ChatGPT

Using Synthea and Synthetic Data Generators in a .NET CLI for HL7 v2, FHIR, and CCDA

Introduction

Synthetic healthcare data generation tools enable developers to create **realistic but fake patient data** for testing, development, and research without privacy concerns [1] [2]. This guide provides a comprehensive overview of **Synthea** – an open-source patient simulator – and similar tools, focusing on how to use them in a .NET C# command-line interface (CLI) to generate data in multiple standard formats (HL7 v2 messages, FHIR resources, C-CDA documents, etc.). We cover Synthea's features, CLI options, configuration, and customization, then survey other synthetic data generators (e.g. **MDClone**, **Inferno**, and various FHIR data generator libraries). We also discuss strategies for invoking these tools from .NET, extending Synthea's modules, leveraging standard medical code sets (SNOMED CT, LOINC, ICD-10, etc.), and validating the output against the respective standards.

Overview of Synthea: Synthetic Patient Generator

Synthea is a widely used open-source **Synthetic Patient Population Simulator** developed by The MITRE Corporation. It simulates lifetimes of synthetic patients from birth to death, generating comprehensive electronic health records (EHRs) that reflect realistic medical histories [3]. Synthea's default configuration models demographics on real population statistics (Massachusetts census by default) and includes an extensive library of modular clinical logic (**~90 disease modules**) covering common conditions and wellness behaviors [3]. Key features of Synthea include:

- **Longitudinal Records:** Each synthetic patient has a full longitudinal record of encounters, conditions, medications, labs, procedures, etc., potentially spanning decades [3]. Deceased patients are included to reflect full lifecycles.
- **Modular Disease Modeling:** Health events are driven by modular scripts (in a Generic Module Framework) that encapsulate progression and treatment guidelines for diseases [4] [5]. These modules were informed by clinical protocols and statistical research (CDC, NIH data). Developers can add or customize modules to simulate specific scenarios.
- **Configurable Demographics:** Synthea uses configuration-based demographics and statistics. By default it simulates a population resembling Massachusetts, but this can be adjusted to other locales or custom stats via config files [6]. You can specify state, city, gender, age ranges, etc. on the command line to tailor the population.
- **Standardized Output Formats:** Synthea can export synthetic health records in a variety of standard formats, including **HL7 FHIR** (supported versions: DSTU2, STU3, and R4), **C-CDA** (Consolidated Clinical Document Architecture), and CSV files, among others [7]. By tweaking settings, it can also output **Bulk FHIR** NDJSON files and a claims-oriented **CPCDS** format [8]. These outputs make Synthea data immediately usable with many health IT systems and standards.

Synthea CLI Usage and Options

Synthea is typically run as a standalone Java application (JDK 11+ required). Developers can use the provided `run_synthea` script or the fat JAR (`synthea-with-dependencies.jar`) to generate data via simple CLI commands [9]. The basic usage is:

```
run_synthea [options] [STATE [CITY]]
```

Options: Synthea's CLI supports various parameters to control the simulation. Common options include:

- **Population Size (`-p`)** – number of patient records to generate (e.g. `-p 100` for 100 patients) [10].
- **Random Seed (`-s`)** – seed for the random number generator to produce repeatable results [11] [12]. Using the same seed yields identical patient data each run.
- **State and City Filter** – specifying a state (and optionally city) will generate patients geographically located in that region (e.g. `run_synthea Massachusetts Boston`) [13]. Synthea uses built-in demographics for many U.S. states.
- **Gender (`-g`)** – limit generated patients to a specific gender (`M` or `F`) [14]. For example, `-g M` generates male patients only.
- **Age Range (`-a`)** – define a min-max age range for the generated population (e.g. `-a 60-65` for only senior patients) [14].
- **Output Directory (`--exporter.baseDirectory`)** – override the base output folder (default is `./output`). For example: `--exporter.baseDirectory="./output_tx"` to output files to a custom directory [14].
- **Custom Configuration File (`-c`)** – path to a custom synthea.properties file for overriding settings [15]. This allows using a tailored config per run.
- **Module Directory (`-d`)** – path to a directory of custom modules to include in the simulation [15]. Use this when you have created or modified module JSON files and want Synthea to use them instead of (or in addition to) the default modules.
- **Initial/Updated Population Snapshot (`-i`, `-u`)** – paths to input and output population snapshots, used for iterative runs. These options let you start from a previously generated population state and simulate forward (`-t` specifies the time period in days to advance) [15]. This is useful for continuing a simulation or applying updates to an existing synthetic population.

Example: To generate 100 female patients in Utah (Salt Lake City area) with a fixed seed, one might run:

```
./run_synthea -s 12345 -p 100 -g F Utah "Salt Lake City"
```

This produces 100 synthetic patient records of women in Salt Lake City, using seed 12345 for reproducibility.

Disease Module filtering (`-m`): Synthea also supports a `-m` **option to focus on a particular clinical module** or condition. For example, `-m Asthma` will ensure the *Asthma* module is loaded, attempting to simulate asthmatic patients [10]. **Note:** This does **not** force every patient to have that condition – it merely includes that module in all simulations. Patients still develop conditions according to incidence probabilities. In other words, if asthma has ~5% prevalence in the population, roughly 5 out of 100 generated patients might have asthma in that run [16]. You would need to generate a larger population or adjust module logic if you want more instances of a specific condition.

Running Synthea will print progress to the console and produce output files in the designated output directory. By default, each run creates multiple files per patient or per resource type, depending on format. For instance, **FHIR** records are exported as one JSON file per patient (containing a Bundle of that patient's resources), and **C-CDA** records are one XML document per patient, while **CSV** output is split into multiple files (patients.csv, encounters.csv, conditions.csv, etc.) representing normalized tables [17]. A single run can generate *dozens of files* if multiple formats are enabled. By default, Synthea outputs FHIR R4 data; enabling other formats requires configuration changes (discussed next).

