**Project Name: Solution Architectural Design**

**Document Overview**

This document outlines the architectural design and considerations for the [Project Name] project. It provides an end-to-end view of the solution architecture, from initial planning to deployment and beyond.

**Table of Contents**

1. **Introduction**
   * Project Overview
   * Purpose of the Document
2. **Business Requirements**
   * Stakeholders and User Groups
   * Functional Requirements
   * Non-Functional Requirements
3. **Architecture Design**
   * High-Level Architecture
   * System Components and Interactions
   * Data Flow Diagrams
4. **Technology Stack**
   * Programming Languages
   * Frameworks and Libraries
   * Databases and Data Storage
5. **Infrastructure Design**
   * Deployment Environment
   * Scalability and Availability
   * Security Measures
6. **Data Management**
   * Data Sources and Integration
   * Data Processing and Storage
   * Data Privacy and Compliance
7. **Integration and APIs**
   * Third-Party Integrations
   * API Design and Documentation
   * Authentication and Authorization
8. **User Experience**
   * User Interface Design
   * Accessibility Considerations
   * User Journeys
9. **Performance and Optimization**
   * Performance Testing
   * Caching Strategies
   * Optimization Techniques
10. **Security Architecture**
    * Threat Analysis
    * Authentication and Authorization
    * Encryption and Data Protection
11. **Monitoring and Logging**
    * Monitoring Tools
    * Logging Guidelines
    * Incident Response Plan
12. **Deployment Strategy**
    * Staging and Production Environments
    * Continuous Integration and Deployment
    * Rollback and Recovery Plan
13. **Maintenance and Support**
    * Ongoing Maintenance Tasks
    * Bug Tracking and Issue Resolution
    * Upgrades and Enhancements
14. **Conclusion**
    * Summary of Architectural Decisions
    * Next Steps in the Project

**Introduction**

This section provides a brief overview of the project, its goals, and the purpose of the document.

**Project Overview**

[Provide a high-level overview of the project, its objectives, and the problem it aims to solve.]

**Purpose of the Document**

[Explain why this document is important, i.e., to communicate the architectural decisions, design rationale, and technical considerations for the project.]

**Business Requirements**

This section outlines the business needs that drive the project.

**Stakeholders and User Groups**

[List the stakeholders involved in the project and the different user groups that will interact with the solution.]

**Functional Requirements**

[List the functional features and capabilities the solution needs to provide.]

**Non-Functional Requirements**

[List the non-functional aspects such as performance, scalability, security, and usability requirements.]

**Architecture Design**

Here, the high-level architecture and key components of the solution are described.

**High-Level Architecture**

[Provide a diagram and description of the system's high-level architecture, including the various components and their interactions.]

**System Components and Interactions**

[List and explain the various components/modules of the system and how they communicate with each other.]

**Data Flow Diagrams**

[Include data flow diagrams to illustrate the movement of data within the system.]

**Technology Stack**

This section outlines the technologies that will be used to build the solution.

**Programming Languages**

[List the programming languages that will be used for various components.]

**Frameworks and Libraries**

[List the frameworks and libraries that will be used to expedite development.]

**Databases and Data Storage**

[Detail the databases and data storage solutions that will be utilized.]

**Infrastructure Design**

This section focuses on the infrastructure aspects of the project.

**Deployment Environment**

[Describe the target deployment environment, such as cloud providers, servers, and networking configurations.]

**Scalability and Availability**

[Explain how the solution will handle scalability and ensure high availability.]

**Security Measures**

[Detail the security measures in place, including firewalls, intrusion detection, and access controls.]

**Data Management**

This section outlines the data-related aspects of the project.

**Data Sources and Integration**

[Explain the sources of data and how they will be integrated into the solution.]

**Data Processing and Storage**

[Detail how data will be processed, transformed, and stored within the system.]

**Data Privacy and Compliance**

[Explain how data privacy regulations and compliance requirements will be met.]

**Integration and APIs**

This section focuses on integration with external systems and API design.

**Third-Party Integrations**

[List the third-party systems or services that the solution will integrate with.]

**API Design and Documentation**

[Explain the design principles for APIs and how they will be documented for external developers.]

**Authentication and Authorization**

[Detail how authentication and authorization will be implemented for APIs.]

