SUNNY: From Models to Interactive Web Apps for (almost) free

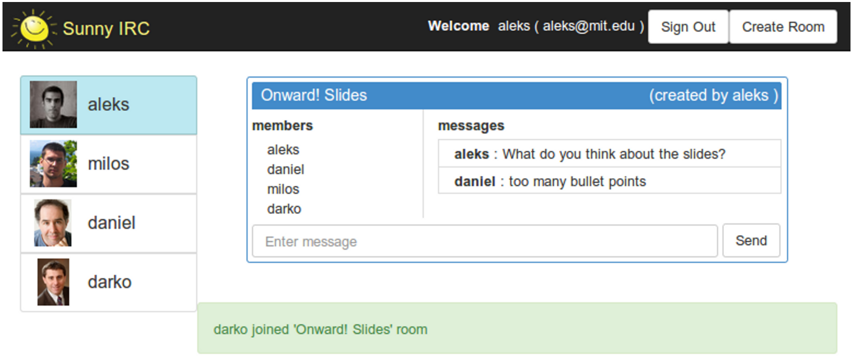
Aleksandar Milicevic
Daniel Jackson
{aleks,dnj}@csail.mit.edu

Milos Gligoric
Darko Marinov
{gliga,marinov}@illinois.edu

Onward! 2013
Indianapolis, IN

A simple web app: Sunny IRC

custom-tailored internet chat relay app



Sunny IRC

Welcome aleks (aleks@mit.edu)

Sign Out | Create Room

- aleks
- milos
- daniel
- darko

Onward! Slides (created by aleks)

members

- aleks
- daniel
- milos
- darko

messages

aleks : What do you think about the slides?

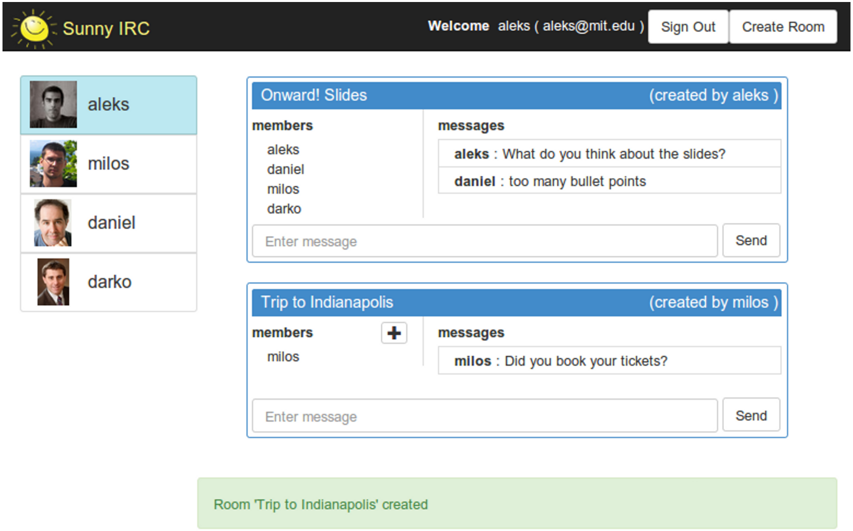
daniel : too many bullet points

Enter message | Send

darko joined 'Onward! Slides' room

A simple web app: SUNNY IRC

custom-tailored internet chat relay app



Onward! Slides (created by aleks)

members: aleks, daniel, milos, darko

messages

aleks : What do you think about the slides?

daniel : too many bullet points

Trip to Indianapolis (created by milos)

