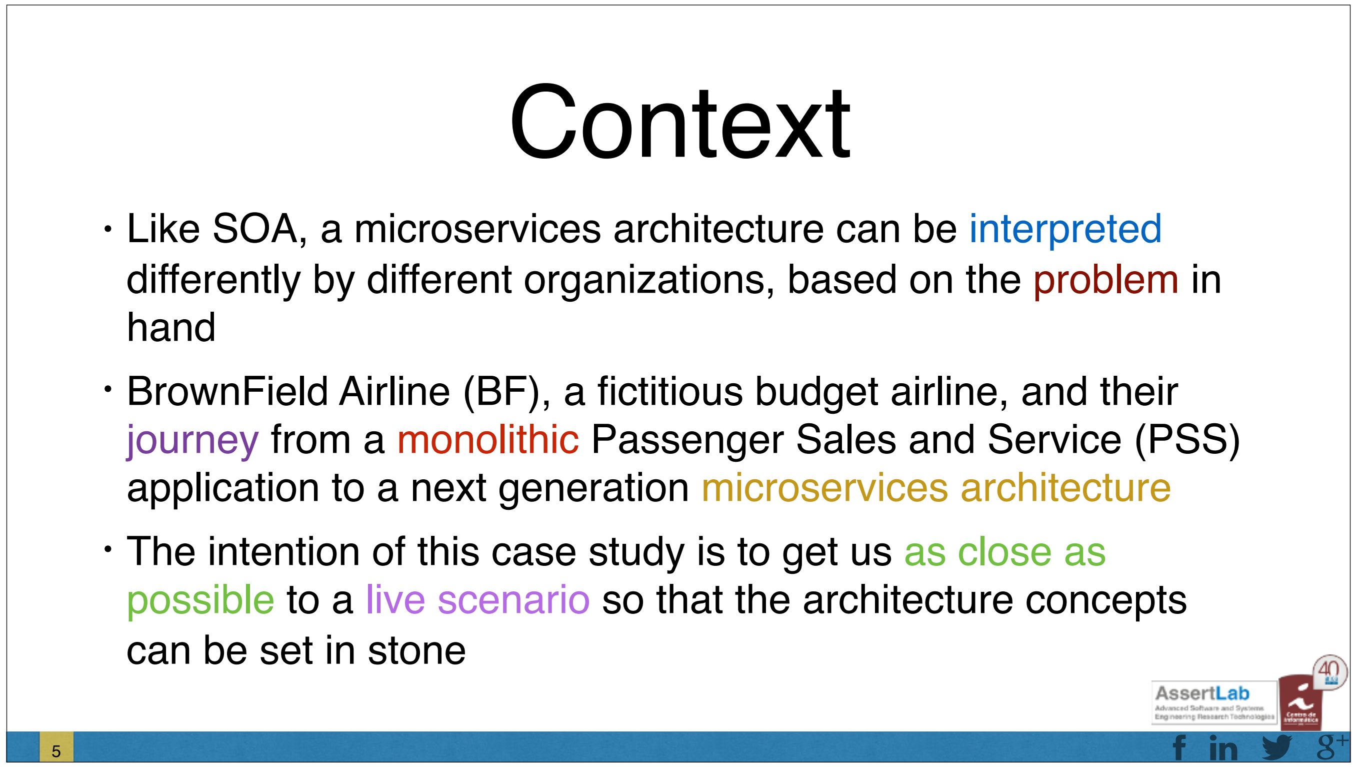
Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Building Microservices: Designing Fine-Grained Systems
  - Spring Microservices
  - Spring Boot: Acelere o desenvolvimento de microsserviços
  - Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers
  - Migrating to Cloud-Native Application Architectures
  - Continuous Integration
  - Getting started guides from spring.io



Microservices Evolution – A Case Study

AssertLab
Advanced Software and Systems Engineering Research Technologies



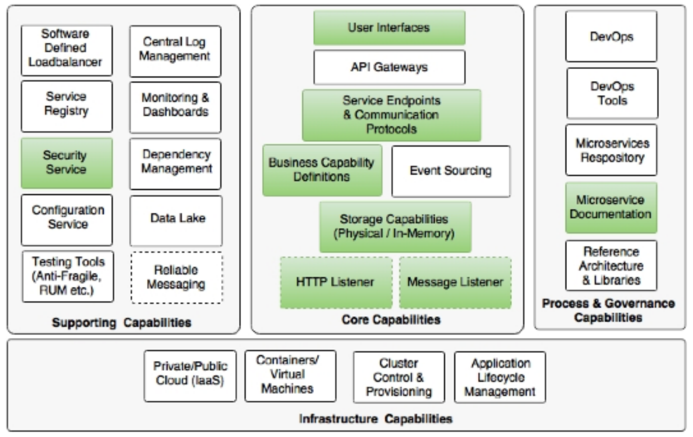
Context

- Like SOA, a microservices architecture can be interpreted differently by different organizations, based on the problem in hand
- BrownField Airline (BF), a fictitious budget airline, and their journey from a monolithic Passenger Sales and Service (PSS) application to a next generation microservices architecture
- The intention of this case study is to get us as close as possible to a live scenario so that the architecture concepts can be set in stone

This chapter examines the PSS application in detail, and explains the challenges, approach, and transformation steps of a monolithic system to a microservices-based architecture, adhering to the principles and practices that were explained in the previous chapter.

Reviewing the microservices capability model

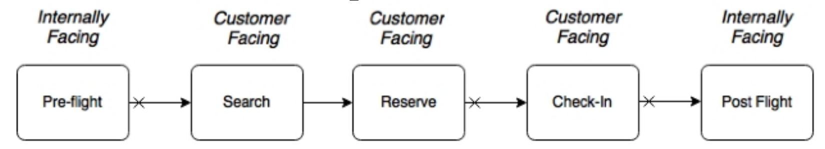
- The examples in this lecture explore the following microservices capabilities from the microservices capability model discussed in last lecture
  - HTTP Listener
  - Message Listener
  - Storage Capabilities (Physical/In-Memory)
  - Business Capability Definitions
  - Service Endpoints & Communication Protocols
  - User Interfaces
  - Security Service
  - Microservice Documentation


Software Defined Loadbalancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

Understanding the PSS application

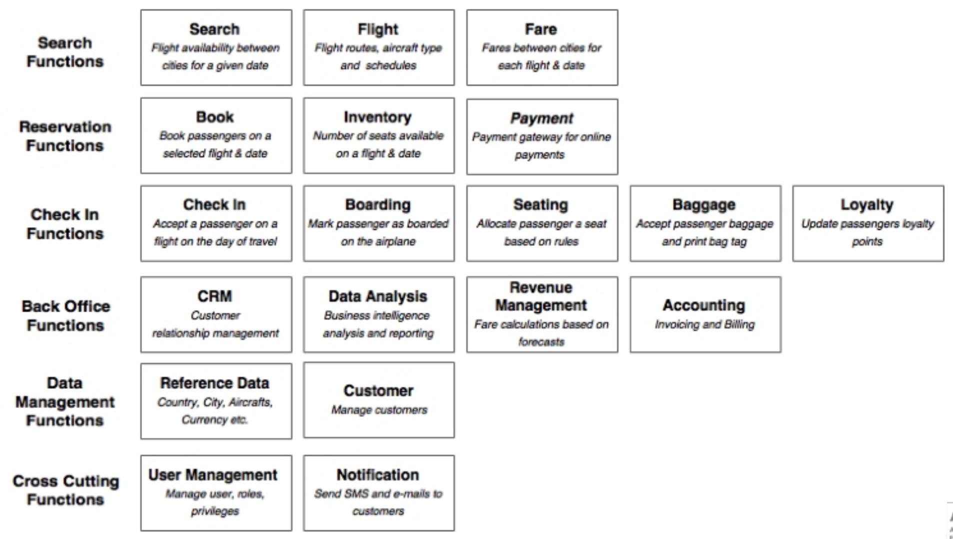
- BrownField Airline is one of the fastest growing low-cost, regional airlines, flying directly to more than 100 destinations from its hub
- As a start-up airline, BrownField Airline started its operations with few destinations and few aircrafts
- BrownField developed its home-grown PSS application to handle their passenger sales and services

Business process view

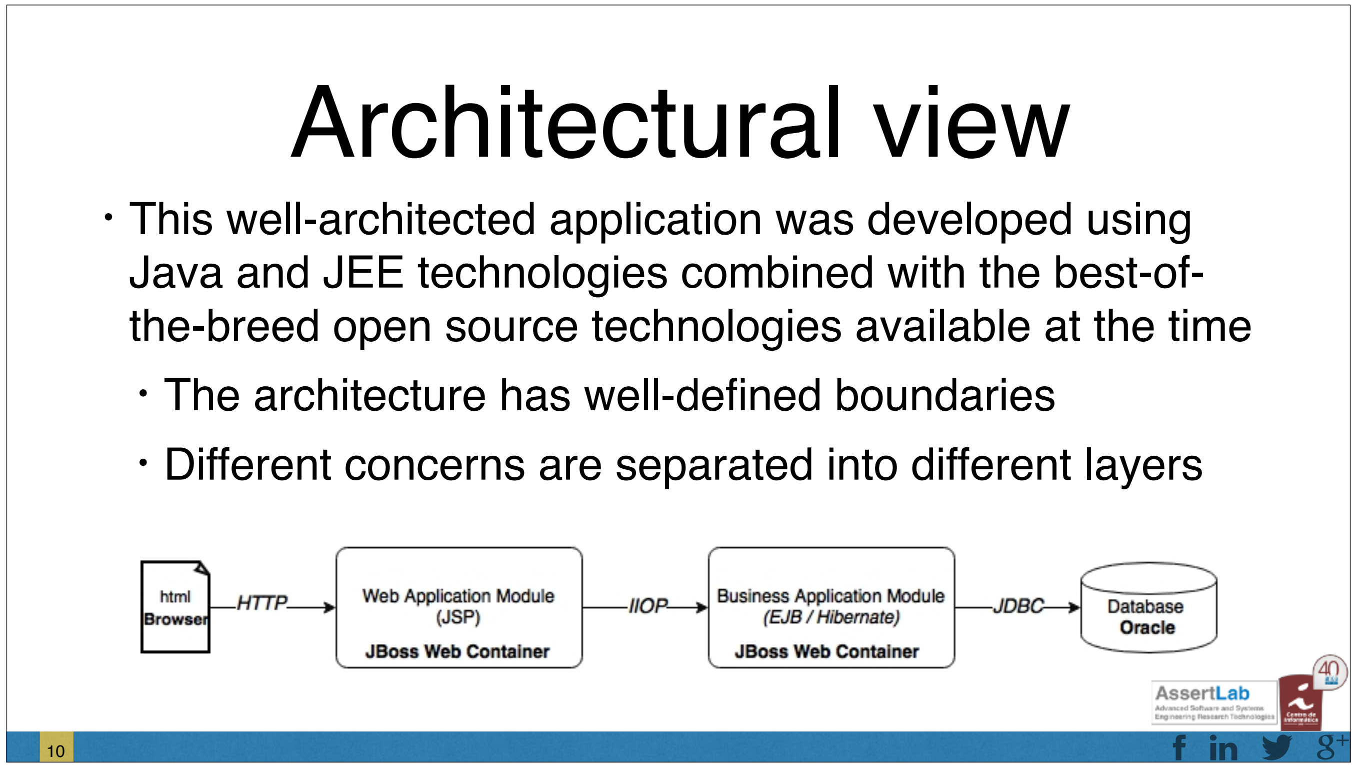


- There are two internally facing functions, **Pre-flight** and **Post-flight**.
  - **Pre-flight** functions include the planning phase, used for preparing flight schedules, plans, aircrafts, and so on
  - **Post-flight** functions are used by the back office for revenue management, accounting, and so on
- The **Search** and **Reserve** functions are part of the **online seat reservation process**, and the **Check-in** function is the process of accepting passengers at the airport
  - The **Check-in** function is also accessible to the end users over the Internet for **online check-in**

Functional view

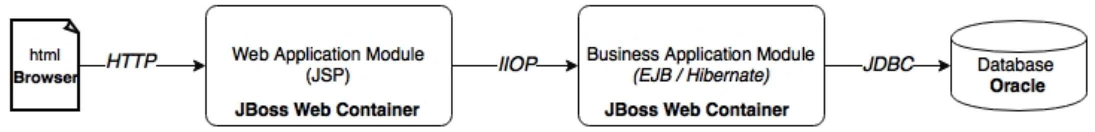

Search Functions
Search
Flight availability between cities for a given date
Flight
Flight routes, aircraft type and schedules
Fare
Fares between cities for each flight & date
Reservation Functions
Book
Book passengers on a selected flight & date
Inventory
Number of seats available on a flight & date
Payment
Payment gateway for online payments
Check In Functions
Check In
Accept a passenger on a flight on the day of travel
Boarding
Mark passenger as boarded on the airplane
Seating
Allocate passenger a seat based on rules
Baggage
Accept passenger baggage and print bag tag
Loyalty
Update passengers loyalty points
Back Office Functions
CRM
Customer relationship management
Data Analysis
Business intelligence analysis and reporting
Revenue Management
Fare calculations based on forecasts
Accounting
Invoicing and Billing
Data Management Functions
Reference Data
Country, City, Aircrafts, Currency etc.
Customer
Manage customers
Cross Cutting Functions
User Management
Manage user, roles, privileges
Notification
Send SMS and e-mails to customers

Each subfunction shown in the preceding diagram explains its role in the overall business process. Some subfunctions participate in more than one business process. For example, inventory is used in both search as well as in booking. To avoid any complication, this is not shown in the diagram. Data management and cross-cutting subfunctions are used across many business functions.



