University of St Andrews School of Computer Science

CS2003 — Internet and the Web — 2025/26

Assignment: W11
Deadline: 2025-11-26 Credits: 50% of coursework mark (30% of module mark)

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

The aim of this coursework is to build a ticket purchasing system for Ticket Chief, a large company that sells tickets for concerts over the World Wide Web. The user (customer) must be able to purchase tickets for their favourite musicians and bands. Since many bands are very popular, users are placed in a queue. The queue may take some time but eventually, when they are first in the queue, they will be able to purchase a ticket. Multiple clients must be able to queue at the same time. You are required to create both the client-side and server-side components of the system. The client will be developed using JavaScript, and the server will be implemented in Java. You are provided with the following resources:

- HTML and CSS files for the front-end, supplied by the design team at Ticket Chief to get you started.
- A specification for the initial RESTful API that defines how the client and server should interact. You will have to modify this for later parts of the coursework. The initial version, used in part 1, supports purchasing a ticket single type of ticket for a single event (such as a concert or gig).
- A JSON file containing a set of tickets that are on sale, which should be used to populate the server with initial data.

This assignment is split into several parts of increasing complexity. It is essential to complete each part in sequence and ensure the solutions to the more

basic tasks are correct before progressing to advanced tasks. Submit all parts as a single ZIP file, with subdirectories for each part that you have attempted.

The learning objectives of this coursework are to gain experience with building client and server systems in different languages, with implementing an API specification, and to better understand a widely-used application-layer protocol (HTTP).

The Ticket Chief API Specification

The Ticket Chief API has two URLs: `/ticket` and `/queue`. These are used to manipulate the two kinds of resouces in the system: tickets and ticket purchase requests.

Tickets

`/tickets` exposes one HTTP method: `GET /tickets` that retrieves the current number of tickets and details about the event.

Request:

- **Accept** header: `application/json`
- Request body: Empty

Response:

- **Content-Type** header: `application/json`
- Response body: A JSON object representing the tickets that can be requested.
  - `count` (number): A count of the number of tickets remaining.
  - `artist` (string, max 64 characters): name of the band or musician.
  - `venue` (string, max 64 characters): the name of the venue.

- `datetime` (ISO 8601 UTC string): The timestamp for the time and date of the gig. You should use the simplified version of ISO 8601 specified in RFC 3339.

Status codes:

- 200: Successfully retrieved ticket information.
- 500: Server error (returns an error message).

Example Request:

```
GET /tickets HTTP/1.1
Accept: application/json
```

Example Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "count": 42,
  "artist": "The Electric Owls",
  "venue": "St Andrews Student Union",
  "datetime": "2025-11-22T19:30:00Z"
}
```

Queue

`/queue` exposes two HTTP methods: `POST /queue` and `GET /queue/:id`. These endpoints manage the queue of ticket purchase requests.

Tickets and request IDs are represented as numbers. A request's `position` indicates its place in the queue; when `position` reaches `0`, the tickets are considered purchased and `ticketIds` is populated.

POST /queue

Create a new ticket purchase request and add it to the queue.

Request:

- **Accept**: `application/json`
- **Content-Type**: `application/json`
- **Body** (JSON):
  - `tickets` (number, required): The number of tickets requested.

Response:

- **Content-Type**: `application/json`
- **Body** (JSON):
  - `id` (number): The unique ID of the newly created ticket purchase request.
- **Headers**:
  - `Location`: `/queue/:id` (URL of the created resource)

Status codes:

- **201 Created**: Request successfully created; returns the `id`.
- **200 OK**: The request was not created because there are no more tickets available.
- **400 Bad Request**: Invalid request (e.g., missing or invalid fields).
- **500 Internal Server Error**: Server error (returns an error message).

Example Request:

```
POST /queue HTTP/1.1
Host: example.com
```

```
Accept: application/json
Content-Type: application/json

{
"tickets": 2
}
```

Example Response (201 Created):

```
HTTP/1.1 201 Created
Location: /queue/57
Content-Type: application/json

{
"id": 57
}
```

GET /queue/:id

Retrieve the full current status of a ticket purchase request.

Request:

- **Accept**: `application/json`
- **Body**: Empty

Response:

- **Content-Type**: `application/json`
- **Body** (JSON):
  - `id` (number): The unique ID of the ticket purchase request.
  - `tickets` (number): The number of tickets requested.
  - `position` (number): The current position of the request in the queue. A value of `0` means the tickets have been purchased.

  - `ticketIds` (array of strings): The IDs of the purchased tickets (populated when `position` is `0`; otherwise must be empty).

Status codes:

- **200 OK**: Successfully retrieved the ticket purchase request.
- **404 Not Found**: No ticket purchase request with the given `id`.
- **500 Internal Server Error**: Server error (returns an error message).

Example Request:

```
GET /queue/57 HTTP/1.1
Accept: application/json
```

Example Response (200 OK):

```
HTTP/1.1 200 OK
Content-Type: application/json

{
"id": 57,
"tickets": 2,
"position": 3,
"ticketIds": []
}
```

Requirements

There are three sets of requirements and you have been provided with starter code for each of these. You should submit a separate set of code for each part that you are attempting. You can reuse code from Part 1 for Part 2 and Part 3, but do not build your implementation of Part 3 on top of Part 2, because the requirements are slightly different.

Requirements – Part 1

Server Implementation

1. **Basic Server:** Implement the server in Java. It must run correctly on the CS lab Linux machines.
2. **Properties Configuration:** Use the provided `cs2003-C3.properties` file for runtime configuration (e.g., port numbers). Do not hard-code parameters such as port numbers, file paths, or usernames. All configuration values should be read from this file. Markers must be able to run your programs on a different machine without having to edit any code.
3. **Serve Public Files:** Your server should serve the files provided in the `starter/public` directory via the endpoint `/`. The exact location of the files should be specified in the properties file as `documentRoot`. When an HTTP client visits the endpoint `/`, it should serve the HTML file `index.html` in its response. If a client sends a `GET` request for a file that exists in `documentRoot`, then the server should serve that file. Otherwise, it should return a `404 Not Found` response.
4. **Load from Storage:** The server should use a provided JSON file to initialise its state at startup. This file contains details about available tickets and the gig (e.g., artist, venue, date/time). You must implement a helper class (for example, `Store`) to read this data from disk at startup and make it accessible to the server while running. New ticket purchase requests created during runtime should be stored in memory only; persistence between server restarts is not required.
5. **Ticket Endpoints:** The server should implement the following endpoints according to the API specification:
    - `GET /tickets` – retrieves the current number of tickets remaining and the gig details.
    - `POST /queue` – submits a new ticket purchase request to the queue.
    - `GET /queue/:id` – retrieves the current status (position) of a specific ticket purchase request in the queue.
6. **Queue Management:** The server must maintain an internal FIFO queue of ticket purchase requests. Each new purchase request added via `POST /queue` should be assigned a unique numeric ID and a queue position. To simulate real-world delays, the server should process each purchase re-

quest after a random delay (e.g., between 5-10 seconds). In other words, clients may not necessarily be added to the queue in the order that they arrive, since each client will have a different random delay. When a request is processed, the number of available tickets should decrease accordingly. If there are no tickets remaining, new requests should return a 200 OK status code with an empty body and not create the resource.

7. **Ticket Identifiers:** Each successfully processed ticket purchase should be assigned a simple unique identifier (e.g., `T-001`, `T-002`, etc.). Clients may later use this identifier to request a refund or check the status of a ticket. Security or authentication is not required for this coursework. The server does not need to remember the ticket identifiers.

Client Implementation

1. **Basic Client:** Implement the client using JavaScript, designed to run in the Firefox web browser. You may use the provided HTML, CSS, and JavaScript files. They can be modified as you see fit. These files can be found in the `starter/public` folder on Studres. files if needed.
2. **Client-Server Interaction:** When the client loads in the browser, it should retrieve the current available tickets from the server by making a `GET /tickets` request. The page should display the number of tickets remaining and relevant gig information.
3. **Join the Queue:** The client should allow users to purchase tickets by submitting a `POST /queue` request. The client should then display confirmation that the request has been added to the queue and show the assigned queue ID.
4. **Check Queue Position:** After joining the queue, the client should periodically poll the server using `GET /queue/:id` to retrieve the user's current position in the queue and display it on the page. The position should update dynamically until the request has been processed (the simulated delay should make this visible).
5. **Client-Server Separation:** It must be possible to run the client on a different machine from the server. You should test your implementation on the CS lab machines and/or teaching servers to verify that client-server communication functions correctly across machines.
6. **Error Handling:** The client should handle and display server errors gracefully (e.g., 400 or 500 responses). If a purchase request fails or the queue

cannot be retrieved, an appropriate message should be shown to the user.

Requirements – Part 2

1. **Cancelling Purchases:** Extend the server API and the client to allow users to cancel their ticket purchase requests. A purchase request can be cancelled only if it has not yet been processed (i.e., before tickets are issued). You will need to add a `DELETE /queue/{id}` endpoint that removes the corresponding request from the queue. The endpoint should use appropriate HTTP status codes and headers consistent with the existing API specification. You will also need to update the client to provide a cancel button for any pending purchase, which sends the appropriate request and updates the displayed queue accordingly. You may need to adjust your queueing delay in order to test this feature appropriately.