members: milos

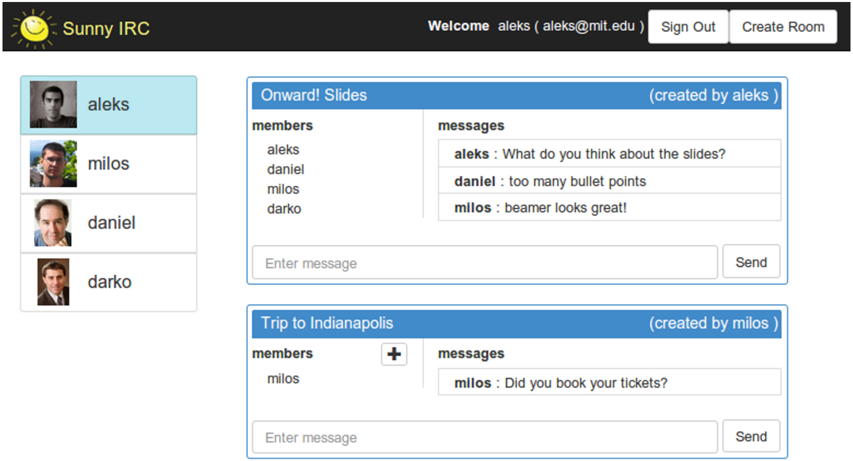
messages

milos : Did you book your tickets?

Room 'Trip to Indianapolis' created

A simple web app: SUNNY IRC

custom-tailored internet chat relay app



aleks : What do you think about the slides?

daniel : too many bullet points

milos : beamer looks great!

milos : Did you book your tickets?

Conceptually simple, but in practice...

Conceptually simple, but in practice...

- **distributed system**
  - → concurrency issues
  - → keeping everyone updated

Conceptually simple, but in practice...

- **distributed system**
  - → concurrency issues
  - → keeping everyone updated
- **heterogeneous environment**
  - → rails + javascript + ajax + jquery + ...
  - → html + erb + css + sass + scss + bootstrap + ...
  - → db + schema + server config + routes + ...

Conceptually simple, but in practice...

- **distributed system**
  - → concurrency issues
  - → keeping everyone updated
- **heterogeneous environment**
  - → rails + javascript + ajax + jquery + ...
  - → html + erb + css + sass + scss + bootstrap + ...
  - → db + schema + server config + routes + ...
- **abstraction gap**
  - → high-level problem domain
  - → low-level implementation level

Conceptually simple, but in practice...

- **distributed system**
  - → concurrency issues
  - → keeping everyone updated
- **heterogeneous environment**
  - → rails + javascript + ajax + jquery + ...
  - → html + erb + css + sass + scss + bootstrap + ...
  - → db + schema + server config + routes + ...
- **abstraction gap**
  - → high-level problem domain
  - → low-level implementation level

MDD: how far can it get us?

exercise:

sketch out a model (design, spec)
for the Sunny IRC application

Sunny IRC: data model

```
record User < WebUser do
  # inherited fields
  #   name: String,
  #   email: String,
  #   pswd_hash: String,
end
```

```
record Msg do
  refs text: Text,
       sender: User
end
```

```
record ChatRoom do
  refs name: String,
       members: (set User)
  owns messages: (set Msg)
end
```

- record-like data structures with typed fields
- automatically persisted

Sunny IRC: machine model

```
machine Client < WebClient do
  # inherited fields
  #   auth_token: String
  refs user: User
end
```

```
machine Server < WebServer do
  # inherited fields
  #   online_clients: (set WebClient)
  owns rooms: (set ChatRoom)
end
```

- generic network architecture
- machines are records too ($\Longrightarrow$ persisted, have fields)
- assumes certain (standard) properties of web severs and clients

Sunny IRC: event model

```
event JoinRoom do
  from   client: Client
  to     serv:   Server
  params room:   ChatRoom

  requires     { !room.members.include?(client.user) }
  ensures      { room.members << client.user }
  success_note { "#{client.user.name} joined '#{room.name}' room" }
end
```

- core functionality of the system

Sunny IRC: event model

```
event JoinRoom do
  from   client: Client
  to     serv:   Server
  params room:   ChatRoom

  requires     { !room.members.include?(client.user) }
  ensures      { room.members << client.user }
  success_note { "#{client.user.name} joined '#{room.name}' room" }
end
```

- core functionality of the system
- other IRC events: `CreateRoom`, `SendMsg`
- included library events: CRUD operations, user Auth events

Modeling done. What next?

challenge

how to make the most of this model?

Modeling done. What next?

challenge

how to make the most of this model?

goal

make the model executable as much as possible!