Architectural view

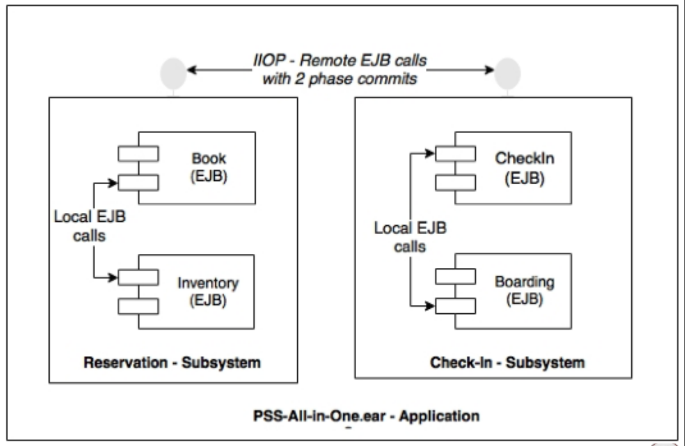
- This well-architected application was developed using Java and JEE technologies combined with the best-of-the-breed open source technologies available at the time
  - The architecture has well-defined boundaries
  - Different concerns are separated into different layers



The web application was developed as an N-tier, component-based modular system. The functions interact with each other through well-defined service contracts defined in the form of EJB endpoints

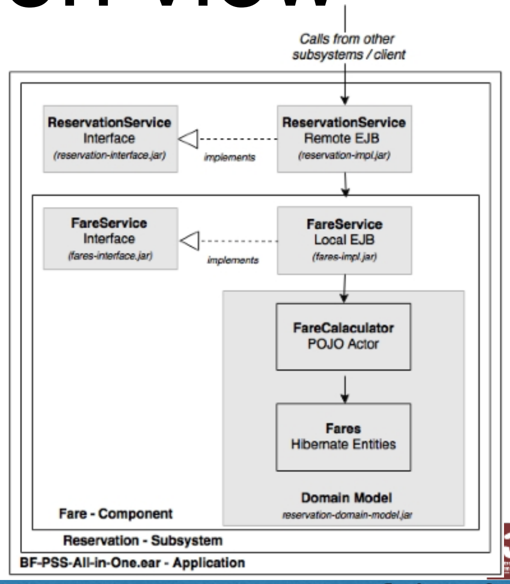
Design view

- Subsystems interact with each other through remote EJB calls using the IIOP protocol
- The transactional boundaries span across subsystems
- Components within the subsystems communicate with each other through local EJB component interfaces
- In theory, since subsystems use remote EJB endpoints, they could run on different physically separated application servers.
- This was one of the design goals



Implementation view

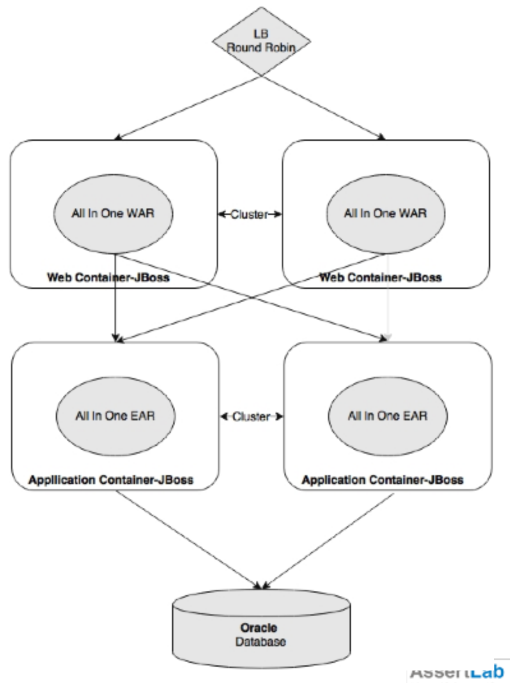
- The gray-shaded boxes are treated as different Maven projects, and translate into physical artifacts
- Interfaces are packaged as separate JAR files so that clients are abstracted from the implementations
- Local EJBs are used as component interfaces
- Finally, all subsystems are packaged into a single all-in-one EAR, and deployed in the application server


Calls from other subsystems / client
ReservationService Interface (reservation-interface.jar)
implements
ReservationService Remote EJB (reservation-impl.jar)
FareService Interface (fares-interface.jar)
implements
FareService Local EJB (fares-impl.jar)
FareCalaculator POJO Actor
Fares Hibernate Entities
Domain Model reservation-domain-model.jar
Fare - Component
Reservation - Subsystem
BF-PSS-All-in-One.ear - Application

Subsystems and components are designed adhering to the program to an interface principle

Deployment view

- The web modules and business modules were deployed into separate application server clusters
- The application was scaled horizontally by adding more and more application servers to the cluster
- Zero downtime deployments were handled by creating a standby cluster, and gracefully diverting the traffic to that cluster
- The standby cluster is destroyed once the primary cluster is patched with the new version and brought back to service



Death of the monolith

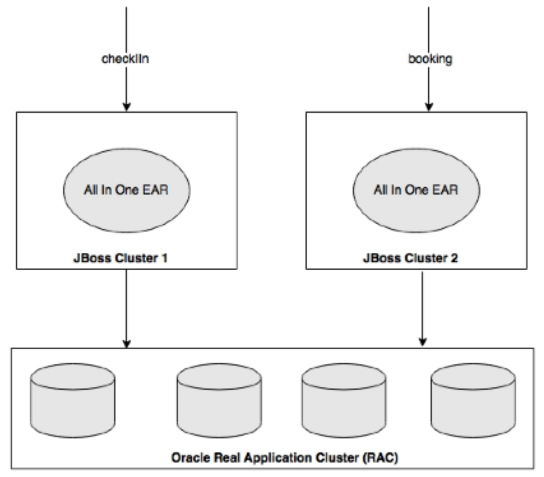
- The PSS application was performing well, successfully supporting all business requirements as well as the expected service levels
  - The system had no issues in scaling with the organic growth of the business in the initial years
- The business has seen tremendous growth over a period of time.
  - The fleet size increased significantly, and new destinations got added to the network.
  - As a result of this rapid growth, the number of bookings has gone up, resulting in a steep increase in transaction volumes, up to **200 - to 500** - fold of what was originally estimated

Pain points

- The rapid growth of the business eventually put the application under pressure
- Weaknesses of the system as well as the root causes of many failures
    - Stability - primarily due to stuck threads, which limit the application server's capability to accept more transactions
    - Outages - outage window increased largely because of the increase in server startup time
    - Agility - changes became harder to implement

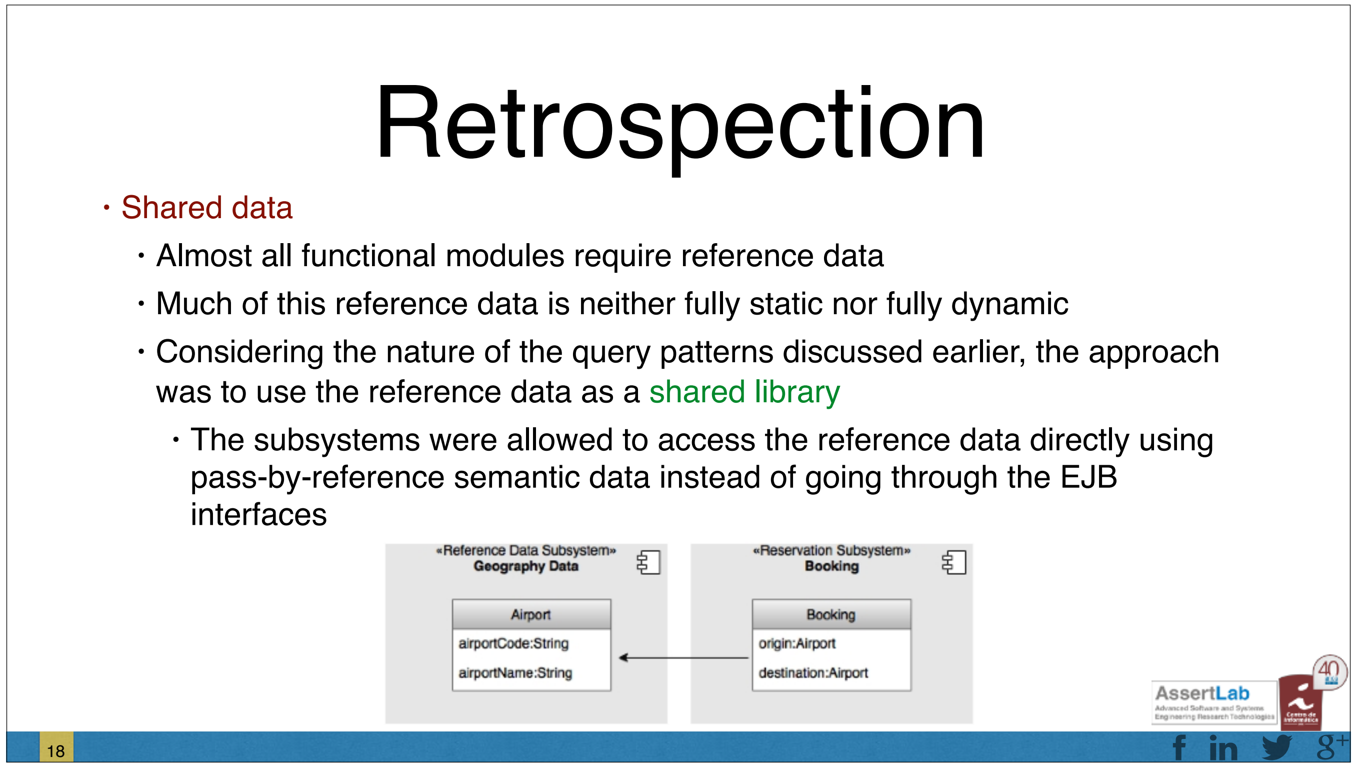
Stop gap fix

- Performance issues were partially addressed by applying the Y-axis scale method in the scale cube
- This new scaling model reduced the stability issues, but at a premium of increased complexity and cost of ownership
- The technology debt also increased over a period of time, leading to a state where a complete rewrite was the only option for reducing this technology debt



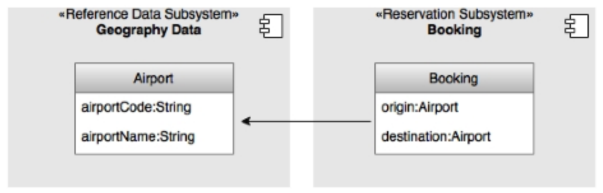
Retrospection

- Although the application was well-architected, there was a clear segregation between the functional components
- They were loosely coupled, programmed to interfaces, with access through standards-based interfaces, and had a rich domain model
- How come such a well-architected application failed to live up to the expectations? What else could the architects have done?



Retrospection

- Shared data
  - Almost all functional modules require reference data
  - Much of this reference data is neither fully static nor fully dynamic
  - Considering the nature of the query patterns discussed earlier, the approach was to use the reference data as a shared library
    - The subsystems were allowed to access the reference data directly using pass-by-reference semantic data instead of going through the EJB interfaces

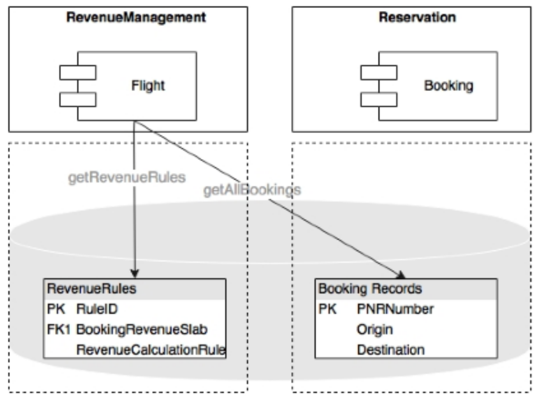


AssertLab

For instance, say a user wishes to see all the flights to a country. In this case, the flow of events could be as follows: find all the cities in the selected country, then all airports in the cities, and then fire a request to get all the flights to the list of resulting airports identified in that country.
As depicted in the preceding diagram, the **Booking entity** in the reservation subsystem is **allowed** to use the **reference data entities**, in this case Airport, as part of their **relationships**.