**User Experience**

This section covers the user interface and experience design.

**User Interface Design**

[Describe the design approach for the user interface and provide wireframes or mockups.]

**Accessibility Considerations**

[Explain how accessibility standards will be met to ensure an inclusive user experience.]

**User Journeys**

[Outline typical user journeys and interactions within the application.]

**Performance and Optimization**

This section focuses on ensuring the solution's optimal performance.

**Performance Testing**

[Detail the performance testing strategies that will be employed.]

**Caching Strategies**

[Explain how caching mechanisms will be used to enhance performance.]

**Optimization Techniques**

[List techniques for optimizing code, database queries, and other critical components.]

**Security Architecture**

This section outlines the security measures in place to protect the solution.

**Threat Analysis**

[Identify potential threats and risks to the system and how they will be mitigated.]

**Authentication and Authorization**

[Detail the mechanisms for user authentication and authorization.]

**Encryption and Data Protection**

[Explain how sensitive data will be encrypted and protected.]

**Monitoring and Logging**

This section focuses on monitoring the health and performance of the solution.

**Monitoring Tools**

[List the tools that will be used for monitoring system health.]

**Logging Guidelines**

[Explain the logging practices and guidelines for capturing relevant information.]

**Incident Response Plan**

[Detail the plan for addressing and responding to incidents or outages.]

**Deployment Strategy**

This section outlines the deployment process and strategies.

**Staging and Production Environments**

[Explain how staging and production environments will be set up and managed.]

**Continuous Integration and Deployment**

[Describe the CI/CD pipeline and automation for deployment.]

**Rollback and Recovery Plan**

[Detail the plan for rolling back changes and recovering from failures.]

**Maintenance and Support**

This section covers ongoing maintenance and support considerations.

**Ongoing Maintenance Tasks**

[List the regular maintenance tasks required to keep the system running smoothly.]

**Bug Tracking and Issue Resolution**

[Explain how bugs and issues will be tracked and resolved.]

**Upgrades and Enhancements**

[Detail how upgrades and new feature enhancements will be managed.]

**Conclusion**

Summarize the key architectural decisions and next steps in the project.

**Summary of Architectural Decisions**

[Provide a concise summary of the main architectural decisions made.]

**Next Steps in the Project**

[Outline the upcoming phases of the project, including development, testing, and deployment.]

**.NET Azure Cost estimation**

Estimating the cost of a .NET-based Azure microservices architecture involves considering various factors, such as the services you'll use, their configuration, usage patterns, data storage, networking, and more. Azure offers a pay-as-you-go model, so the cost can vary based on your specific requirements. Here are steps to help you estimate the costs:

1. **Identify Services and Resources**: Determine which Azure services you will use for your microservices architecture. This could include Azure Kubernetes Service (AKS) or Azure Service Fabric for orchestration, Azure Functions for serverless computing, Azure Cosmos DB or Azure SQL Database for data storage, Azure API Management for API gateway, and more.
2. **Define Resource Requirements**: For each service, define the required resources, such as the number of VM instances, CPU and memory specifications, storage capacities, etc.
3. **Usage Patterns**: Understand your application's expected usage patterns. Consider factors like the number of requests, data transfer rates, storage usage, and peak loads. Azure bills you based on these usage metrics.
4. **Azure Pricing Calculator**: Use the Azure Pricing Calculator (<https://azure.com/pricing/calculator>) to estimate the costs of various Azure services. Input the details of the resources you plan to use, and the calculator will provide an estimate of your monthly costs.
5. **Azure Cost Management**: Azure offers tools like Azure Cost Management and Azure Advisor that can help you track and manage your spending. These tools provide insights into your actual usage and costs, helping you optimize your expenses.
6. **Reserved Instances and Discounts**: Consider using reserved instances for services like VMs, which offer significant discounts compared to on-demand pricing. Azure also provides various cost-saving options like Azure Hybrid Benefit, which allows you to apply your existing Windows Server licenses to Azure VMs.
7. **Data Storage Costs**: If your microservices architecture involves data storage, understand the pricing model for the chosen storage service (e.g., Azure Blob Storage, Azure Cosmos DB). Costs are often based on factors like storage volume, data transactions, and data egress.
8. **Network Costs**: Consider data transfer costs, especially if your microservices interact extensively. Azure charges for both data ingress (incoming data) and egress (outgoing data), so factor in your expected data transfer volumes.
9. **Monitoring and Logging**: Depending on the level of monitoring and logging you implement, there might be costs associated with services like Azure Monitor and Azure Log Analytics.
10. **Scalability**: Microservices architecture often allows for better scalability. Consider the implications of scaling up or down based on usage. Azure Autoscale features can help manage this dynamically.

