CS130: Software Engineering

Lecture 10: Class APIs

https://forms.gle/9jZ6TDt4XpQvZCc

- A word: What's your favorite slang term right now? (PG-13 max)
- A tweet: Do you like your current API? Why or why not?

Mid-quarter update

CS130 Goal

"Write a webserver"

CS130 Goal

Actually:

- Code reviews
- Testing
- Revision control
- Teamwork
- Tools, like coverage and static analysis

- Practice writing readable code
- Practice writing maintainable code
- Use a continuous build
- Learn how to design a good API
- Learn how to work with other people's code
- Learn how to work with frameworks and libraries

CS130 Anti-patterns

- Clever implementations
- Expecting unambiguous right and wrong answers

Expectations reminder

- Write and submit code
    - Readable code
- Review code
- Write tests
- Check your tests
    - Keep your build passing!

Class API Discussion

Outline

- Request handlers, dispatch, and configuration
- Review what everyone has done
- Consider some questions that arise
- Come up with a unified proposal
- This is a discussion!

- Assignment 6:
  Adopt the unified proposal

Config file format

Warm-up: comments

1. #
2. //
3. /* … */

Considerations:

- Can these appear in valid places in the config?
- What if they appear inside quoted string?

Specifying locations

1. Location-major untyped:

```
location /static1/ {
    root /files/;
}
```

2. Location-major typed:

```
location /static StaticHandler {
    root ./test_static_sites/basic;
}
```

3. Type-major:

```
static {
    /files /static_data;
    /photos /Desktop/pictures;
}
```

Considerations:

- Ease of writing
- Ambiguity when parsing
- Specifying multiple handlers
- Overlapping locations
- Abstraction of handlers
- Extensibility

Handler arguments

1. Location-major flat:

```
location /static StaticHandler
     ./test_static_sites/basic;
```

2. Location-major block, untyped:

```
location /static StaticHandler {
    /test_static_sites/basic;
}
```

3. Location-major block, typed:

```
location /static/ StaticHandler {
    root "../test_static_sites/basic";
}
```

Considerations:

- Ease of writing
- Ambiguity when parsing
- Specifying multiple arguments
- Duplicate arguments
- Extensibility

Handler arguments

1. Type-major flat:
```
static /static ./test_static_sites/basic;
```

2. Type-major block, untyped:
```
static {
    /static /test_static_sites/basic;
}
```

3. Type-major block, typed:
```
static {
    location "/static";
    root "../test_static_sites/basic";
}
```

Considerations:

- Ease of writing
- Ambiguity when parsing
- Specifying multiple arguments
- Duplicate arguments
- Extensibility

Misc

- Name of "/static" thing?
- Support quoted strings?
  a. Quotes inside?
- Filesystem paths
  a. Relative allowed
  b. Absolute only
- Trailing slashes on paths
  a. Optional
  b. Required
  c. Prohibited

Considerations:

- Ease of writing
- Ease of parsing
- Ease of testing
- Ease of use

Request handlers

Handler instantiation

1. Statically

```
RequestHandler* CreateMe(const string& name) {
  if (name == kStaticHandler) {
    return new StaticHandler(); }
  if (name == kEchoHandler) {
    return new EchoHandler(); }
}
```

2. Dynamically

Considerations:

- Some arguments (e.g. root path) come from config
- Some arguments (e.g. file name) come from request

Handler instantiation

1. Statically
2. Dynamically

```
static Registry::RegisterHandler(
    const string& name,
    RequestHandlerFactory factory) {
  _map[name] = factory;
}
REGISTER_HANDLER(Echo) =
  Registry::RegisterHandler(Echo::kName,
Echo::Init)

class EchoHandler :: public RequestHandler {
  static RequestHandler* Init(...) {
    return new EchoHandler(...);
  }
}
```

Considerations:

- Some arguments (e.g. root path) come from config
- Some arguments (e.g. file name) come from request

Handler initialization

1. With constructor:

```
new EchoHandler(
    const string& path, NginxConfig& config);
```

2. With post-construct method:

```
static EchoHandler::Init(
    const string& path, NginxConfig& config);
```

Considerations:

- Force uniform construction interface?
- What information is available vs needed at construction time?
- Which is easier to test?

Handler configuration

1. With server config block:

```
new StaticHandler(
    const string& path, NginxConfig& config);
```

2. With parsed arguments:

```
new StaticHandler(
  const string& path, const string& root_dir);

// or …
StaticArgsBuilder args;
args.set_root_dir(config->root_dir);
new StaticHandler(
    config string& path,
    StaticArgsBuilder args);
```

Considerations:

- Who should own knowledge about object construction?
- If you add a new Handler, where do you want to add construction code?

Pluggability

1. Server knows hard-coded types:

```
if (type == "static") new StaticHandler
```

2. Handlers create themselves:

```
static map<string type, Handler*>
   server_handler_map;

// In static_handler.cc:
server_handler_map.put(
    "StaticHandler", new StaticHandler());
```

Considerations:

- Extensibility
- Code changes when adding new handler
- Ease of writing code
- Ease of reading code

Handler lifetime

1. Long
    - Created at server startup
2. Short
    - Created per request

Considerations:

- Cost of instantiating handler
- Thread safety
- Configuring handler

Dispatching

Dispatching

1. Dispatcher knows paths:

```
if (prefix == "/static") then route
```

2. Dispatcher queries handlers:

```
class RequestHandler {
   bool can_serve(string& uri);
}
...
if handler.can_serve(url) then route
```

3. Dispatcher has map:

```
static map<string& path, Handler*>
   server_handler_map;
...
server_handler_map.get(path).handle(req)
```

Considerations:

- Must parse request path to dispatch?
- Does the handler know its own path?
- How is the path matched?
  By server?
  By handler?

Misc

- Does the handler receive the full path? or the relative path?
- What about routing precedence?
  a. First match in config?
  b. First match in map?
  c. Longest common prefix?

Considerations:

- Flexibility
- Correctness
- Ease of development

RequestHandler API

Requests and responses

1. Typed objects:

```
virtual Response HandleRequest(
   const Request& request) = 0;
```

2. Strings and individual values:

```
std::string get_response(
   size_t bytes_transferred,
   char* data_);
```

Considerations:

- Ease of reading
- Code reuse
- Flexibility
- Ease of writing

Returning data

1. Copy of response
```
virtual response handle_request(
    const request& req);
```

2. Handler-allocated response
```
std::shared_ptr<reply> HandleRequest(
    const request& request);
```

3. void (w/ outparam)
```
virtual void handle_request(
    const request& req, reply& rep);
```

4. status (w/ outparam)
```
http::server::reply::status_type ...
```

Considerations:

- Efficiency
- Memory management
- Extensibility
- Importance of data

Misc

- Name of handle method:
  a. HandleRequest()
  b. handle_request()
  c. handle()
- What kind of Request/Response object?
- How to handle large data?

Considerations:

- Consistency
- Readability
- Code reuse
- Extensibility
- Robustness