Retrospection

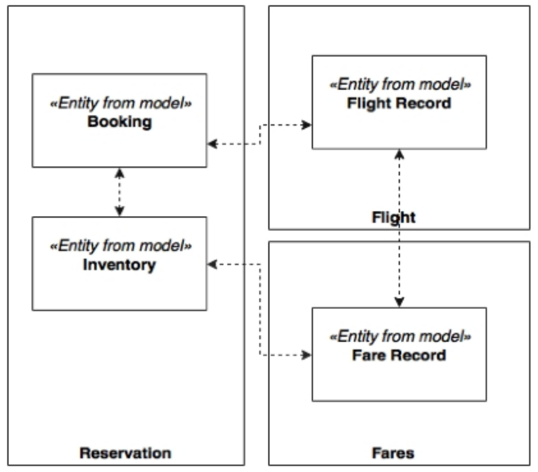
- Single database
  - The single schema approach opened a plethora of issues
  - Native queries
  - Stored procedures



Assume this results in N booking records. Now, for each booking record, Accounting has to execute a database call to find the applicable rules based on the fare code attached to each booking record. This could result in N+1 JDBC calls, which is inefficient.

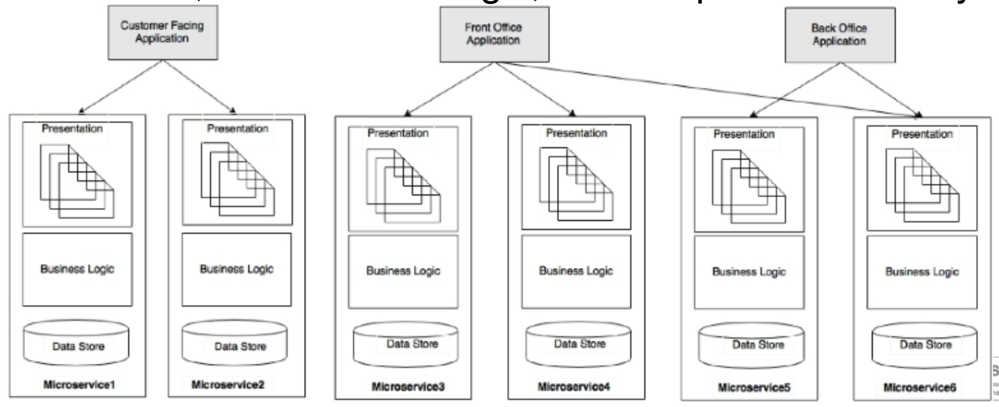
Retrospection

- Domain boundaries
  - Though the domain boundaries were well established, all the components were packaged as a single EAR file
  - As depicted in the following diagram, hibernate relationships were created across subsystem boundaries


«Entity from model» Booking
«Entity from model» Inventory
Reservation
«Entity from model» Flight Record
Flight
«Entity from model» Fare Record
Fares

Microservices to the rescue

- The objective is to move to a microservices-based architecture aligned to the business capabilities. Each microservice will hold the data store, the business logic, and the presentation layer



The approach taken by BrownField Airline is to build a number of web portal applications targeting specific user communities such as customer facing, front office, and back office. The advantage of this approach lies in the flexibility for modeling, and also in the possibility to treat different communities differently. For example, the policies, architecture, and testing approaches for the Internet facing layer are different from the intranet-facing web application. Internet-facing applications may take advantage of CDNs (Content Delivery Networks) to move pages as close to the customer as possible, whereas intranet applications could serve pages directly from the data center.

The business case

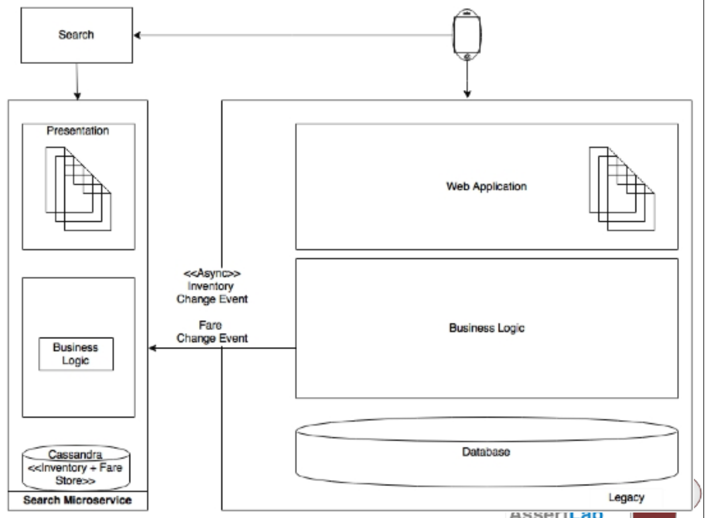
- Microservices offers a full list of benefits (see lecture 2)
  - Service dependencies
  - Physical boundaries (also technology)
  - Selective scaling
  - Technology obsolescence

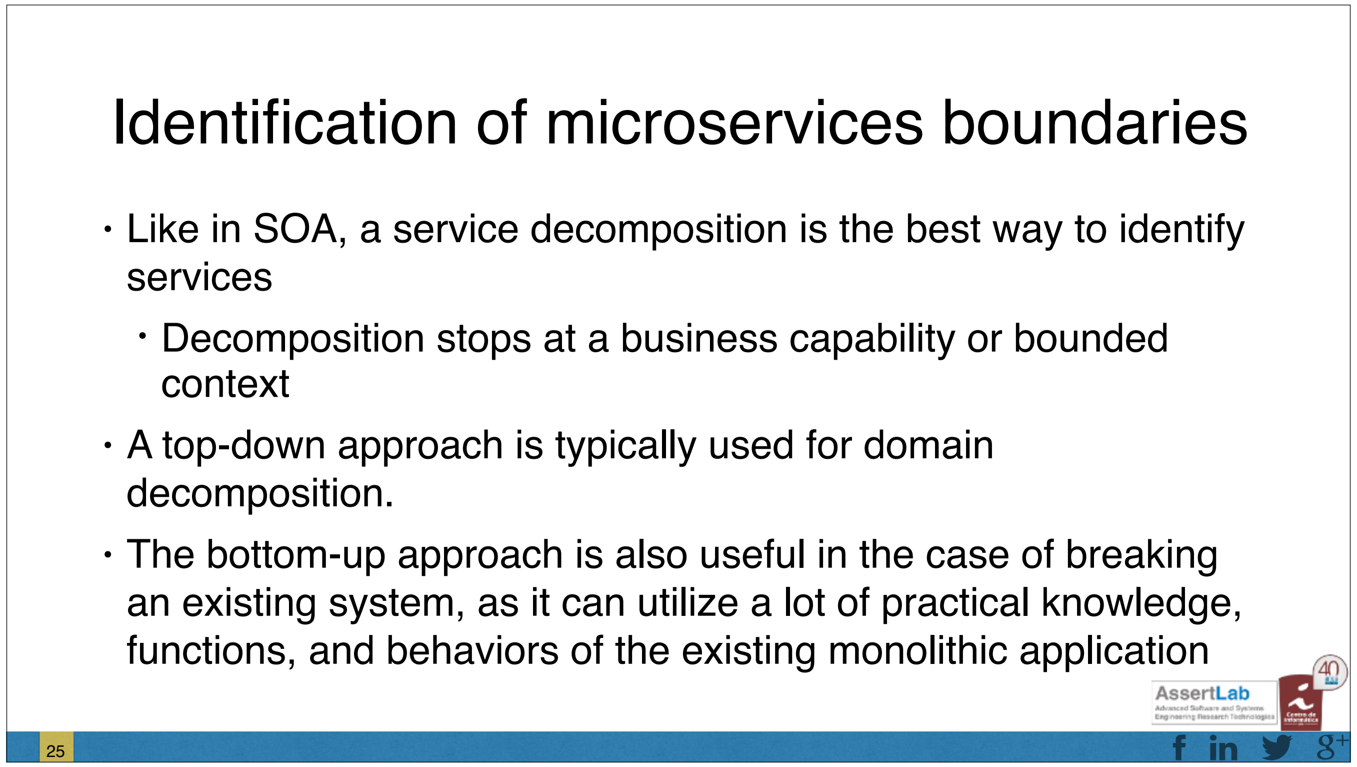
Plan the evolution

- It is not simple to break an application that has millions of lines of code, especially if the code has complex dependencies
- How do we break it?
- More importantly, where do we start, and how do we approach this problem?

Evolutionary approach

- At every step, a microservice will be created outside of the monolithic application, and traffic will be diverted to the new service
- A number of key questions need to be answered from the transition point of view
  - Identification of microservices' boundaries
  - Prioritizing microservices for migration
  - Handling data synchronization during the transition phase
  - Handling user interface integration, working with old and new user interfaces
  - Handling of reference data in the new system
  - Testing strategy to ensure the business capabilities are intact and correctly reproduced
  - Identification of any prerequisites for microservice development such as microservices capabilities, frameworks, processes, and so on





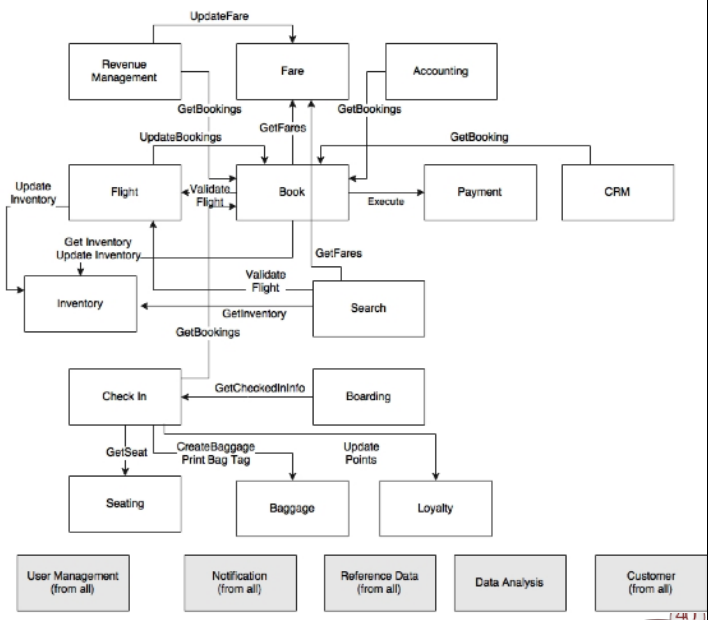
Identification of microservices boundaries

- Like in SOA, a service decomposition is the best way to identify services
  - Decomposition stops at a business capability or bounded context
- A top-down approach is typically used for domain decomposition.
- The bottom-up approach is also useful in the case of breaking an existing system, as it can utilize a lot of practical knowledge, functions, and behaviors of the existing monolithic application

This is the most interesting part of the problem, and the most difficult part as well.

Analyze dependencies

- Analyze the dependencies between the initial set of candidate microservices that we created
  - Dependency graph
- The bottom layer shows cross-cutting capabilities that are used across multiple modules



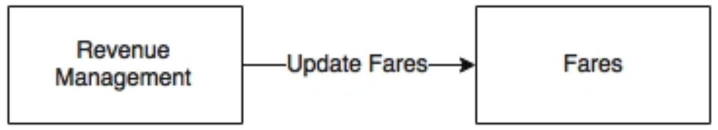
AssertLab Advanced Software and Systems Engineering Research Technologies

One way to produce a dependency graph is to list out all the components of the legacy system and overlay dependencies. This could be done by combining one or more of the approaches listed as follows:

- Analyzing the manual code and regenerating dependencies.
- Using the experience of the development team to regenerate dependencies.
- Using a Maven dependency graph. There are a number of tools we could use to regenerate the dependency graph, such as PomExplorer, PomParser, and so on.
- Using performance engineering tools such as AppDynamics to identify the call stack and roll up dependencies

Events as opposed to query

- Dependencies could be query-based or event-based



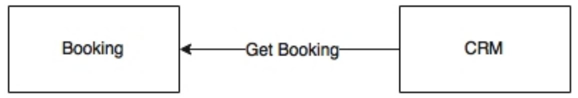
- Revenue Management is a module used for calculating optimal fare values, based on the booking demand forecast. In case of a fare change between an origin and a destination, Update Fare on the Fare module is called by Revenue Management to update the respective fares in the Fare module.

Event-based is **better** for **scalable systems**. Sometimes, it is possible to convert query-based communications to event-based ones. In many cases, these dependencies exist because either the business organizations are managed like that, or by virtue of the way the old system handled the business scenario.

