CSCI427/927 Systems Development

Service-oriented architectural patterns
Microservices (cont.)

Acknowledgement: Materials on these slides are adapted from "Engineering Software Products: An Introduction to Modern Software Engineering" by Ian Sommerville

RESTful services

- REST (REpresentational State Transfer): One of the most used approaches for service interaction
  - **Transferring representations of digital resources** from **a server** to a **client**.
  - You can think of **a resource as any chunk of data** such as credit card details, an individual’s medical record.
  - Resources are accessed via their **unique Uniform Resource Identifier (URI)** and RESTful services operate on these resources.

- This is the **fundamental approach used in the web** where the resource is a page to be displayed in the user’s browser.
  - An HTML representation is generated by the server in response to an HTTP GET request and is transferred to the client for display by a browser or a special-purpose app.

RESTful service principles

- ***Use HTTP verbs***
  The basic methods defined in the **HTTP protocol** (GET, PUT, POST, DELETE) must be used to access the operations made available by the service.
- ***Stateless services***
  Services must never maintain internal state.
- ***URI addressable***
  All resources must have a URI, with a **hierarchical structure**, that is used to access sub-resources.
- ***Use XML or JSON***
  Resources should normally be represented in JSON or XML or both. Other representations, such as audio and video representations, may be used if appropriate.

RESTful service operations

- ***Create***
  Implemented using **HTTP POST**, which creates the resource with the given URI. If the resource has already been created, an error is returned.

- ***Read***
  Implemented using **HTTP GET**, which reads the resource and returns its value. GET operations should never update a resource so that successive GET operations with no intervening PUT operations always return the same value.

- ***Update***
  Implemented using **HTTP PUT**, which modifies an existing resource. PUT should not be used for resource creation.

- ***Delete***
  Implemented using **HTTP DELETE**, which makes the resource inaccessible using the specified URI. The resource may or may not be physically deleted.

Road information system

- Imagine a system that maintains **information about incidents**, such as traffic delays, roadworks and accidents on a national road network. This system can be accessed via a browser using the URL:
  - https://trafficinfo.net/incidents/
- **Users can query the system** to discover incidents on the roads on which they are planning to travel.
- When implemented as a RESTful web service, you need to design the resource structure so that **incidents are organized hierarchically**.
  - For example, incidents may be recorded according to the road identifier (e.g. A90), the location (e.g. stonehaven), the carriageway direction (e.g. north) and an incident number (e.g. 1). Therefore, each incident can be accessed using its URI:
  - https://trafficinfo.net/incidents/A90/stonehaven/north/1