Configuration and Export Formats

Synthea's behavior is extensively configurable via a properties file (`synthea.properties`). Key settings control which data formats are exported, the versions of standards, and various clinical parameters (population demographics, disease prevalence, etc.). **By default**, Synthea generates **FHIR R4** data in JSON (FHIR R4 is the default FHIR version) [18]. Other formats are available but disabled by default to save processing time and disk space [19]. You can activate these by editing the properties file or overriding settings on the CLI. Important configuration toggles include:

- **FHIR STU3 / DSTU2:** To generate FHIR STU3 (v3.0.1) or DSTU2 (v1.0.2) instead of R4, set the corresponding flags in `synthea.properties`. For example, to get STU3 output, you would disable R4 and enable STU3:

  ```
  exporter.fhir.export = false
  exporter.fhir_stu3.export = true
  ```

  Only one FHIR version should be true at a time, matching the version your target system expects [20]. By default, `exporter.fhir.export` (R4) is true and others false.

- **Bulk FHIR (NDJSON):** Synthea can produce **Bulk Data** files in NDJSON format (newline-delimited JSON) as per the FHIR Bulk Data specification. Enable this with:

  ```
  exporter.fhir.bulk_data = true
  ```

  When enabled, after a run you'll get NDJSON files for each resource type (Patient, Observation, Encounter, etc.) containing all patients' data, instead of per-patient bundles [21] [22]. This is useful for loading into systems via bulk import.

- **C-CDA Documents:** By default Synthea's C-CDA exporter is off. Turn it on with:

  ```
  exporter.ccda.export = true
  ```

Each patient will then have a C-CDA document (an XML file) summarizing their record [22] [19]. The C-CDA reflects encounters, medications, problems, etc., and is useful for testing document-based interoperability (e.g. hospital discharge summaries). Ensure you have the MDHT libraries if you plan to validate these (see Validation section).

- **CSV Files:** Synthea can output CSVs for analysis or for loading into relational databases. Activate with:

  ```
  exporter.csv.export = true
  ```

  When enabled, Synthea produces a set of CSV files (patients.csv, observations.csv, conditions.csv, etc.) in an `output/csv` directory [23]. These files form a relational representation of the generated data (a data dictionary is available to explain each field [17]).

- **CPCDS (Claims data):** Synthea can generate **CPCDS** (Common Payer Consumer Data Set) files, which mimic claims data in CSV form. Enable with:

  ```
  exporter.cpcds.export = true
  ```

  CPCDS is useful if you need synthetic claims for insurance or billing workflows. It includes fields like payer information, claim amounts, procedure and diagnosis codes, etc., aligned with the CARIN Blue Button data model.

All these settings (and any in the properties file) can be overridden on the CLI using `--config` flags. For example, one could run:

```
./run_synthea -p 10 --config exporter.csv.export=true --config
exporter.ccda.export=true
```

This would generate 10 patients and produce **both** CSV and C-CDA outputs (in addition to the default FHIR) in one run [14]. This approach is convenient for one-off runs or scripting Synthea without editing the properties file each time. Synthea's documentation notes that virtually *any* property can be passed this way (`--config <key>=<value>`) [24].

Output Locations: By default, Synthea creates an `output/` directory in the working folder. Within it, subfolders like `fhir`, `fhir_stu3`, `fhir_dstu2`, `csv`, `ccda` etc. will appear depending on what's enabled. For FHIR R4, the output may go to `output/fhir` or `output/fhir_r4` (the exact path can be configured in the properties). Ensure your .NET application knows where to find these files (or configure a custom `exporter.baseDirectory`). Synthea will print where it is writing files in the console log.

Standards Conformance: Synthea's generated data is meant to be syntactically and semantically valid according to the target standards. In fact, Synthea's maintainers state that the synthetic records are **conformant with HL7 FHIR (R4/STU3/DSTU2), the US Core implementation guide, C-CDA, and other**

formats [25]. This means the data uses proper coding (e.g. LOINC codes for labs, SNOMED CT for problems, etc.) and should pass basic validation (though always test, as noted later). The FHIR output, for example, is aligned with US Core profiles for key resources (Patients, Observations, Allergies, etc.), which is critical for interoperability testing.

Customizing and Extending Synthea Modules

One of Synthea's strengths is its **extensible modular architecture**. Clinical knowledge is encoded in **JSON module files** describing state machines (initial state, intermediate states for events, terminal state) that simulate a patient's progression through a health condition or care plan [4] [26]. Synthea includes modules for many conditions (diabetes, hypertension, COPD, etc.), preventive care (annual wellness visits, immunizations), and behaviors (smoking, exercise). Developers can modify these or create new modules to simulate custom scenarios.

Module Structure: A Synthea module is a JSON defining states (e.g. `ConditionOnset`, `MedicationOrder`, `Encounter`, `Delay`, `Death`, etc.) and transitions between them. Each module can have logic for when a patient enters it (e.g. entering the **Asthma** module if a patient develops asthma) and what events occur over time. The Generic Module Framework will evaluate modules at each time step of the simulation for each patient [26]. Modules often contain probabilistic branching (to reflect percentages of patients following different paths) and may reference external statistics (e.g. incidence rates by age). The Synthea GitHub repository contains all default modules in `src/main/resources/modules`. MITRE also provides companion guides and a **Module Builder** tool (a web-based editor) to help authors create modules with proper structure and logic (see Synthea Wiki for Module Companion Guides).

Adding Custom Modules: To add a new module or override an existing one, you can create a JSON file (following the GMF schema) and then run Synthea with the `-d` **option pointing to your module directory**. Synthea will load modules from that directory in addition to the built-in ones (or exclusively, if you disable default modules via config) [27]. For example:

```
./run_synthea -p 50 -d modules/my_custom_modules/
```

This would generate 50 patients using the modules in `my_custom_modules` (plus core modules). If your module is meant to replace a default one, ensure it has a unique name or disable the default via configuration. When developing modules, it's often useful to test with a small population and a fixed seed for repeatability, and to use the **Graphviz exporter** to visualize the module logic. Synthea can generate Graphviz `.dot` diagrams of modules by running `./gradlew graphviz`, which helps in debugging the flow of states.