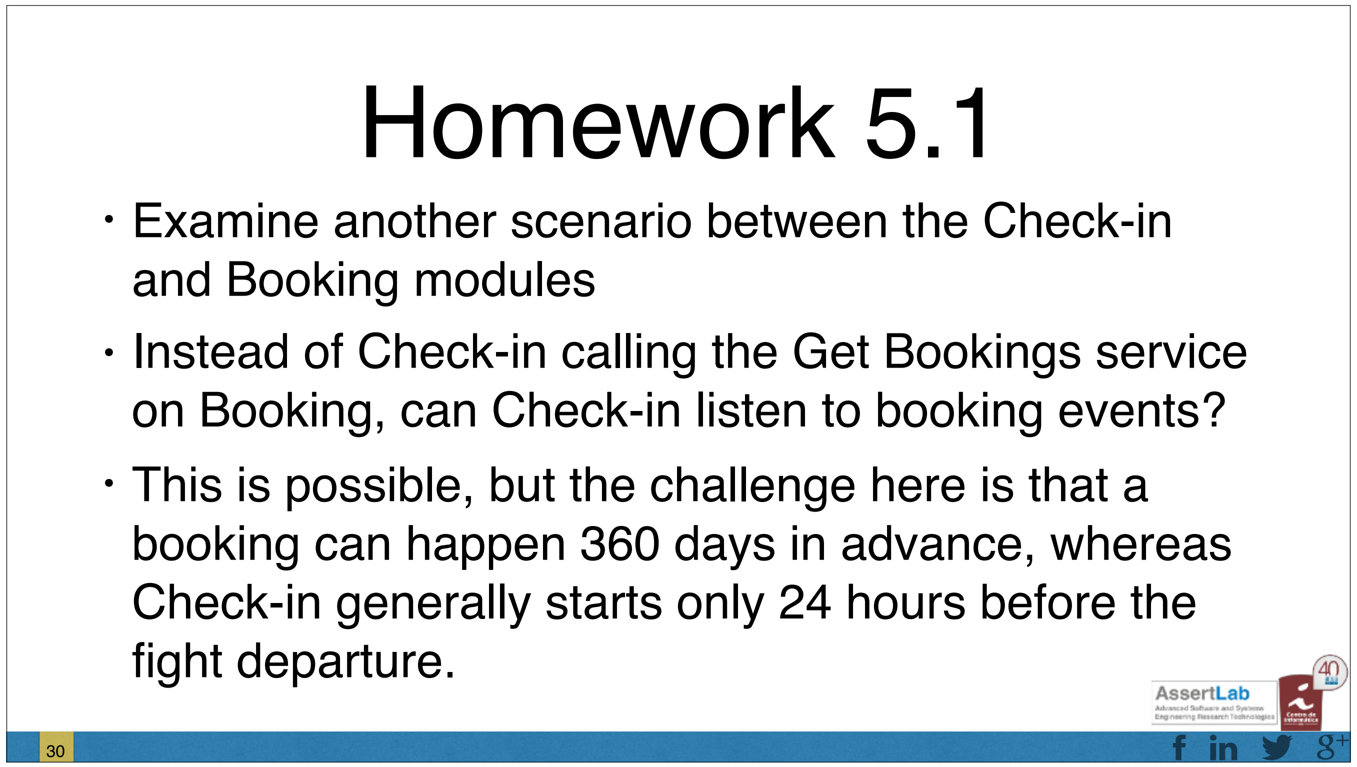
Events as opposed to query

- An alternate way of thinking is that the Fare module is subscribed to Revenue Management for any changes in fares, and Revenue Management publishes whenever there is a fare change.
- This reactive programming approach gives an added flexibility by which the Fares and the Revenue Management modules could stay independent, and connect them through a reliable messaging system

Events as opposed to query



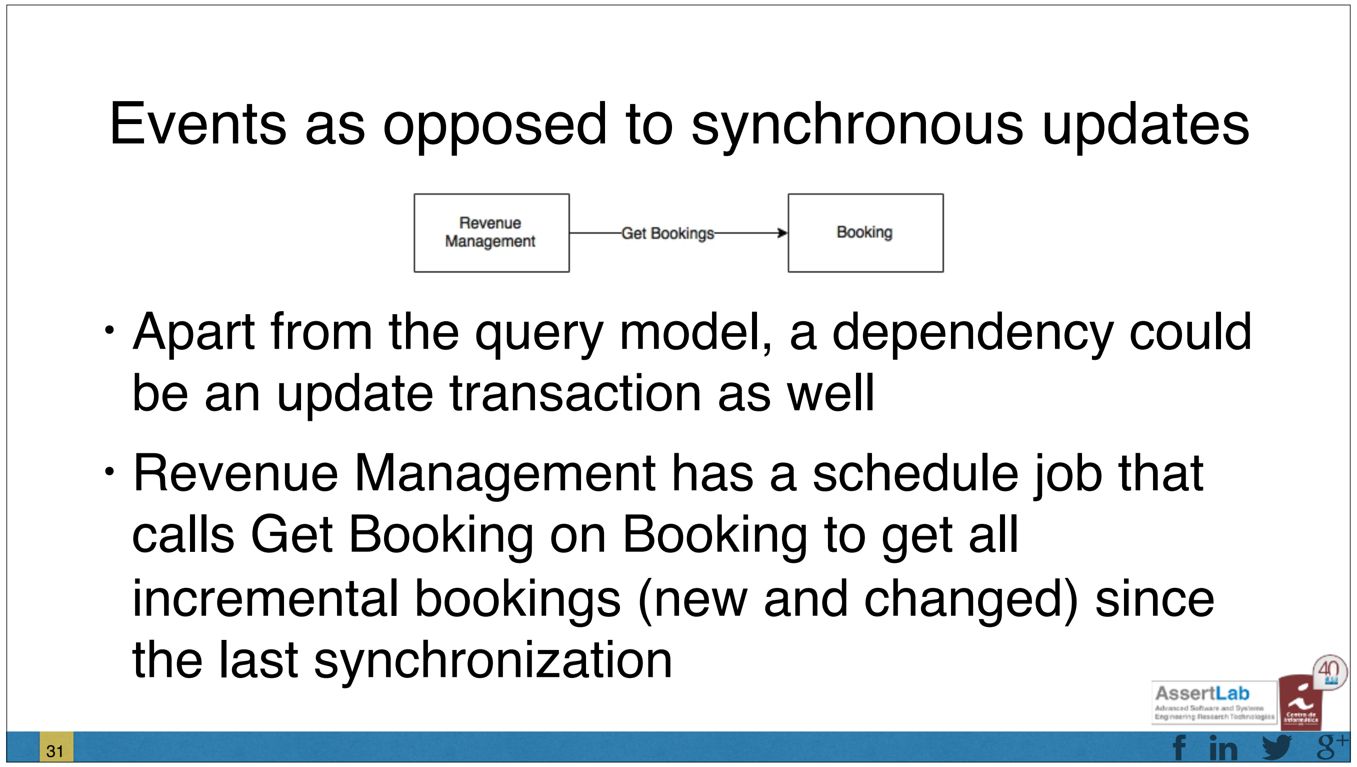
- The CRM module is used to manage passenger complaints. When CRM receives a complaint, it retrieves the corresponding passenger's Booking data
  - In reality, the number of complaints are negligibly small when compared to the number of bookings
- If we blindly apply the previous pattern where CRM subscribes to all bookings, we will find that it is not cost effective



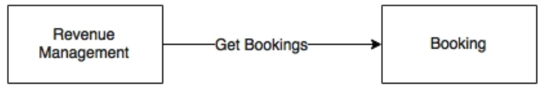
Homework 5.1

- Examine another scenario between the Check-in and Booking modules
- Instead of Check-in calling the Get Bookings service on Booking, can Check-in listen to booking events?
- This is possible, but the challenge here is that a booking can happen 360 days in advance, whereas Check-in generally starts only 24 hours before the fight departure.

An alternate option is that when check-in opens for a flight (24 hours before departure), Check-in calls a service on the Booking module to get a snapshot of the bookings for a given flight. Once this is done, Check-in could subscribe for booking events specifically for that flight. In this case, a combination of query-based as well as event-based approaches is used. By doing so, we reduce the unnecessary events and storage apart from reducing the number of queries between these two services.



Events as opposed to synchronous updates

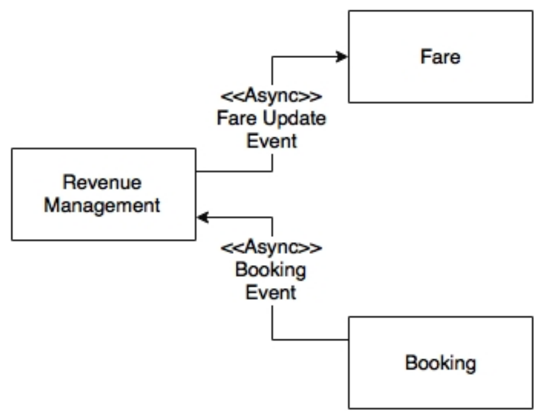


- Apart from the query model, a dependency could be an update transaction as well
- Revenue Management has a schedule job that calls Get Booking on Booking to get all incremental bookings (new and changed) since the last synchronization

An alternative approach is to send new bookings and the changes in bookings as soon as they take place in the Booking module as an asynchronous push.

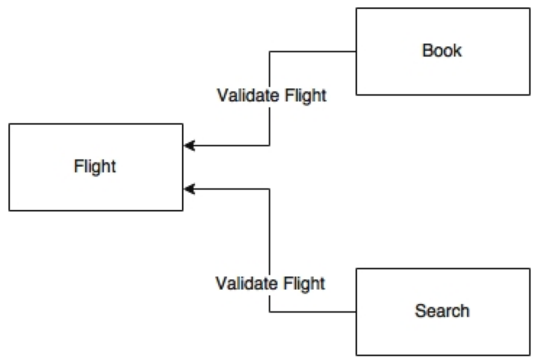
Events as opposed to synchronous updates

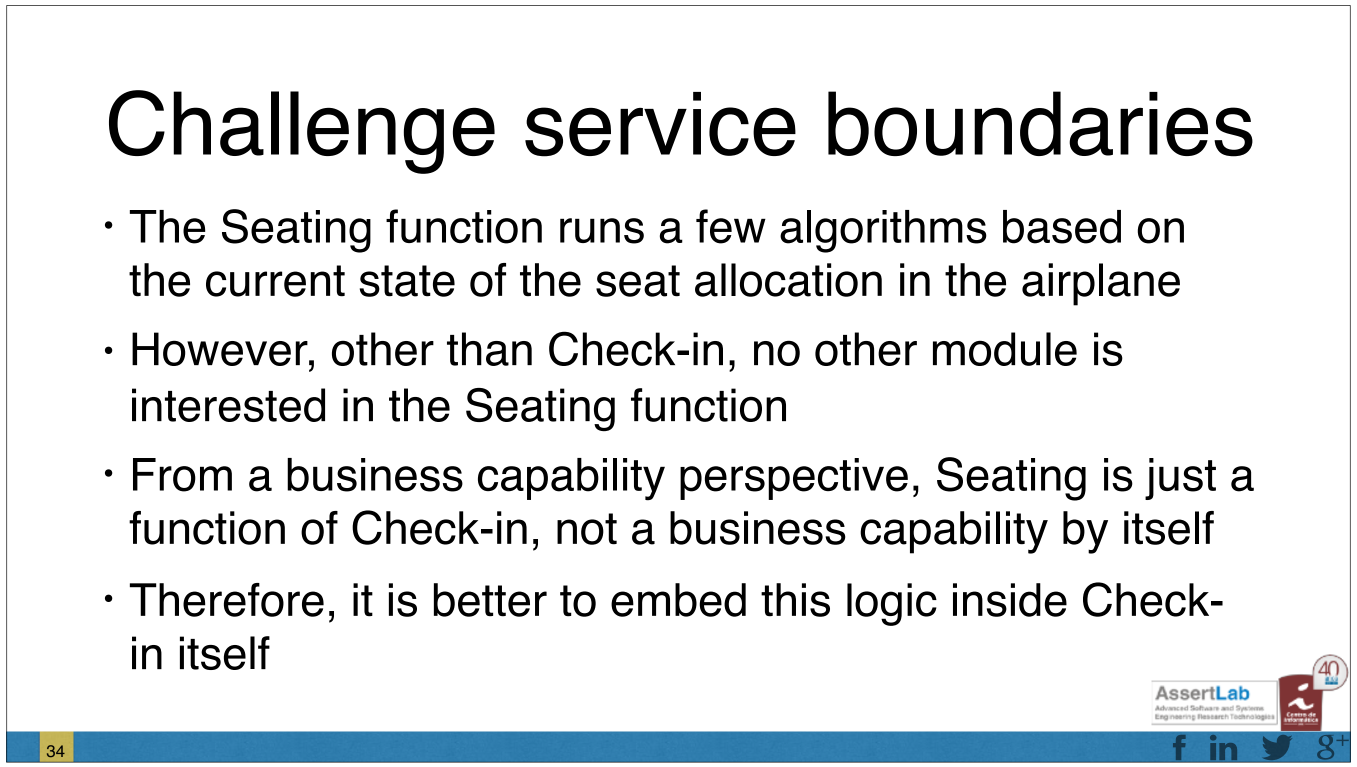
- The same pattern could be applied in many other scenarios such as from Booking to Accounting, from Flight to Inventory, and also from Flight to Booking



