**CS 361 – Module 3 Notes – Architecture and Diagramming**

**Microservices and UML Sequencing**

* **Microservice architectural style** is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. They are built around business capabilities and independently deployable by fully automated deployment machinery.
  + These are compared to **monolithic applications** that put all of their functionality into a single process. Everything works at once and if changes or tweaks are needed, the entire process must be rebuilt. **Microservices** put each element of functionality into a separate service, allowing for easy access and modification with little re-building required. Different services apart of the same service can be written in different languages
* **Common Characteristics of Microservice Architecture**
  + **Componentization via Services**
    - the authors’ definition of a **component** in this case is a unit of software that is independently replaceable and upgradable
    - **Microservices** will use libraries but their main way of componentizing their own software is by breaking down into services. These **libraries** are components that are linked into a program and called using in-memory function calls, while **services** are out-of-process components who communicate with a mechanism such as a web service request or remote procedure call.
    - Using services like these have downsides though, such as remote calls being more expensive than in-process calls and thus, remote APIs need to be coarser-grained, which is often more awkward to use
  + **Organized around Business Capabilities** 
    - Large applications often require multiple teams such as a UI team, database team, server-side logic team, etc. Often times, small simple changes can cause a cross-team project to take more time and require more budget approval. Smart teams will optimize around this and take the lesser of two evils – force the logic into whichever application they have access to.
    - **Microservice** approach to division is different. Generally it splits up into services organized around **business capability**. Teams are cross-functional and have a full range of skills (i.e. the UI team can also work on the database and server-side and vice-versa)
  + **Products, not Projects**
    - Most application development involves a project model, a team to build it and once complete, this software is handed over to a maintenance team and the build team is disbanded
    - **Microservices** tend to avoid this model and prefer a build team own a product over its full lifetime. Similar to Amazon’s notion of “if you build it, you run it” and a dev team takes full responsibility of software in production.
  + **Smart Endpoints and Dumb Pipes**
    - **Microservices** prefer a different approach than the general communication process where structures between different processes put significant smarts into the communication mechanism itself.
    - They tend to use the ***smart endpoints and dumb pipes*** path: they aim to be as decoupled and as cohesive as possible – they own their own domain logic and act more as filters in the classical Unix sense – receiving a request, applying logic as appropriate, and producing a response. REST-ish protocols are generally used.
    - The two protocols most commonly used are the HTTP request-response with resource APIs and lightweight messaging
  + **Decentralized Governance**
    - **MS(microservices)** tend to prefer the idea of producing useful tools that other devs can use to solve similar problems to the ones they are facing. This opposes the common path of following a set of defined standards written down somewhere on paper.
  + **Decentralized Data Management**
    - This idea presents itself in a number of different ways but at the abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view.
    - Monolithic apps prefer one main database for all data, where a **MS** prefers a database for each service, making data modifying and access easier and simpler
    - Although this may seem like the better solution, this kind of path creates problems across multiple services/integration. Distributed transactions are notoriously difficult to implement
  + **Infrastructure Automation**
    - This topic has evolved enormously with the evolution of the cloud and AWS (Amazon Web Services) and reduces the complexity of building, deploying, and operating microservices
    - Many products or services being built with microservices are being built by teams with extensive experience of **Continuous Delivery** (software built in a way that it can be released to production any time)and its precursor, **Continuous Integration** (team members integrating their work frequently, often daily)
    - These topics include automated tests and automated deployment
  + **Design for Failure**
    - A consequence of using services as components is that apps need to be designed so that they can tolerate the failure of services, such as a service call failing or servers going down
    - Constant monitoring and testing is required to spot bad behavior quickly and possible flaws in the program. A **MS** includes sophisticated monitoring and many metrics appearing if things were to go down.
  + **Evolutionary Design**
    - This involves giving devs the opportunity to change things or update their monolith services to microservices, and so forth
* **UML – Unified Modeling Language** – A modeling toolkit that guides the creation and notation of many types of diagrams, including behavior diagrams, interaction diagrams, and structure diagrams
  + A **sequence diagram** is a type of interaction diagram because it describes how – and in what order – a group of objects works together. Used by software devs and business professionals to understand requirements for a new system or to document an existing process.
  + There are two types of diagrams, **UML** and **code-based diagrams**. The latter is sourced from programming code
* **Benefits from Sequence Diagrams**
  + Represent the details of a UML use case
  + Model the logic of a sophisticated procedure, function, or operation
  + See how objects and components interact with each other to complete a process
  + Plan and understand the detailed functionality of an existing or future scenario
* **Use Cases for Sequence Diagrams** – the following scenarios are ideal for using a sequence diagram
  + **Usage scenario:** A diagram of how your system could potentially be used. It’s a great way to make sure that you have worked through the logic of every usage scenario for the system
  + **Method Logic:** Just as you use a UML to explore the logic of use case, it can be used to explore the logic of any function, procedure, or complex process
  + **Service Logic:** Helpful for high-level methods used by different clients
  + **Sequence Diagram Visio:** A sequence diagram software to create the diagrams
* **Basic Symbols**
  + **Object Symbol –** represents a class or object in UML. Demonstrates how an object will behave in the context of the system
  + **Activation Box –** Represents the time needed for an object to complete a task. Longer the task will take, the longer the box is
  + **Actor Symbol –** Shows entities that interact with or are external to the system
  + **Package Symbol –** Used in UML 2.0 notation to contain interactive elements of the diagram. Also known as a frame.
  + **Lifeline Symbol –** Represents the passage of time as it extends downward. Dashed vertical line shows the sequential events that occur to an object during the charted process
  + **Option Loop Symbol –** Used to model if/then scenarios
  + **Alternate Symbol –** Symbolizes a choice (that is usually mutually exclusive) between two or more message sequences. Conditions can also be represented within the symbol.
  + **See rest of symbols in the reading**
* **How to make a sequence diagram**
  + Open a blank document or start with a template
  + To the left of the editor, click “Shapes” to open the Shape Library Manager
  + Check “UML” to enable all of the UML shape libraries or “UML” to enable shapes specific to UML sequence diagrams. Click “Save”
  + Drag the symbols you need from the toolbox to the canvas
  + Then model the process flow by drawing lines between shapes while adding text