Example Use-Case: Suppose you want to simulate a new condition not in Synthea, or tweak an existing one (e.g. a custom COVID-19 progression module). You could copy an existing module JSON as a template, modify the states (e.g. add a new medication or outcome), and then run Synthea with your module. Synthea's maintainers encourage community contributions, so you might even contribute your module back to the project. The **Synthea Wiki and GitHub Discussions** have FAQs and tips on module creation and evaluation. For instance, one common pitfall is forgetting that **all modules run continuously** – a module without delays or guards will execute its sequence immediately at patient birth, which might not be intended [28] . Thus, adding realistic entry conditions (like an onset state triggered at a certain age or risk factor) is important.

Customization via Config: Beyond modules, many high-level aspects of Synthea can be customized via properties. For example, `generate.demographics.default_file` can be changed to use a different population distribution file (if simulating a country other than the US, for instance). You can also adjust `exporter.years_of_history` to limit how much retrospective history each patient has by default [29] (Synthea typically generates a number of years of history for each patient to simulate past records). Fine-tuning such parameters can make your synthetic dataset more suitable for specific needs (e.g., shorter records for performance testing, or focusing on pediatric patients, etc.).

Quick Start and Cheat Sheets: For newcomers, Synthea provides a **Basic Setup and Running** guide on GitHub, and there are community "cheat sheets" and blogs summarizing usage. For example, one quick-start suggests: install JDK, download the Synthea JAR, then run a command like:

```
java -jar synthea-with-dependencies.jar -p 100 -g F -m Asthma --
exporter.csv.export true
```

This single line generates 100 patients (`-p 100`), female only (`-g F`), ensuring the Asthma module is included (`-m Asthma`), and outputs CSV files in addition to the default FHIR [10] . Such examples demonstrate how multiple options can be combined to tailor data generation. Always remember that Synthea will produce *a mix of conditions* unless you specifically modulate the epidemiological inputs – which is closer to real life. If you request 100 patients with `-m Asthma`, you might only get a handful of asthmatics, along with many other comorbidities, reflecting realistic prevalence [17] . This is by design, to keep the data clinically plausible.

Other Synthetic Health Data Generators and Tools

While Synthea is a go-to open source solution for synthetic patient records, there are several other tools and platforms for generating or using synthetic health data. Each has different strengths, and some may integrate more directly with certain environments (including .NET). Below we outline a few noteworthy options:

MDClone

MDClone is a commercial platform for on-demand synthetic data generation, widely used in healthcare organizations. Unlike Synthea's model-driven simulation, MDClone uses a **data-driven approach**: it takes real patient datasets and generates synthetic versions that preserve the statistical properties and correlations of the original data without any one-to-one mapping [30] . In effect, MDClone analyzes cohorts of real records and produces new, artificial records that **mimic the real population** distribution (often called the "derivation" method of synthetic data generation [31]). The synthetic data is *non-reversible* – you cannot pinpoint an actual patient – but it's statistically similar, allowing research or testing with minimal privacy risk [32] .

Key points about MDClone:

- It covers a broad range of healthcare data types (EHR data, claims, etc.), not just the clinical records that Synthea simulates.
- MDClone's platform includes a query interface where users define a cohort or dataset, then generate a synthetic version of that data. This makes it powerful for organizations with large real databases who want to safely share or analyze data.
- The output can often be obtained in formats like CSV or potentially FHIR/HL7 (depending on how the platform is set up), but MDClone is not an open-source CLI tool – it's a self-service web/cloud platform. Integration in a .NET app would likely mean using MDClone's API (if available) or exporting data from MDClone for use in your system.
- MDClone emphasizes **preserving multivariate statistics**. For example, if in the real data hypertension and diabetes are correlated, the synthetic data will reflect that correlation to the same degree. This is valuable for machine learning training or validating algorithms on "realistic" patterns.

From a .NET developer's perspective, you wouldn't embed MDClone in your application (unless your institution has it deployed and provides API access). Instead, you might use MDClone to **generate a synthetic dataset upfront**, then use that data as input files for your .NET application or database. MDClone ensures things like ICD-10 codes, lab values distributions, etc., closely mirror reality, since it's derived from actual patient data (subject to the quality of the source data).

ONC Inferno (and Crucible) – Test Suites with Synthetic Data

Inferno is an open-source testing suite for FHIR developed (and hosted) by the ONC. Its primary purpose is to **validate FHIR servers and apps for compliance** with standards (e.g., SMART on FHIR and US Core). While Inferno itself isn't a data generator in the way Synthea is, it *incorporates* synthetic data generation for testing purposes. Specifically, the Inferno **Community Edition** includes scripts that **generate synthetic patient data** conforming to the US Core Implementation Guide [33]. Under the hood, Inferno leverages Synthea to create these patients, then selects a small set that collectively have all required data elements for testing various FHIR API interactions [34] [35]. For example, the Inferno team provides a "**uscore-data-script**" that runs Synthea and picks out a minimal set of patients covering every US Core profile (there's even a so-called "Mr. Burns" patient that has *all* possible conditions, for edge-case testing) [36].

Crucible is another earlier MITRE project – a web-based FHIR testing tool – which also had the ability to **generate synthetic patient records** to populate a test FHIR server [37]. Crucible would use synthetic data to score implementations (e.g., checking if a server could handle a full set of patient data).

For .NET integration, you typically wouldn't use Inferno/Crucible to generate data on the fly in your application. Instead, you might use them in a **QA phase**: for example, after your .NET CLI generates FHIR or HL7 messages, you could run those outputs through Inferno's tests or validators to ensure they meet the expected standard and profiles. If your .NET CLI is actually serving as a FHIR server (less likely in this context), Inferno can be used to test your server's responses using its synthetic dataset.

