



MICROSERVICES

DR. ISAAC GRIFFITH

IDAHO STATE UNIVERSITY

After today's lecture you will:

- Have an understanding of Microservices
  - Specifically how Microservices differ from Components
- Have an understanding of the Microservices Lite Practice including:
  - Microservices Lite Alphas
  - Microservices Lite Products
  - Microservices Lite Activities
- Understand the the Value of the Kernel to the Microservices Practice
- Understand the impact of Microservices Lite Practice for the team



# Microservices

---

CS 3321



- Developing software with **microservices is an evolution of designing software with components**
  - which facilitates a modular approach to building the software system.
- A **component** can be described as an element containing code and data.
  - Only the code “inside” the component (object) can modify the data inside the object and it does that when another component sends a message with a request to do that.
  - This idea is known as “**data hiding**” (data is hidden to other components) and is an accepted best practice in developing maintainable software.
- What Microservices has added is support for components all the way from design, code to deployment

# Components vs. Microservices



- Like components
  - microservices are interconnected via interfaces over which messages are sent to allow communication.
  - each microservice can evolve separately from other microservices, thus making it easy to introduce new functionality
- In a software system built from microservice, each microservice runs a unique executing process.
  - There may be several such executions or instances of the same program running in parallel.
- What microservices bring to software beyond what components already did is the **ability to also deploy the microservices independently without stopping the execution of the entire software system.**

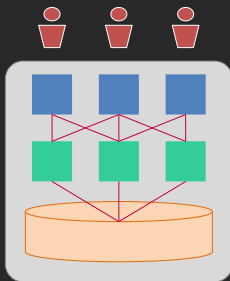
# Microservices Big Picture



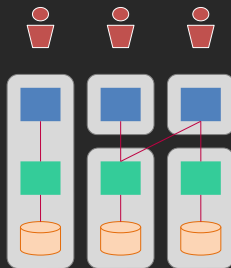
Idaho State  
University

Computer  
Science

Monolithic Architecture



Microservice Architecture



## Legend



User



User Interface



Application Logic



Datastore



Container

- **User Interface** – A user interface is the part of a software system that users interact with. It is the screens, and buttons, and so on.
- **Application Logic** – The code behind the user interface that performs computation, move data around, etc.
- **Data Store** – The data retrievable by the application logic lives in a data store.
- **Containers** – Containers are components of a software system that can be managed separately (i.e. started, stopped, upgraded, and so on)

# Advantages of Microservices



- Each microservice runs
  - as a separate process,
  - possibly in its own container or virtual machine
  - it has its own programming language, user interface, application logic and data store.
- This architecture allow developers to upgrade each microservice independently
  - For example you can upgraded a microservices from Java 8 to Java 9 or a data store without impacting other microservices.
  - If however, code for different logical software element were to run in the same process or virtual machine, an upgrade of one element may inadvertently impact another element.
  - Thus, **enhancing the functionality of an existing microservice is easier** than that of a monolithic software system.

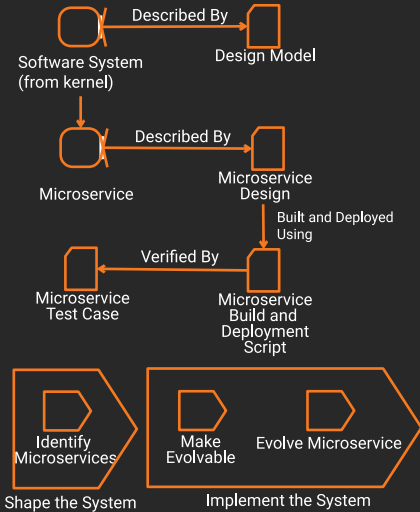
- For a deep dive on Microservices other literature provides complete and detailed presentations
- Our **goal is to show how to essentialize a Practice**
  - This is a Lite practice because we have selected what we deem as a minimal core of the practice;
    - minimal for demonstration but in the real world would require much more technical detail
- How to describe Microservices Lite Practice using Essence?
  - The first questions we always ask when essentializing a practice are:
    - What are the things you need to work with?
    - What are the activities you do?