Challenge requirements

- The Validate Flight call is to validate the input flight data coming from different channels
- An alternate way of solving this is to adjust the inventory of the flight based on these given conditions
- As far as Search and Booking are concerned, both just look up the inventory instead of validating flights for every request

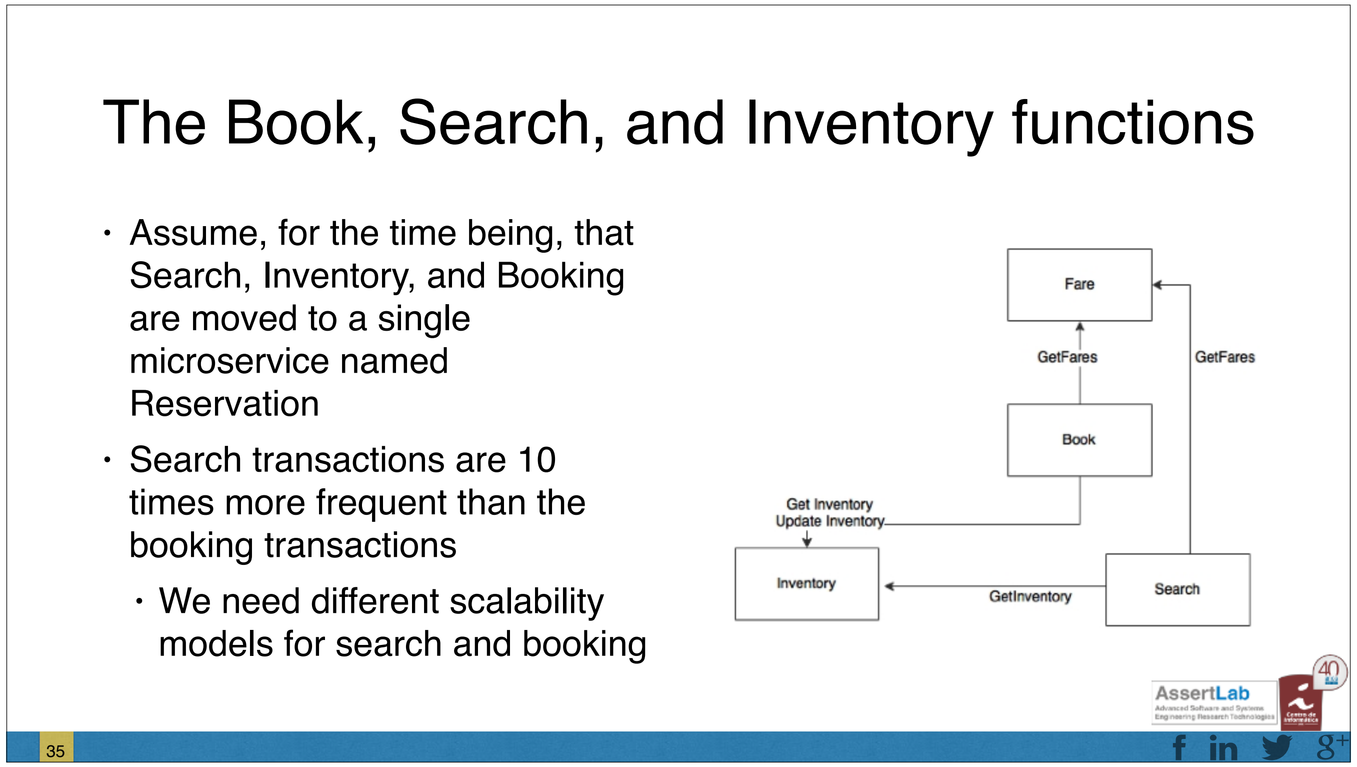




Challenge service boundaries

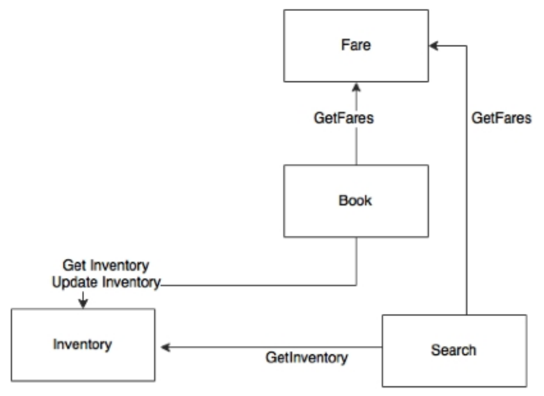
- The Seating function runs a few algorithms based on the current state of the seat allocation in the airplane
- However, other than Check-in, no other module is interested in the Seating function
- From a business capability perspective, Seating is just a function of Check-in, not a business capability by itself
- Therefore, it is better to embed this logic inside Check-in itself

The same is applicable to Baggage as well. BrownField has a separate baggage handling system. The Baggage function in the PSS context is to print the baggage tag as well as store the baggage data against the Check-in records. There is no business capability associated with this particular functionality. Therefore, it is ideal to move this function to Check-in itself.

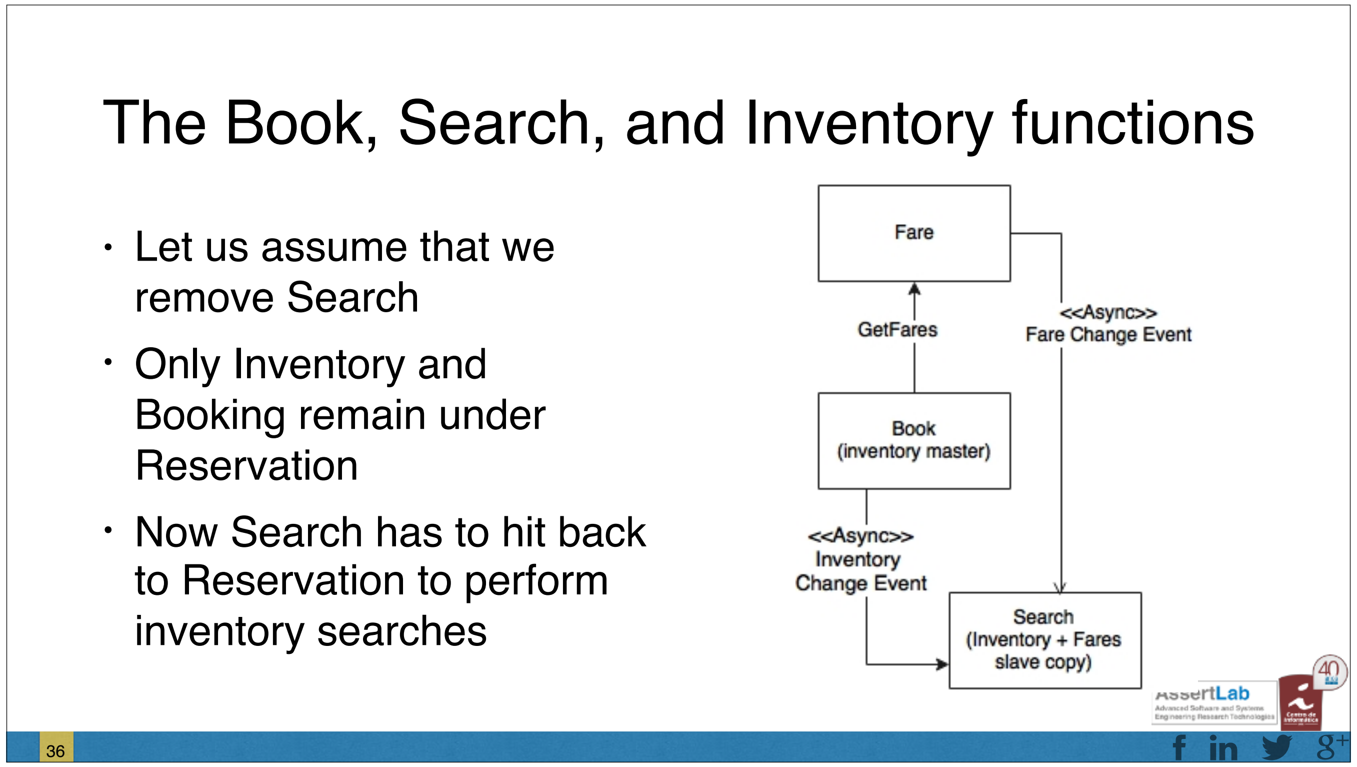


The Book, Search, and Inventory functions

- Assume, for the time being, that Search, Inventory, and Booking are moved to a single microservice named Reservation
- Search transactions are 10 times more frequent than the booking transactions
  - We need different scalability models for search and booking

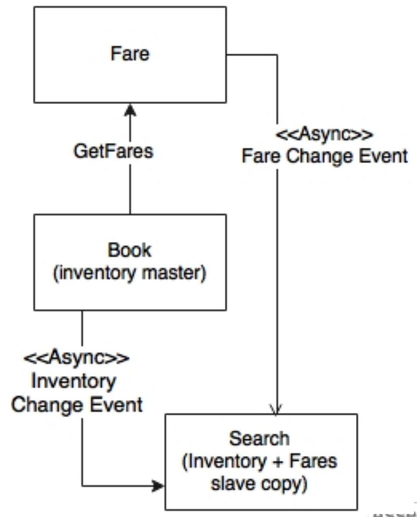


Similarly, Inventory and Search are more supporting functions of the Booking module. They are not aligned with any of the business capabilities as such. Similar to the previous judgement, it is ideal to move both the Search and Inventory functions to Booking.



The Book, Search, and Inventory functions

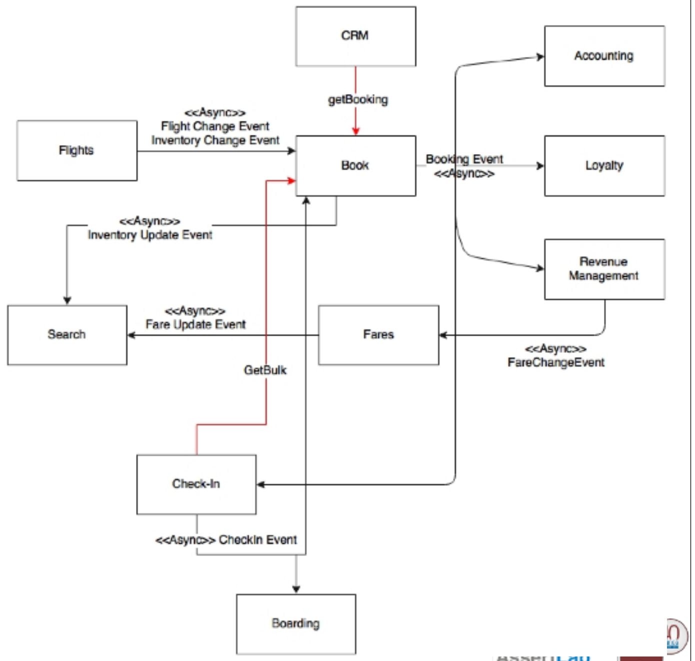
- Let us assume that we remove Search
- Only Inventory and Booking remain under Reservation
- Now Search has to hit back to Reservation to perform inventory searches