Remember that Azure's pricing can change over time, and it's important to regularly review your usage and adjust your estimates accordingly. Additionally, consider engaging with a Azure sales representative or partner to get more personalized cost estimation based on your specific requirements.

**RFP**

Certainly, I can help you understand how to create a Request for Proposal (RFP) for a microservices container-based architecture. An RFP is a document that outlines the requirements and specifications for a project and invites potential vendors or service providers to submit their proposals.

Here's a general outline of what you might include in your RFP for a microservices container-based architecture:

1. **Introduction:**
   * Briefly explain the purpose of the RFP.
   * Provide a high-level overview of the project and its goals.
   * Include important dates such as the deadline for proposal submission.
2. **Background:**
   * Describe the current state of your organization's technology infrastructure.
   * Explain the motivation for adopting a microservices container-based architecture.
   * Highlight any existing challenges that this architecture aims to address.
3. **Scope of Work:**
   * Clearly define the scope of the project. What aspects of your application or system will be built using microservices and containers?
   * Specify the key features and functionalities you expect to be implemented.
   * Include any specific requirements, such as scalability, performance, security, and integration with existing systems.
4. **Technical Requirements:**
   * Detail the technical specifications for the microservices architecture and containerization approach.
   * Specify the programming languages, frameworks, and tools you want to be used.
   * Describe the expected communication patterns between microservices and how data will flow between them.
   * Address requirements for monitoring, logging, and troubleshooting.
5. **Infrastructure and Deployment:**
   * Specify the containerization platform you intend to use (e.g., Docker, Kubernetes) and any preferences for cloud providers.
   * Define the deployment process and considerations for scaling, load balancing, and high availability.
   * Outline your expectations for infrastructure management, including automated scaling and updates.
6. **Security and Compliance:**
   * Define security requirements, such as encryption, authentication, and authorization mechanisms.
   * Address data privacy and compliance concerns, especially if your application deals with sensitive data.
   * Specify any industry-specific regulations or standards that need to be adhered to.
7. **Project Timeline:**
   * Provide an estimated project timeline, including key milestones and deliverable dates.
   * Allow vendors to propose their own timelines if necessary.
8. **Evaluation Criteria:**
   * Outline how proposals will be evaluated. This could include factors like technical expertise, relevant experience, cost, and proposed solution.
   * Define the weighting of each criterion to help potential vendors understand your priorities.
9. **Submission Guidelines:**
   * Provide instructions on how vendors should structure and format their proposals.
   * Specify the required documents, such as technical architecture diagrams, cost breakdowns, and references.
10. **Contact Information:**
    * Provide a point of contact for vendors to ask questions or seek clarification.
11. **Budget and Payment Terms:**
    * Include details about your budget range and any preferred payment terms.
    * Be open to negotiations and provide vendors with an opportunity to propose different pricing models.

Azure Microservices using .NET generally involves building and deploying small, independently deployable services that work together to deliver a complete application. These services are designed to be modular, scalable, and maintainable. Microsoft provides various tools and services that facilitate building microservices-based applications on the Azure platform.

Here's a general outline of how you might approach building Azure microservices with .NET:

1. **Service Development**: Each microservice is developed as a separate application. This can be done using .NET Core or .NET 5+ (now .NET 6+). These services typically have a well-defined API that they expose for communication with other services.
2. **Containerization**: To ensure consistency and portability, each microservice is often containerized using technologies like Docker. This helps in packaging the application along with its dependencies.
3. **Azure Kubernetes Service (AKS)**: Microsoft's Azure Kubernetes Service is a popular choice for orchestrating and managing containerized applications. AKS allows you to deploy, scale, and manage containerized applications using Kubernetes.
4. **Azure Service Fabric**: Azure Service Fabric is another option for building microservices-based applications. It provides tools for deploying and managing microservices, whether they're stateless or stateful.
5. **API Management**: Azure API Management helps you manage your APIs and provide a gateway for external consumers to access your microservices. It can handle authentication, rate limiting, and other API-related tasks.
6. **Azure Functions**: While not traditional microservices, Azure Functions provide a serverless option for executing code in response to events. They can be used for specific tasks within your architecture