In summary, **Inferno and Crucible** are **validation tools**: they come with synthetic data to facilitate testing. They demonstrate an important concept – using synthetic data to test interoperability. If your goal is to ensure your generated data is valid, you could leverage these tools or their data. For instance, after generating FHIR records with Synthea, you might run the HL7 FHIR Validator or use Inferno's suite to verify conformance (more on validation below).

FHIR Data Generator Libraries (Python and Others)

Beyond full frameworks like Synthea, there are lighter-weight tools for generating synthetic FHIR data, which can sometimes be easier to integrate or customize in a development pipeline:

- **SMART-on-FHIR Sample Patient Generator (Python):** The SMART team has a utility that generates FHIR **DSTU2/STU3** patient data using templates and data tables [38]. It's a Python project (`smart-on-fhir/sample-patients` on GitHub) where you can tweak CSV data files (for names, conditions, etc.) and run a script to produce a set of FHIR **Bundle** files. It's not as clinically sophisticated as Synthea (no longitudinal disease progression), but it's useful for quickly synthesizing patients with plausible data for demos or tests. You can run it via command line (`python generate.py --write-fhir`) to output a bunch of FHIR JSON files [39]. This could be invoked from .NET by calling the Python process, or you could pre-generate sample data and include those JSON files in your application.

- **FHIRGenerator (Python package):** There's a community Python package called `fhirgenerator` (available on PyPI) that programmatically creates FHIR R4 resources based on a configuration-driven approach [40] [41]. You define in a JSON or dict how many patients, the date range of encounters, distributions of conditions, etc., and it will generate FHIR Bundles accordingly. This is more lightweight and user-controlled than Synthea – you essentially tell it what kind of data to make. However, it may not capture complex clinical realism on its own. From .NET, using this directly would require calling out to Python as well, but it might be suitable if you need to script very specific test data (for example, "generate 5 patients all aged 40 with condition X").

- **Make Data (FHIR)**: There have been tools (often showcased at hackathons) like "**MakeData**" which generate FHIR data with a geographic or epidemiologic bent. For instance, one LinkedIn reference noted a *MakeData FHIR generator* that creates patients and data with realistic geographic distribution (this may refer to an extension or configuration of Synthea or a separate dataset-based approach). Although not widely documented, it indicates the ecosystem's effort to create shareable synthetic FHIR datasets for varied scenarios.

- **HL7 Reference Implementations:** HL7 organizations sometimes publish *example FHIR data* (e.g., sample patients for implementation guides). These are static examples, not random generators, but are useful for testing. For instance, the US Core IG includes sample resources. These aren't generators, but your CLI might incorporate them as baseline cases or for validation.

For a .NET developer, using a Python-based generator means managing a dual runtime (C# and Python). This could be done by having your CLI call an external Python script (perhaps configurable via your CLI's parameters) and then post-process the results in C#. Another approach is to re-implement small-scale data generation directly in C# using libraries: for example, using the **Firely .NET FHIR SDK** to construct FHIR objects and a library like **Bogus** (a .NET data faker) to fill in random demographics. This gives full control in C#, but be aware – creating *realistically correlated* medical data from scratch is very challenging. Tools like Synthea or MDClone encapsulate a lot of domain knowledge that simple randomization won't replicate.

Thus, a hybrid approach is often best: use Synthea (or similar) to generate a baseline of realistic data, then use .NET code to perhaps transform or filter that data, or convert it into other formats (like HL7 v2).

Other Notable Tools

A few other synthetic data initiatives worth mentioning:

- **Syntegra (Generative AI for Healthcare):** Syntegra is a startup that uses generative models (think advanced AI) to create synthetic EHR data. It's somewhat like MDClone in that it learns from real data, but uses modern machine learning to generate new patient records. It's not open-source, but it signifies a trend towards AI-driven synthetic data. If accessible, such a tool could produce very realistic data with complex correlations. Integration is again via APIs or downloading datasets, rather than embedding in a .NET CLI directly.

- **Synthetic Data Vault (SDV):** An open-source library (in Python) for generating synthetic data for tabular or relational datasets. Not healthcare-specific, but one could train it on healthcare data and generate similar data. For example, one could feed SDV a de-identified patient table and get synthetic patients out. The downside is you need real data to train it (which Synthea and others aim to eliminate the need for).

- **Mimic Simulator / EHRSim:** Some academic efforts have tried generating synthetic data based on real hospital data (like the MIMIC ICU dataset). These are more one-off and not widely available as tools, but worth knowing the landscape.

- **HL7 v2 Message Generators:** While not as common as FHIR generators, there are tools to create random HL7 v2 messages. For example, the community around integration engines like Mirth Connect has plugins to generate HL7 ADT messages for testing. InterSystems IRIS community has an **"HL7v2Gen"** tool that can generate HL7 v2 messages from templates [42]. These typically let you define a message structure and then produce many variants with random data. If your .NET CLI needs to output HL7 v2 directly, you might leverage a .NET HL7 library (like NHapi – a .NET port of the HAPI HL7v2 library) to programmatically construct messages, possibly feeding in data from Synthea's output (see next section).

In summary, **Synthea remains the most comprehensive open-source solution** specifically for health records. MDClone and Syntegra cater to organizations with real data, producing statistically faithful synthetic copies. Inferno and similar tools ensure your data or system meets standards by testing with synthetic data. And various smaller generators can fill specific gaps or be embedded with a bit of development effort. Many of these can complement each other – for example, you might use Synthea to create FHIR, then an HL7v2 generator to convert those records to messages, etc.