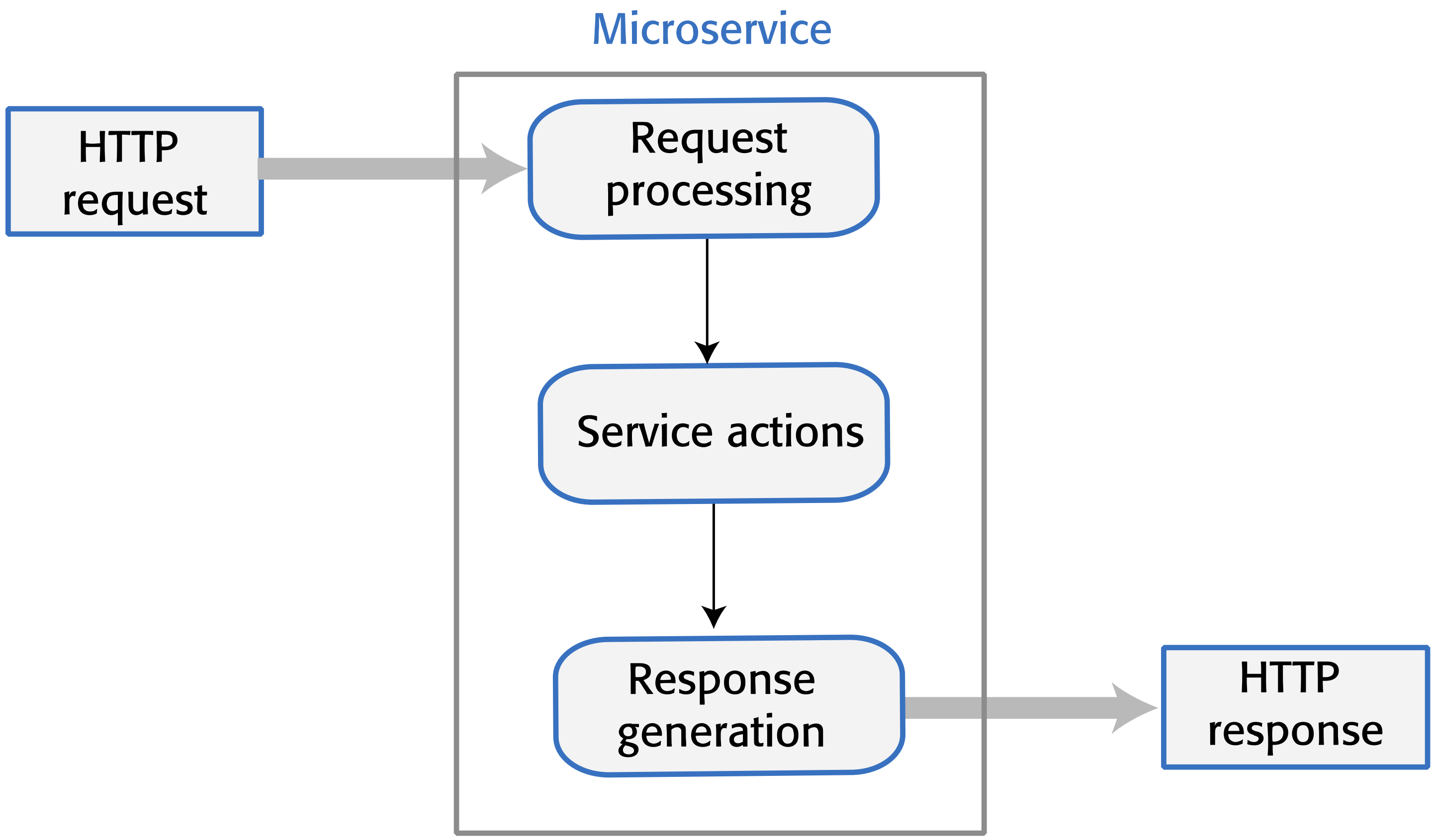
Incident description

- Incident ID: A90N17061714391
- Date: 17 June 2017
- Time reported: 1439
- Severity: Significant
- Description: Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes

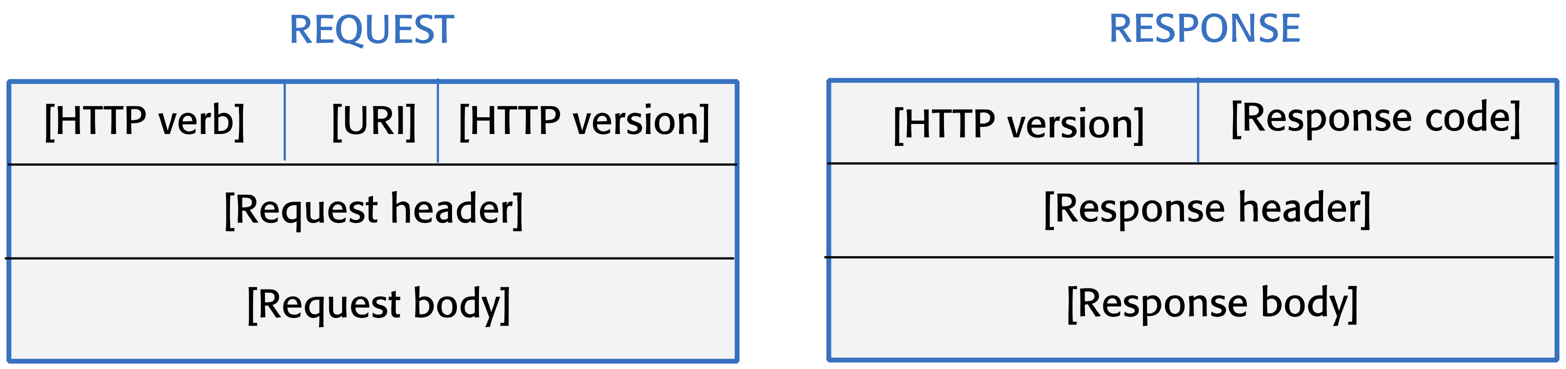
Service operations

- Retrieve
  - Returns information about a reported incident or incidents. Accessed using the **GET** verb.
- Add
  - Adds information about a new incident. Accessed using the **POST** verb.
- Update
  - Updates the information about a reported incident. Accessed using the **PUT** verb.
- Delete
  - Deletes an incident. The **DELETE** verb is used when an incident has been cleared.

HTTP request and response processing



HTTP request and response message organization



XML and JSON descriptions

- ***JSON***
- {
  id: “A90N17061714391”,
  “date”: “20170617”,
  “time”: “1437”,
  “road_id”: “A90”,
  “place”: “Stonehaven”,
  “direction”: “north”,
  “severity”: “significant”,
  “description”: “Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.”
  }

XML and JSON descriptions

- ***XML***
- <id>
  A90N17061714391
  </id>
  <date>
  20170617
  </date>
  <time>
  1437
  </time>
  …
  <description>Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.
  </description>

A GET request and the associated response

REQUEST

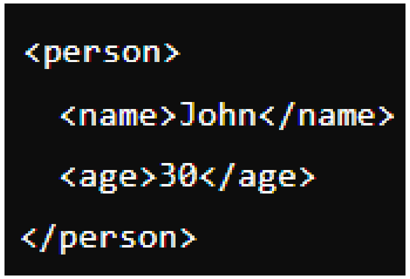
```
GET | incidents/A90/stonehaven/ | HTTP/1.1
Host: trafficinfo.net
...
Accept: text/json, text/xml, text/plain
Content-Length: 0
```

RESPONSE

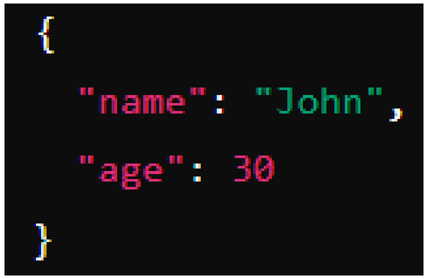
```
HTTP/1.1 | 200
...
Content-Length: 461
Content-Type: text/json

{
   “number”: “A90N17061714391”,
   “date”: “20170617”,
   “time”: “1437”,
  “road_id”: “A90”,
  “place”: “Stonehaven”,
  “direction”: “north”,
  “severity”: “significant”,
  “description”: “Broken-down bus on north
   carriageway. One lane closed. Expect delays
of up to 30 minutes.”
}
{
   “number”: “A90S17061713001”,
   “date”: “20170617”,
   “time”: “1300”,
  “road_id”: “A90”,
  “place”: “Stonehaven”,
  “direction”: “south”,
  “severity”: “minor”,
  “description”: “Grass cutting on verge. Minor
delays”
}
```

JSON VS XML

- **Syntax Overhead =>** File Size



```
<person>
  <name>John</name>
  <age>30</age>
</person>
```



```
{
  "name": "John",
  "age": 30
}
```

- **Data Types:** XML doesn't have native data types (e.g., numbers, strings, booleans), but text. JSON natively supports data types such as strings, numbers, booleans, arrays, and objects.
- **Readability and Parsing**: verbose tagging (opening and closing) in XML makes it harder to read and more resource-intensive to parse, especially for large documents