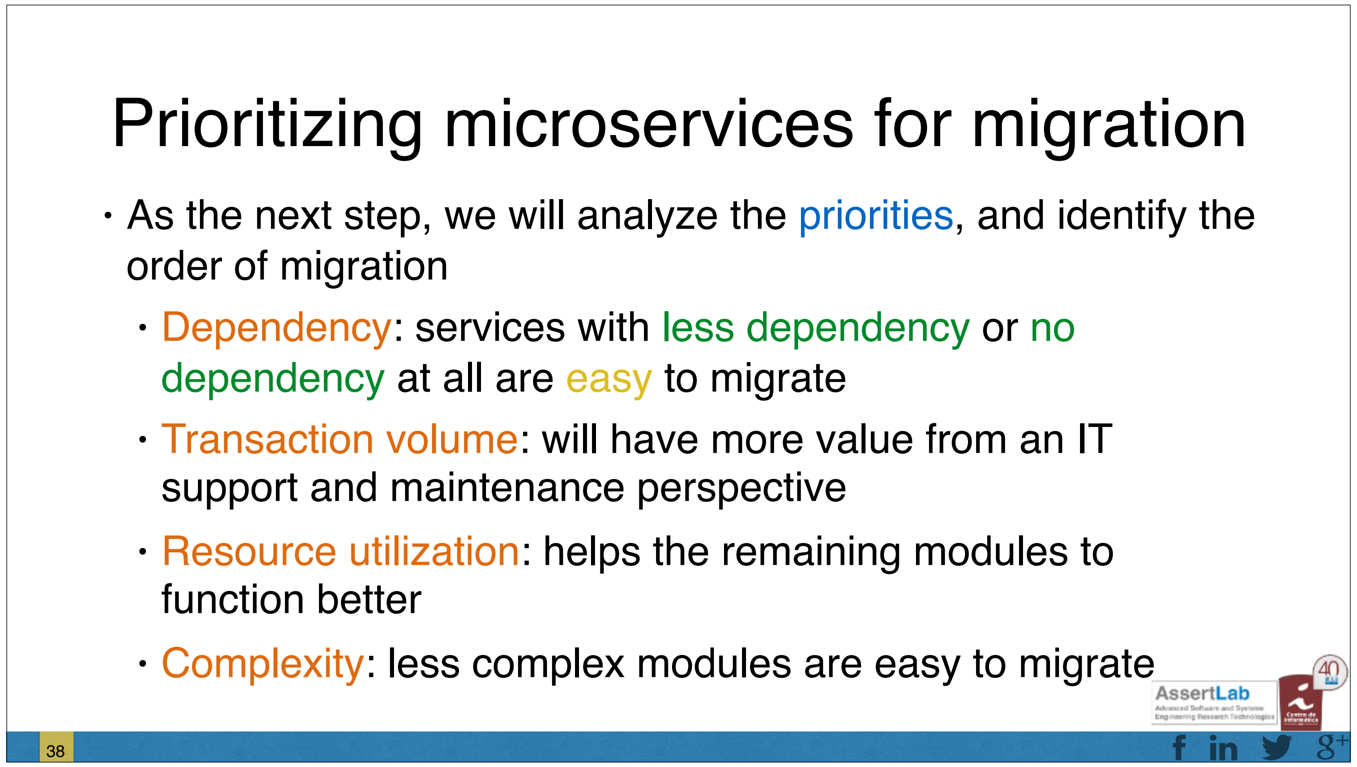
A better approach is to keep **Inventory along** with the **Booking** module, and keep a **read-only copy of the inventory under Search**, while continuously synchronizing the inventory data over a **reliable messaging system**

Final dependency graph

- There are still a few synchronized calls, which, for the time being, we will keep as they are
- By applying all these changes, the final dependency diagram will look like the following one



Now we can safely consider each box in the preceding diagram as a microservice



Prioritizing microservices for migration

- As the next step, we will analyze the priorities, and identify the order of migration
  - Dependency: services with less dependency or no dependency at all are easy to migrate
  - Transaction volume: will have more value from an IT support and maintenance perspective
  - Resource utilization: helps the remaining modules to function better
  - Complexity: less complex modules are easy to migrate

- Accounting, Loyalty, CRM, and Boarding have less dependencies as compared to Booking and Check-in.
- Services with the highest transaction volumes will relieve the load on the existing system
- Migrating resource intensive services out of the legacy system provides relief to other services.

Prioritizing microservices for migration

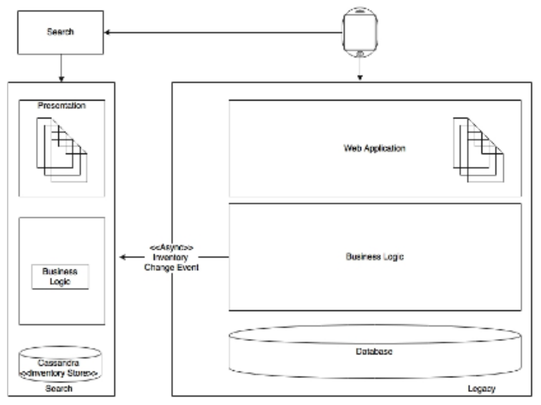
- Business criticality: highly critical modules deliver higher business value
- Velocity of changes: indicates the number of change requests targeting a function in a short time frame
- Innovation: innovations in legacy systems are harder to achieve as compared to applying innovations in the microservices world

Data synchronization during migration

- During the transition phase, the legacy system and the new microservices will run in parallel
  - synchronize the data between the two systems at the database level by using any data synchronization tool
    - built on the same data store technologies
  - we allow a backdoor entry, hence exposing the microservices' internal data store outside

Data synchronization during migration

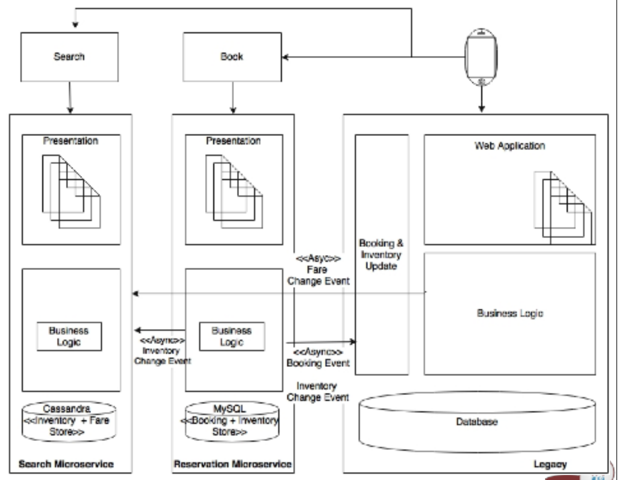
- Let us assume that we use a NoSQL database for keeping inventory and fares under the Search service
- In this particular case, all we need is the legacy system to supply data to the new service using asynchronous events
- The Search service then accepts these events, and stores them locally into the local NoSQL store



The following diagram shows the data migration and synchronization aspect once Search is taken out.

Data synchronization during migration

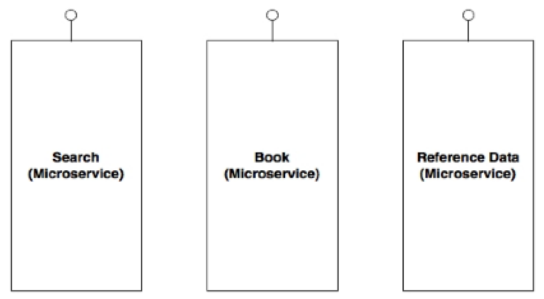
- The new Booking microservice sends the inventory change events to the Search service
- In addition to this, the legacy application also has to send the fare change events to Search
- Booking will then store the new Booking service in its MySQL data store



This is a bit more tedious in the case of the complex Booking service.
The most complex piece, the Booking service, has to **send the booking events and the inventory events back to the legacy system**. This is to **ensure** that the functions in the legacy system **continue to work as before**.
The simplest approach is to write an update component which accepts the events and updates the old booking records table so that there are no changes required in the other legacy modules.

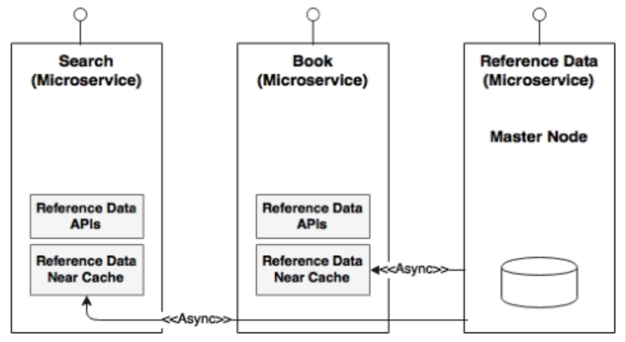
Managing reference data

- One of the biggest challenges
- A simple approach is to build the reference data as another microservice itself
- In this case, whoever needs reference data should access it through the microservice endpoints
- This is a well-structured approach, but could lead to performance issues as encountered in the original legacy system



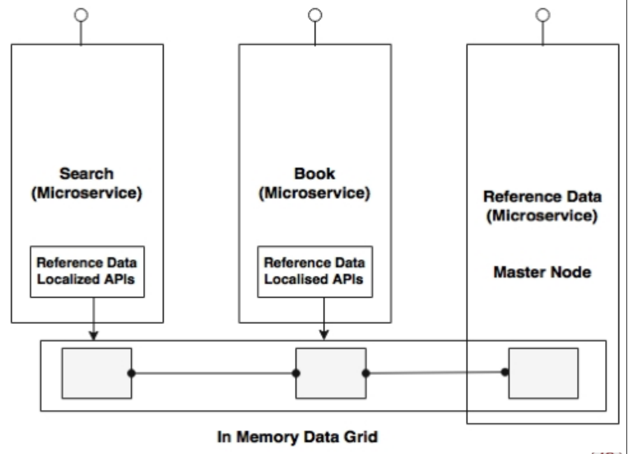
Managing reference data

- An alternate approach is to have reference data as a microservice service for all the admin and CRUD functions
  - A near cache will then be created under each service to incrementally cache data from the master services
  - A thin reference data access proxy library will be embedded in each of these services
- The reference data access proxy abstracts whether the data is coming from cache or from a remote service.

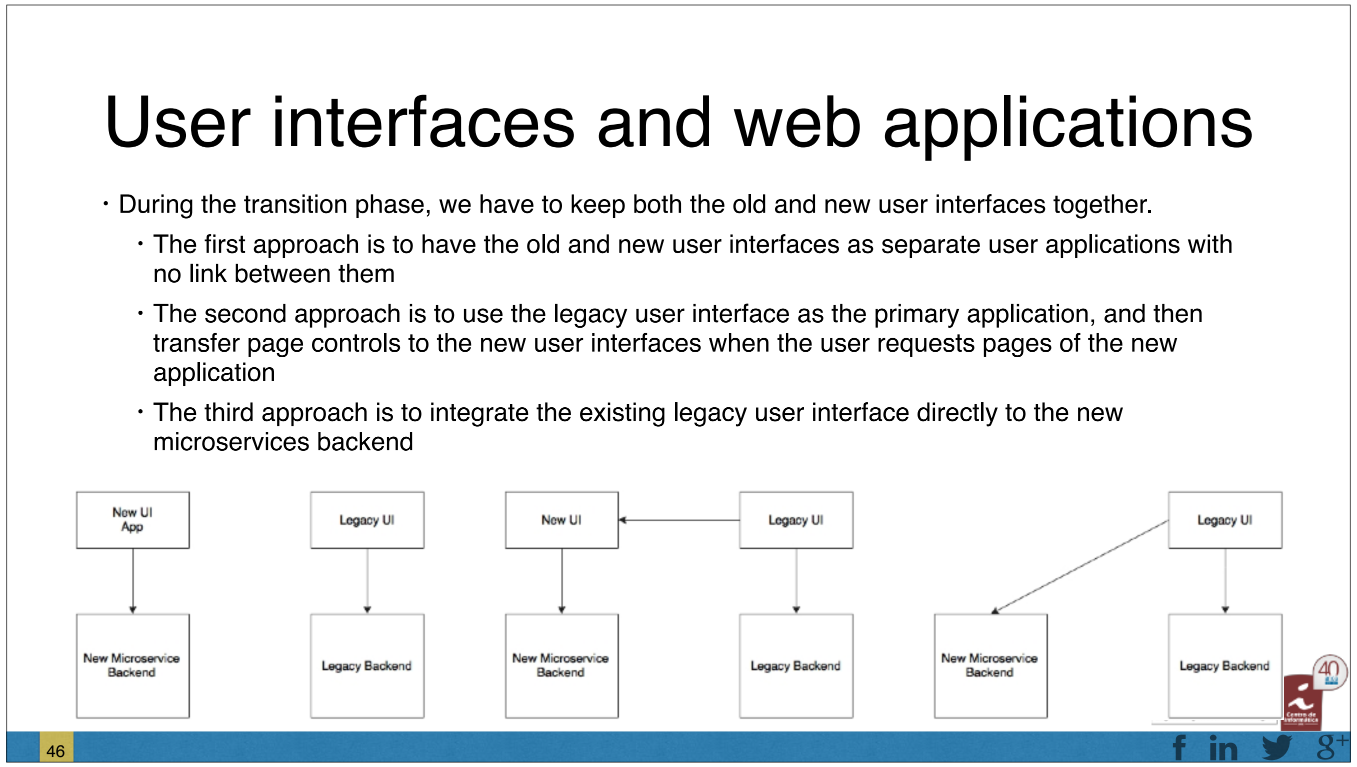


Managing reference data

- The challenge is to synchronize the data between the master and the slave
- A subscription mechanism is required for those data caches that change frequently
- A better approach is to replace the local cache with an in-memory data grid

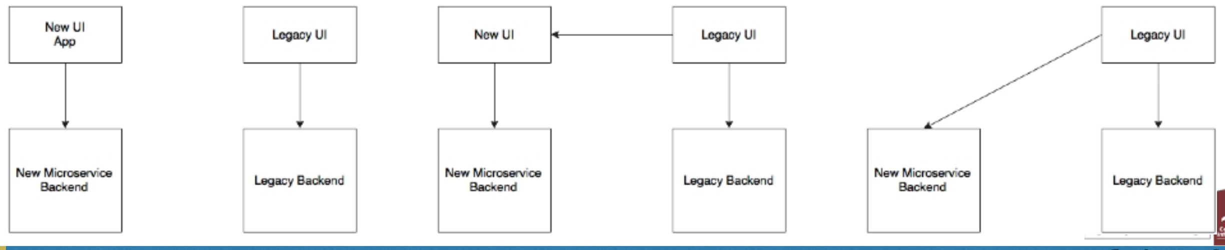

Search (Microservice)
Reference Data Localized APIs
Book (Microservice)
Reference Data Localised APIs
Reference Data (Microservice)
Master Node
In Memory Data Grid

The reference data microservice will write to the data grid, whereas the proxy libraries embedded in other services will have read-only APIs. This eliminates the requirement to have subscription of data, and is much more efficient and consistent.