# Microservice Model

- The **Software System** is described by a **Design Model** and decomposed into **Microservices**
- Each Microservice is:
  - described by a **Microservice Design**
  - built and deployed using **Microservice Build and Deployment Script**
  - Verified by a **Microservice Test Case**
- The practice has several activities:
  1. Identify Microservices
  2. Make Evolvable
  3. Evolve Microservice



# Microservice Elements

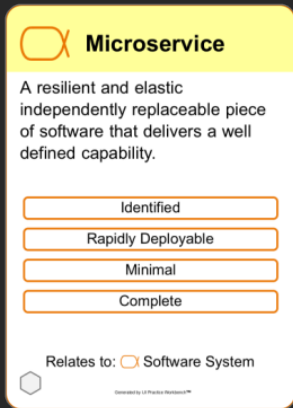
---

CS 3321

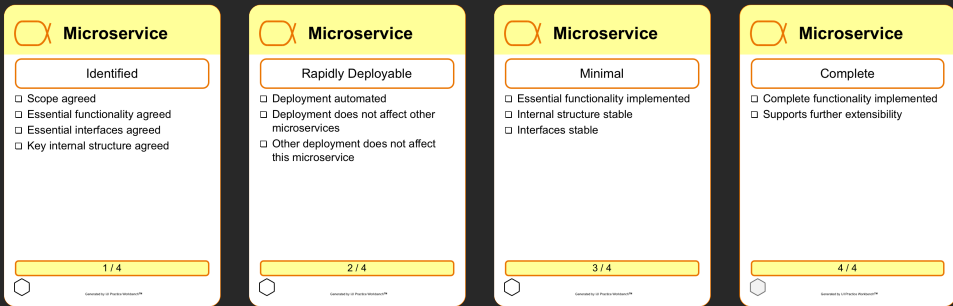
The Microservice alpha represents a resilient and elastic piece of the software (i.e. part of the Software System) that delivers a well-defined capability

Microservices progresses through the following states:

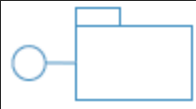


- **Identified** – The scope of the microservice as well as the functionality delivered must be clear.
- **Rapidly Deployable** – A major goal is the ability to quickly change it and re-deploy it to the production environment. It requires low coupling and high cohesion as well as build and deployment scripts
- **Minimal** – team work iteratively to realize the required functionality of the microservice starting with a minimal set of requirements so that it can be integrated and tested in collaboration with other microservices
- **Complete** – fulfilled all the required interfaces of the microservice. Also, refined the structure of the microservice so that it is extensible.



# Microservice Alpha States



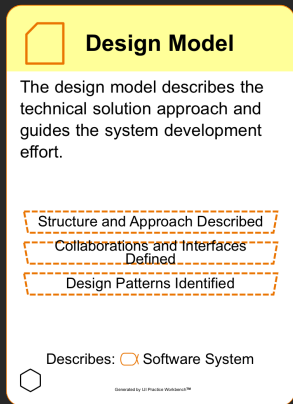
- The alpha state cards above help understand and agree when the Alpha has achieved a state with specific checklists

Element Type	Notation	Description
Subsystem		Looks like a Folder, a package contains a group of code or classes to offer some functionalities, which clients might want to invoke
Provided Interface		Looks like a lollipop, it is the outward functionality, which a subsystem offers. When using the functionality, the client does not need to understand what goes on within the subsystem
Required Interface		Looks like a socket. You can plug in stuff into the socket to provide more functionality into a subsystem. For example, on your computer, you can plug-in hard disks, cameras, etc into your USB port.

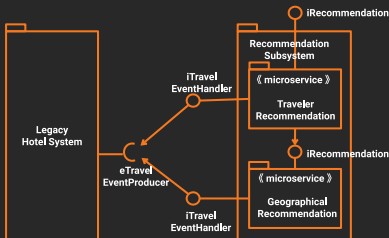
The Design Model is a description of the Software System. It depicts the elements in the Software System and how they interact with one another.

A Design Model can be at different levels of detail:

- **Structure and Approach Described** – the Design Model describes clearly the elements the Software System. It will describe how the different parts are organized and the purpose of each part
- **Collaborations and Interfaces Defined** – the roles and responsibilities of each part are more detailed
- **Design Patterns identified** – common design patterns that can be shared across elements are identified and described. A design pattern is a common design solution to a common design problem



# Recommendation Design



- The legacy hotel system provide an interface, referred to an **eTravelEvent** used to push out traveler events
- The Recommendation subsystem is designed using two microservices
  - the **Traveler Recommendation** microservice dealing with specific travelers or groups of travelers,
  - **Geographical Recommendation** microservice dealing with specific recommendations related to the traveler's current or near term planned geographical locations
- Once the system is deployed there would be multiple microservice processes running
  - one for each traveler group and for each geographical area
- each with its own datastore for Traveler Recommendations.

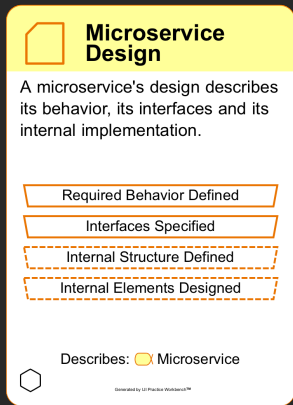
# Microservice Design Product



The Microservice Design is a work product describing the design of a microservice, from its interfaces to its behavior and internal design.