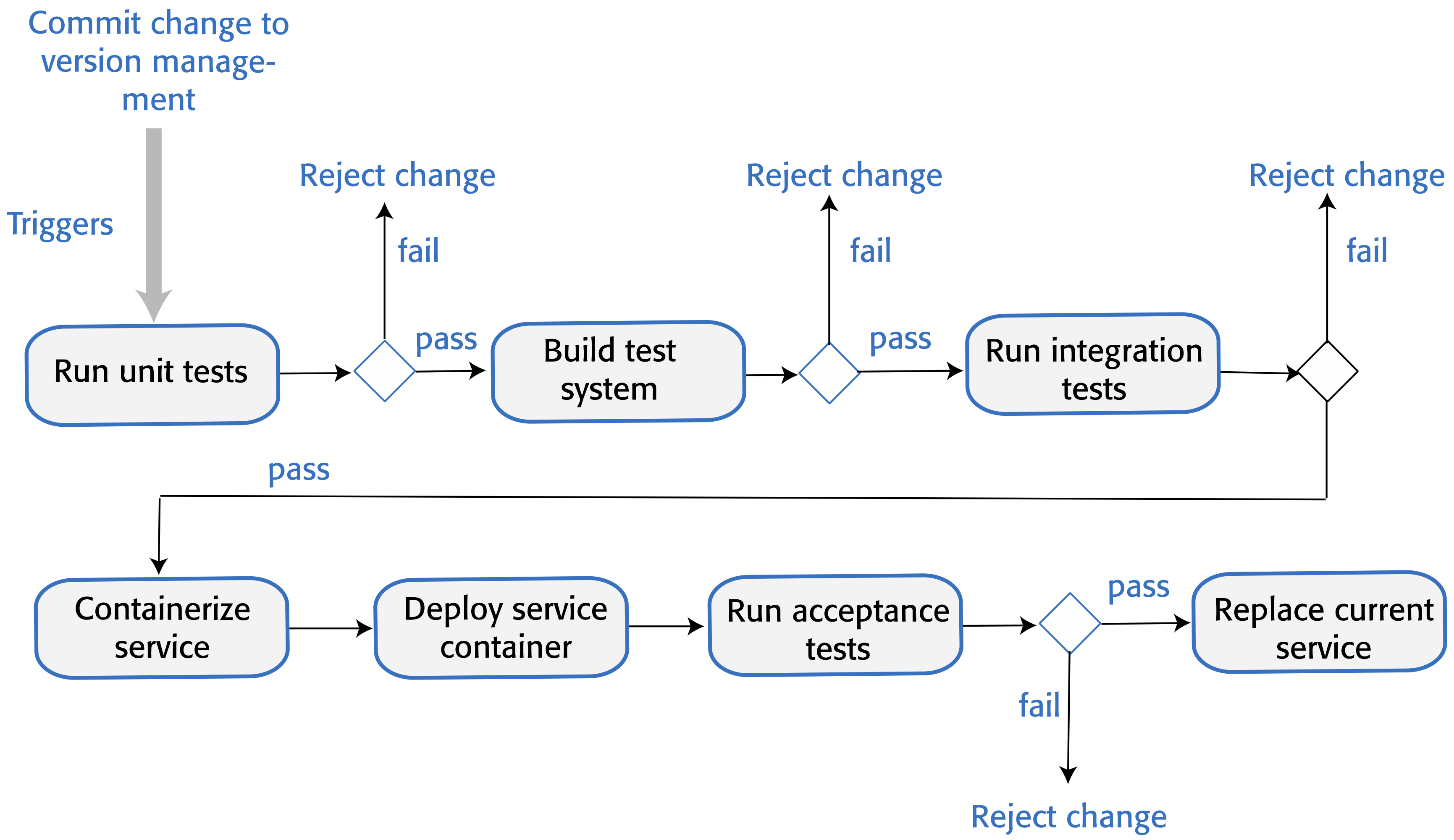
Service deployment

- After a system has been **developed and delivered**, it has to be **deployed** on servers, monitored for problems and updated as new versions become available.
- When a system is composed of tens or even hundreds of microservices, deployment of the system is more **complex** than for monolithic systems.
- The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is **no 'standard' deployment configuration** for all services.
- It is now normal practice for **same development teams** to be responsible for deployment and service management as well as software development and to use continuous deployment.
- **Continuous deployment** means that as soon as a change to a service has been made and validated, the modified service is redeployed.