**.NET Core best practices:**

1. **Target the Latest Version**: Always aim to use the latest version of .NET, as it includes performance improvements, security updates, and new features. .NET follows a release cycle, and staying up to date ensures you are benefiting from the latest advancements.
2. **Use Dependency Injection**: .NET Core promotes the use of built-in dependency injection. Design your classes with dependency injection in mind, as it improves testability, maintainability, and separation of concerns.
3. **Configure Services Properly**: Use the **Startup** class to configure services for your application. This includes setting up your database connections, adding middleware, and configuring logging.
4. **Follow SOLID Principles**: Apply SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to create maintainable and extensible code.
5. **Separation of Concerns**: Organize your code into meaningful layers, such as presentation, business logic, and data access. This makes the codebase more maintainable and easier to understand.
6. **Use Async/Await**: Take advantage of asynchronous programming to improve the responsiveness and scalability of your application. However, be mindful of deadlocks and ensure proper exception handling.
7. **Configure Logging**: Utilize the built-in logging features in .NET Core to capture application events and diagnose issues. Configure different logging levels and log destinations as needed.
8. **Entity Framework Core Best Practices**: If you're using Entity Framework Core for data access, use best practices such as using migrations, avoiding the N+1 query problem, and optimizing database queries.
9. **Applying Security**: Implement security measures like proper authentication and authorization using built-in ASP.NET Core features. Always validate input and sanitize data to prevent security vulnerabilities.
10. **Use Middleware**: ASP.NET Core offers a rich set of middleware for various purposes like routing, authentication, and exception handling. Utilize these to streamline common tasks.
11. **Performance Optimization**: Profile and optimize performance-critical areas using tools like profiling, caching, and asynchronous programming. Also, consider using application performance monitoring tools to identify bottlenecks.
12. **Versioning APIs**: When building APIs, use versioning to manage changes and ensure backward compatibility for clients.
13. **Unit Testing**: Write unit tests for your code to ensure that it behaves as expected. Utilize testing frameworks like xUnit or NUnit, and practice test-driven development (TDD) if possible.
14. **Containerization**: Consider using Docker to containerize your applications. This helps in maintaining consistency across different environments and simplifying deployment.
15. **Continuous Integration and Deployment (CI/CD)**: Implement CI/CD pipelines to automate building, testing, and deploying your applications. Tools like Azure DevOps, Jenkins, and GitHub Actions can be helpful.
16. **Documentation**: Maintain clear and concise documentation for your codebase, APIs, and any libraries or packages you develop.
17. **Error Handling**: Implement proper error handling and graceful degradation to provide users with meaningful error messages and prevent crashes.
18. **Code Reviews**: Encourage code reviews within your team to ensure code quality, share knowledge, and catch potential issues early.
19. **Monitoring and Telemetry**: Use monitoring tools to gather insights into application usage, performance, and errors. Application insights or similar tools can provide valuable data for improving your application over time.
20. **Keep Learning**: Stay up-to-date with the latest .NET Core/.NET updates, tools, and best practices by participating in the .NET community, reading blogs, attending conferences, and exploring relevant online resources.

best practices for developing ASP.NET Web API applications:

1. **Use Attribute Routing**: ASP.NET Web API supports both convention-based routing and attribute-based routing. Attribute routing is more flexible and allows you to define routes directly on the controller actions using attributes like **[Route]** and **[HttpGet]**.
2. **Version Your APIs**: Design your APIs with versioning in mind. Use versioning techniques to manage changes to your API while maintaining backward compatibility for existing clients.
3. **Input Validation and Data Sanitization**: Always validate and sanitize user input to prevent security vulnerabilities like SQL injection and cross-site scripting (XSS). Use data annotations, validation attributes, and input validation libraries.
4. **Use HTTP Verbs Correctly**: Use the appropriate HTTP verbs (GET, POST, PUT, DELETE, etc.) for the intended actions. Following RESTful principles, map actions to the corresponding HTTP verbs.
5. **Model Binding and Validation**: Leverage model binding to automatically bind incoming data to action parameters. Use data annotations for validation rules, and return appropriate HTTP status codes and error responses for validation failures.
6. **Use DTOs (Data Transfer Objects)**: Consider using DTOs to shape the data you expose through your APIs. This provides better control over the data you send and receive, avoiding overexposing sensitive information.
7. **Content Negotiation**: Support multiple data formats (JSON, XML, etc.) using content negotiation. Ensure your API can respond with the format requested by the client.
8. **Paging and Sorting**: Implement paging and sorting for collection resources to improve performance and user experience when dealing with large data sets.
9. **Exception Handling**: Implement a consistent and well-structured exception handling mechanism. Create custom exception classes and use a global exception filter to handle errors gracefully and provide meaningful error responses.
10. **Use HTTP Status Codes Correctly**: Return appropriate HTTP status codes to indicate the result of API requests (e.g., 200 OK, 404 Not Found, 500 Internal Server Error). This helps clients understand the outcome of their requests.
11. **HATEOAS (Hypermedia as the Engine of Application State)**: Consider adding links to related resources in your API responses. This helps clients navigate your API more efficiently and discover available actions.
12. **Authentication and Authorization**: Secure your API endpoints using appropriate authentication and authorization mechanisms. Utilize built-in ASP.NET Core features or third-party libraries like IdentityServer for OAuth/OpenID authentication.
13. **Caching**: Implement caching for frequently requested data to improve performance. Use cache headers and mechanisms like the **MemoryCache** or distributed caching solutions.
14. **Logging and Tracing**: Implement logging and tracing to monitor your API's behavior and diagnose issues. Use a logging framework like Serilog or NLog to capture important events.
15. **Unit Testing**: Write unit tests for your API controllers and services. Use mocking frameworks to isolate components and ensure reliable and consistent testing.
16. **Integration Testing**: Perform integration tests to validate the behavior of your API as a whole. Use tools like **HttpClient** or specialized testing frameworks.
17. **Documentation**: Create clear and comprehensive API documentation using tools like Swagger or OpenAPI. Good documentation helps developers understand how to use your API effectively.
18. **Security Headers**: Implement security headers to protect against common web vulnerabilities like Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).
19. **Dependency Injection**: Use dependency injection to manage dependencies and promote modularity and testability in your codebase.
20. **Continuous Integration and Deployment (CI/CD)**: Implement CI/CD pipelines to automate building, testing, and deploying your API. This ensures consistent and reliable releases

Microservice architecture is a design approach where a software application is built as a collection of loosely coupled and independently deployable services. Each service focuses on a specific business capability and communicates with other services via well-defined APIs. Here are some best practices for designing and implementing a microservice architecture:

1. **Single Responsibility Principle (SRP)**: Design each microservice around a specific business capability or domain. Keep each service focused on its own responsibility, which promotes modularity and maintainability.
2. **Decentralized Data Management**: Aim for separate data storage for each microservice. This avoids tight coupling and allows each service to choose the best-suited database technology for its needs.
3. **API First Design**: Design well-defined APIs for communication between microservices. Consider using REST, gRPC, or GraphQL for clear and standardized communication.
4. **Asynchronous Communication**: Use asynchronous messaging patterns (e.g., message queues, event-driven architecture) to decouple services and handle communication between them. This improves scalability and fault tolerance.
5. **Fault Isolation**: Isolate failures within a single microservice to prevent cascading failures across the entire system. Implement mechanisms like circuit breakers and timeouts to manage failures gracefully.
6. **Infrastructure Automation**: Use infrastructure-as-code (IaC) tools to automate the provisioning and scaling of microservice infrastructure. Tools like Docker and Kubernetes are commonly used for containerization and orchestration.
7. **Continuous Integration and Deployment (CI/CD)**: Implement robust CI/CD pipelines to automate testing, building, and deploying microservices. This ensures fast and reliable delivery of new features and bug fixes.
8. **Independent Deployment**: Each microservice should be deployable independently. This allows teams to release updates to specific services without affecting the entire system.
9. **Monitoring and Observability**: Implement monitoring and observability tools to gain insights into the health, performance, and behavior of your microservices. Use tools like Prometheus, Grafana, and distributed tracing solutions.
10. **Service Discovery**: Use service discovery mechanisms to help microservices locate and communicate with each other dynamically. Tools like Consul, Eureka, or Kubernetes' built-in service discovery can be helpful.
11. **Centralized Configuration**: Centralize configuration management to ensure consistency and make it easier to manage configuration changes across all services.
12. **Stateless Services**: Strive for stateless microservices. Store any necessary state in separate databases or caching layers. This makes scaling and maintenance easier.
13. **Resilience Patterns**: Implement resilience patterns like retry mechanisms, circuit breakers, and timeouts to handle transient failures and maintain system stability.
14. **Scaling Strategies**: Consider auto-scaling based on actual usage patterns. Utilize horizontal scaling to add more instances of a service in response to increased demand.
15. **Security**: Apply security best practices, including proper authentication and authorization mechanisms, encryption for data in transit and at rest, and regular security assessments.
16. **Team Structure**: Organize development teams around microservices, allowing each team to have ownership and accountability for their respective services.
17. **Documentation**: Maintain comprehensive documentation for each microservice, including API documentation, architecture diagrams, and deployment instructions.
18. **Testing Strategies**: Implement unit, integration, and end-to-end tests for your microservices. Use tools like consumer-driven contract testing to ensure API compatibility.
19. **Evolutionary Design**: Embrace the fact that the architecture may evolve over time. Be prepared to refactor and adapt as the system's requirements change.
20. **Domain-Driven Design (DDD)**: Apply DDD principles to model and understand your application's domain. This helps in creating well-defined and meaningful microservices.