A level of details for this work product are:

- **Required Behavior Defined** – the scope of a microservice is described in words.
- **Interfaces Specified** – the scope of the microservice is described using interfaces.
- **Internal Structure Defined** – Once the external behavior is agreed, this level of detail describes the elements within the microservice. Developers can then start to write code with a good understanding of this structure.
- **Internal Elements Designed** – For complex microservices and elements, therein, more details are needed.

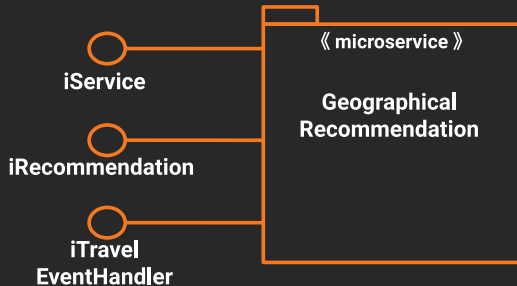




# Geographical Recommender Design



- In addition to the interfaces highlighted in the design model,
- a microservice also needs interfaces to manage its execution,
  - such as setting configuration parameters, and controlling its execution (starting, pausing, resuming, stopping, reset.)
- The Geographic Recommendation microservice has
  - `iService` to manage its execution
  - `iRecommendations` to manage its recommendations
  - `iTravelerEventHandler` to handle traveler events

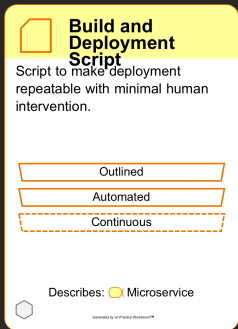


# Build and Deploy Script Product



The Build and Deployment Script is an automated script that supports rapid production and deployment of each microservice independently

- To make the build and deployment process as repeatable as possible, otherwise there is no real advantage of using microservices



A level of details for this work product are:

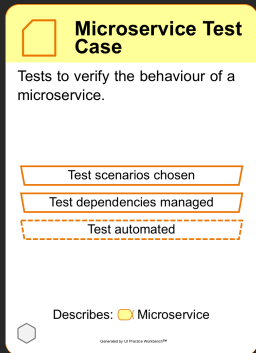
- **Outlined** – there is an agreement as to what rapidly evolvable entails and steps to achieve it are agreed and described.
  - There is not an actual runnable script available yet.
- **Automated** – This is where the real work has been done.
  - The team has written the actual build and deployment script and it has made sure that it works within the development and deployment environment.
- **Continuous** – This is a higher level of detail that ensures that the script runs in continuous support of microservice upgrades without disruption to other microservices.

# Microservice Test Case Product



The testing of a microservice follows a similar approach to testing user stories, and use case slices

- The execution of a microservice will likely depend on other microservices. So, you might have to mock out the surrounding dependencies



A level of details for this work product are:

- **Test Scenarios Chosen** – different scenarios in which the microservice is used are enumerated systematically and prioritized.
- **Test Dependencies Managed** – the scope for each test case is agreed, including dependencies, which will be mocked or stubbed.
  - Mocked means to create extra test code that simulates the other side of the interface. Stubbed means rather than simulate the interface to just ensure the test code will execute without the program crashing.
- **Test Automated** – At this level of detail, the test cases are scripted and automated. They usually run as part of the build and deployment process.

# Identify Microservices Activity



Idaho State  
University

Computer  
Science



## Identify Microservices

Identify the services and their responsibilities required to support emerging functionalities.

☐ Requirements: Bounded

Shape the System



Analysis



Development

☐ Software System: Architecture Selected



Design Model: Structure and Approach Described



Microservice: Identified



Microservice Design: Required Behavior Defined

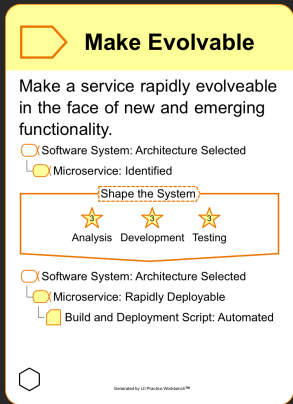


Generated by US Practice Workbench™

- Identifying microservices requires both the Development and Analysis competencies.
  - These competencies are needed to identify microservices that exhibit high cohesion, low coupling and well-defined interfaces.
  - Identifying microservices with these characteristics helps teams achieve the Software System alpha, Architecture Selected state
- The card indicates that:
  - The *Software System* alpha is at **Architecture Selected** state and Design Model product details are at **Structure and Approach described**
  - Microservice* Alpha achieves the **Identified** state by having
    - The *Microservice Design* Product details at **Required Behavior Defined**

# Make Evolvable Activity

Make Microservice Evolvable activity is about the **ability to make rapid changes to a software system in the production environment.**



Goal is to be able to replace a single microservice quickly without affecting other microservices, ensuring

- the modularity and extensibility of each microservice
  - so that requirements changes are localized to individual microservices,
- development and production environment
  - so that changes to microservices are repeatable, reliable and fast.