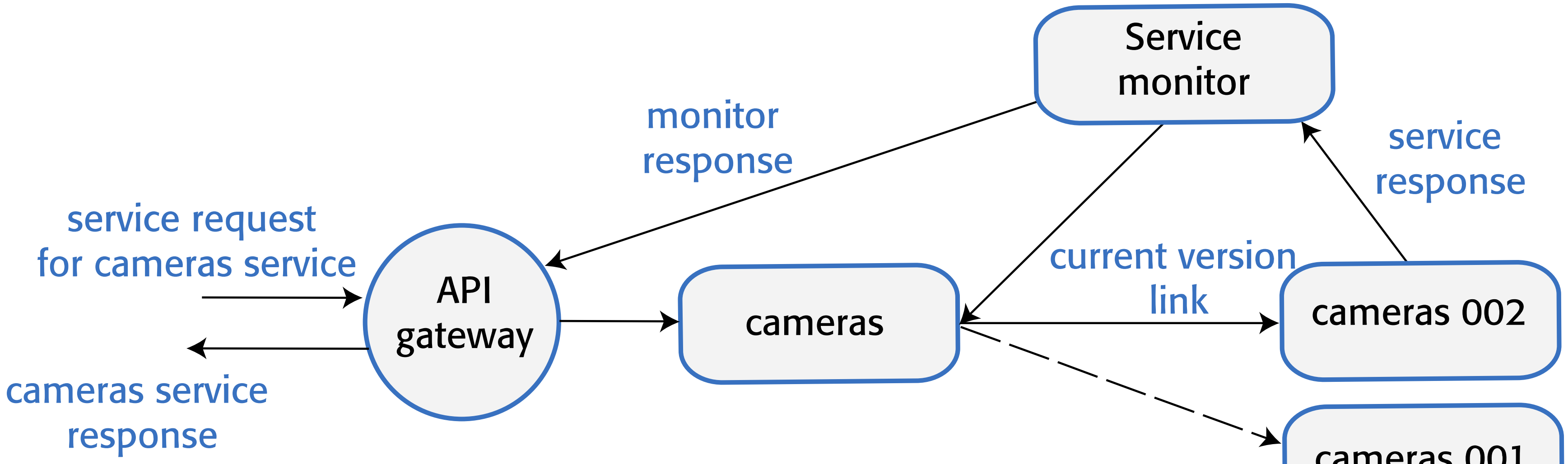
Deployment automation

- **Continuous deployment** depends on automation so that as soon as a **change is committed**, a series of automated activities is triggered to test the software.
- If the software **'passes' these tests**, it then enters another automation pipeline that packages and deploys the software.
- The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git.
- This triggers a set of automated tests that run using the modified service. If all service tests run successfully, a new version of the system that incorporates the changed service is created.
- Another set of automated system tests are then executed. If these run successfully, the service is ready for deployment.

A continuous deployment pipeline



Versioned services



Summary

- A microservice is an **independent** and **self-contained** software component that runs in its own process and communicates with other microservices using lightweight protocols.

- Microservices in a system can be implemented using **different programming languages** and **database technologies**.

- Microservices have a **single responsibility** and should be designed so that they can be **easily changed** without having to change other microservices in the system.

- Microservices architecture is an **architectural style** in which the system is constructed from communicating microservices. It is well-suited to **cloud based systems** where each microservice can run in its **own container**.

- The two most important responsibilities of architects of a microservices system are to decide **how to structure the system into microservices** and to decide how microservices should **communicate** and be **coordinated**.

Summary (cont.)

- Communication and coordination decisions include deciding on microservice **communication protocols**, **data sharing**, whether services should be **centrally coordinated**, and **failure management**.

- The **RESTful** architectural style is widely used in microservice-based systems. Services are designed so that the HTTP verbs, GET, POST, PUT and DELETE, map onto the service operations.

- The RESTful style is based on **digital resources** that, in a microservices architecture, may be represented using **XML** or, more commonly, **JSON**.

- **Continuous deployment** is a process where new versions of a service are put into production as soon as a service change has been made. It is a completely **automated process** that relies on automated testing to check that the new version is of ‘production quality’.

- If continuous deployment is used, you may need to maintain **multiple versions** of deployed services so that you can switch to an older version if problems are discovered in a newly-deployed service.