2. **Refunding Tickets:** Extend the server API to allow users to return tickets they have already purchased. This should be done using a `POST /tickets/refund` endpoint, which accepts a JSON object containing a list of valid ticket IDs. When the request is successful, the number of available tickets should increase by the corresponding amount. The server does not have to validate that the ticket exists (as we are unconcerned with security in this coursework). Refunds can be initiated from the client interface.
3. **Multiple Concerts:** Extend the data model and server to support multiple concerts. Each concert should have its own artist, venue, date/time, and available ticket count, all specified in the JSON file loaded at startup. The API should be extended to distinguish between concerts. For example:
    - `GET /tickets/{concert_id}` – retrieves ticket information for a specific concert.
    - `POST /queue/{concert_id}` – adds a new purchase request to the queue for the specified concert.
    - `GET /queue/{concert_id}/{id}` – retrieves the queue status for a specific purchase request associated with a given concert.
4. **Client Extensions:** The client should be updated to display multiple concerts. The user should be able to select a concert from a list or dropdown

menu, view its available tickets, and join that concert's queue. Each concert should display its own queue position updates independently.

Requirements – Part 3

For part 3 you will consider (some) security. The aims of security that we covered (or will cover, depending on when you read this) in lectures are confidentiality, integrity and availability. The use of queues in our Ticket Chief system attempts to address availability. But we would also like you to consider the other aims.

1. **Implementation of one security aim:** You should attempt to implement some confidentiality or integrity in the Ticket Chief system. You are permitted to use any standard libraries, but you are not permitted to use TLS, as the aim is to implement security at the application layer. Apart from this, we are not prescriptive about how you attempt this.
2. **Considering of another security aim:** You should document in your report how you might attempt to achieve the other aim of security. You should also discuss some potential attacks on Ticket Chief that might be prevented by considering the three aims of security.

Testing

You should provide evidence of adequate testing of all attempted requirements. You should also test that your programs are robust against unexpected input or protocol violations. To do this, you should use the program `curl` which is installed on the CS Linux machines and was introduced in exercise classes. You may also consider other testing strategies as you see appropriate.

Report

Your report must show evidence of testing, including descriptions of any issues you encountered and you attempted to resolve them.

Your report *must* use the structure outlined in the report_template directory on studres. In particular you should pay attention to how to structure your testing

section.

Hints

- This is a bigger piece of coursework than the previous two, so allocate appropriate time for this.
- We are issuing the specification before everything has been covered in lectures so that you can prepare. But with this in mind, it would be a good idea to attend lectures so that you understand the underlying concepts that you have been asked to implement.
- Do not attempt Parts 2 or 3 until you have completed Part 1.
- We suggest starting with the server and using curl to simulate the client. Look to the exercise classes and tutorials for examples.
- Remember that HTTP is a very simple string-based format. So your server should mainly be reading, parsing and writing strings. Start with a server that can serve a single file (e.g. index.html) with the correct headers.
- Make appropriate use of regular expressions, e.g. for parsing headers.
- Look at the FAQ as we will keep this updated.

Submission

A single file containing your code and PDF report in ZIP format must be submitted electronically via MMS by the deadline. Submissions in any other format will be rejected.

The specific mark descriptors for this piece of coursework are described below. Note that there are no extensions; to achieve excellent marks you should meet all of the requirements with exceptional clarity and demonstrated insight. This includes an excellent report and comprehensive testing.

Assessment Criteria

Marking will follow the guidelines given in the school student handbook (see link in next section).

Some specific descriptors for this assignment are given below:

0–6
A failure to submit any working code or report.

7–10
Code and a report fulfilling part 1 of the requirements, but the code and/or report is poor quality or does not work, or insufficiently meets the requirements.

11–13
Working code and a report fulfilling part 1 of the requirements, but poor testing or poor quality of code or report.

14–16
Good quality code, testing, and a report fulfilling part 1 of the requirements.

17–18
Excellent quality code, along with thorough testing and a well-written report fulfilling parts 1 and 2 of the requirements.

19–20
Excellent quality code, along with thorough testing and a well-written report fulfilling all requirements.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook: https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme A: 1 mark per 24-hour period, or part thereof): https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/education/handbook/good-academic-practice/