Traditional MVC Approach

Traditional MVC Approach

- boilerplate:
  - → write a matching DB schema
  - → turn each record into a resource (model class)
  - → turn each event into a controller and implement the CRUD operations
  - → configure URL routes for each resource

Traditional MVC Approach

- boilerplate:
  - → write a matching DB schema
  - → turn each record into a resource (model class)
  - → turn each event into a controller and implement the CRUD operations
  - → configure URL routes for each resource
- aesthetics:
  - → design and implement a nice looking GUI

Traditional MVC Approach

- boilerplate:
  - → write a matching DB schema
  - → turn each record into a resource (model class)
  - → turn each event into a controller and implement the CRUD operations
  - → configure URL routes for each resource
- aesthetics:
  - → design and implement a nice looking GUI
- to make it interactive:
  - → decide how to implement server push
  - → keep track of who's viewing what
  - → monitor resource accesses
  - → push changes to clients when resources are modified
  - → implement client-side Javascript to accept pushed changes and dynamically update the DOM

Traditional MVC Approach

in SUNNY:

- ~~boilerplate:~~
  - → ~~write a matching DB schema~~
  - → ~~turn each record into a resource (model class)~~
  - → ~~turn each event into a controller and implement the CRUD operations~~
  - → ~~configure URL routes for each resource~~
- aesthetics:
  - → design and implement a nice looking GUI
- ~~to make it interactive:~~
  - → ~~decide how to implement server push~~
  - → ~~keep track of who's viewing what~~
  - → ~~monitor resource accesses~~
  - → ~~push changes to clients when resources are modified~~
  - → ~~implement client-side Javascript to accept pushed changes and dynamically update the DOM~~

GUIs in SUNNY: dynamic templates

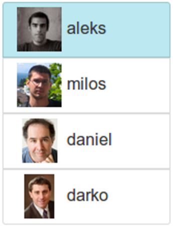
- like standard templating engine (ERB) with data bindings
- automatically re-rendered when the model changes

GUIs in SUNNY: dynamic templates

- like standard templating engine (ERB) with data bindings
- automatically re-rendered when the model changes

online_users.html.erb

```
<div class="list-group">
<% server.online_clients.user.each do |user| %>
  <%= img_tag_for user %>
  <div class="... <%= (user == client.user) ? 'me' : '' %>">
    <h4 class="..."><%= user.name %></h4>
  </div>
<% end %>
</div>
```

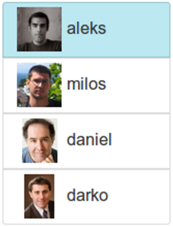


GUIs in SUNNY: dynamic templates

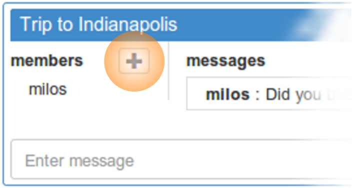
- like standard templating engine (ERB) with data bindings
- automatically re-rendered when the model changes

online_users.html.erb

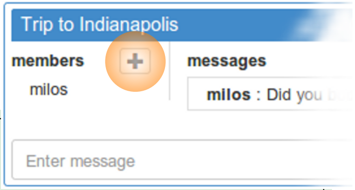
```
<div class="list-group">
<% server.online_clients.user.each do |user| %>
  <%= img_tag_for user %>
  <div class="... <%= (user == client.user) ? 'me' : '' %>">
    <h4 class="..."><%= user.name %></h4>
  </div>
<% end %>
</div>
```



GUIs in SUNNY: binding to events



GUIs in SUNNY: binding to events

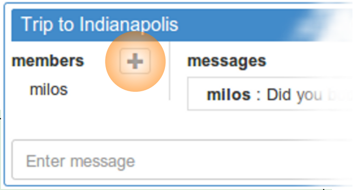


milos : Did you

room_members.html.erb

```
<% unless chat_room.members.member?(client.user) %>
  <button class="..." type="button"
          data-trigger-event="JoinRoom"
          data-param-room="${new ChatRoom(<%= chat_room.id %>)}">...</button>
<% end %>
```