User interfaces and web applications

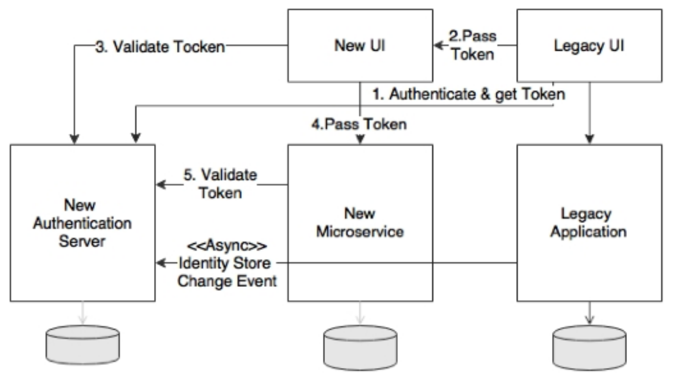
- During the transition phase, we have to keep both the old and new user interfaces together.
  - The first approach is to have the old and new user interfaces as separate user applications with no link between them
  - The second approach is to use the legacy user interface as the primary application, and then transfer page controls to the new user interfaces when the user requests pages of the new application
  - The third approach is to integrate the existing legacy user interface directly to the new microservices backend



2nd -> In this case, since the old and the new applications are web-based applications running in a web browser window, users will get a seamless experience. SSO has to be implemented between the old and the new user interfaces.

Session handling and security

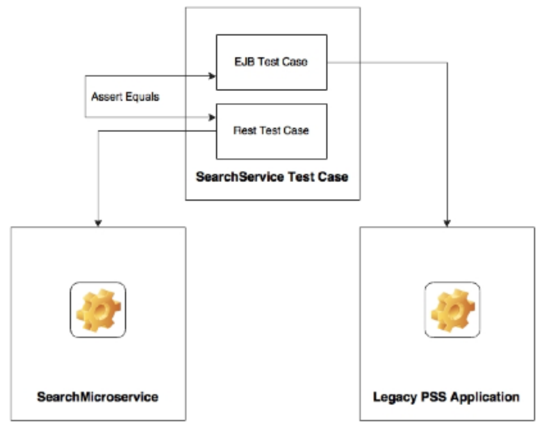
- Assume that the new services are written based on Spring Security with a token-based authorization strategy, whereas the old application uses a custom-built authentication with its local identity store
- The simplest approach, as shown in the preceding diagram, is to build a new identity store with an authentication service as a new microservice using Spring Security.
  - This will be used for all our future resource and service protections, for all microservices.



The existing user interface application authenticates itself against the new authentication service, and secures a token. This token will be passed to the new user interface or new microservice. In both cases, the user interface or microservice will make a call to the authentication service to validate the given token. If the token is valid, then the UI or microservice accepts the call.

Test strategy

- One important question to answer from a testing point of view is how can we ensure that all functions work in the same way as before the migration?
- Integration test cases should be written for the services that are getting migrated before the migration or refactoring



Building ecosystem capabilities

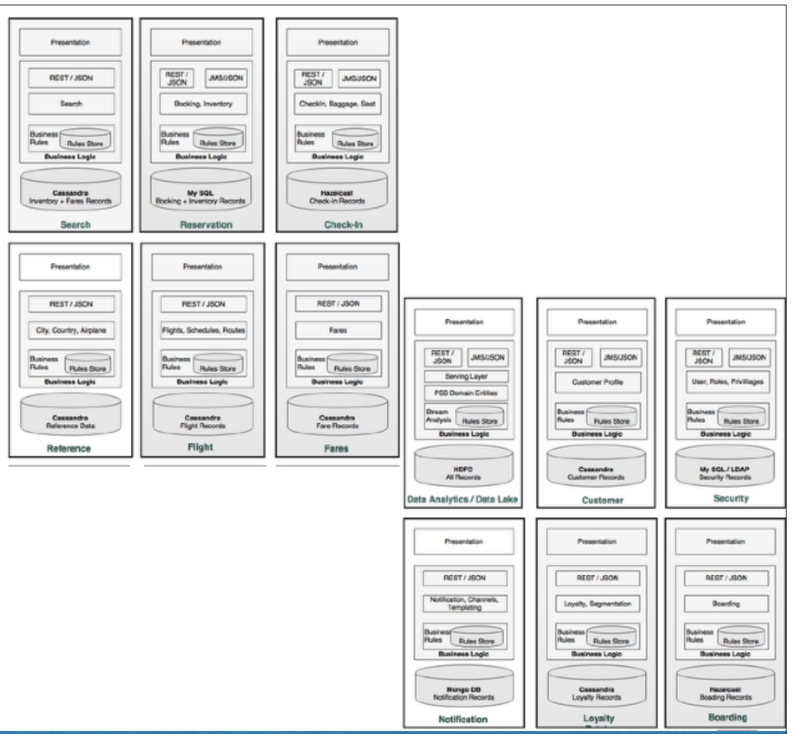
- Before we embark on actual migration, we have to build all of the microservice's capabilities mentioned under the capability model
- In addition to these capabilities, certain application functions are also required to be built upfront
  - Reference data, security and SSO, and Customer and Notification
  - Data warehouse or a data lake is also required as a prerequisite
- An effective approach is to build these capabilities in an incremental fashion, delaying development until it is really required

Migrate modules only if required

- It is important to understand that it is not necessary to migrate all modules to the new microservices architecture, unless it is really required
    - A major reason is that these migrations incur cost

Target architecture

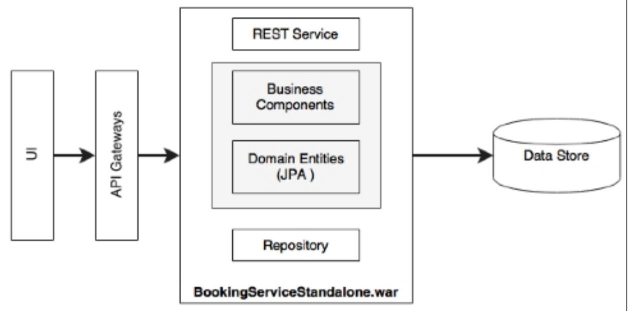
- Each block in the diagram represents a microservice
- The shaded boxes are core microservices, and the others are supporting microservices
- The diagram also shows the internal capabilities of each microservice



Each service has its own architecture, typically consisting of a presentation layer, one or more service endpoints, business logic, business rules, and database. As we can see, we use different selections of databases that are more suitable for each microservice. Each one is autonomous with minimal orchestration between the services. Most of the services interact with each other using the service endpoints.

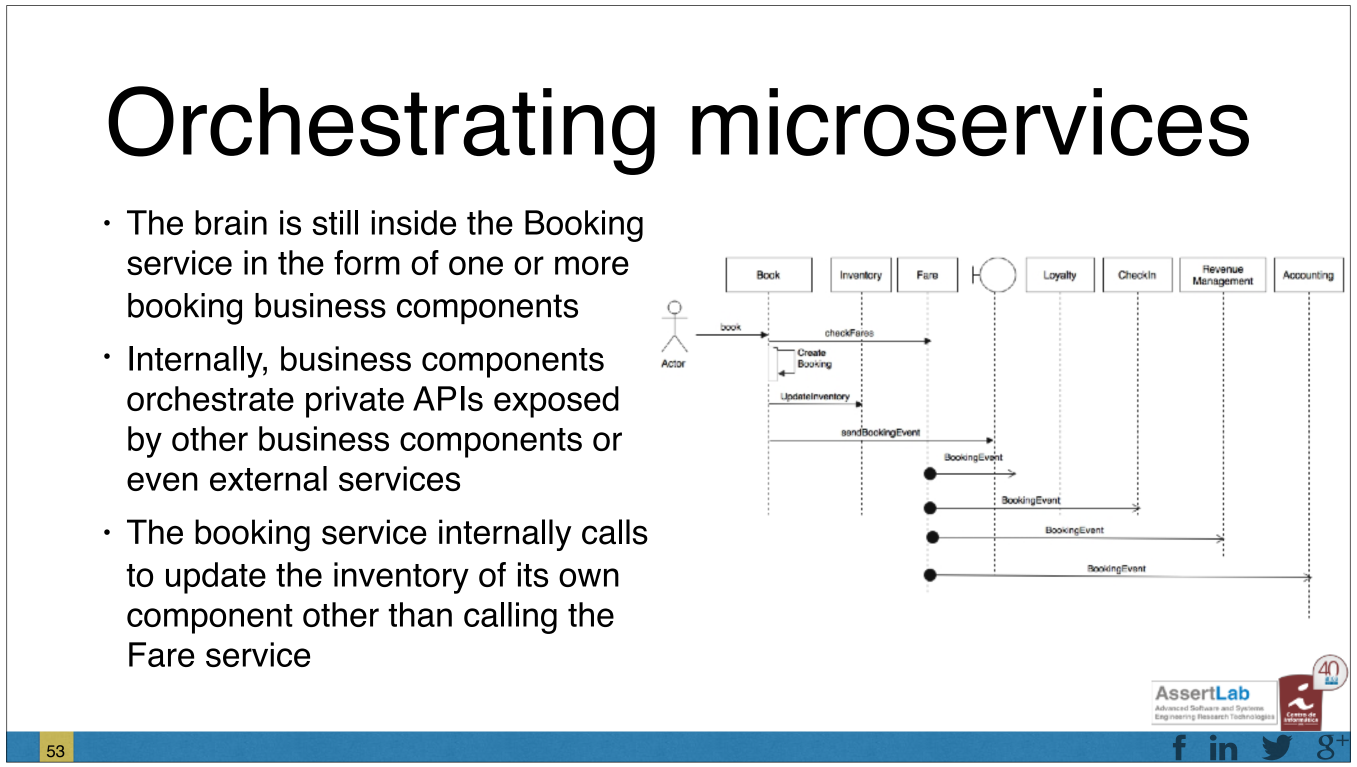
Internal layering of microservices

- There is no standard to be followed for the internal architecture of a microservice
- The rule of thumb is to abstract realizations behind simple service endpoints



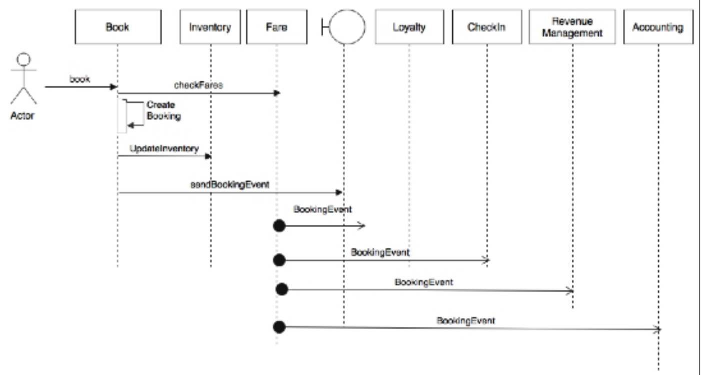
AssertLab Advanced Software and Systems Engineering Research Technologies

The UI accesses REST services through a service gateway. The API gateway may be one per microservice or one for many microservices—it depends on what we want to do with the API gateway. There could be one or more rest endpoints exposed by microservices. These endpoints, in turn, connect to one of the business components within the service. Business components then execute all the business functions with the help of domain entities. A repository component is used for interacting with the backend data store.



Orchestrating microservices

- The brain is still inside the Booking service in the form of one or more booking business components
- Internally, business components orchestrate private APIs exposed by other business components or even external services
- The booking service internally calls to update the inventory of its own component other than calling the Fare service



AssertLab Advanced Software and Systems Engineering Research Technologies

Is there any orchestration engine required for this activity? It depends on the requirements. In complex scenarios, we may have to do a number of things in parallel. For example, creating a booking internally applies a number of booking rules, it validates the fare, and it validates the inventory before creating a booking. We may want to execute them in parallel. In such cases, we may use Java concurrency APIs or reactive Java libraries.