Invoking Synthetic Data Generators from a .NET C# Environment

Integrating these tools into a .NET CLI involves either calling external executables or using available libraries/APIs:

Running Synthea from C#

Since Synthea is a Java application, the typical integration pattern is to **treat it as an external process**. In your C# CLI program, you can use `System.Diagnostics.Process` to invoke Synthea. For example, you could package the Synthea JAR with your application or require the user to have it installed, then run:

```
ProcessStartInfo psi = new ProcessStartInfo("java",
    $"-jar synthea-with-dependencies.jar -p 1000 -c custom.properties");
psi.RedirectStandardOutput = true;
Process proc = Process.Start(psi);
proc.WaitForExit();
string outputLog = proc.StandardOutput.ReadToEnd();
```

This would launch Synthea to generate 1000 patients with a custom properties file. You'd likely want to monitor the exit code and handle any errors (e.g., if Java isn't installed or the jar path is wrong). One consideration is **performance and memory** – generating tens of thousands of patients can be memory-intensive. You might need to adjust JVM options (like `-Xmx` for heap size) which can also be done via the `ProcessStartInfo` arguments. In practice, generating a few thousand patients is quite fast (minutes) but very large populations (hundreds of thousands) could strain resources.

Another approach is using **Docker**. If Synthea is available as a Docker image (and indeed, community images exist [43] [44]), your .NET CLI could call `docker run` commands to execute Synthea in a container. For example, the command:

```
docker run --rm -v "$PWD/output":/data synthea/synthea -p 1000 Minnesota
```

could run Synthea in a container and output results to a mounted volume. Your C# code can trigger this via a Process call as well. Docker adds a dependency, but it simplifies having Java and Synthea installed locally. It's useful if you distribute your CLI with Docker or use it in CI pipelines.

After running Synthea, your .NET code can then **consume the output** – for instance, reading the generated FHIR JSON files into memory. Here the **Firely .NET FHIR SDK** is extremely handy: you can use `FhirJsonParser` to parse each Bundle, or even use the `Bundle` class to read from stream. Similarly, for CSV output, you can use .NET CSV parsers to ingest the data (if your CLI needs to further process or combine it). If the goal is simply to generate files and hand them off to the user, your CLI might not need to parse them at all, just orchestrate the generation.

Tip: If you plan to generate data repeatedly, consider keeping Synthea warm (in memory). Spawning a new JVM each time has overhead. One could envision a service mode for Synthea (not natively supported out-of-the-box) or reusing the same process for multiple generation tasks. However, simplest is often to run it per request.

Using .NET Libraries for HL7 v2 and FHIR

If your CLI needs to output HL7 v2 messages or do format conversion, you might combine Synthea's output with .NET libraries:

- For **FHIR**, the official **HL7 FHIR .NET API (Firely)** allows you to create, manipulate, and serialize FHIR resources in C#. You could load Synthea's FHIR JSON and then use the library to, say, post them to a FHIR server or convert to XML if needed. You could also generate FHIR data from scratch with this library (but as noted, it won't be clinically rich unless you supply the logic).

- For **HL7 v2**, libraries like **NHapi** (a .NET port of the HAPI Java library) or the **Microsoft HL7 SDK** can be used to construct and parse HL7 v2 messages. You might take Synthea's CSV output and map columns to HL7 v2 segments. For example, for each synthetic patient, you could create an ADT^A01 admit message: use patient demographic fields for PID segment, encounter data for PV1, diagnoses for DG1 segments, etc. This requires knowledge of HL7 v2 structure. Alternatively, if you have Synthea's FHIR, you can transform it to HL7 v2.

One clever solution from the community: use an interface engine or script to **convert Synthea FHIR to HL7v2**. For instance, an InterSystems IRIS-based tool was created to transform Synthea FHIR R4 resources into HL7 v2 messages using mapping logic [45] [46]. It maps FHIR fields and code systems to HL7 segments and codes (e.g., FHIR condition SNOMED codes to an HL7 DG1 using ICD-10). While that particular solution runs in IRIS (Dockerized), the concept applies generally – you might write a .NET conversion module or use a Python script with the **HL7apy** library to do the transformation. If implementing yourself, keep in mind HL7 v2 has different required fields and code systems (for example, admission type, PV1-2, etc., that FHIR might not directly have, and HL7 v2 often expects ICD-10 or local codes for diagnoses by default).

In summary, your .NET CLI can act as a **controller** that invokes specialized tools (like Synthea) and then uses .NET code to massage or route the data. This way you leverage Synthea's realism and .NET's ecosystem for any additional processing.

Integrating Other Tools

For other tools:

- **MDClone:** If you have access to MDClone, integration might be through exporting data (e.g., MDClone could output a CSV or SQL dump which your .NET app then imports). MDClone doesn't have a publicly available CLI to generate data that you can run locally; it's more of a platform. So this might be a manual or higher-level integration.

- **Inferno:** You likely wouldn't integrate Inferno into your CLI, but you might call its tests as part of a QA pipeline. For example, after your CLI generates FHIR, you could launch Inferno tests (Inferno is itself a Ruby app with a web interface, so integration is not trivial unless you use its API mode). An easier approach might be to use the **HL7 FHIR Validator** (a Java tool provided by HL7) to validate your FHIR files – you can call that from .NET similar to how you call Synthea. There's also a **HAPI FHIR CLI** that can validate FHIR resources. These could be bundled or invoked to give users feedback if the generated data has errors.

- **Python generators:** If using a Python tool (like the `fhirgenerator` or SMART sample patients), you'd again call it via `Process.Start("python", "...")`. Ensure the environment has the necessary Python packages installed. You might package those tools with your app or provide setup instructions. It might be easier to generate needed data ahead of time and include it, but that sacrifices flexibility.

Logging and Error Handling: When invoking external tools, capture their output and errors. Synthea outputs logs to STDOUT/STDERR; capturing those can be helpful for debugging (or even showing progress to the user in your CLI). For instance, you can read Synthea's log and display a progress bar or at least a "generation complete" message when it finishes.