GUIs in SUNNY: binding to events

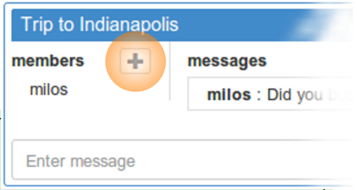

Trip to Indianapolis
members
milos
messages
milos : Did you
Enter message

room_members.html.erb

```
<% unless chat_room.members.member?(client.user) %>
  <button class="..." type="button"
          data-trigger-event="JoinRoom"
          data-param-room="${new ChatRoom(<%= chat_room.id %>)}">...</button>
<% end %>
```

- html5 `data` attributes specify event type and parameters
- dynamically discovered and triggered asynchronously
- no need to handle the Ajax response
  - → the data-binding mechanism will automatically kick in if the event makes any changes

GUIs in SUNNY: binding to events



room_members.html.erb

```
<% unless chat_room.members.member?(client.user) %>
  <button class="..." type="button"
          data-trigger-event="JoinRoom"
          data-param-room="${new ChatRoom(<%= chat_room.id %>)}">...</button>
<% end %>
```

- html5 `data` attributes specify event type and parameters
- dynamically discovered and triggered asynchronously
- no need to handle the Ajax response
  → the data-binding mechanism will automatically kick in if the event makes any changes

demo

- responsive GUI without messing with javascript

Adding New Features: adding a field

implement user status messages

Adding New Features: adding a field

implement user status messages

- all it takes:

```
record User < WebUser do
  refs status: String
end
```

```
<%= autosave_fld user,
                 :status,
                 :default => "...statusless..." %>
```

Adding New Features: adding a field

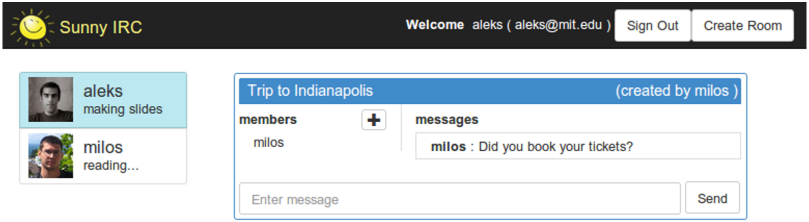
implement user status messages

- all it takes:

```
record User < WebUser do
  refs status: String
end
```

```
<%= autosave_fld user,
                 :status,
                 :default => "...statusless..." %>
```

demo



milos : Did you book your tickets?

Adding New Features: adding a ’write’ policy

forbid changing other people’s data

- by default, all fields are public
- policies used to specify access restrictions

Adding New Features: adding a ’write’ policy

forbid changing other people’s data

- by default, all fields are public
- policies used to specify access restrictions

```
policy EditUserData do
  principal client: Client

  @desc = "Can’t edit other people’s data"
  write User.*.when do |user| client.user == user end
end
```

Adding New Features: adding a ’write’ policy

forbid changing other people’s data

- by default, all fields are public
- policies used to specify access restrictions

```
policy EditUserData do
  principal client: Client

  @desc = "Can’t edit other people’s data"
  write User.*.when do |user| client.user == user end
end
```

- declarative and independent from the rest of the system
- automatically checked by the system at each field access

Adding New Features: adding ’read’ policies

hide status messages in certain cases

- show only if the two users share a room

Adding New Features: adding ’read’ policies

hide status messages in certain cases

- show only if the two users share a room

```
@desc = "Must share a room to see user’s status message"
read User.status.when do |user|
  client.user == user ||
    server.rooms.some?{|room| room.members.contains?([user, client.user])}
end
```

Adding New Features: adding 'read' policies

hide status messages in certain cases

- show only if the two users share a room

```
@desc = "Must share a room to see user's status message"
read User.status.when do |user|
  client.user == user ||
    server.rooms.some?{|room| room.members.contains?([user, client.user])}
end
```

invisible users

- hide users whose status is “busy”

Adding New Features: adding ’read’ policies

hide status messages in certain cases

- show only if the two users share a room

```
@desc = "Must share a room to see user’s status message"
read User.status.when do |user|
  client.user == user ||
    server.rooms.some?{|room| room.members.contains?([user, client.user])}
end
```

invisible users

- hide users whose status is “busy”

```
@desc = "Hide ’busy’ users"
restrict Client.user.when do |c|
  c != client && c.user.status == "busy"
end
```