Remember that while these best practices provide a strong foundation for designing microservice architectures, the specific implementation may vary based on your organization's needs, the technology stack you're using, and the complexity of your application. Continuously monitor and iterate on your architecture to ensure it meets your goals effectively.

**RFP Response docs**

Responding to a Request for Proposal (RFP) requires a well-structured and comprehensive document that addresses the client's needs, demonstrates your capabilities, and showcases your solution. Here's a template to help you create an effective RFP response:

**[Your Company Name] RFP Response**

**1. Executive Summary:** Briefly summarize your proposed solution, highlighting key benefits and how your company is uniquely positioned to meet the client's needs.

**2. Introduction:** Introduce your company, its history, expertise, and relevant experience.

**3. Understanding of Requirements:** Demonstrate a clear understanding of the client's requirements and challenges. Highlight any specific pain points mentioned in the RFP.

**4. Proposed Solution:** Detail your proposed solution and how it directly addresses the client's needs. Include information on technology, methodology, and tools you plan to use.

**5. Key Features and Benefits:** List the features of your solution that align with the client's requirements. Explain how these features will benefit the client's business.

**6. Project Timeline:** Present a detailed timeline that outlines the project phases, milestones, and estimated completion dates.

**7. Team Expertise:** Introduce the team members who will be working on the project, including their relevant experience and roles.

**8. Case Studies/References:** Provide examples of similar projects you've successfully completed. Include testimonials or references from clients who have benefited from your solutions.

**9. Pricing:** Present a clear and transparent pricing structure. If applicable, break down costs by project phase or service.

**10. Implementation Approach:** Explain how you plan to implement the solution, from initial requirements gathering to final delivery.

**11. Support and Maintenance:** Outline your post-implementation support and maintenance offerings. Include details about response times, issue resolution, and ongoing updates.

**12. Quality Assurance and Testing:** Describe your quality assurance and testing processes to ensure the solution's reliability and accuracy.

**13. Risk Management:** Identify potential risks associated with the project and provide strategies for mitigating or managing them.

**14. Compliance and Security:** Highlight any industry standards, compliance certifications, or security measures your solution adheres to.

**15. Sustainability and Scalability:** Explain how your solution can evolve with the client's needs and how it contributes to long-term sustainability.

**16. Conclusion:** Reiterate your enthusiasm for the project and your commitment to delivering a successful solution.

**17. Appendices:** Include any additional materials that support your proposal, such as detailed technical specifications, diagrams, or legal documents.

**18. Contact Information:** Provide contact details for your sales representative or project lead for any further inquiries.

Remember to tailor each RFP response to the specific needs outlined in the RFP and to showcase how your solution aligns with the client's goals. Proofread your response for clarity, accuracy, and professionalism before submitting it.

Top of Form