Medical Code Sets and Enhancing Realism

Synthetic data is most useful when it uses **real medical codes and terminology**, so it looks and behaves like real data in systems. Synthea is designed with this in mind: it leverages standard code systems that are freely usable. By default, Synthea **does not use ICD-10** or other licensed codes, because of licensing restrictions [47]. Instead:

- Diagnoses and conditions are coded in **SNOMED CT** (a comprehensive clinical terminology that is free in many jurisdictions for use in synthetic data). SNOMED codes appear in problem lists, condition entries, etc.
- Lab tests and observations are coded with **LOINC** codes (Logical Observation Identifiers Names and Codes), which are freely available for use. For example, a lab result for glucose might use the appropriate LOINC code for "Glucose [Mass/volume] in Blood".
- Medications are represented using **RxNorm** codes (for drug ingredients or clinical drugs) and sometimes NDCs (National Drug Codes) for specific packages. RxNorm is a public U.S. standard for medications, making it suitable for synthetic data [47].
- Procedures and interventions in Synthea often use **SNOMED CT** as well, since CPT and HCPCS (common billing codes for procedures) are not freely licensed. Synthea's focus is clinical realism over billing, so using SNOMED for procedures is acceptable in many cases.
- Immunizations use **CVX codes** (CDC's immunization coding system), which are public.
- Synthea also includes socio-demographic data and social determinants; these might use standard codes when available (for example, US Census race codes, or LOINC for survey instruments, etc.).

For developers aiming to increase realism or integrate with systems expecting other code systems, there are a few approaches:

1. **Mapping SNOMED to ICD-10:** If your target system (say HL7 v2 messages or certain analytics) expects ICD-10-CM diagnosis codes, you'll need to map Synthea's SNOMED codes to ICD-10. There are publicly available maps (e.g., the NLM provides a SNOMED CT to ICD-10-CM map [48]). You could incorporate such a map into your CLI. For example, after generating data, post-process each condition: look up the SNOMED code in a mapping table and assign the corresponding ICD-10 code (perhaps in an additional field or as a translation in FHIR). Be cautious: mappings are often one-to-many or context-dependent, so this is a rough conversion. But for testing, it may suffice to at least include plausible ICD codes.

2. **Use ICD-9 (if needed):** As noted in a Synthea discussion, ICD-9 is public domain [49]. If ICD-10 is problematic, one could use ICD-9 codes, though those are outdated. Usually sticking to SNOMED/LOINC is fine for most modern use cases (since FHIR and C-CDA support SNOMED and LOINC natively).

3. **LOINC and UCUM units:** Synthea lab results come with LOINC codes and values with units (often in UCUM format). This is great for realism. Ensure when converting formats that you preserve these. For instance, if generating HL7 v2 OBX segments from Synthea, use the LOINC code as OBX-3 (Observation Identifier) and UCUM unit in OBX-6.

4. **Terminology Services:** For advanced usage, you might integrate a terminology service (like an instance of UMLS or FHIR Terminology Server) if you want to perform code translations or expansions. For example, to validate that all codes used are valid and maybe to translate SNOMED to patient-friendly text. This is beyond basic generation but can add value to your synthetic data pipeline (especially if presenting data to users).

5. **Realistic Values and Distributions:** Synthea already provides realistic ranges (e.g., lab results will fall in physiologically plausible ranges, vital signs vary with age, etc.). If you generate any data manually, try to follow known distributions or use reference ranges. For example, if you generate random blood pressures, ensure they fall in a human range (like systolic 90-180). Synthea modules handle this, but if you extend modules or create new ones, be mindful of the clinical realism of the values you assign.

In short, **Synthea's use of SNOMED CT, LOINC, RxNorm, and other open standards means the synthetic data will closely resemble real EHR data**. Many EHRs use these codes internally or can accept them. And since Synthea avoids proprietary codes by default, you don't have licensing headaches. If your use case absolutely needs ICD-10 or CPT codes (for example, a billing system test), you'll need to layer those on via mappings or using a different dataset (e.g., Synthea's CPCDS output might include some placeholder codes for claims). The community is aware of this gap – there have been requests for ICD-10 support, but it remains a licensing issue [50].

Remember to cite the source of codes in any documentation of your CLI to make users aware (e.g., "Diagnosis codes are synthetic SNOMED CT codes, not real patient codes"). While synthetic, these codes are real codes from the terminologies, so any terminology-aware software (like a FHIR server with a terminology module) can treat them normally (e.g., look up a SNOMED code's description).

Validation and Testing of Generated Data

After generating synthetic data in HL7 v2, FHIR, or C-CDA formats, it's crucial to **validate** that the data conforms to the standards and can be ingested by target systems (EHRs, analytics platforms, etc.). Best practices and tools include:

FHIR Validation

For FHIR output, the gold standard is the **HL7 FHIR Validator** tool. HL7 provides a Java CLI tool (`validator.jar`) that can validate FHIR resource files (JSON or XML) against the base spec and specific profiles. You can run it like:

```
java -jar validator.jar patient123.json -version 4.0.1 -profile
hl7.fhir.us.core/StructureDefinition/us-core-patient
```

This would validate a patient resource against FHIR R4 and US Core Patient profile. Given that Synthea aims to produce US Core conformant data [25], most resources should pass, but it's not uncommon to find small issues (e.g., a missing must-support element). Running the validator will catch these. As a .NET developer, you can include this validator jar and invoke it from your CLI (similar to invoking Synthea). Alternatively, **HAPI FHIR** has a validation library (and a CLI in the HAPI FHIR distribution) which you could integrate if you prefer a pure Java or embedded approach.

Inferno Testing: If you want deeper testing, the **Inferno** tool mentioned earlier can be used to test a set of FHIR resources or a FHIR server. Inferno's **Program Edition** is used for ONC certification and tests things like authentication and required search support. For static data validation, the Community Edition focusing on US Core is more relevant. It will check each resource to ensure it meets the US Core profiles and that all "Must Support" data elements are present at least somewhere in the patient set. The Inferno tests essentially simulate what a client application would need (e.g., can we get a smoking status Observation for each patient, etc.). Using Inferno might be overkill for basic validation, but it's a great way to **QA the realism and completeness** of your synthetic dataset. If your CLI is generating data to, say, preload a FHIR server, you might run Inferno against that server as a final verification.