Adding New Features: adding 'read' policies

hide status messages in certain cases

- show only if the two users share a room

```
@desc = "Must share a room to see user's status message"
read User.status.when do |user|
  client.user == user ||
    server.rooms.some?{|room| room.members.contains?([user, client.user])}
end
```

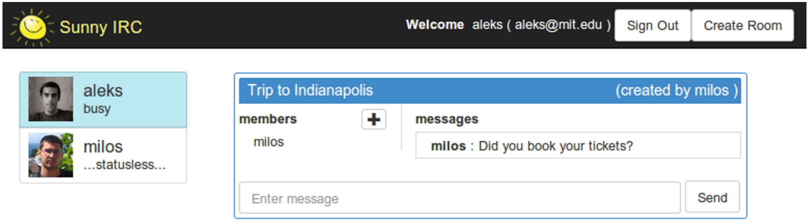
invisible users

- hide users whose status is “busy”

```
@desc = "Hide 'busy' users"
restrict Client.user.when do |c|
  c != client && c.user.status == "busy"
end
```

no GUI templates need to change!

Demo: defining access policies independently



Welcome aleks (aleks@mit.edu)

aleks
busy

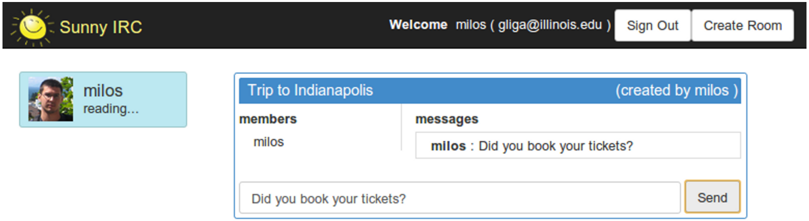
milos
...statusless...

Trip to Indianapolis (created by milos)

members
milos

messages
milos : Did you book your tickets?

Enter message



Welcome milos (gliga@illinois.edu)

milos
reading...

Trip to Indianapolis (created by milos)

members
milos

messages
milos : Did you book your tickets?

Did you book your tickets?

More cool policy examples

- private messages: message text starts with @username

```
@desc = "filter out messages that start with '@' but not '@#{client.user.name} '"
filter ChatRoom.messages.reject do |room, msg|
  msg.sender != client.user &&
    msg.text.starts_with?("@") &&
    !msg.text.starts_with?("@#{client.user.name} ")
end
```

More cool policy examples

- private messages: message text starts with @username

```
@desc = "filter out messages that start with '@' but not '@#{client.user.name} '"
filter ChatRoom.messages.reject do |room, msg|
  msg.sender != client.user &&
    msg.text.starts_with?("@") &&
    !msg.text.starts_with?("@#{client.user.name} ")
end
```

- private rooms: if room name starts with "private", show messages to members only

```
@desc = "if room name starts with '#private', show messages only to members"
restrict ChatRoom.messages.when do |room|
  !room.members.include?(client.user) &&
    room.name.starts_with?("#private")
end
```

SUNNY IRC: what was hard?

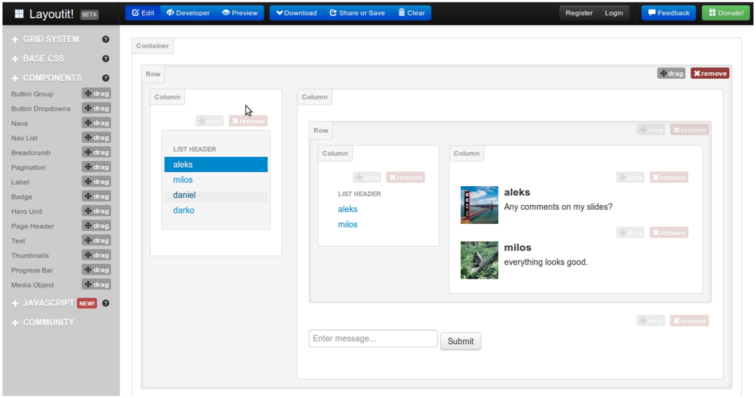
HTML & CSS for GUI templates

- least fun, most tedious

SUNNY IRC: what was hard?