Integration with other systems

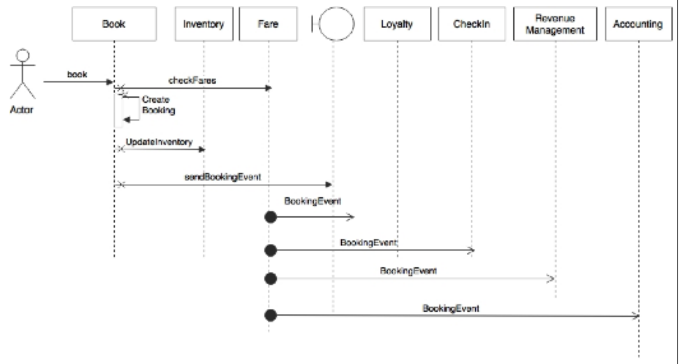
- In the microservices world, we use an API gateway or a reliable message bus for integrating with other non-microservices
- Let us assume that there is another system in BrownField that needs booking data
  - Unfortunately, the system is not capable of subscribing to the booking events that the Booking microservice publishes
- An Enterprise Application integration (EAI) solution could be employed, which listens to our booking events, and then uses a native adaptor to update the database

Managing shared libraries

- Certain business logic is used in more than one microservice
- Search and Reservation, in this case, use inventory rules
- In such cases, these shared libraries will be duplicated in both the microservices

Handling exceptions

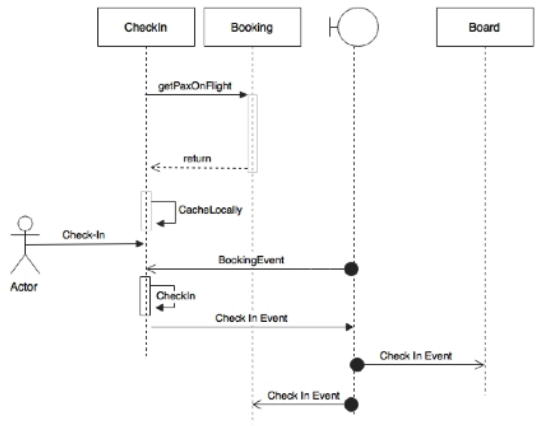
- There is a synchronous communication between Booking and Fare
- In case the Fare service is not available, we trust the incoming request, and accept the Booking
- We will use a circuit breaker and a fallback service which simply creates the booking with a special status, and queues the booking for manual action or a system retry



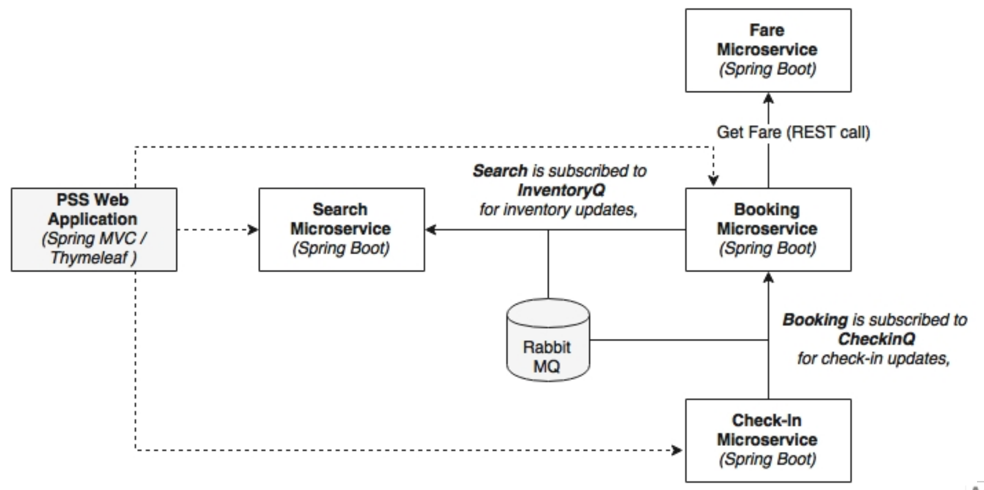
What if creating the booking fails? If creating a booking fails unexpectedly, a better option is to throw a message back to the user. We could try alternative options, but that could increase the overall complexity of the system. The same is applicable for inventory updates

Handling exceptions

- Consider a scenario where the Check-in services fail immediately after the Check-in Complete event is sent out
- The other consumers processed this event, but the actual check-in is rolled back.
  - This is because we are not using a two-phase commit
- This could be done by catching the exception, and sending another Check-in Cancelled event



Target implementation view


Fare Microservice (Spring Boot)
Get Fare (REST call)
Search is subscribed to InventoryQ for inventory updates,
PSS Web Application (Spring MVC / Thymeleaf)
Search Microservice (Spring Boot)
Booking Microservice (Spring Boot)
Rabbit MQ
Booking is subscribed to CheckinQ for check-in updates,
Check-In Microservice (Spring Boot)

As shown in the preceding diagram, we are implementing four microservices as an example: Search, Fare, Booking, and Check-in.

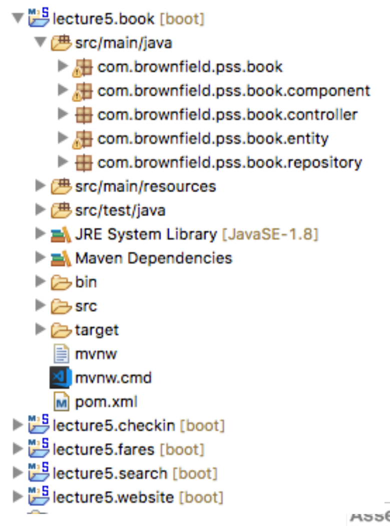
Implementation projects

- The basic implementation of the BrownField Airline's PSS microservices system has five core projects as summarized in the following table

Microservice	Projects	Port Range
Book msvc	lecture5.book	8060-8069
Check-in msvc	lecture5.checkin	8070-8079
Fare msvc	lecture5.fares	8080-8089
Search msvc	lecture5.search	8090-8099
Website	lecture5.website	8001

Implementation projects

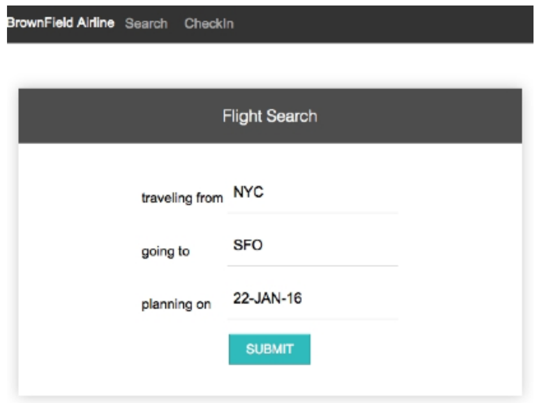
- The root folder (com.brownfield.pss.book) contains the default Spring Boot application.
- The component package hosts all the service components where the business logic is implemented.
- The controller package hosts the REST endpoints and the messaging endpoints. Controller classes internally utilize the component classes for execution.
- The entity package contains the JPA entity classes for mapping to the database tables.
- Repository classes are packaged inside the repository package, and are based on Spring Data JPA.



```
▼ lecture5.book [boot]
  ▼ src/main/java
    ▶ com.brownfield.pss.book
    ▶ com.brownfield.pss.book.component
    ▶ com.brownfield.pss.book.controller
    ▶ com.brownfield.pss.book.entity
    ▶ com.brownfield.pss.book.repository
  ▶ src/main/resources
  ▶ src/test/java
  ▶ JRE System Library [JavaSE-1.8]
  ▶ Maven Dependencies
  ▶ bin
  ▶ src
  ▶ target
    mvnw
    mvnw.cmd
    pom.xml
▶ lecture5.checkin [boot]
▶ lecture5.fares [boot]
▶ lecture5.search [boot]
▶ lecture5.website [boot]
```

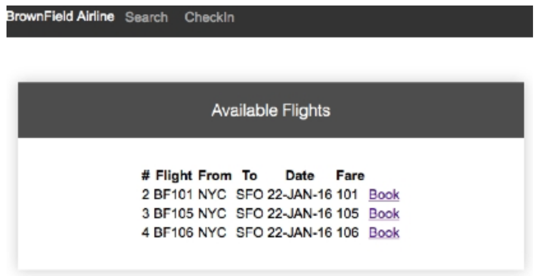
Running and testing the project

```
Follow below steps to run the application.

1. Open terminal Window and move to project home.
mvn -Dmaven.test.skip=true install

2. Start Rabbit MQ.
rabbitmq-server

3. Run below commands from the respective project folders, in separate
terminal windows.
java -jar lecture5.fares/target/fares-1.0.jar
java -jar lecture5.search/target/search-1.0.jar
java -jar lecture5.checkin/target/checkin-1.0.jar
java -jar lecture5.book/target/book-1.0.jar
java -jar lecture5.website/target/website-1.0.jar

4. Open Browser Window and paste below URL.
http://localhost:8001

5. When asked for credentials use guest/guest123

6. Click Search Menu for search and Booking

7. Click CheckIn Menu for check-in
```



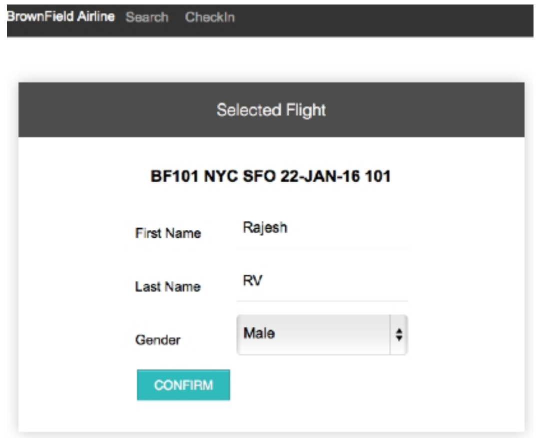
traveling from NYC
going to SFO
planning on 22-JAN-16

Booking screen


BrownField Airline Search CheckIn

Available Flights

#	Flight	From	To	Date	Fare	
2	BF101	NYC	SFO	22-JAN-16	101	Book
3	BF105	NYC	SFO	22-JAN-16	105	Book
4	BF106	NYC	SFO	22-JAN-16	106	Book


BrownField Airline Search CheckIn

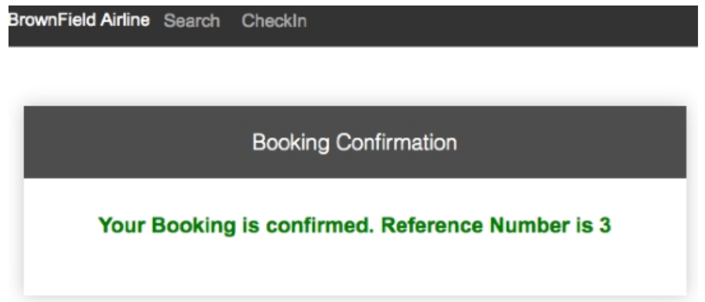
Selected Flight

BF101 NYC SFO 22-JAN-16 101

First Name	Rajesh
Last Name	RV
Gender	Male

CONFIRM

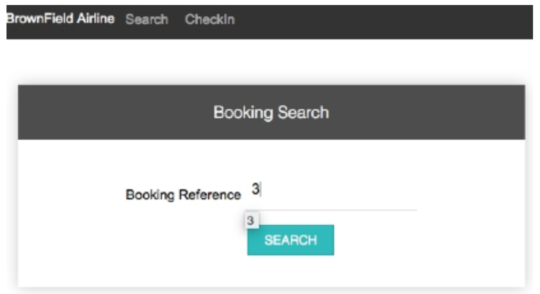
Booking screen

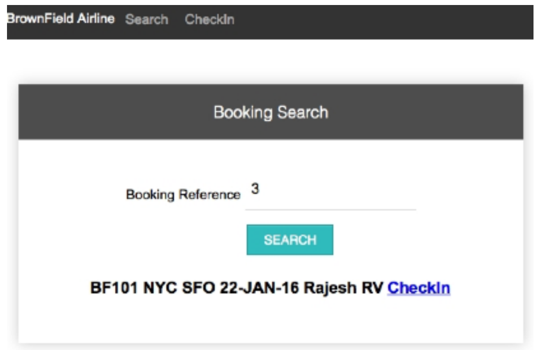

BrownField Airline Search CheckIn

Booking Confirmation

Your Booking is confirmed. Reference Number is 3

Check-in microservice





BF101 NYC SFO 22-JAN-16 Rajesh RV CheckIn

Homework 5.2

- Try to running the BrownField PSS microservices system