This requires the streamlining of the deployment pipeline with **plenty of automation embodied in the Build and Deployment Script** work product

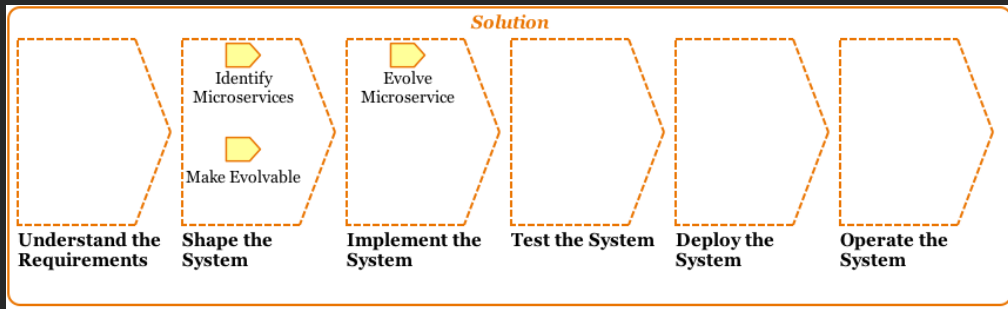
The card indicates that:

- The *Software System* alpha is at **Architecture Selected** state
- *Microservice* alpha achieves the **Rapidly Deployable** state by having
- The *Build and Deployment Script* Product details at **Automated**

# Impact on the Team



- Recall that both User Story Lite and Use Case Lite did not provide any guidance on how to implement the software system.
  - These two practices focused on requirements and tests of the software system.
- The Microservice Lite practice addresses implementation guidance
  - Providing guidance on Shape the System and Implement the System activity spaces



- Having many microservices each running separately raises other problems such as:
  - how to manage and coordinate their execution
  - how to propagate the change of data from one microservice to other microservices
  - how to manage the security of each microservice
  - and so on.
- Cloud providers offer standard mechanisms to solve them.
  - This allows developers to focus on the application, realizing user requirements, push out functionality to the users quickly, get user feedback, and innovate
    - Rather than the low-level infrastructure plumbing which now happens behind the scenes
    - Such rapid cycles are ultimately the value of microservices



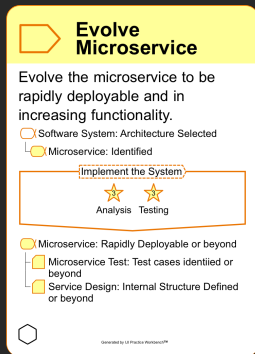
- Smith's team found that the Microservices practice help them provide structure to shape and implement the system
  - What the team realized was that the **Microservice alpha state provided a way to help them plan and implement all the tasks** to get the Software System to a state where the customer would be happy with the result.
- By the time they explicitly applied the Scrum Lite, Use-Case Lite, and the Microservice Lite practices, they had quite a number of alphas additional to the kernel alphas:
  - Sprint, Product Backlog-Item, Use-Case Slice, Use Cases, Microservice
- Someone outside Smith's team was looking at how they were running development with Essence and the practice.
  - He asked "for a small endeavor like ours, isn't this too many things to check? It seems that you have many cards!"
  - The usually quiet Grace was quick to reply, "When we were doing the actual work, **each of us would only focus on achieving a few alpha states**, for example Use-Case Slice Verified, and Microservice Rapidly Deployable. The **states are like micro-checklists** for the small chunks of work we do that could be completed each day. **They help us split the work into small chunks, and gave us a sense of progress** during the day."



# Evolve Microservice Activity



Once a microservice has been made rapidly deployable and its interfaces identified, it becomes straightforward to evolve each microservice, and thereby introduce new functionality to the entire software system.



The card indicates that from:

- The *Software System* alpha is at **Architecture Selected** state
- *Microservice* alpha at **Selected** state

This activity will lead to implement the system by:

- *Microservice* Alpha achieves **Rapidly Deployable** or beyond state
- *Microservice Test* product details at **Identified** or beyond
- *Service Design* product details at **Internal Structure Defined** or beyond

# For Next Time



Idaho State  
University

Computer  
Science

- Review this Lecture
- Come to Class
- Watch Lecture 16 Video





# Are there any questions?