HTML & CSS for GUI templates

- least fun, most tedious
- **future work**: the SUNNY approach lends itself to MBUI builders



Related Model-Driven Technologies

scaffolding (as in Rails)

- uses transient models for one-off code generation
  - → beneficial mostly for the first prototype application

Related Model-Driven Technologies

scaffolding (as in Rails)

- uses transient models for one-off code generation
  - → beneficial mostly for the first prototype application
- in SUNNY
  - → permanent models, fundamental part of the running system

Related Model-Driven Technologies

scaffolding (as in Rails)

- uses transient models for one-off code generation
  - → beneficial mostly for the first prototype application
- in SUNNY
  - → permanent models, fundamental part of the running system

traditional MDD

- permanent models, but external to the running system
  - → code generation used to generate an implementation
  - → roundtrips possible, but limited and discouraged

Related Model-Driven Technologies

scaffolding (as in Rails)

- uses transient models for one-off code generation
  - → beneficial mostly for the first prototype application
- in SUNNY
  - → permanent models, fundamental part of the running system

traditional MDD

- permanent models, but external to the running system
  - → code generation used to generate an implementation
  - → roundtrips possible, but limited and discouraged
- in SUNNY
  - → first-class models, interpreted at runtime
  - → the SUNNY modeling language is embedded in standard Ruby
  - → no code generation needed beforehand
  - → the models are the running code (reduces the paradigm gap)

Related “Web 3.0” Technologies

Meteor

- low-level mechanism for automatic data propagation
- all javascript framework
- no explicit system model, no type information
  - → doesn’t get many of the MDD benefits

Related “Web 3.0” Technologies

Meteor

- low-level mechanism for automatic data propagation
- all javascript framework
- no explicit system model, no type information
  - → doesn’t get many of the MDD benefits
- SUNNY
  - → strives to provide a higher-level programming paradigm
    - addresses software design questions
    - imposes a more structured (model-based) approach
    - aims to bridge the gap between formal specification and executable implementation

Related “Web 3.0” Technologies

Meteor

- low-level mechanism for automatic data propagation
- all javascript framework
- no explicit system model, no type information
  → doesn’t get many of the MDD benefits
- SUNNY
  → strives to provide a higher-level programming paradigm
    - addresses software design questions
    - imposes a more structured (model-based) approach
    - aims to bridge the gap between formal specification and executable implementation

  → another implementation of SUNNY could be built on top of Meteor

SUNNY: the big picture

SUNNY: the big picture

centralized *unified model* of the system

- formal, analyzable modeling language (inspired by Alloy)
- fully executable

SUNNY: the big picture

centralized *unified model* of the system

- formal, analyzable modeling language (inspired by Alloy)
- fully executable

goal: *maximize* benefits of *model-driven* development

- automatic data persistence and ORM
- sequential semantics of a distributed system
- automatic data propagation
- automatic policy checking
- generic model-based UI builder
- formal analysis, verification, model checking, model-based testing

SUNNY: the big picture

centralized *unified model* of the system

- formal, analyzable modeling language (inspired by Alloy)
- fully executable

goal: *maximize* benefits of *model-driven* development

- automatic data persistence and ORM
- sequential semantics of a distributed system
- automatic data propagation
- automatic policy checking
- generic model-based UI builder
- formal analysis, verification, model checking, model-based testing

applications: event-driven distributed systems, web apps, robots

SUNNY: the big picture

centralized *unified model* of the system

- formal, analyzable modeling language (inspired by Alloy)
- fully executable

goal: *maximize* benefits of *model-driven* development

- automatic data persistence and ORM
- sequential semantics of a distributed system
- automatic data propagation
- automatic policy checking
- generic model-based UI builder
- formal analysis, verification, model checking, model-based testing

applications: event-driven distributed systems, web apps, robots

Thank You!

SUNNY: coming for holidays 2013