HL7 v2 Validation

HL7 v2 messages are traditionally validated using profile conformance tools. A variety of options are available:

- **NIST HL7v2 Validation Suite:** NIST offers a web-based tool and downloadable library for HL7 v2 conformance [51]. This tool can validate messages against HL7 standard definitions or specific implementation guides (like immunization messaging, ELR, etc.). The NIST tools have an interface where you paste a message and get a report of errors (missing required fields, wrong value length, etc.). There's also a GitHub project encapsulating the NIST validator logic [52] which could be used for automated checks. For instance, CDC's immunization group provides a NIST validator for VXU messages. If your synthetic data includes immunization HL7v2 messages, that's a good choice.

- **HL7apy (Python):** This Python library can parse and validate HL7 v2 messages structure. You could write a small script to load each message and see if it conforms to HL7 v2.x structure (using the built-in HL7apy message definitions). This would catch structural issues like segments out of order or missing required segments. It won't enforce semantic constraints beyond structure unless you add them.

- **Integration Engine Tools:** Tools like **Mirth Connect** (now NextGen Connect) have message templates and a message generator plugin [42]. You might use Mirth in a testing capacity: import your synthetic HL7 messages into Mirth and see if it parses them without errors. If you have access to an EHR interface engine, that's often the true test – feeding the synthetic HL7 messages and verifying they are accepted. Mirth also has a built-in message structure viewer which is handy for spot-checking messages.

- **Community Validators/Parsers:** There are various small utilities (often on GitHub or as part of interface engines) for HL7. For example, the **Redox** engine (a healthcare integration platform) published an open source HL7 v2 parser/generator in Node.js [53] – not directly for validation, but it shows the structure. In .NET, not many modern HL7 v2 open validators exist aside from NHapi's parsing (which ensures the message can be parsed into an object model given a message structure definition).

When validating HL7 v2, pay attention to: required segments (e.g., PID must have certain fields, some segments like MSH need specific values in certain components), data types (e.g., date formats), and coding systems. If you converted Synthea data to HL7 v2, check that any code you put in, say, DG1-3 (Diagnosis Code) is valid for the coding system you specified (e.g., if using SNOMED, some systems might not expect that). In HL7 v2, often an "HL7 table" number or identifier accompanies coded fields (e.g., PV1-2 patient class uses a specific HL7 table of allowed values). Ensure you're populating those with allowed values.

C-CDA Validation

C-CDA documents (XML documents for patient summaries) are complex and require schema and schematron validation. Tools to validate C-CDA include:

- **MDHT (Model Driven Health Tools):** MDHT is an Eclipse-based toolkit that includes a C-CDA validator. The Synthea team used MDHT libraries to ensure their C-CDA conform to the C-CDA implementation guide. You can use the MDHT command-line or the **ONC C-CDA Validator** service. ONC provides a free **C-CDA Scorecard** web tool which not only validates conformance but also gives a "score" for how complete the document is. You could manually upload a few Synthea-generated CCDAs there to see if any errors arise. For automated use, you could incorporate MDHT in a Java utility that you call from .NET (again via a process call) – there isn't a simple out-of-the-box command-line for MDHT, but it's possible to write one. Alternatively, the **Trifolia CDA Validator** (from Aegis) is an option, although it's typically behind a login.

- **Schematron and XSD:** The C-CDA spec provides XSD and schematron files that define the rules. In theory, you could run an XML validation in .NET against those. For instance, use an XSLT processor to apply the schematron rules to each C-CDA XML file. This is a bit heavy on implementation, so using MDHT (which already has those rules encoded) is easier.

Synthea's C-CDA should be quite good; however, minor issues have been found by users in the past (for example, template IDs or code values that may not perfectly match the latest spec). Running validation helps catch those. If you find issues, you could correct the C-CDA output via post-processing or by adjusting Synthea's templates if you venture into the code.

Best Practices

- **Test Small Batches:** When setting up your .NET CLI and these generators, first run with a small population (e.g., 5 patients) and validate that output thoroughly. It's easier to debug formatting issues on a small set. Once it passes, scale up the generation.
- **Use Real Systems for Validation:** Nothing validates like trying to load the data into a real system. If you have access to a test FHIR server (e.g., HAPI FHIR server, or Azure API for FHIR), try uploading the FHIR resources – any issues will surface as errors from the server (which often use the same HL7 validator under the hood). Similarly, if you have an HL7 v2 consumer (like an open source EMR or a public health system test harness), send a few messages.
- **Automate QA:** Integrate validation into your workflow. For instance, if your CLI generates files, you can optionally have it run a validation step (perhaps behind a command flag, like `--validate`) which triggers the validators and summarizes results. This adds confidence for users that the data is standards-conformant. It's also useful if you modify your generator logic – you can quickly catch if a change introduced a conformance error.
- **Update Standards and Profiles:** Keep an eye on updates. For example, US Core profiles evolve (e.g., from STU3 to STU4 versions, not to be confused with FHIR STU3 vs R4). Synthea might lag or lead in certain areas. If a new required element comes (say US Core mandates something new), your synthetic data might need to adapt. Being able to validate against specific profiles (maybe user-selectable) can future-proof your CLI.
- **Document Assumptions:** In the output of your CLI or its documentation, note any known deviations. For example: "Note: HL7 v2 messages are generated using SNOMED codes for diagnoses in DG1 segments, which may not be typical in all systems. Map to ICD-10 as needed." This way, end-users know what to expect and how to handle the data.

Finally, leverage the community. Projects like Synthea have active communities (GitHub issues, forums) where people discuss issues like data validity and conversions. If you encounter trouble (say the FHIR validator flags an issue in Synthea output), chances are someone has reported it or there's a known workaround on those forums.

Conclusion

Building a .NET C# CLI to generate synthetic health data is highly achievable by combining the power of Synthea with targeted tools for specific formats. **Synthea** provides a rich, ready-to-use engine for generating realistic patient records in bulk, complete with standard codes and longitudinal detail. By mastering Synthea's CLI options and configuration, you can tailor the synthetic data to your needs (different locales, specific subsets of patients, various output formats). Through customization of modules, you can extend Synthea to cover new clinical scenarios, ensuring the data stays relevant as healthcare evolves.

Beyond Synthea, understanding **complementary tools** allows you to cover all bases: use **FHIR generators** or libraries for quick synthetic data in FHIR format when full clinical realism isn't required, and consider solutions like **MDClone** if you need statistically-grounded synthetic replicas of real datasets. Incorporating **HL7 v2 generation** might involve conversion steps or additional libraries, but it is facilitated by community contributions (like FHIR-to-HL7v2 mappers). Throughout, adherence to **standard code systems** like SNOMED CT and LOINC ensures your fake data looks real to software systems, with the option to map to others like ICD-10 if needed [47].

Crucially, a robust validation workflow with tools such as HL7's FHIR Validator, NIST's HL7 v2 test suite, and MDHT for C-CDA will give confidence that the synthetic data meets interoperability standards and can be consumed by healthcare applications without issues. Many organizations and government bodies (ONC, HL7, etc.) even run **data challenges and provide guidance** around using synthetic data [25] – reflecting a growing acceptance of these tools in accelerating innovation.

In summary, using Synthea and similar tools in a .NET CLI involves orchestrating cross-platform components: Java-based generators, possibly Python scripts, and .NET code all working in concert. With thoughtful design – capturing the strengths of each – you can deliver a one-stop solution that generates synthetic HL7 v2 messages, FHIR bundles, and C-CDA documents at the press of a button. This empowers developers and testers to have realistic health data on demand, fueling the development of health IT systems without compromising real patient information.

References and Resources:

- Synthea GitHub Repository and Wiki (documentation, module guides, issues) [11] [3]
- Synthea Technical Guides by ONC (PDFs and tips for using Synthea) [25]
- MITRE's Synthea website (module gallery and dataset downloads) [54] [55]
- MDClone Website and Publications (for understanding derivation method) [30]
- HL7 Inferno and Community Edition documentation (FHIR testing)
- FHIR Generator (Python) on PyPI and SMART Sample Patients on GitHub [38] [40]
- HL7 FHIR Validator Tool documentation (HL7 official site)
- NIST HL7 v2 Conformance Testing Toolkit [51] and MDHT C-CDA Validator resources.

By leveraging these tools and resources, you can ensure your .NET-based synthetic data CLI is both **comprehensive** in capabilities and **rigorous** in output quality – a valuable asset for health IT development and testing.

---

[1] [2] [4] [5] [54] [55] Synthea
https://synthetichealth.github.io/synthea/

[3] [6] [7] [9] [10] [16] [17] [23] Synthea: Do-It-Yourself Data. It is difficult to find patient-level... | by Robert (Bob) Hoyt MD FACP ABPM-CI FAMIA | Medium
https://rehoyt.medium.com/synthea-do-it-yourself-data-6ebe4d850db6

[8] [11] [12] [13] [14] [15] [19] [21] [22] [24] GitHub - synthetichealth/synthea: Synthetic Patient Population Simulator
https://github.com/synthetichealth/synthea

[18] [20] Import and export FHIR resources using Cloud Storage | Cloud Healthcare API | Google Cloud
https://cloud.google.com/healthcare-api/docs/how-tos/fhir-import-export

[25] [PDF] Synthetic Health Data Generation to Accelerate Patient-Centered ...
https://www.healthit.gov/sites/default/files/page/2022-03/20220314_Synthetic%20Data%20Final%20Report_508.pdf

[26] [27] [28] [29] How are custom modules created in the Synthea Module Builder evaluated? · synthetichealth synthea · Discussion #1040 · GitHub
https://github.com/synthetichealth/synthea/discussions/1040

[30] [32] Synthea: An approach, method, and software mechanism for ...
https://academic.oup.com/jamia/article/25/3/230/4098271

[31] Leveraging Artificial Intelligence and Synthetic Data Derivatives for ...
https://pmc.ncbi.nlm.nih.gov/articles/PMC10538345/

[33] [34] [35] [36] GitHub - inferno-framework/uscore-data-script: Generate realistic and complete synthetic test data for US Core v3.1.0
https://github.com/inferno-framework/uscore-data-script

[37] Testing Tools
https://help.edifecsfedcloud.com/FHIREducationCenter/Content/Testing_Tools.htm

[38] [39] GitHub - smart-on-fhir/sample-patients: Utilities to generate sample data as FHIR Resources
https://github.com/smart-on-fhir/sample-patients

[40] [41] fhirgenerator · PyPI
https://pypi.org/project/fhirgenerator/

[42] Sample HL7 message generator? - Forums - Mirth Community
https://forums.mirthproject.io/forum/open-source-hit/general-discussion-aa/5485-sample-hl7-message-generator

[43] smartonfhir/synthea - Docker Image
https://hub.docker.com/r/smartonfhir/synthea

[44] irisdemo-base-synthea - Package - InterSystems
https://openexchange.intersystems.com/package/irisdemo-base-synthea

[45] [46] Package
https://openexchange.intersystems.com/package/ks-fhir-gen

[47] [49] [50] Module for COVID-19 · Issue #679 · synthetichealth/synthea · GitHub
https://github.com/synthetichealth/synthea/issues/679

[48] [PDF] USER Manual CMS Synthetic RIF Files
https://data.cms.gov/sites/default/files/2023-05/d51e1218-68c3-4c7c-9598-0b81f22fe903/User%20Guide%20-%20CMS%20Synthetic%20RIF%20Files%20May%202023_AM508_v2.pdf

[51] HL7 v2 Conformance Testing Tools | NIST
https://www.nist.gov/itl/ssd/systems-interoperability-group/health-it-testing-infrastructure/testing-tools/hl7-v2

[52] CDCgov/lib-hl7v2-nist-validator - GitHub
https://github.com/CDCgov/lib-hl7v2-nist-validator

[53] RedoxEngine/redox-hl7-v2: Redox's in-house HL7v2 parser/generator
https://github.com/RedoxEngine/redox-hl7-v2